

HeapVision: Debugging by Interactive Heap Navigation

Jibin Ou

Master's Thesis
March 2014

Supervisor:

Prof. Dr. Otmar Hilliges, Prof. Dr. Martin Vechev

Co-supervisor:

Prof. Dr. Jan Borchers

ETH zürich



Advanced Interactive
Technologies

SRL
SOFTWARE RELIABILITY LAB

Abstract

With the proliferation of online source code repositories such as GitHub and BitBucket, programmers have instant access to countless code examples. With the availability of these resources the focus in software development shifts away from writing code towards understanding source code. Many common algorithms, that see a lot of reuse across different problem domains, involve complex manipulations of data structures and hence the program's heap (e.g., sorting algorithms). Understanding these manipulations and the effect onto the heap's structure is a cognitively demanding and complex task. However, despite the importance of such algorithms for many applications, there is little tool support build into current IDEs (Integrated development environment) to help programmers in understanding and debugging such algorithms and data structures. In particular, there currently does not exist any sophisticated tool to visually explore the heap structure and to interactively experience the impact of the algorithms instructions on the heap throughout the execution of the program.

In the course of this thesis we have developed a tool called HeapVision, which allows for inspection of and interaction with dynamic visual representations of the heap structure. The interface allows programmers to more easily understand how a data structure and it's content are modified by a particular sequence of instructions. Furthermore, we have developed a novel pen and touch based interface that allows developers to more directly interact with the running process and to concentrate on the essence of the program. Pen and touch gestures can be used to dynamically change the visual representation of the heap, allowing the user to concentrate on important parts of the heap. We leverage formal program analysis tools in order to automatically abstract away unimportant aspects of the heap, while keeping the parts of a data structure that are being manipulated concrete. In other words we aim to capture the essence of an algorithm's manipulations so that the user may understand the underlying working principle. Our application can improve the comprehension of algorithms, especially for algorithms that manipulate data structures recursively – such as but not limited to reversing linked lists.

Finally, we have conducted a number of low-level system evaluations on the main components of the developed application. Showing that our application allows debugging of different sized data structures at interactive rates without slowing down the debug process itself. Moreover, we have elicited preliminary user feedback from actual software developers. While this feedback is at this point only informal it suggests that the developed visual debugging tool can help developers in understanding heap manipulating algorithms more easily.

The contribution of this thesis are:

- A novel combination of state-of-the program reasoning tools and an advanced human-computer interface for the dynamic inspection of a running process.
- A formalized method to abstract away unimportant aspects of dynamic data structures, based on shape analysis;
- A framework for the visualization of and interaction with such semi-abstract heaps, based on pen and touch interaction.

Acknowledgment

I would like to thank Prof. Dr. Otmar Hilliges and Prof. Dr. Martin Vechev for supervising this project. In addition, I would like to also thank Chat Wacharamanotham and Prof. Dr. Jan Borchers for the co-supervising. I thank you all to enable this valuable opportunity to combine the knowledge of human computer interaction and program analysis. Although these 9 months were really challenging, I did gained a lot of knowledge, which is useful in my future study and career. Therefore, I will continue to pay effort on the research of end-user programming and try to have a position in the research community.

I would also express my gratitude to my parents and my girlfriend. Without your support, I could not easily handle the difficulties in my master study.

Contents

List of Figures	vii
List of Tables	ix
1. Introduction	1
1.1. Heap Manipulating Data Structures	2
1.2. Our Approach	2
1.3. Outline of Thesis	5
2. State of the Art	7
2.1. Program Visualization for Education	8
2.2. Program State and Heap Visualization	9
2.2.1. Heap visualization	10
2.3. Shape Analysis	13
2.4. Pen and Touch Interaction	17
2.5. Live Programming	18
3. Enabling Deep Program Interaction	21
3.1. Visual Notation	21
3.2. Visual Interaction	23
3.3. Computing The Initial Abstract Heap	28
3.4. Incremental Graph Drawing Algorithms	29
4. Implementation	35
4.1. Architecture of Application	35
4.2. User Interface Design	36

Contents

4.3.	Graph Drawing Framework	37
4.3.1.	Graph visualization	38
4.3.2.	Implementation of graph layout algorithm	41
4.3.3.	Graph Interaction	42
4.4.	Custom Protocol for Communication	44
4.5.	Debug Plugin	45
4.5.1.	Java Debug Interface	45
4.5.2.	Heap traversal algorithm	46
5.	Evaluation	49
5.1.	Experiment: Performance of basic algorithms	49
5.1.1.	Test program and method	50
5.1.2.	Results and discussion	50
5.2.	User evaluation	52
5.2.1.	Overview of program comprehension evaluation	52
5.2.2.	Initial questions	53
5.2.3.	Methodology	54
5.2.4.	Results	54
5.3.	Discussion	55
5.4.	Limitations	56
6.	Conclusion and Future Work	57
6.1.	Future work	58
A.	Appendix	61
A.1.	Definitions	61
A.2.	Figures	62
A.3.	Questionnaire	63
	Bibliography	71

List of Figures

1.1.	Concrete states of a linked list reversal program	3
1.2.	Two abstracted states of the linked list reversal program	4
1.3.	Two abstracted states in the middle of a linked list reversal program	4
2.1.	Graphical model of stack and heap in Java	10
2.2.	Variables' view in Eclipse IDE	10
2.3.	A double linked list in DDD Debugger	11
2.4.	Control flow graph of the creation program	16
2.5.	Prototype of live programming by Victor	19
3.1.	An example of the abstraction and concretization	26
4.1.	Architecture of HeapVision	36
4.2.	Interactions between different components	36
4.3.	User interface design of the frontend application	37
4.4.	Software prototype using Prefuse framework	40
4.5.	Comparison between original and current graph rendering layers	42
4.6.	Class diagram of debug models in JDI	46
4.7.	Workflow of the backend when a breakpoint is hit	46
5.1.	Execution time of heap traversal and graph data serialization\transmission	51
5.2.	Execution time of graph data deserialization and TVLA's <i>blur</i> process in the backend	51
5.3.	Summation of the execution time used for producing a visualization for the linked list	52
5.4.	Results of qualitative questions in the online survey	55

List of Figures

6.1. Current visualization of a double linked list	59
6.2. Ideal visualization and interaction of a double linked list	59
A.1. Sequence diagram of the whole HeapVision application	62

List of Tables

2.1.	Truth table in three-valued logic	14
2.2.	Example of predicates in TVLA	15
2.3.	Possible heap shapes of different states in the linked list creation program . . .	16
3.1.	Visualizing elements in Java	22
3.2.	Visual notations in HeapVision	22
3.3.	List of gestures for graph visualization and manipulation	24
3.4.	Shorthand notations of α and γ algorithms	25
3.5.	Abstraction and Concretization Maps for Nodes and Edges	26
3.6.	Predicates used for abstracting a singly-linked list	28
3.7.	<i>Blur</i> process for linked list abstraction	29
3.8.	Drawing graphs using the <i>GraphViz</i> layout engine	31
3.9.	Objectives and constraints in GraphViz and DynaDAG	32
4.1.	Comparison between existing graph drawing frameworks	39
4.2.	Input and output of the incremental graph layout engine	41
4.3.	Animations in HeapVision	43
4.4.	Elements in the custom protocol	44
4.5.	Messages of the custom protocol for communication between backend and front-end	45
5.1.	Execution time of different components	50
A.1.	Formal definition of variables and values in Java	61
A.2.	Formal definition of types in Java	62

List of Tables

1

Introduction

In the last few decades, programming has become ubiquitous, yet fundamentally, the way developers interact with programs has mostly remained the same. Indeed, the interface to building, understanding and debugging programs has undergone only small changes. That is, the main physical devices and interfaces for programming and program understanding have remained virtually identical to three decades ago, when the WIMP paradigm was used for designing the graphical user interface. This lack of progress has negative effects: for instance, these days, it is common for programmers to stitch code pieces from multiple sources such as GitHub [Preston-Werner et al. 2008] or Stackoverflow [Atwood and Spolsky 2008] and then interactively explore the resulting program using program debuggers to study whether it is doing what the programmer expected. Without appropriate tools support for navigation and understanding, the programmer can easily make mistakes and waste valuable time.

At the same time, the fields of automated reasoning, program analysis and human-computer interaction have undergone massive progress in the last few decades. For instance, program analysis techniques can now be used to answer deep questions about the program behavior while new interactive touch interfaces are ubiquitous in end user devices. A key question then is:

How can we best combine and fuse advances in human-computer interaction and program reasoning in order to build the next generation program debugging and program navigation systems?

In this thesis, we focus on addressing this question. To make progress and to study the inherent trade-offs, approaches and limitations, we concentrate on a particular application domain, namely that of heap-manipulating data structures.

1.1. Heap Manipulating Data Structures

The focus of this thesis is on a program navigation system targeting the domain of heap manipulating data structures such as lists, trees, graphs and others. The reasons we focused on these domain is:

- Importance: these are some of the most fundamental programs in computer science meaning that they are widely used and invariably taught in undergraduate courses.
- Complexity: it is often not easy to build correct versions of these data structures as they involve tricky manipulating, pointer chasing, complex intermediate invariants and other issues.
- Visual form: these data structures and their invariants can often be described in visual form, and in fact that is how they are often described in textbooks [Cormen et al. 2001], making them an attractive case study for advanced user interfaces.
- Poor IDE support: existing IDEs and techniques provide poor support for navigating and interacting with heap manipulating programs [Cornelissen et al. 2009].

Collectively, the above four factors dictate that any progress towards a program navigating system that aids in natural interaction with heap manipulating data structures can lead to lasting impact on programming and education. For instance, such a system could eventually be used to aid in teaching undergraduate students[Lieberman and Fry 1995][Oechsle and Schmitt 2002], as well as helping programmers understand foreign code and potentially improve overall software quality.

1.2. Our Approach

Next, via a classic example, we illustrate the features of our interactive navigation system and how it helps in debugging heap manipulating data structures. The aim of this example is to information illustrate *what* our system can do. Technical details of how this is accomplished are discussed in later chapters.

```

1 class ListNode{
2     ListNode next;
3     int value;
4 }
5
6 ListNode reverse(ListNode head, int n){
7     ListNode node = head, prev = null, next = null;
8     for(int i = 0; i < n; i++){
9         next = node.next;
10        node.next = prev;
11        prev = node;
12        node = next;
13    }
14    head.next = node;
15    return prev;
16 }

```

Listing 1.1: A Java program which reverses a linked list

An Example: Reverse of a Linked List Consider the example shown in Listing 1.1. This program reverses a singly linked list and can have fairly complex invariants (it is used as a poster problem for *shape analysis* [Sagiv et al. 2002]). The program takes the head reference of a linked list and its length as input. Then it initializes `prev` as the head of the second linked list. An head of the original linked list is moved to the head of the second linked list in the loop. Finally, the head of the second list is returned. In the program, we define the entry of the loop (between line 8 and line 9) as **State 1** (Figure 1.1(a)) and the end of the loop (between line 12 and line 13) as **State 2** (Figure 1.1(b)). The main difference between these two states is that the red node is moved from one list to another list. Second, references `prev`, `next` and `node` point to different nodes. Therefore, our visualization needs to capture these differences, while abstraction away the parts which are the same. The resulting abstraction of **State 1** and **State 2** can be succinctly expressed as two abstracted shapes, shown in Figure 1.2.

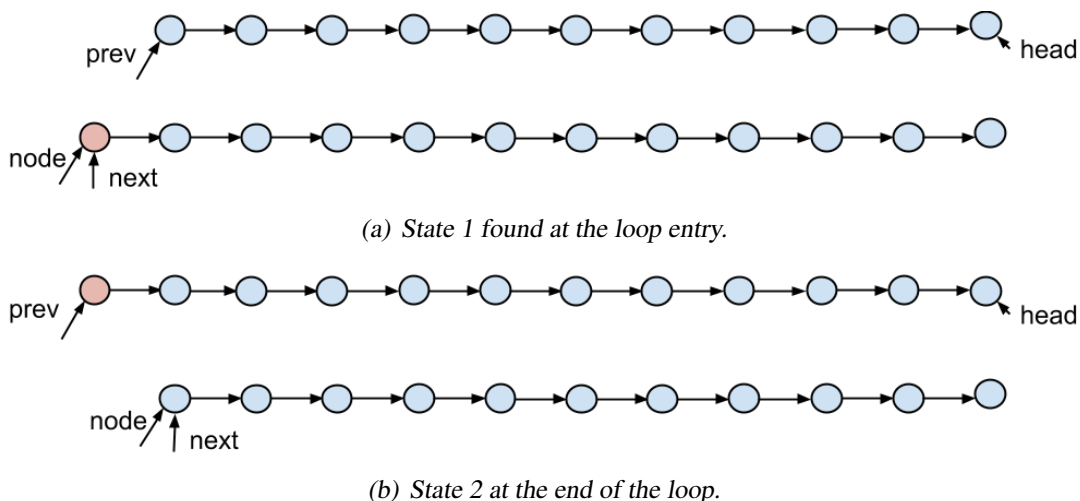
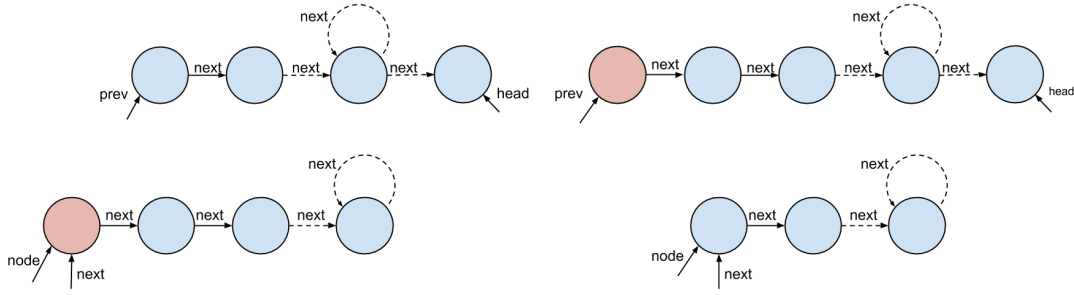


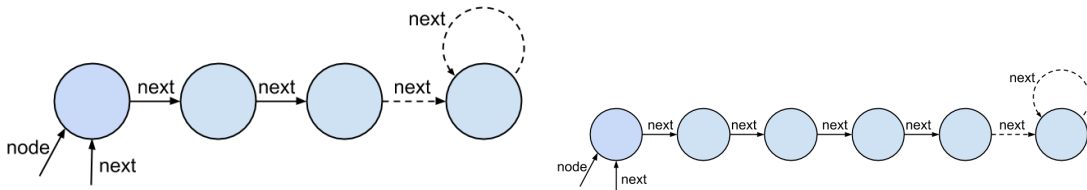
Figure 1.1.: Concrete states of a linked list reversal program

1. Introduction



(a) State 1 represents the abstract shape at the loop entry. (b) State 2 represents the abstract shape at the end of the loop.

Figure 1.2.: Two abstracted states of the linked list reversal program



(a) The abstract shape at the loop entry capturing the concrete State 1. (b) State 2 represents the abstracted status in the end of the loop.

Figure 1.3.: Two abstracted states in the middle of a linked list reversal program

In the debugging process, inspection of the data structures allows programmers to find potential problems. An inspection of the abstracted memories is basically materializing the concrete nodes from the abstracted one. These process can be expressed in Figure 1.3. Two concrete nodes are exposed from the abstracted node in the concretization process.

Benefits of Our Approach Our approach enjoys the following benefits:

- **Visualization of data structure.** With a graph visualization, the relation of data structures can be revealed. This helps developer understand their actual relations in the heap memory. We formally define a concrete heap graph, which can be used to describe objects and their references in the Java heap.
- **Flexible abstraction and concretization.** User can specify different rules for abstraction and toggle different levels of abstraction to inspect the data structure. Current works about data structure visualization do not consider abstracting the data structure or simply perform the abstraction based on some fixed rules. We adopt the abstract interpretation techniques of shape analysis, and allow user to specify abstraction predicates using first order logic with transitive closure.
- **Pen and touch interaction for graph manipulation.** We take advantage of the fabulous pen and touch interaction technique to convey navigation and interactions in graph visualization. Existing graph visualization researches only focus on how to visualize the data, rather than how to manipulate the data based on the current visualization. We modify a current graph visualization framework to support graph manipulation as well as visual-

ization.

- **Low coupling architecture.** The application is designed as a non-intrusive graphical front-end of debugger. It acts as an auxiliary tool, which allows user to control the debugger and read source code directly from the front-end.

1.3. Outline of Thesis

The following is the outline of this thesis.

- **Chapter 2** will describe the current state of the art in program analysis, visualization and interactive interfaces that are relevant to the problem of program navigation and debugging.
- **Chapter 3** describes the technical core of our approach. Basically abstraction, concretization methods and predicates which can be used in the interactions will be described. Meanwhile, it introduces different frameworks for building a modern debugger.
- **Chapter 4** concerns with the issues of implementation. It shows how to combine many existing tools and frameworks to deliver a user interface with better user experience.
- **Chapter 5** demonstrate the setup and result of experiments. Two of the user experiments are intended for evaluating the usability and efficiency of the whole system. One experiment is for evaluating the performance of interactions.
- **Chapter 6** concludes the project and summarizes shortly the most important points. Future works will be proposed for further investigation.

1. Introduction

2

State of the Art

Great efforts have been put forth into making programmers' work easier, and we will discuss them in this chapter. In this chapter, we will discuss three research directions, which memory visualization technique is used to solve problems in their domains. Meanwhile, we introduce shape analysis and pen and touch interaction, which are the two important techniques in this thesis. Finally, we mention a new programming paradigm, which inspires researchers.

1. **Program Visualization for Education**(section 2.1). we focus on the works which use graphs to represent the behavior of a program. This technique has been used computer science education(CSE) community to visualize data structure and algorithms for helping students and novice programmers learn programming.
2. **Program State and Heap Visualization**(section 2.2). Program state visualization is beneficial in the debugging and profiling process in software engineering. In debugging, IDEs visually demonstrates program states and allow programmer to inspect them. In profiling, the whole heap is usually visualized. Programmer can facilely find issues related to memory consumption, like memory leak and aliasing.
3. **Shape Analysis**(section 2.3) is a static program analysis technique, which discovers and verifies properties of programs which have dynamically allocated data structures.
4. **Pen and Touch Interaction**(section 2.4) combines advantages of two input methods and provides intuitive interactions in different domains.
5. **Live Programming**(section 2.5) is a programming paradigm which merges programming and debugging.

2.1. Program Visualization for Education

Program Visualization(PV) is a subset of Software Visualization, which uses graphical representations of data structures and motion to illustrate the higher-level run-time behavior of algorithms. It is broadly used for educational purpose. A few intuitive advantages of PV has been given by Stasko [Stasko et al. 1993]. However, most of existing evaluations are based on different systems, which leads to a variation in the evaluation results. Some works show a pessimistic result. Stasko and his colleagues later proved that the animation only help student understanding in a limited extent [Stasko et al. 1993]. According to Naps [Naps et al. 2002]’s conclusion, the visualization may not be educationally beneficial in a user’s perspective, while it creates overhead in the teacher’s side. On the other hand, there is optimistic results in recent years. The prevalence of MOOC(Massive Open Online Course) gives a good chance for students to use PV to assist their study. *Online Python Tutor* [Guo 2013] has been tested by hundreds of thousands of users and got a significant recognition.

To characterize different PV systems, many taxonomies [Sorva et al. 2013] [Myers 1990] have been give. A most famous review in the recent years is given by Naps [Naps et al. 2002]. He concluded that the success of a PV system is based on the engagement level of the participants. An engagement taxonomy is given to characterize a PV system.

- *No Viewing*. There is no visualization.
- *Viewing*. The visualization is only looked at without any other form of engagement.
- *Responding*. The visualization is able to answer some of the questions in the current context.
- *Changing*. Modification of visualization is allowed.
- *Constructing*. It refers whether a user is possible to construct an algorithm directly in the visualization system.
- *Presenting*. Learners present visualization to others for feedback and discussion.

From his point of view, a successful system should guarantee the aspects above. To be specific, a good PV system should be highly interactive, flexible, and responsive regarding to the context. We can use this taxonomy to judge the systems we mentioned in the beginning. In Stasko’s experiment, he used Polka Animation System [Stasko and Kraemer 1993], which uses predefined animations. Although the appearance can be changed to match a specific data structure, the whole animation can not be changed once it is built. On the contrary, the appearance can not be changed in *Online Python Tutor*. Arrays and arbitrary objects will be presented in a table. It merely visualizes the link between objects as well as their values. However, its usage is not limited to showing values and states in a specific algorithm. It can be used when a user is practicing his own algorithm.

Compared to our work, the target user is different in two aspects. First, the target of PV for education is students and novice programmers. How to motivate and encourage them to learn is one of the important considerations in the design process of application. The usage of HeapVision is not only for learning but also for debugging and understanding source code in software development process. Second, specific and predefined visualizations are used in PV, for exam-

ple an array is displayed as a series of adjacent boxes and a linked list is boxes in a chain. These visualizations are good for understanding some basic algorithms in a small data set. However, when dealing with a large amount of data, the visualization will become complicated.

2.2. Program State and Heap Visualization

Stuggles with tracing and program state is one of the difficulties in program comprehension [Sorva et al. 2013]. It directly influences the efficiency in debugging. Interactive debugging is performed by intercepting a running program and inspecting the current running state. By observing transitions between two states, user is able to compare two states and reason about the cause of changes. A program state includes many dimensions. Here are the three common views which are provided by a normal IDE.

- **Variable view** shows field variables of objects as well as local variables of a stack frame.
- **Stack frame view** visualizes the call hierarchies of threads. User can trace the execution route of one thread through the call hierarchy of it.
- **Thread view** shows the running threads in the program. It is usually organized in the same view with stack frames.

This thesis focuses on visualizing the data structures, which are commonly shown in the variable view. Meanwhile, the implementation of our back-end is mainly based on Java language and its runtime environment. We will talk about how memory is stored in the Java Virtual Machine(JVM). First we will go through memory structures in JVM, Heap, Global and Stack. Heap is used for storing all dynamic data structures, namely all structures created by operation new. Global, which is also called PermGen(Permanent Generation), stores all objects associated with classes as well as interned strings. Stack is section of memory used to store temporary information, which will be lost after a method returns. The relation between Heap and stack can be illustrated in Figure 2.1.

Current IDEs, like Visual Studio, Eclipse and XCode, use tree view[Bogdan et al. 1999] as the UI widget for displaying variables. We criticize tree view in three aspects. At first, although tree view is an ideal widget to demonstrate hierarchical information, a tree view fails in revealing back pointers and reference cycles. Second, inspecting a complex and big data structure consumes a lot of space, since to visualize a value which is "deep" in the tree view requires all its ancestors to be open(Figure 2.2(a)). Third, as a consequence of the second disadvantage, user need to explicitly toggle a lot of variables to visualize a value.

Eclipse partially solves the second and third problems by defining the *logical structure* of a data structure. Logical structure acts as the structure of representation in the variable view. For example, a linked list can be represent as an array, which is easier to navigate in the tree view(Figure 2.2(b)). However, a programmer still needs to specify the structure manually. At the same time, the original structure will be covered by the logical structure. To sum up, using tree view to display variables has the following shortcomings: 1. visualization can not reveal the heap structure concisely; 2. many redundant interactions are needed to inspect values.

2. State of the Art

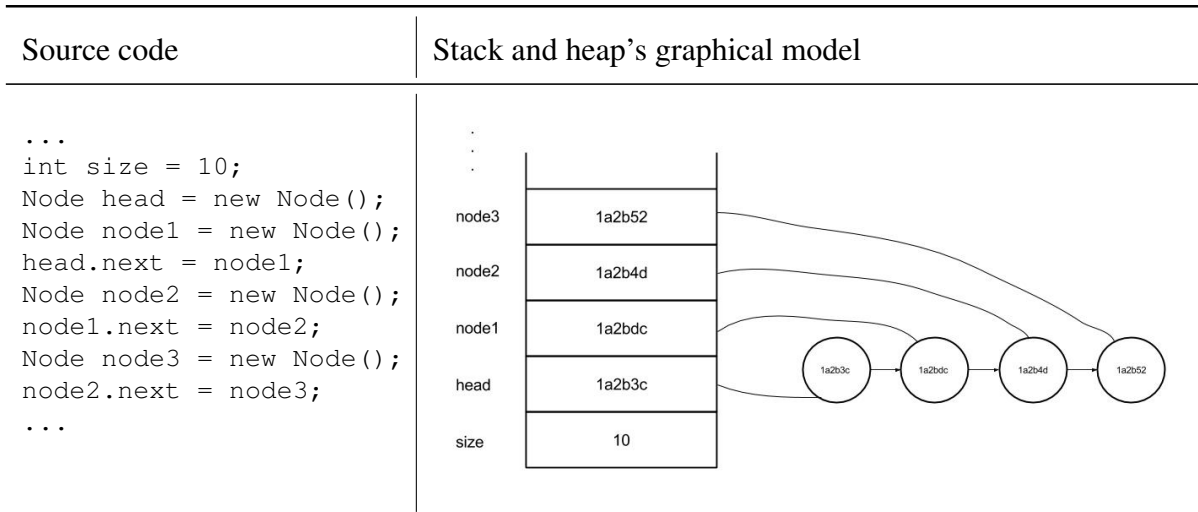
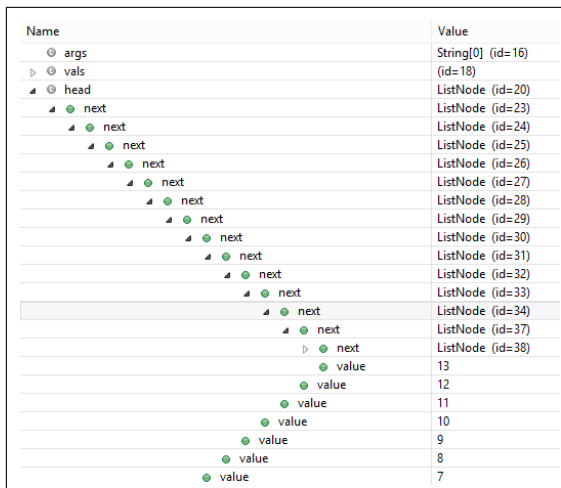
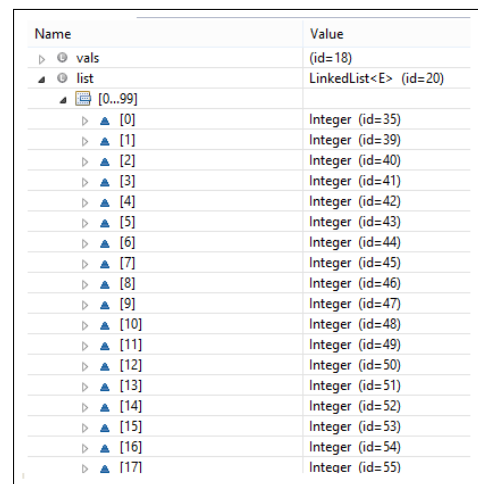


Figure 2.1: Graphical model of stack and heap in Java. The stack frame stores the references of heap memory as well as values of local primitive variables. Objects in the heap can be depicted as vertices in a graph, while their references are edges. All valid heap objects are reachable from the stack. Otherwise, the unreachable objects are called "garbage", which will be collected and erased in the garbage collection process.



(a) To inspect a linked list, programmer needs to manually toggle many variables in the tree view.



(b) With logical structure, a double linked list can be visualized as an array. However, the shape information is lost.

Figure 2.2: Variables' view in Eclipse IDE

2.2.1. Heap visualization

To overcome the shortcomings in the existing IDEs, researchers have tried to use visualization techniques to demonstrate data structures in debugging and profiling. We show a representative in each domain. In debugging, researchers visualize the whole reachable parts of heap to allow inspection. In profiling, a concise heap hierarchy is provided to expose the hidden information in the memory.

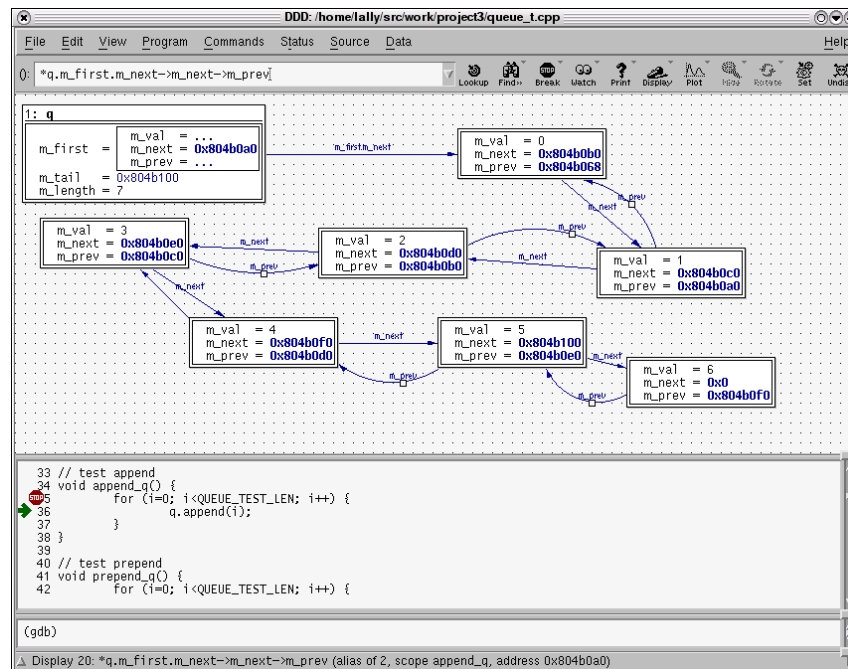


Figure 2.3.: A double linked list in DDD Debugger

Concrete heap visualization

DDD(Data Display Debugger) [Zeller and Lütkehaus 1996] is a free graphical front-end for UNIX debuggers, such as GDB and DBX. Besides common features like a combined view of source code and breakpoints, it offers *graphical data display*, where data structure can be displayed as a graph or in a chart. Its user interface is based on WIMP paradigm. User can select pointers in the code and visualize the target objects in the canvas. Instances of *struct* and *class* are treated as basic units in the canvas. Complex data structures can be explored using mouse click incrementally and interactively. Meanwhile, DDD will automatically layout all the on-screen units. However, all the expansions of pointers require manual interactions. If you need to navigate a very large linked list, it will lead to too many duplications of effort. In a continuous work of DDD, Zimmermann and Zeller [Zimmermann and Zeller 2002] present an idea of *memory graph*. Unlike DDD, all visible units should be unveiled manually. *Memory graph* displays the whole graph in one time, so that user can view the whole hierarchy immediately. With its help, alias, back pointer and reachable objects can be easily found. By comparing two graphs, user can directly perceive the changes between two states.

Heap visualization with abstraction

Compared to *memory graph* [Zimmermann and Zeller 2002], some recent projects go one step further [Aftandilian et al. 2010, Marron et al. 2013]. They provide a visualization of a program heap based on a heap dump. The difficulty in this visualization stands out in abstraction of the heap. *Memory graph* provides a visualization of the concrete heap, but it becomes meaningless when the number of concrete objects is huge. Since many objects are stored in an array or in a recursive data structure. These objects are basically equivalent, so that they can be squeezed to

2. State of the Art

a highly abstracted level. HeapViz [Aftandilian et al. 2010] and HeapDbg [Marron et al. 2013] use the same rules to abstract the heap dump. Before introducing the abstraction rules, concrete, abstract heap and their relation should be defined.

Concrete heap A concrete heap is a labeled directed graph in tuples: $(\text{root}, \text{null}, \text{Ob}, \text{Pt}, \text{Ty})$. Heap objects (Ob) are defined as nodes in the graph, while Pointers (Pt) act as edges. Roots stands for the objects which is pointed by a special *root* pointer in the heap. Since all edges in a graph should have a head and a tail node and a label from the set $\text{Label } \text{Pt} \subseteq \text{Ob} \times \text{Ob} \times \text{Label}$, null object is defined to leverage the null pointers. Since Java is strongly typed, every object is associated to a type in the set Type . This map is defined as $\text{Ty} : \text{Ob} \rightarrow \text{Type}$. A *region* of memory $C \subseteq \text{Ob} \setminus \{\text{null}, \text{root}\}$ is part of the concrete heap objects, except the root or null nodes.

Concrete heap properties After the definition of the concrete heap, some properties are introduced to describe a concrete heap. As is defined above, C is a *region* of memory in the concrete heap.

- *Type*. The Type of C is the set of types of the objects in C : $\{\text{Ty}(o) \mid o \in C\}$.
- *Cardinality*. The amount of objects in C is $|C|$.
- *Nullity*. The object in the head node of an edge defines the nullity of a pointer.
- *Injectivity* The pointers labeled p from two different objects o_1 and o_2 to different objects t_1 and t_2 are *injective*.
- *Shape*. Graph theoretic notations of trees, directed acyclic graphs(DAG) and general graphs are used to describe the shape of C .

Abstract heap An abstract heap is defined as the following tuple:

$$(\text{root}, \text{null}, \text{Ob}^\#, \text{Pt}^\#, \text{Ty}^\#, \text{Cd}^\#, \text{Ij}^\#, \text{Sh}^\#)$$

The $\#$ attribute stands for the corresponding abstracted parameter in the abstracted heap. A abstracted heap can be regarded as an abstraction of an concrete heap, if there exists an abstraction function μ holds the following predicates:

- *Embedded*. $\text{Embed}(\mu, \text{Ob}, \text{Pt}, \text{Ob}^\#, \text{Pt}^\#) \Leftrightarrow \mu(\text{root}) = \text{root} \wedge \mu(\text{null}) = \text{null} \wedge \forall o_1 \xrightarrow{l} o_2 \in \text{Pt}. \exists l. \mu(o_1) \xrightarrow{l^\#} \mu(o_2) \in \text{Pt}^\# \wedge l \in \gamma_L(I^\#)$
- *Typing*. $\text{Typing}(\mu, \text{Ob}, \text{Ty}, \text{Ob}^\#, \text{Ty}^\#) \Leftrightarrow \forall o \in \text{Ob}. \text{Ty}(o) \in \text{Ty}^\#(\mu(o))$
- *Counting*. $\text{Counting}(\mu, \text{Ob}, \text{Ob}^\#, \text{Cd}^\#) \Leftrightarrow \forall n \in \text{Ob}^\#. |\mu^{-1}(n)| \in \text{Cd}^\#(n)$
- *Injective*. $\text{Injective}(\mu, \text{Pt}, \text{Pt}^\#, \text{Ij}^\#) \Leftrightarrow \forall (n_1, n_2, l) \in \text{Pt}^\#. \text{Ij}^\#(n_1, n_2, l) \Rightarrow \forall p \in \gamma_L(l). \text{inj}(\mu^{-1}(n_1), \mu^{-1}(n_2), p)$
- *Shape*. $\text{Shape}(\mu, \text{Pt}, \text{Pt}^\#, \text{Sh}^\#) \Leftrightarrow \forall (n, L, \text{tree}) \in \text{Sh}^\#. \text{tree}(\mu^{-1}(n), \gamma_L(L))$

In words, all concrete pointers should be embedded in their corresponding abstracted pointers. The type of every node is included in the type set of its abstracted node. The amount of concrete nodes which map to the same abstracted node is the cardinality of this abstracted node. Every set of pointers which is injective in concrete heap graph is still injective in the abstract graph. Finally, the Shape relation guarantees the concrete shape predicates.

Abstraction rules Based on the concrete and abstract heap graph relation, different rules can be applied to achieve a abstract graph. Following rules shows a good example to abstract concrete heaps which contain array and different recursive data structure. They are shared by HeapViz and HeapDbg.

1. *Same Recursive Data Structure Objects.* $\text{recursive_data}(o) \Leftrightarrow \exists o_1, \exists p := (o_1, o, l), p \in Pt \wedge Ty(o_1) = Ty(o)$ In words, if two objects have a reference, and they share the same type, they will be merged as one node;
2. *Equivalent on Abstract Predecessors.* $\text{equi_predecessor}(o) \Leftrightarrow$ If two objects which share the same type and have the same set of predecessor objects, they will be as well merged as one node.

Conclusion of heap visualization

Since no abstraction is offered in the DDD debugger, it merely acts as an small improvement of the current variable view. Compared to HeapVision, it is the closest previous work. Our work brings not only the graph-based visualization as well as memory abstraction and novel user interactions. Formal definitions of concrete heap and abstract heap is given to enable the future study in HeapDbg. However, it minimizes the levels of detail. A user can only get a shallow understanding of a summarized node. Actually, compared to *memory graph*, *HeapViz* and *HeapDbg* have different objectives. They act as a profiling tool to measure memory consumption in sparse states. Apart from the abstraction of the whole heap structure, HeapViz provides an interactive graph view based on the Prefuse[Heer et al. 2005] toolkit. HeapDbg is built as a plugin for Visual Studio and the heap graph can be directly displayed in the IDE. To sum up, two current works about abstracted heap visualization have a different purpose than than HeapVision. Meanwhile, the abstraction technique will reduce detailed information which is useful in debugging.

2.3. Shape Analysis

As shown above, Marron [Marron et al. 2013] gives simple abstraction rules to abstract the memory. However, these abstraction rules are fixed. And the user is not possible to change the visualization after the heap was abstracted, which makes the application not suitable for debugging. We introduce shape analysis, which is a static program analysis framework for programs which manipulate dynamic allocated memory. It is related to our project since it provides a flexible way to abstract memory. Meanwhile, the formal definition of the abstraction offers space for extensions in the future.

2. State of the Art

We will go through some background knowledge in the beginning. Static program analysis is used for analysing computer program without running the program, which is performed by an automatic tool on the source code. Programmer often uses static analysis to prove invariants in the program, such as division by zero, accessing uninitialized variables, array out-of-bounds errors and so on. Shape analysis is a static analysis technique based on abstract interpretation [Cousot and Cousot 1977], that finds and verifies the manipulation of dynamic allocated memory in a program. It is one of the difficult problems in static analysis, due to two main issues:

- **Destructive updating through pointers.** Aliasing relationships are pervasive.
- **Dynamic storage allocation.** Runtime data structures has no upper bound.

Sagiv has provided a solid theoretical background for parametric shape analysis[Sagiv et al. 2002]. Three-Valued Logic Analyzer(TVLA)[Lev-Ami and Sagiv 2000] is an implementation for the parametric framework. It stands out among static analysis tools for its flexibility, so that it can be regarded as the Yacc¹ for shape analysis. The framework can be used in different ways by filling different predicates, which determine the tracked properties. It can be used to analyse memory errors, such as dereferencing NULL pointers, dereferencing dangling pointers and memory leaks. The main characteristic of TVLA is the use of Kleene's Three-Valued Logic(TVL) [Kleene 1952]. Besides True = 1 and False = 0 values in Two-Valued Logic, there is an Unknown = $\frac{1}{2}$ value in TVL, which makes the truth table in TVL as below:

$A \wedge B$	A	$A \vee B$	A	A	$\neg A$
	1		1	1	0
	$\frac{1}{2}$		$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
B	1	B	1	0	1
	$\frac{1}{2}$		$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
	0		0	0	0
	0		$\frac{1}{2}$	0	1
	0		0	0	0
(a) AND		(b) OR			(c) NOT

Table 2.1.: Truth table for AND, OR, NOT logical operations in Three-valued logic

The import of Unknown is useful, because it can be used to represent the unbounded data structures which share the same properties. Since there is no bound on the size of runtime data structures allocation, in traditional pointer analysis approach, it is not efficient to cover all possible states. With the help of TVL, memory states with the same properties can be abstracted for further analysis. As mentioned above, predicates, more specifically unary predicates, are used for shaping a state. First order logic with transitive closure(FO(TC)) is used for defining a predicate. To verify a property, user need to explicitly describe it using FO(TC) and provide it as well as a TVP file translation of the source code[Lev-Ami and Sagiv 2000]. TVLA will automatically generate a report with graphical representations of memory.

¹A program which generates parser for computer language in the Unix operating system

Example: creation of a linked list We take an example in Listing 2.1, which simply create a single linked list. We apply a set of program independent predicates (Table 2.2) and verify properties. The goal of the analysis is to verify that neither memory leak nor pointer alias exists in the program. Each state of the execution process has been examined. And all possible shapes can be inferred by the analyzer. Figure 2.4 and Table 2.4 show result of the analysis, that is, no leaks or aliases is found in the program. The first step of a verification is the *blurring* process.

```

1 //list.h
2 typedef struct node{
3     struct node *n;
4     int data;
5 } *List;
6 //create.c
7 void create(List x, int size)
8 {
9     List f = NULL;
10    for (i=0; i<size; i++) {
11        f = malloc(sizeof(struct node)
12    );
13        f->n = NULL;
14        f->n = x;
15        x = f;
16    }
17 }

```

Listing 2.1: A program which create a linked list based on an existing linked list

Predicate	Meaning
$f(v)$	Value is pointed by pointer f
$x(v)$	Value is pointed by pointer x
$n(v_1, v_2)$	There is a pointer with label n between value v_1 and v_2
$t_n(v_1, v_2)$	v_2 is reachable from v_1 via pointer with label n
$r_{n,f}(v)$	v is reachable from stack pointer f via pointer with label n
$r_{n,x}(v)$	v is reachable from stack pointer x via pointer with label n

Table 2.2.: Predicates for the verification of a linked-list's creation

All the concrete two-valued structures will be evaluated by the abstraction predicate set. The boolean result of each predicate will be combined as a bit vector. The structures which have the same result will be grouped together. During the execution of each concrete operational semantics in the programming language, one of each abstraction predicate will be combined with a corresponding *predicate-update formula* to transform the source structure to a target struction, which is denoted as, $c(v_1, \dots, v_k) = \tau_{c,e}(v_1, \dots, v_k)$. Further more, to guarantee the precision, *focus*, *coerce* operations as well as different instrumentation predicates are introduced. Here is the result of verifying the created linked list is not shared by any other pointers, and there is no memory leak during the execution. A control flow graph is shown to clarify different states. Corresponding graphs are presented in the right for all possible states. It can be inferred from $L1$ that x is equal to null when it is passed to the function. About the annotation in the graph, an edge between two nodes represents a binary predicate, such as $n(v_1, v_2)$. An edge without source node indicates a unary predicate. A dash edge means the certain predicate is evaluated to $\frac{1}{2}$, while a concrete edge means its value is 1. Finally, a circle elaborates a concrete store, while a node with double circles means it is an abstracted node, which shows that there may be an object in the actual case.

2. State of the Art

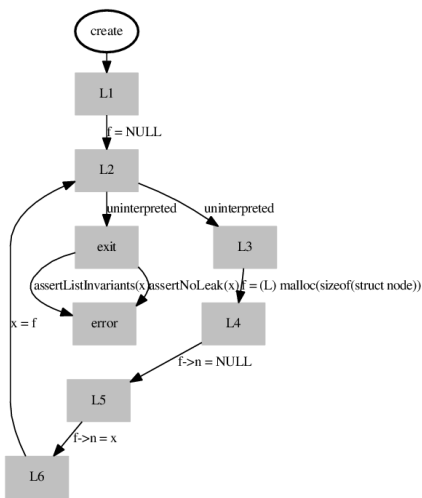


Figure 2.4.: Control flow graph of the linked list creation program. Each gray box represents a state of the program. The edge between two states means that a transition exists between these two states. Sequences of possible consecutive states form the whole control flow of the program.

State	Structure 1	Structure 2	Structure 3
L1	Structure with empty universe(Empty)		
L2	Empty		
L3	Empty		
L4			
L5			
L6			
Exit	Empty		

Table 2.3.: The input structure(variable x) of the analysis is $NULL$. TVLA statically analyses each state and shows possible heap shapes, which are shown in each row. Heap shapes which are in the same row means they share the same structure. Images are generated by TVLA and GraphViz[Ellson et al. 2004]. TVLA analyses the source code and provides graphical reports in dot format. GraphViz parses the file into images.

Although the use case of TVLA is different from our project, it is enlightening in two aspects: 1. Use of TVL in static analysis; 2. Predicates in FO(TC) to represent program properties. The differences between static analysis and debugging lie on:

- **Representation.** Strictly speaking, it is a *diagram* instead of a *graph* that is used to represent a state in TVLA. A graph is $G : \langle V \times E \rangle$ with $E : \langle V, V \rangle$, while in the diagram of TVLA, an edge without a source node denotes a unary predicate and an edge with both source and target implies a binary predicate. As arity of a predicate increases, different visual constructs should be introduced. From a user's perspective, the more complex a graph is, the less intuitive it is going to be. The representation of graph can not be directly borrowed from TVLA.
- **Predicate.** Abstraction predicates $\mathcal{A} - abstract$ are first used in the *blurring* process. Since predicates are used to examine the properties of the heap, a node is defined by a unique bitvector. If there are n types of structures and m predicates, they have a relation of $\log_2 m \leq n$. However, if $m \in [\log_2 m, n)$, modifying one predicate in the predicate set may change the evaluated result of the certain predicate, which leads to a global change. If n predicates can identify n nodes, it is actually using a single predicate to tag one kind of structure. $\mathcal{A} - abstract$ will expand according to the size of concrete nodes, which leads to the loss of efficiency.

2.4. Pen and Touch Interaction

Pen and touch interaction technique has been widely studied in the recent years. Hinckley [Hinckley et al. 2010] has concluded various benefits of this bimanual interaction technique. The main advantage is that it offers flexibility and accuracy for interaction simultaneously. Secondly, highly intuitive interactions can be designed using a combination of operations, which are based on pen and touch interaction. Pen and touch interaction has been applied in different domains.

- **Drawing.** Both Hinckley [Hinckley et al. 2010] and Brandl [Brandl et al. 2008] have developed prototypes of drawing application. Pen can provide fine-grained interaction, while touch can provide natural and flexible interaction. Meanwhile, user needs to switch different tools and models frequently in a drawing process.
- **Editing.** Sketching is one of the aspects which differs pen from other single-point input devices. Hinckley [Hinckley et al. 2012] has taken advantages of this property to facilitate annotation and notes collection during the on-screen reading process.
- **Gaming.** Researchers have applied the interaction on a real-time strategy (RTS) game. Player can perform accurate and responsive interactions on a touch screen with pen, which acts as a good replacement of mouse in RTS game.
- **Data Exploration.** Walny [Walny et al. 2012] has conducted user studies to explore applying pen and touch interaction for information visualization on an interactive white-board. They categorized different usages of the interaction techniques in data exploration tasks, and provided good suggestions for designing pen and touch enabled information

2. State of the Art

visualization interfaces.

Researchers in *infoVis* community has considered applying this technique in graph visualization [North et al. 2009a]. However, they considered that pen-based interface is close to mouse-based interface, which has been widely studied. Meanwhile, what pen-based interface offers was no more than lasso selection and marking-menu-based command activation. No more detailed investigation has been tried. In this thesis, we think that pen and touch interaction is useful in graph visualization and manipulation, due to the following to aspects.

Success of touch input At first, touchable interface has become a crucial input device in the post-WIMP era[Shneiderman 1993]. A lot of successful applications [Dietz and Leigh 2001] [Han 2005] have been developed based on the touchable interface, which, however, has not been widely supported in IDEs. Traditionally, IDE is regarded as a tool for source code editing [Murphy et al. 2006]. Code editing is a form of text editing process, in which mouse and keyboard are the dominating and most efficient input devices [CARD et al. 1978] [Forlines et al. 2007]. However, as we mentioned in Chapter 1, the work of code editing will be reduced with the help of open-sourced source code repositories. Programmers do not need to write codes themselves. Instead, they can clue and test codes to construct a program. It leads to the fact that touch input might be more beneficial in the future. Especially in the user interface of HeapVision, the visualization is for visualizing a graph structure. Touch interaction is proved to be useful in the work of North's[North et al. 2009a].

Need for input accuracy Our application can be based on two sizes of input devices, touchable monitor and tablet computer. The input area of both of devices is smaller than the sizes of the Microsoft Surface, which is used in the experiment of Walny's[Walny et al. 2012]. The accuracy of operations varies in different sizes of display [Brewster 2002]. The bar hand gestures may not perform as well as in a smaller-sized display. The pen input can highly enhance the pointing input accuracy. Second, the pen is also used for text input in the original design. However, due to the time constraint, we could not integrate this part in the thesis. Finally, the combination of pen and touch commands can yield new intuitive commands, which can be used for building up the vocabulary of heap exploration.

To sum up, Pen and touch inputs have different interaction properties. Combining both inputs yields new interaction techniques which can be applied in different domains and use cases. The potential of pen and touch interaction in graph visualization and manipulation is still waiting to be explored.

2.5. Live Programming

Live programming aims to provide an instant feedback during code editing, which increases the programmer's awareness and understanding of the code behavior. It is a way of mixing programming and debugging experience. In terms of technical aspect and implementation, it is highly depending on the programming language and their environment. The world has seen many live programming tools, which deal with declarative programming languages. Any

HTML file editor which can render the page while you are editing it is a sound example. DataPlay [Abouzied et al. 2012] is an example for querying a database using SQL in a live way. However, live programming using Imperative language remains a research problem. In this section, *programming* refers to writing software programs using an imperative language.

Liveness in programming was first explored by a few visual languages, such as VIVA[Tanimoto 1990], Form/3[Burnett et al. 1998] and a more productive framework Morphic[Maloney and Smith 1995]. They go beyond the normal feedbacks provided by IDEs, such as code completion, type checking and syntax highlight. They address the feedback of different program behaviors. Victor's demo about live programming in 2012 [Victor 2012] has inspired a few researchers in a user-oriented perspective. In his demo, he addresses three aspects which can influence the future of programming.

- **Direct data visualization and manipulation.** His live programming editor can execute drawing code lively during the editing.
- **Cause and effect relationship visualization.** Programmer can refer to the line of code which leads to the visual effect.
- **Visible flow.** A time-travel functionality allows to retrieve previous states and visualize the change history between them.

In , Victor shows a visualization which allows a programmer to play back the execution trace. In the right panel, the points in vertical direction can be regarded as a control flow, while the points in horizontal direction can be regarded as anchor points to show program states in different iterations.

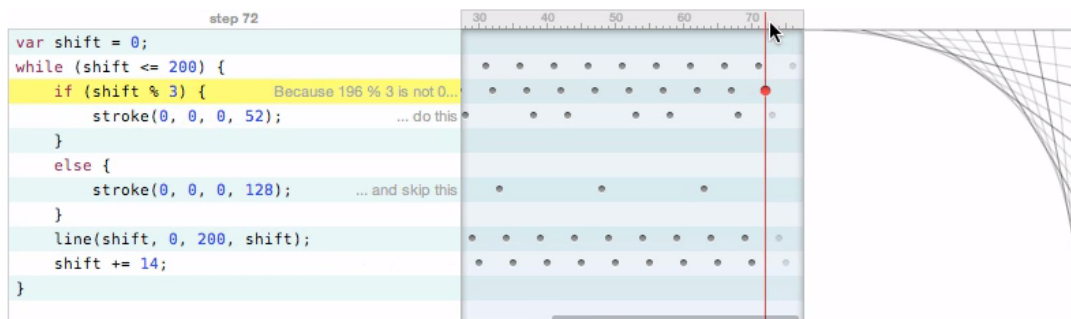


Figure 2.5.: Prototype of live programming by Victor, which shows traces of drawing lines

McDirmid [McDirmid 2013] provides an implementation called TaiChi based on Victor's exploration. TaiChi He focuses on three features, *probing*, *Tracing* and *Time-travelling*. *Probing* offers a live programming by debugging experience, which allows user to execute methods directly in the editing mode. By merging the boundary between implementation and testing, programmers eliminate potential problems before it is carried to the next procedure. *Tracing* helps to build a global awareness of the execution history by abstracting current and previous results into a print-like trace statements. *Probing* can be regarded as a inspection operation, while *tracing* corresponds to logging. Combining both of them, an interactive debugging experience can be achieved directly in code editing. Finally, the design of *Time-traveling* shows a how live programming can embed with visualizable traces. However, since trace

2. *State of the Art*

recording and code modification are happening synchronously. Whether a change in the past should influence the current traces is an open question. Many existing live programming prototypes [Sorensen and Gardner 2010][McDirmid 2007] choose to ensure the consistency of code execution, which should also be reflecting in the traces. Further elaboration about code-trace relation is given by TouchDevelop[Burckhardt et al. 2013]. Traces is defined by program configurations and its source code, while source code is subject to its modifications. Besides, traces compression and abstraction deserve to be mentioned and further considered. For example, a one minute physics simulation includes thousands of frames.

To sum up this section, although computing performance of computer is keep increasing in recent decades, user experience for programming remains in a stage when the Smalltalk-era IDEs is still pervasive. An evolution in programming environments has not been triggered for a long time. Victor has given suggestions in a user-oriented perspective. However, transferring these ideas to productive tools still requires a lot of work. TaiChi is one of the prototypes which reflects Victor's ideas. Compare live programming with interactive debugging, live programming intends to inject features, which are only available in debugging, like inspection of values and execution traces visualization, into code editing process. It is a nice concept, but still requires an evolution in the supporting programming language and development environment. Interactive debugging brings liveness into debugging process, by visualizing data structures and their changes in the execution process. It is a more realistic than live programming, since it can directly take advantage of the existing facilities for debugging.

3

Enabling Deep Program Interaction

In this chapter, we describe the core technical aspects that make our system possible. First, we define the appearance details of the heap and the informal meaning of the elements. Second, we provide a list of interactions that we support. Third, we provide a formal definition of the interactions. Finally, we cover some graph layout algorithms and present our incremental layout algorithm.

3.1. Visual Notation

Before defining the visual notations, we need to specify what is being visualized. Our system can visualize objects and their relations in the heap. The first step to connect an *object* in the heap to a *node* in the graph (that is visualized) is to ensure that we have some way to identify objects. In our system, we will focus on the Java programming language. Although Java is an object-oriented language, not all types inherit the `java.lang.Object` type. Such types are called primitives, while the remaining types are referred to as a reference type. A value, whose type is a reference type, is formally called object. A definition of a type, variable and its value is shown Table A.2 and A.1. An object can be created, modified and assigned by the program, while it can be freed only by the garbage collectors. In addition, an object is associated with an ID, which is unique in the program life cycle. To visualize all program behaviors, the nodes in the graph represent heap objects. In addition, we define elements in the Java language as follows:

3. Enabling Deep Program Interaction

Item	Tuple	Meaning
local variable	$(target, name, type)$	a tuple of associated value, variable name and variable type
field variable	$(source, target, name, type)$	a tuple of owner's value, associated value, variable name and variable type
object	$(type, value, ID)$	object type, object value and its assigned ID

Table 3.1.: Visualizing elements in Java

To visualize the abstract heap, we adopt the visual notation used in TVLA [Sagiv et al. 2002] and extend it.

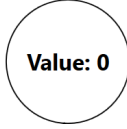
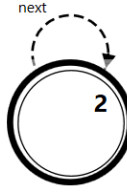

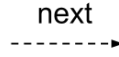
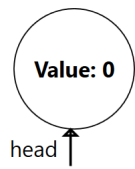
Name	Symbol	Meaning	Example	Details
concrete node	solid circle	object		Text in the middle of node represents the value or the ID of an object.
summary node	double circle	summary of objects		Text in the node represents the amount of abstracted objects.
concrete edge	solid edge	field variable		Text represents the name of field variable.
abstract edge	dash edge	summary of field variables		Text represents the name of field variables, if all abstracted variables share the same name.
arrow	solid edge without source	local variable		Text represents the name of the local variable.

Table 3.2.: Visual notations in HeapVision

3.2. Visual Interaction

We now define a series of interactions on a heap graph. These interactions are used for both graph visualization and manipulation. The difference between visualization and manipulation is that, in visualization, the graph remains isomorphic in all states, while with manipulation, the user changes the graph by inserting and deleting vertices and edges. Previous works focus on visualization of data [Heer et al. 2005] and interactions [Vlaming et al. 2010] to facilitate visualization. Direct manipulations on the graph have been seldom explored. The closest work is from North [North et al. 2009b] where the work ran user studies and tested different gestures on a two-handed, multi-touch surface. However, their test was based on Microsoft Surface (now named Microsoft PixelSense¹) which is much larger than our input devices.

The gestures supported by our system are shown in Table 3.3. Interactions are grouped into four different kinds:

- *View*. Either size, position of the whole graph or position of an individual node is changed.
- *Selection*. Objects are selected for the next operation.
- *Abstraction*. A set of nodes are merged to one summary node.
- *Concretization*. A summary node is separated into two or more nodes.

The pinch and double tap gestures are realized via two graph manipulation algorithms, which manipulates a mapping of concrete nodes and edges. Abstraction reduces the number of nodes by merging nodes into a summary node. Concretization adds more detailed information to the graph by inserting more nodes. Abstraction and concretization correspond to two graph operations in graph theory, *vertex contraction* and *vertex cleaving*. Vertex contraction may occur on any subset of vertices. If this subset of vertices is a clique, the operation is called *edge contraction*. Existing edges between contracting vertices are removed. Edges between contracting vertices and non-contracting vertices are remained. Vertex cleaving, or vertex splitting is the reverse operation of vertex contraction, which means one vertex is being split into two, where these two new vertices are adjacent to the vertices that the original vertex was adjacent to. A series of iterative vertex cleaving operations means one vertex is being split into two or more vertices. The algorithms for these two operations are shown in Algorithm 3.1 and Algorithm 3.3.

¹Microsoft PixelSense, <http://www.microsoft.com/en-us/pixelsense/default.aspx>

3. Enabling Deep Program Interaction

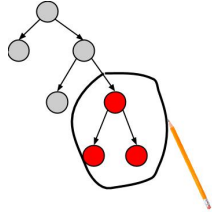
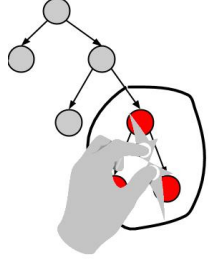
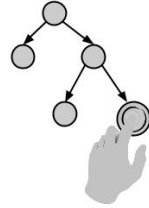
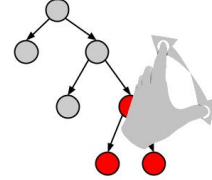
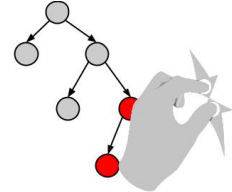
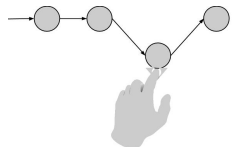
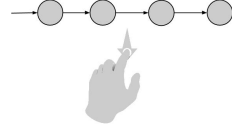
Name	Description	Function	Preview
Group	Draw the boundary of a group of nodes to select them	$\text{group}(V_{select})$	
Abstract	Collapse a set of nodes into a summary node	$\text{abstractGraph}(V_s, G, \alpha_N, \alpha_E)$ $(G', \alpha'_N, \alpha'_E)$	
Concretization	Double tap on a summary node to materialize nodes which are pointed by adjacent nodes	$\text{concretizeGraph}(v_a, G, \alpha_N, \alpha_E)$ $(G', \alpha'_N, \alpha'_E)$	
Zoom in	Perform stretch gesture on an empty space to enlarge the size of the whole graph		
Zoom out	Perform pinch gesture on an empty space to reduce the size of the whole graph		
Drag	Drag a node to an empty space with one finger		
Reposition	Pan on an empty space with one finger to move the whole graph		

Table 3.3.: List of gestures for graph visualization and manipulation

Functions	Gesture	Meaning	Algorithm
$\text{group}(V_{select})$	pen draw	select a set of nodes	
$s(e)$		get the source node of edge e	
$t(e)$		get the target node of edge e	
$\text{reMapVertex}(v_o, v_a, \alpha_N) : \alpha'_N$		map concrete node v_o to abstract node v_a	
$\text{reMapEdge}(e_o, e_a, \alpha_E) : \alpha'_E$		map concrete edge e_o to abstract edge e_a	
$\text{newAEdge}(v_s, v_t, E, \alpha_E) : (e, E', \alpha'_E)$		create a new edge with source v_s , target v_t	
$\text{concretizeNodes}(V_s, G, \alpha_N, \alpha_E) : (G', \alpha'_N, \alpha'_E)$		make a set of nodes concrete V_s	Alg. 3.2
$\text{concretizeGraph}(v_a, G, \alpha_N, \alpha_E) : (G', \alpha'_N, \alpha'_E)$	double tap	concretize a set of nodes which are pointed by in-edges of a summary node v_a , and return the new node and edge mappings	Alg. 3.3
$\text{alphaGraph}(V_s, G, \alpha_N, \alpha_E) : (G', \alpha'_N, \alpha'_E)$	pinch	map the selected nodes to a node v_a , map the related edges to new edges and return the new node and edge mappings	Alg. 3.1

Table 3.4.: Shorthand notations of algorithms

The Definition of Abstraction and Concretization

To capture abstraction, we maintain four mappings, defined in Table 3.5. Here, the set CN denotes the set of concrete nodes and AN denotes the set of abstract nodes. Then, α_V is a mapping between a concrete node in CN and an abstract node in AN , while α_E is a mapping between a concrete edge in CE and an abstract edge in AE .

3. Enabling Deep Program Interaction

Definition	Meaning
$\alpha_V(v) : CN \rightarrow AN$	a surjective function which maps a concrete node to an abstract node
$\alpha_E(e) : CE \rightarrow AE$	a surjective function which maps a concrete edge to an abstract edge
$\gamma_V(v) : AN \rightarrow CN$	a surjective function which maps an abstract node to a set of concrete nodes
$\gamma_E(e) : AE \rightarrow CE$	a surjective function which maps an abstract edge to a set of concrete edges

Table 3.5.: Abstraction and Concretization Maps for Nodes and Edges

Based on the abstraction and concretization maps, we can define the corresponding algorithms for pinch and double tap gestures in Alg. 3.1 and Alg. 3.3. For an example of the two operations, consider Figure 3.1. Here, two nodes u_1, u_2 are selected in the grey area. Edges b and f share the same source node. They are merged into one edge b' while edges a, d, e, g are merged to one self edge a' .

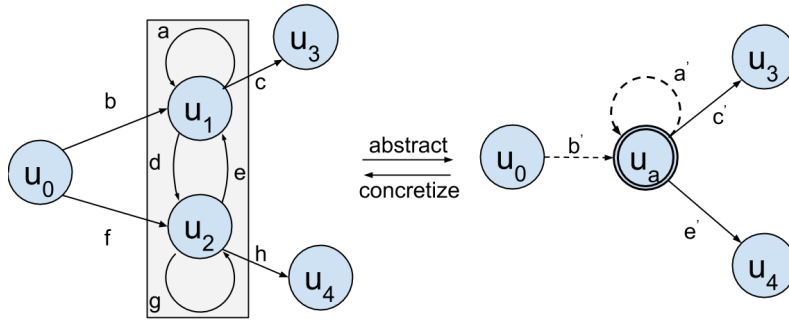


Figure 3.1.: An example of the abstraction and concretization processes

```

1 abstractGraph( $V_s, G, \alpha_N, \alpha_E$ ) : ( $G, \alpha'_N, \alpha'_E$ )
2    $inEdges \leftarrow \{(v_s, v_t) | v_s \notin V_s \wedge v_t \in V_s\}$ 
3    $outEdges \leftarrow \{(v_s, v_t) | v_t \notin V_s \wedge v_s \in V_s\}$ 
4    $insideEdges \leftarrow \{(v_s, v_t) | v_t \in V_s \wedge v_s \in V_s\}$ 
5
6    $v_a \leftarrow \exists v \in V_s$ 
7   foreach  $v \in V_s \setminus v_a$ 
8      $\alpha_N \leftarrow \text{reMapVertex}(v, v_a, \alpha_N)$ 
9   if  $selfEdges \neq \emptyset$ 
10    ( $selfEdge, E, \alpha_E$ )  $\leftarrow \text{newAEdge}(v_a, v_a, E, \alpha_E)$ 
11    foreach  $e \in insideEdges$ 
12       $\alpha_E \leftarrow \text{reMapEdge}(e, selfEdge, \alpha_E)$ 
13   foreach  $e \in inEdges$ 

```

```

14     if  $\exists(v_s, v_t) \in E : \text{source}(e) = v_s \wedge \text{target}(e) = v_a$ 
15          $edge \leftarrow \exists(v_s, v_t) \in E' : \text{source}(e) = v_s \wedge \text{target}(e) = v_a$ 
16     else
17          $(edge, E, \alpha_E) \leftarrow \text{newAEdge}(\text{source}(e), v_a, E, \alpha_E)$ 
18      $\alpha_E \leftarrow \text{reMapEdge}(e, edge, \alpha_E)$ 
19 foreach  $e \in \text{outEdges}$ 
20     if  $\exists(v_s, v_t) \in E : \text{target}(e) = v_t \wedge \text{source}(e) = v_a$ 
21          $edge \leftarrow \exists(v_s, v_t) \in E' : \text{source}(e) = v_s \wedge \text{target}(e) = v_a$ 
22     else
23          $(edge, E, \alpha_E) \leftarrow \text{newAEdge}(v_a, \text{target}(v_a), E, \alpha_E)$ 
24      $\alpha_E \leftarrow \text{reMapEdge}(e, edge, \alpha_E)$ 
25 return  $\leftarrow (G, \alpha_N, \alpha_E)$ 

```

Listing 3.1: Collapse a set of nodes to one node

```

1 concretizeNodes $(V_s, G, \alpha_N, \alpha_E) : (G', \alpha'_N, \alpha'_E)$ 
2     foreach  $v \in V_s$ 
3         if  $\alpha_V(v) = v$ 
4              $oV \leftarrow \gamma_V(v)$ 
5             if  $\text{count}(oV) = 1$ 
6                  $\text{return} \leftarrow (G, \alpha_N, \alpha_E)$ 
7              $\alpha_V \leftarrow \text{reMapVertex}(v, v, \alpha_V)$ 
8              $\text{inEdges} \leftarrow \{(v_s, v_t) | v_t = v\}$ 
9              $v_{\text{new}} \leftarrow \exists v \in oV \setminus \{v\}$ 
10            foreach  $e \in \text{inEdges}$ 
11                 $\alpha_E \leftarrow \text{reMapEdge}(e, (\text{source}(e), v_{\text{new}}), \alpha_E)$ 
12
13             $\text{outEdges} \leftarrow \{(v_s, v_t) | v_s = v\}$ 
14            foreach  $e \in \text{outEdges}$ 
15                 $\alpha_E \leftarrow \text{reMapEdge}(e, (v_{\text{new}}, \text{target}(e)), \alpha_E)$ 
16
17            if  $\alpha_V(v_c) = v_c$ 
18                 $\text{return} \leftarrow (G, \alpha_N, \alpha_E)$ 
19             $\alpha_N \leftarrow \text{reMapVertex}(v_c, v_c, \alpha_N)$ 
20             $\text{relatedEdges} \leftarrow \{(v_s, v_t) | \forall (v_s, v_t) \in E, v_s = v_c \vee v_t = v_c\}$ 
21            foreach  $e \in \text{relatedEdges}$ 
22                if  $\text{source}(e) = v_c$ 
23                     $\text{newSource} = v_c$ 
24                     $\text{newTarget} = \alpha_V(\text{target}(edge))$ 
25                else
26                     $\text{newSource} = \alpha_V(\text{source}(edge))$ 
27                     $\text{newTarget} = v_c$ 
28                if  $\exists(v_s, v_t) \in E' : v_s = \text{newSource} \wedge v_t = \text{newTarget}$ 
29                     $edge \leftarrow \exists(v_s, v_t) \in E : v_s = \text{newSource} \wedge v_t = \text{newTarget}$ 
30                else
31                     $(edge, E, \alpha_E) \leftarrow \text{newAEdge}(\text{newSource}, \text{newTarget}, E, \alpha_N)$ 
32                 $\alpha_E \leftarrow \text{reMapEdge}(e, edge, \alpha_E)$ 
33            return  $\leftarrow (G, \alpha_N, \alpha_E)$ 

```

Listing 3.2: Concretize a set of vertices in the graph

3. Enabling Deep Program Interaction

```

1 concretizeGraph( $v_a, G, \alpha_N, \alpha_E$ ) : ( $G', \alpha'_N, \alpha'_E$ )
2    $cInEdges \leftarrow \{(v_s, v_t) \mid \forall (v_s, v_t) \in E, \alpha_V(v_t) = v_a \wedge \alpha_V(v_s) \neq v_a\}$ 
3   ( $G, \alpha_N, \alpha_E$ )  $\leftarrow$  concretizeNodes ( $\{\text{target}(e) \mid \forall e \in cInEdges\}, G, \alpha_N, \alpha_E$ )

```

Listing 3.3: Concretize the vertices which are pointed by in edges

3.3. Computing The Initial Abstract Heap

A key question that we need to address is how to compute the *initial* abstract shape. A natural answer can be found in the way TVLA computes abstractions. Informally, it keeps as concrete all nodes directly pointed to by local variables while all other nodes are abstracted. More formally, given a set of predicates Ω which represent the local pointer variables, we compute the abstract shape based on these predicates. To see how this process works, we give an example. Consider the predicates in Table 3.6 used in the case of a singly linked list and the example in Table 3.7 which shows how the concrete (Java) heap is abstracted into an abstract shape using the particular predicates. Indeed, this is the process we used in order to obtain the α and γ maps we defined earlier and which are the core of our system. It is important to note that after the initial abstraction we no longer need to deal with predicates, but the TVLA style initial abstraction does provide an intuitive and natural starting point for exploration.

Predicate	Meaning	Formula
$\{x(v) : x \in LVar\}$	local variable x points to value v	
$\{n(v_1, v_2) : n \in FVar\}$	all fields pointer with label n is between value v_1 and v_2	
$\{r_{n,x}(v) : x \in LVar\}$	v is reachable from stack pointer f via pointer with label n	$\exists v_x. x(v_x) \wedge n^*(v_x, v)$
$c_n(v)$	value v is in a cycle	$n^+(v, v)$

Table 3.6.: Predicates used for abstracting a singly-linked list

	Concrete	Abstract																																																																										
Structure																																																																												
Array	<table border="1"> <thead> <tr> <th></th> <th>u_1</th> <th>u_2</th> <th>u_3</th> <th>u_4</th> <th>u_5</th> <th>u_6</th> </tr> </thead> <tbody> <tr> <td>$x(u)$</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>$y(u)$</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>$r_{n,x}(u)$</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td> </tr> <tr> <td>$r_{n,y}(u)$</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> <td>1</td> </tr> <tr> <td>$c_n(v)$</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> </tbody> </table> <p>(a) Unary predicates and their results</p>		u_1	u_2	u_3	u_4	u_5	u_6	$x(u)$	1	0	0	0	0	0	$y(u)$	0	0	0	1	0	0	$r_{n,x}(u)$	1	1	1	1	1	1	$r_{n,y}(u)$	0	0	0	1	1	1	$c_n(v)$	0	0	0	0	0	0	<table border="1"> <thead> <tr> <th></th> <th>u_1</th> <th>u_a</th> <th>u_4</th> <th>u_b</th> </tr> </thead> <tbody> <tr> <td>$x(u)$</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>$y(u)$</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>$r_{n,x}(u)$</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td> </tr> <tr> <td>$r_{n,y}(u)$</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>$c_n(v)$</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> </tbody> </table> <p>(c) Unary predicates and their results</p>		u_1	u_a	u_4	u_b	$x(u)$	1	0	0	0	$y(u)$	0	0	1	0	$r_{n,x}(u)$	1	1	1	1	$r_{n,y}(u)$	0	0	1	1	$c_n(v)$	0	0	0	0		
		u_1	u_2	u_3	u_4	u_5	u_6																																																																					
	$x(u)$	1	0	0	0	0	0																																																																					
	$y(u)$	0	0	0	1	0	0																																																																					
$r_{n,x}(u)$	1	1	1	1	1	1																																																																						
$r_{n,y}(u)$	0	0	0	1	1	1																																																																						
$c_n(v)$	0	0	0	0	0	0																																																																						
	u_1	u_a	u_4	u_b																																																																								
$x(u)$	1	0	0	0																																																																								
$y(u)$	0	0	1	0																																																																								
$r_{n,x}(u)$	1	1	1	1																																																																								
$r_{n,y}(u)$	0	0	1	1																																																																								
$c_n(v)$	0	0	0	0																																																																								
	<table border="1"> <thead> <tr> <th></th> <th>u_1</th> <th>u_2</th> <th>u_3</th> <th>u_4</th> <th>u_5</th> <th>u_6</th> </tr> </thead> <tbody> <tr> <td>u_1</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>u_2</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>u_3</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>u_4</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>u_5</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>u_6</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> </tbody> </table> <p>(b) Binary predicates and their results</p>		u_1	u_2	u_3	u_4	u_5	u_6	u_1	0	1	0	0	0	0	u_2	0	0	1	0	0	0	u_3	0	0	0	1	0	0	u_4	0	0	0	0	1	0	u_5	0	0	0	0	0	1	u_6	0	0	0	0	0	0	<table border="1"> <thead> <tr> <th></th> <th>u_1</th> <th>u_a</th> <th>u_4</th> <th>u_b</th> </tr> </thead> <tbody> <tr> <td>u_1</td> <td>0</td> <td>$\frac{1}{2}$</td> <td>0</td> <td>0</td> </tr> <tr> <td>u_a</td> <td>0</td> <td>$\frac{1}{2}$</td> <td>$\frac{1}{2}$</td> <td>0</td> </tr> <tr> <td>u_4</td> <td>0</td> <td>0</td> <td>0</td> <td>$\frac{1}{2}$</td> </tr> <tr> <td>u_b</td> <td>0</td> <td>0</td> <td>0</td> <td>$\frac{1}{2}$</td> </tr> </tbody> </table> <p>(d) Binary predicates and their results</p>		u_1	u_a	u_4	u_b	u_1	0	$\frac{1}{2}$	0	0	u_a	0	$\frac{1}{2}$	$\frac{1}{2}$	0	u_4	0	0	0	$\frac{1}{2}$	u_b	0	0	0	$\frac{1}{2}$
	u_1	u_2	u_3	u_4	u_5	u_6																																																																						
u_1	0	1	0	0	0	0																																																																						
u_2	0	0	1	0	0	0																																																																						
u_3	0	0	0	1	0	0																																																																						
u_4	0	0	0	0	1	0																																																																						
u_5	0	0	0	0	0	1																																																																						
u_6	0	0	0	0	0	0																																																																						
	u_1	u_a	u_4	u_b																																																																								
u_1	0	$\frac{1}{2}$	0	0																																																																								
u_a	0	$\frac{1}{2}$	$\frac{1}{2}$	0																																																																								
u_4	0	0	0	$\frac{1}{2}$																																																																								
u_b	0	0	0	$\frac{1}{2}$																																																																								

Table 3.7.: process of linked list abstraction, the predicate set P from Table 3.6

3.4. Incremental Graph Drawing Algorithms

After the problem of interactions with the graph, the next step is how the position of the graph nodes should be specified. Arranging positions of vertices and paths of edges is a basic problem in graph layout. We compared a *force-directed layout algorithm* with a *hierarchical layout algorithm* and concluded that hierarchical layout is the most suitable style for our project. Furthermore, to build up the “mental map” and minimize the cognitive distance between graphs, the algorithm should change the graph incrementally and gradually. For our experiments, we selected an existing dynamic hierarchical layout algorithm called *DynaDAG*. In the future, we will study more refined algorithms.

The work of Dwyer et al. [Dwyer et al. 2009] has performed a comparison between user-generated and automatic graph layouts using multi-touch interaction on a tabletop display. In their experiment, users were asked to optimize the layout for aesthetics and analytical tasks of

3. Enabling Deep Program Interaction

a social network. Their results showed that users always tend to minimize the distance between vertices which share similar properties and reduce crossings of edges to make relations more straightforward.

From their results, we derive two criteria for static graph drawing: 1. few edge crossings; 2. Euclidean distance between two vertices should be minimized. Two kinds of graph layout algorithms are trying to optimize the graph following these criteria, discussed next.

Force-directed layout algorithm Force-directed layout algorithm (FD algorithm) is an algorithm which physically simulates the distance between vertices by assigning forces among the edges and vertices. The algorithm simply iterates until it finds the lowest energy. Typically, an edge is treated as a spring, which maintains a certain distance between the source and target vertices. It is widely used in *infoVis* for the following reason. At first, it can provide good quality result, which guarantees uniform edge length and uniform vertex spread. Second, the drawing process can be dynamic. Third, the algorithm can be highly parameterized through modifying the force of individual edge and vertex.

However, in software visualization, FD algorithms are rarely used. Its advantages are based on its high computational complexity. Generally, a typical FD algorithm has a running time equivalent of $O(n^3)$. If a domain-specific constraint is already embedded in the graph, the programmer also needs to encode this constraint in the parameter which causes an overhead in implementation.

Hierarchical layout algorithm Hierarchical layout algorithm or layered graph drawing algorithm is a type of graph drawing algorithm which draws vertices in horizontal layers with the edges downwards. It was originally used for drawing DAGs (directed acyclic graphs). By applying circle removal as a pre-processing stage, directed cyclic graph can be transferred to a DAG. It was first developed by Sugiyama in the 1980s and became famous and broadly used when GraphViz was developed [Ellson et al. 2004]. The heuristics of the hierarchical layout algorithm are: 1. exposing layered structure; 2. preventing edge crossings and sharp bends; 3. shortening edges; 4. maintaining symmetry and balance.

A typical hierarchical layout algorithm has the following steps:

- **Preprocessing.** Cyclic graph will be transferred to acyclic graph. The backward edges will be inverted.
- **Ranking.** Vertices are grouped into layers. Intuitively, a topological sorting can be applied. In GraphViz, it is treated as a linear programming problem. All connected vertices should be vertically close. A Network Simplex Algorithm (NSA) is used to solve this optimization problem.
- **Cross minimization.** Sorting is performed to reduce the crossing of edges between two adjacent layers.
- **Positioning.** This can again be treated as a linear programming problem. All connected vertices should be close horizontally.
- **Edge drawing.** Edges can be either drawn as straight lines or curves. In GraphViz, edges

are drawn as splines to avoid overlapping.

The whole drawing process can be regarded as a global optimization algorithm. Every isomorphic graph corresponds to a unique drawing result. So it might be possible that a change in a graph varies the drawing result a lot. For example, assigning a next pointer to null is a common pointer manipulation. This operation can be described as cutting an edge in a graph. Table 3.8 shows the difference between visualizing a graph using a *static* algorithm(GraphViz) and a *dynamic* algorithm(*DynaDAG*). In Figure 3.8(a), five nodes are linked. The edge between node4 and node5 is cut. Then the result is treated as an individual graph and repositioned by GraphViz(Figure 3.8(b)). On the other hand, only an edge removal operation is processed in the incremental algorithm, so the shape of the original graph remains(Figure 3.8(c)).

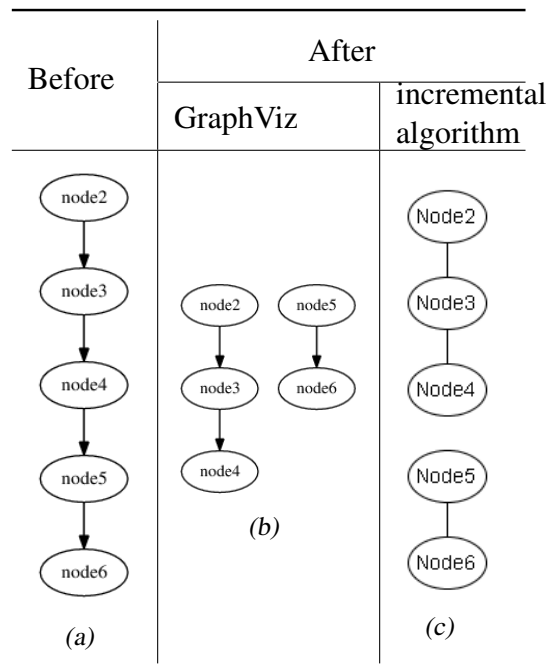


Table 3.8.: Drawing graphs using the GraphViz layout engine. Two graphs on the right side are derived from the graph on the left side by deleting an edge between *node4* and *node5*.

A sudden change in the graph will be visually displeasing to a programmer. Moreover, minimizing the change between two adjacent states can help building the user's "mental map". Previous works [Misue et al. 1995] [Diehl and Görg 2002] showed that by forming a good mental map, the user can comprehend the change between different states better. GraphViz and other static hierarchical layout algorithms are not built for incremental graph drawing.

DynaDAG: Incremental hierarchical layout algorithm

In incremental or dynamic graph drawing problems, changes in the graph and its drawing of the graph are possible by inserting, deleting, or modifying vertices and/or edges. FD algorithms natively support dynamic graph drawing, since they involve an iterative updates in the drawing process. For static hierarchical layout algorithms, an intuitive way to transfer a static algorithm to a dynamic one is to apply constraints, which keep the unrelated parts of the graph steady and only change the sibling vertices. Branke[Branke 2001] suggests two constraints:

3. Enabling Deep Program Interaction

- **Ranking.** The set of vertices belonging to the same layer should remain in the same layer.
- **Ordering.** Vertices in the same layer should remain in the same order.

However, if there is a limitation in drawing space, these constraints cannot be applied. Vertices in the same layer cannot be distributed to the other layer, if the current layer is full. The constraint should be designed as a compensation parameter in the optimization process. The DynaDAG algorithm is the algorithm which uses this approach. It adds incremental drawing features which accept operations:

$$\{insert, delete, modify\} \times \{vertex, edge\}$$

Generally, DynaDAG uses a similar pipeline as GraphViz, which is 1. preprocessing; 2. ranking; 3. ordering; 4. positioning; 5. edge drawing. It reuses the network simplex solver, which is used for ranking and ordering in GraphViz. In addition, it adds constraints, which balances the satiability and aesthetic criterion. The stability is archived through modifying two optimization problems in the pipeline. Table 3.9 shows a comparison between these two problems, GraphViz uses a global optimization, while DynaDAG optimizes, but also adds a penalty, which is derived from the original graph.

GraphViz	DynaDAG
$\min \sum_{(u,v) \in E} \omega_{gr}(u,v)(\rho(u) - \rho(v))$ <p>with $\forall (u,v) \in E, \rho(u) - \rho(v) \geq \delta(u,v)$</p>	$\min \left(\sum_{(u,v) \in E} \omega_{gr}(u,v)(\rho(u) - \rho(v)) + \sum_{u \in V} \theta(u) \cdot c(u) \right)$ <p>with $\forall (u,v) \in E, \rho(u) - \rho(v) \geq \delta(u,v)$</p>
$\min \sum_{(u,v) \in E} \omega_{gp}(u,v) x(u) - x(v) $ <p>with $\forall (u,v) \in E, x(u) - x(v) \geq \rho(u,v)$</p>	$\min \left(\sum_{(u,v) \in E} \omega_{gp}(u,v) x(u) - x(v) + \sum_{u \in V} \chi(u) \cdot c(u) \right)$ <p>with $\forall (u,v) \in E, \rho(u) - \rho(v) \geq \delta(u,v)$</p>

Table 3.9.: Objectives and constraints in GraphViz and DynaDAG; the linear programming problems for the Ranking process are in the second row; the linear programming problems for the Ordering process are in the third row.

DynaDAG maintains an auxiliary graph CG whose nodes act as variables and edges act as constraints. The downward edges are denoted as *strong* edges, while unconstrained edges are *weak* edges. Strong edges and weak edges which point downwards share a normal weight function $\omega_{gr}(e)$. When a weak edge is pointing upward or parallel to the layer, the weight function is $\omega'_{gr}(e)$, which contains a high penalty. Like GraphViz, DynaDAG uses a network simplex solver to solve the integer programming problem. The solver could not support stability intrinsically. To compensate, additional variables and constraints are added. This is the key trick which helps to reduce the gap between two states.

According to the optimization problems (Table 3.9), the balance between stability and optimization can be tuned by adjusting θ and χ . Apart from the ability to maintain stability, the user can easily adjust the size of objects in the graph as well as the vertical and horizontal separation

3.4. Incremental Graph Drawing Algorithms

between nodes. In the performance perspective, the complexity of a network simplex process is $O(IVE)$. Although I is not polynomial, it is close to linear in practise. Cross minimization has complexity of $O(I'VE)$, where I' is a small predefined constant. In the edge drawing process, complexity is up to $O(V^3)$, but it is close to $O(V^2)$ in practice. However, in the implementation process, we did not use the edge routing from DynaDAG, as instead we draw lines directly between adjacent nodes.

3. *Enabling Deep Program Interaction*

4

Implementation

In this chapter, we are going to describe the implementation process and the design of basic components. We first give a global view of the application's client-server architectures. And then, we mentioned the design of user interface of the frontend application. Next, we are going to describe the system from top to down. In the frontend layer, we show the graph drawing framework, in which we deliver graph visualization, layout and manipulation components. Meanwhile, we briefly mentioned two software prototypes based on different graph visualization frameworks. Close behind is the communication protocol, which provides a low coupling relation with the backend. Last but not least, the debug plug-in in Eclipse acts as the backend, which captures heap information when a breakpoint is hit and send incremental updates of the concrete heap to clients.

4.1. Architecture of Application

Based on the design requirements, we address three important aspects in the implementation: **Pen and touch interaction**, **High coherent components** and **Reusable components**. In the first stage, we considered to build an Eclipse plug-in application, which includes all components. The advantage is that communication between graphical frontend and backend can be easily leveraged by intra-process communication in Java. However, since Java SWT uses native UI elements, it may not support every behavior on all systems. The pen and touch support for Windows is not sufficient for building the frontend. It requires a lot effort before getting a fluid pen and touch interaction. A separation of frontend and backend is the best solution in this scenario. Then, the component for inter-process communication is an indispensable layer between frontend and backend. The hierarchy is defined in Figure 4.1. The relation between different components are illustrated as the sequence diagram in Figure 4.2.

4. Implementation

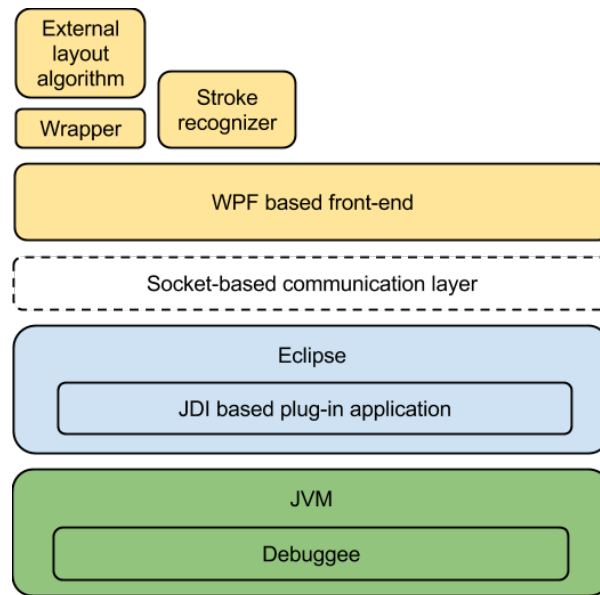


Figure 4.1.: Architecture of HeapVision

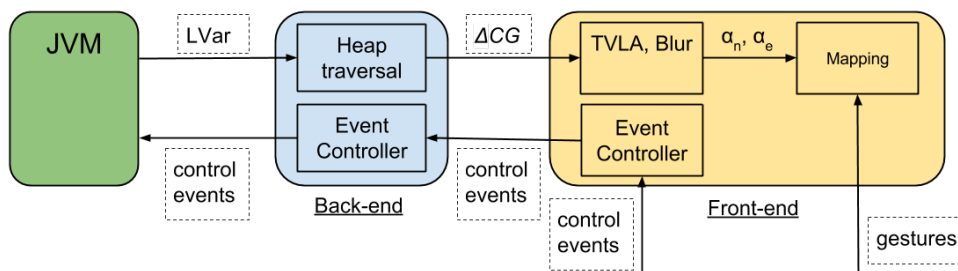


Figure 4.2.: Interactions between different components. Figure A.1 shows a detailed sequence diagram.

From right to left in the sequence diagram, Eclipse acts as a portal towards the virtual machine. User can either perform debug operations, such as *step over*, *step in* and *continue*, through the frontend or directly through the control panel of Eclipse. The Eclipse plugin acts as a backend and uses JDI(Java Debug Interface) to communicate with the virtual machine. When a breakpoint is hit, the backend traverses the heap from local variables in the top stack frame. During the traversal, information is collected to build a concrete heap graph. If a client exists, the backend will broadcast this information. Finally, the frontend parses the graph information and render it on screen. Then, the user can interact with the graph.

4.2. User Interface Design

Besides the current graph viewer, a code viewer, a historical graphs' viewer and a previous graph viewer are built to assist the debugging experience.

- **Historical graphs viewer** is a collection of all the previous states of graph.(Part a in Fig. 4.3) Graphs which are captured in the same line of code are stored in the same horizontal ListView. Listviews are organized sequentially in a vertical ListView. User can retrieve

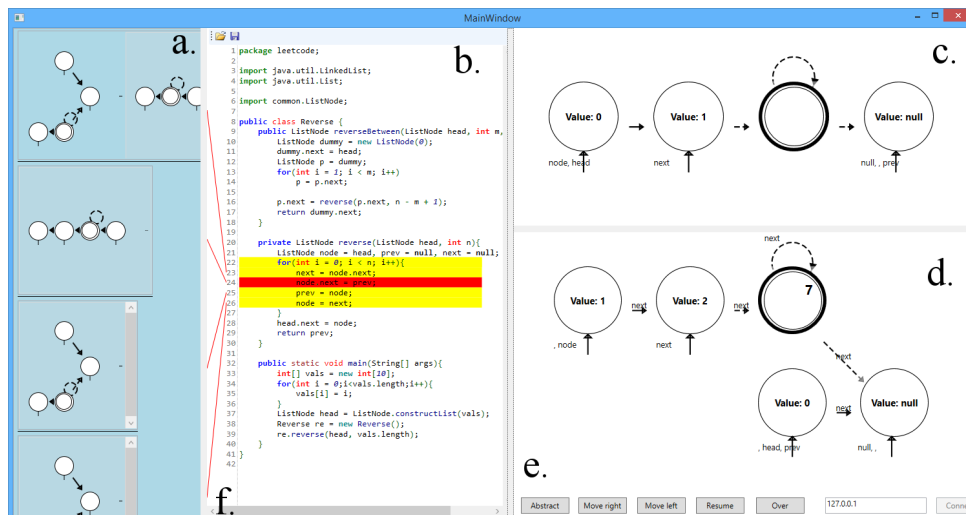


Figure 4.3.: User interface design of the frontend application

information in two dimensions. In horizontal direction, user can see how data is changed in the same position of the program. In vertical direction, user can see how data is changed in the control flow.

- **Code viewer** is the viewer for the debugging file.(Part b in Fig. 4.3) AvalonEdit¹ is used as the text viewer component. It offers all necessary features for building a code editor, including syntax highlighting, auto completion and grammer checking.
- **Previous graph viewer** is the view for a previous state of graph.(Part c in Fig. 4.3) It offers a detail view for a previous graph snapshot. User can drag a thumbnail from the historical graphs viewer and drop it on the viewer. Then, the graph data structure will be used for re-rendering.
- **Current graph viewer** is the view for visualization and manipulation of the current state.(Part d in Fig. 4.3)It will be further described in Section 4.3.
- **Control panel** is for calling basic debug operations.(Part e in Fig. 4.3)They are going to be mentioned in Section 4.4.
- **Mapping view** offers a visual indication for the ralated graphs and lines of source code.(Part f in Fig. 4.3) We insert one panel prior to the view of line numbers in AvalonEdit. It simply draws strings between the lines of code which graph snapshot(s) is available and their corresponding graphs. These strings will be redrawn whenever the viewport is changed in either the code viewer or the historical graphs viewer.

4.3. Graph Drawing Framework

As mentioned above, providing a fluid pen and touch interaction upon the graph visualization is one of the important aspects of implementation. To achieve this goal, we sepearte the imple-

¹AvalonEdit,<https://github.com/icsharpcode/SharpDevelop/wiki/AvalonEdit>

4. Implementation

mentation into a few tasks:

- **Graph visualization.** Find an existing graph drawing framework, which can provide basic interactions, like resizing and repositioning, as well as customizable graph rendering.
- **Graph layout algorithm.** Find an existing hierarchical graph layout algorithm, which can draw graph incrementally and ideally maintain the previous manual changes in different levels.
- **Graph interaction.** Extend the graph drawing framework and support the provided interactions.

4.3.1. Graph visualization

A comparison between different graph drawing frameworks based on different GUI frameworks is given in the first place.

Name	Language	GUI framework	Purpose
Prefuse [Heer et al. 2005]	Java	Swing/AWT	a stand-alone dynamic visualization framework
JUNG ¹	Java	Swing/AWT ²	an universal framework for modeling, analysis, and visualization graph data
Zest ³	Java	SWT ⁴	a graph drawing framework for Eclipse plug-in
Graph# ⁵	C#	WPF ⁶ , WinForm ⁷	a simple graph layout framework in WPF
GraphX ⁸	C#	WPF, WinForm	An extended version of Graph#
GLEE(MSAGL) ⁹	C#	WPF, WinForm	.NET tool for graph layout and viewing
Flare ¹⁰	ActionScript	Flash	an ActionScript library for web-based visualizations in Flash

¹Java Universal Network/Graph [O'Madadhain et al. 2005]

²Abstract Window Toolkit

³Eclipse Zest, <http://www.eclipse.org/gef/zest/>

⁴Standard Widget Toolkit, <http://www.eclipse.org/swt/>

⁵GraphSharp, <http://graphsharp.codeplex.com/>

⁶Windows Presentation Foundation, [http://msdn.microsoft.com/en-us/library/ms754130\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx)

⁷Windows Form, [http://msdn.microsoft.com/en-us/library/dd30h2yb\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/dd30h2yb(v=vs.110).aspx)

⁸GraphX, <https://graphx.codeplex.com/>

⁹GLEE(MSAGL), <http://research.microsoft.com/en-us/projects/msagl/default.aspx>

¹⁰Flare, <http://flare.prefuse.org/>

D3 [Bostock et al. 2011]	JavaScript	HTML	a novel representation-transparent approach to visualization for the web
-----------------------------	------------	------	--

Table 4.1.: Comparison between existing graph drawing frameworks

Software prototypes

To compare different frameworks in a practical level, we have implemented three software prototypes. In the beginning, in order to simplify the communication between frontend and backend, two Eclipse plugins were developed. However, the support for pen and touch input in Java is not sufficient. Then we splitted the application into three components and implemented the frontend based on WPF.

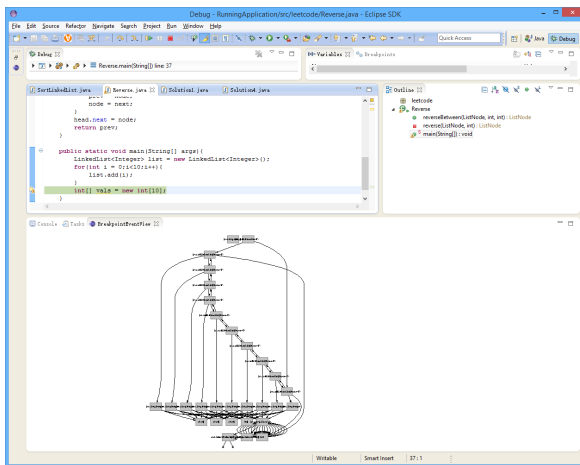
Prefuse based prototype At first, we have tried to use Prefuse to build a graph visualization plugin in Eclipse. Prefuse [Heer et al. 2005] is one of the most famous and successful information visualization frameworks in both *infovis* and HCI community. Many domain-specific visualization tasks can be solved within one stand-alone solution. We have tried to migrate this Swing/AWT based framework to Eclipse, which is a complete SWT based environment. To offer an aesthetic appearance of the graph, we have built a wrapper of GraphViz. However, although it does work with the help of the SWT/AWT Bridge, it keeps redrawing the whole graph whenever resizing or repositioning happens, which introduce a flickering effect on the graph. Meanwhile, without a native support for pen and touch events, both the event handling pipeline and UI elements must be varied.

Zest based prototype We built a second prototype based on Eclipse Zest. Zest is a component of the Graphical Editing Framework(GEF)¹, which is a set of visualization components built for Eclipse. Its library is based on SWT, so it avoids the compatibility problem between AWT and SWT. Moreover, SWT offers more support for touch interface. However, the touch events and pen events are still mixed in SWT, since all events are retrieved from the operating system. In Windows, touch and pen events are grounded as a same type in the event loop. It is possible to separate them only using individual touch and pen events handlers. It is again introducing an overhead in implementation. So, a GUI framework which offers a out-of-box pen and touch support and flexible user interface configuration is considered at the first place in the actual implementation.

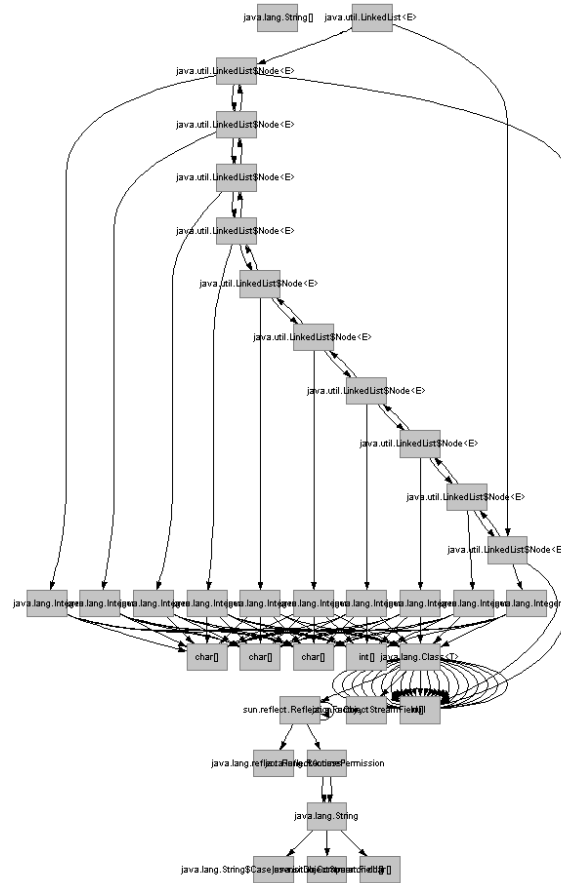
GraphX based prototype Finally, we have abandoned to build an Eclipse plug-in, which is written in Java. A separate frontend application is built for graph visualization and manipulation. In Windows, WPF is a unparallel framework for building rich client-side application. It has established a good support for pen and touch interactions. Besides multiple input devices'

¹<http://www.eclipse.org/gef/>

4. Implementation



(a) Software prototype of an Eclipse plug-in for data structure visualization



(b) A LinkedList with ten entries

Figure 4.4.: Software prototype using Prefuse framework

event handling, internal gesture and stroke recognizers will save efforts in the future development. *GraphX* and *Graph#* are two candidates based on WPF. They share a lot of common features, including customizable UI elements, extensible graph layout algorithms, zoom, drag and pan interactions. One advantage makes *GraphX* prior to *Graph#* is that it is a recent override based on *Graph#*. The project is being maintaining. So *GraphX* becomes the final choice for the graph visualization framework.

GraphX takes advantage of two existing building blocks, which are *QuickGraph* and *Graph#*. *QuickGraph*[Halleux 2012] provides generic directed and undirected graph data structures and graph algorithms for .NET framework. Its graph data structure is used in both *GraphX* and *Graph#*. *Graph#* provides various implementations of graph layout algorithms, including force-directed layout algorithms, hierarchical layout algorithms. In addition, overlapping removal algorithms[Dwyer et al. 2006] and edge routing algorithms[Lozano-Pérez and Wesley 1979] are provided to ease the graph visualization. *GraphX* is responsible for rendering and handling control events. Since WPF is used as the UI stack, a flexible support for interface design is out of the box.

4.3.2. Implementation of graph layout algorithm

In the previous section, we have briefly compared different layout algorithms, and concluded that hierarchical graph is the most suitable graph drawing style for visualizing the heap graph. In this section, the implementation process for migrating an existing incremental graph layout engine DynaGraph [Ellson et al. 2004] will be described. A graph layout engine is a program which takes a graph $G = \langle V \times E \rangle$ or changes of a graph ΔG as input, and returns the coordinates of nodes and paths of edges. The difference between *DynaGraph* and *GraphViz* is that *DynaGraph* tries to balance the optimality of the layout and the stability of an incremental update subject its last state. DynaGraph is written C\C++ and offers three interfaces, a command-line interface which receives specific graph drawing commands, a COM component which enables language-independent communication and a set of C\C++ API. Since the command-line interface can be directly access via the executable of DynaGraph, it has been tried at first. However, the commands only accept operations $\{\text{insert, delete, modify}\} \times \{\text{node, edge}\}$, which means that every insertion, deletion and modification of an node and edge requires a recalculation. Meanwhile, generating and parsing the commands take time in the process. The inefficiency of this approach makes this approach abandoned in our scenario. Finally, we make use of DynaGraph via a C++\CLI wrapper, which facilitates the communication between C\C++ and .Net environment.

The input and output interfaces can be defined. The layout engine receives graph update information $\Delta G = \langle V_+, V_-, E_+, E_- \rangle$ and returns current coordinates of all nodes P . Besides the incremental update, DynaGraph is able to receive a complete graph and return an optimal hierarchical layout. Actually, it can handle different shapes and sizes of nodes as well as provide spline curves for edges. We fix the visual settings to avoid an overhead in implementation.

Value	Explanation	Value	Explanation
$G = \langle V \times E \rangle$	current graph	$u, v, w, \dots \in V$	node
$G' = \langle V' \times E' \rangle$	last graph	$e, f, \dots \in E$	edge
$V_+ = V \setminus (V \cap V')$	inserted nodes	$X(v), Y(v)$	position of node center
$V_- = V' \setminus (V \cap V')$	deleted nodes	$\{(X(v), Y(v)) \mid \forall v \in V\}$	Coordinates of all nodes
$E_+ = E \setminus (E \cap E')$	inserted edges	(b) Output of the incremental graph layout engine	
$E_- = E' \setminus (E \cap E')$	deleted edges		

(a) Input of the incremental graph layout engine

Table 4.2.: Input and output of the incremental graph layout engine

4. Implementation

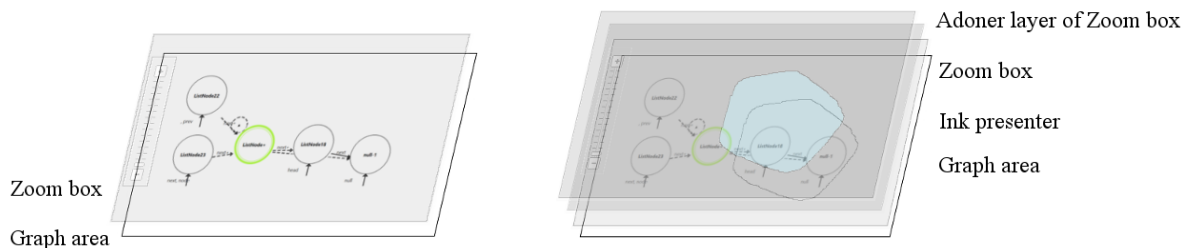
4.3.3. Graph Interaction

As mentioned in Section 3.2, four types of graph interactions are supported. For view related interactions, resizing and repositioning of graph are supported only through mouse input. Supporting gestures is not a big change in the architecture. However, to support the pen-based interactions, we need to modify the framework. Meanwhile, we built animations to narrow the gap between the changes. Therefore, three main tasks will be addressed here.

- **InkPresenter insertion.** An InkPresenter should be added on top of the graph drawing layer to receive and render pen strokes.
- **Abstraction and concretization.** We implement *vertex contraction*(Alg. 3.1) and *vertex cleaving*(Alg. 3.2) algorithms to achieve the abstraction and concretization operations.
- **Inter-frame animation.** Animations are built to avoid sudden changes between states.

InkPresenter insertion

Pen support can be achieved either directly from the input device or from supporting UI framework. Because of the good support for pen and touch interaction, we directly fetch the pen events from WPF. Two widgets support visualizing and collecting digital ink. InkPresenter is the widget which can be covered on an instance of Panel class. The modification of the application structure is shown in Figure 4.5.



(a) Original hierarchy of UI elements in GraphX. ZoomBox is on the top. It is responsible for scaling and translating the GraphArea.

(b) Current hierarchy of UI elements in HeapVision. An InkPresenter is added on top of the GraphArea. It renders all digital inks. The selected area is shown on an AdonerLayer of ZoomBox.

Figure 4.5.: Comparison between original and current graph rendering layers. Fig. a is the original layer, while Fig. b is the current one.

InkPresenter keeps all strokes in a stroke collection. Every stroke is made up of a continuous series of StylusPoint. Programmer can customize the drawing attributes of the InkPresenter to change the strokes' appearance. On the other hand, a separate thread for stroke rendering is created to guarantee a quick response. All inks in a live stroke will be rendered first by this thread. Once the stroke is finished, it can be collected for a further purpose, such as stroke recognition and text recognition. In our scenario, the stroke will be used to form a selection boundary. Then a hit-test will be performed on every node in the view to judge whether it is being selected. Since different coordinate systems in digital ink layer and graph drawing layer, a coordinate translation is necessary.

Inter-frame animations

As mentioned in the previous section, animation plays an important role in bridging two states. Especially when the shapes of two states look like the same, an animation will help a lot to build mental maps of information [Bederson and Boltman 1999]. An animation can be formally define as a tuple of original and target position $animation = \langle X(v'), Y(v'), X(v), Y(v) \rangle$. We list five situations in which animation will happen. WPF provides a good support for animation of heterogeneous properties, including color as well as affine transformations. We use `DoubleAnimation` to leverage color, opacity and position. Currently, the node moves from source to target in a line. Other nodes are possible to be occluded by the moving node. To ease this shortcoming, the node should follow a curved path to avoid overlapping.

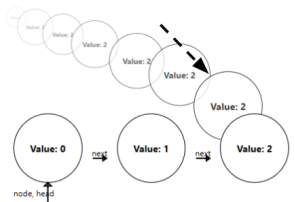
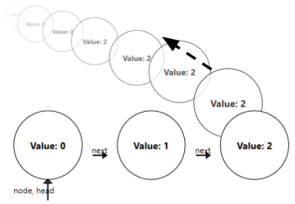
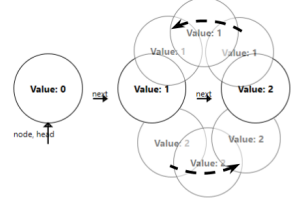
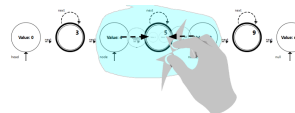
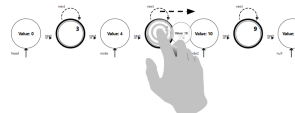
Value	Explanation	Animation
$\langle 0, 0, X(v), Y(v) \rangle$	creation of node	
$\langle X(v'), Y(v'), 0, 0 \rangle$	deletion of node	
$\langle X(v'), Y(v'), X(v'), Y(v') \rangle$	Concrete node's movement	
$\langle X(v'), Y(v'), X(\alpha_N(v')), Y(\alpha_N(v')) \rangle$	Concrete to abstract movement	
$\langle X(\alpha_N(v')), Y(\alpha_N(v')), X(v'), Y(v') \rangle$	Abstract to concrete movement	

Table 4.3.: Five types of node animations in *HeapVision*

4.4. Custom Protocol for Communication

All basic information we need for visualization will be wrapped using XML format. A socket-based component will be used for this bi-directional communication. Since this application is mainly used locally, we do not consider the potential delay as well as the possibility of lost package. Compared to JDWP(Java Debug Wire Protocol)¹, our protocol offers much less information and features. However, using JDWP is not feasible in this case. JDWP is the protocol used for communication between a debugger and the JVM. JDI offers higher-level APIs on top of the JDWP, which makes JDI a more appropriate and usable interface for debugging tools. A more detailed introduction about the JDI and JDWP will be presented in section . If we are going to use JDWP between frontend and JVM, a C# based parser is necessary for parsing the complex messages. It again leads to an overhead in implementation.

Three different types of messages are used during the communication. Their usages and elements are as below. In words, *Graph* message is sent from backend to frontend when the first breakpoint is hit or when a client is connected to a server while there is a running debugging process. It carries a concrete heap graph. *Update* message is as well from backend side and delivers the incremental update information of the graph. *Control* message will be either sent from backend or frontend. The message from frontend is to control debugging process. On the contrary, it carries breakpoint information and running status from backend.

Notation	Explanation
$LVar, LVar'$	current local variables and local variables at previous breakpoint
$LVar_+ = LVar \setminus (LVar \cap LVar')$	new local variables
$LVar_- = LVar' \setminus (LVar \cap LVar')$	invalid local variables
β, BP	the current breakpoint, breakpoints in the document
start, stop	debugging start and stop events
step_in, stop_over	step into the statement, step over a statement
continue	continue execution until the next breakpoint is hit

Table 4.4.: Elements in the custom protocol

¹Java Debug Wire Protocol, <http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html>

Type	Explanation	Elements	Direction
Graph	the initial heap graph	$\langle V \times E \rangle \cup LVar$	back to front
Update	heap graph update	$\{V_+, V_-\} \cup \{E_+, E_-\} \cup \{LVar_+, LVar_-\}$	back to front
Control	debug events	$BP \cup \{\beta\} \cup \{\text{start}, \text{stop}\}$	back to front
		$\{\text{step_over}, \text{step_into}, \text{continue}\}$	front to back

Table 4.5.: Messages of the custom protocol for communication between backend and frontend

4.5. Debug Plugin

In this section, the implementation of debug plugin will be described. At first, JDI(Java Debug Interface) will be described. It models the Java debugging environment and provide a set of elaborated APIs to build a debugger. Second, the algorithm for querying the heap from stack pointers will be illustrated. By starting, we have a brief walkthrough of the control-flow in handling the event when a breakpoint is hit(Figure 4.7). In words, the top stack frame is given as input when a breakpoint is hit. Every local variable is an entry of a traversal. After combining the traversal results, the concrete heap graph is constructed. If it is the first graph, it will be sent directly to all clients. Otherwise, the update of graph and local variables is sent.

4.5.1. Java Debug Interface

At first, we are going to describe JPDA(Java Platform Debugger Architecture), since JDI is one of its interfaces. JPDA is a multi-layered debugging architecture which enables building Java debuggers which run across platforms, virtual machine implementation and JDK versions. Besides JDI, the other two interfaces are JVM TI(Java Virtual Machine Tools Interface) and JDWP(Java Debug Wire Protocol). JVM TI defines the services as virtual machine must provide for debugging. It is usually used for building an agent, which collects information directly in virtual machine at run-time for the purpose of profiling. JDWP defines the format information between the virtual machine and debugger, which is language independent. JDI defines information and communications at a user code level. Through implementing the JDI, a common java debugger frontend can be built.

Important data types in JDI are:

- `IStackFrame` refers to the stack frame in a current running thread, get stack pointers, locate code position in source file and source file's location.
- `IVariable` refers to any variable in the source code, such as a stack pointer, a field variable and an entry of an array.
- `IValue` refers to any value, such as primitives as well as objects.
- `IBreakpoint` refers to any breakpoint in the project, such as line breakpoint, method

4. Implementation

breakpoint and parameter watchpoint.

- ObjectReference acts as a mirror of an actual object in the program.

Debugger is able to manipulate values and call methods during debugging. It enables the possibility for user to change values directly in the graphical frontend. However, every invocation leads to an update of the stack frame, which makes it impossible to invoke methods during the traversal.

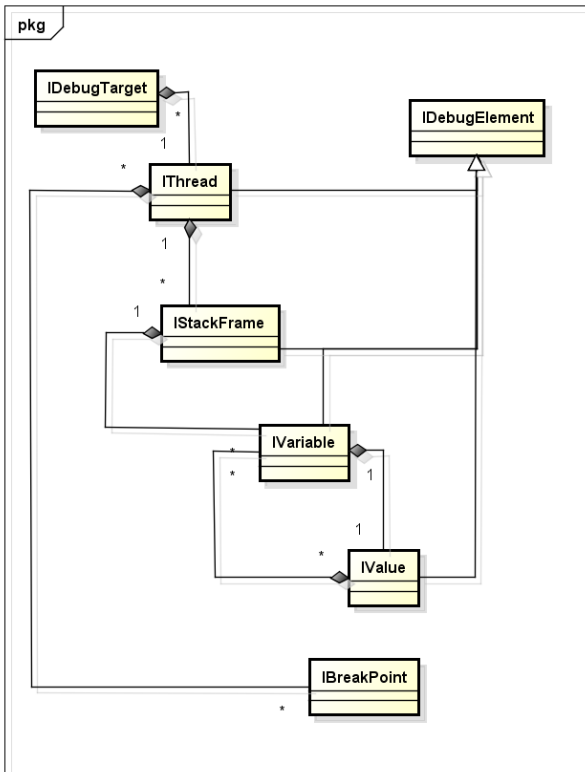


Figure 4.6.: Class diagram of debug models in JDI

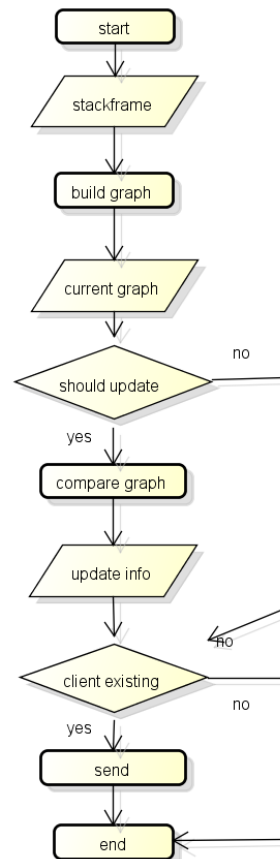


Figure 4.7.: Workflow of the backend when a breakpoint is hit

4.5.2. Heap traversal algorithm

Based on the user-level APIs provided by JDI, a graph traversal algorithm applies to achieve the heap graph. We basically use a depth-first-search algorithm to traverse the heap from each local variable. During the traversal, we explicitly filter out static field variables, since they are shared by all instances and usually will not be modified in its lifetime. We maintain a hash set to store the visited objects. The entry is the object ID, which is explicitly assigned by the JVM TI agent. It will not change in its life cycle.

```

1 Set oldLVar
2 Graph oldGraph
3 Set currentLVar
4 Graph currentGraph
5
6 void onBreakpointHit(StackFrame frame)
7 {
8     oldGraph = currentGraph.clone()
9     currentGraph.clear()
10    oldLVar ← currentLVar
11    stackPointers ← frame.vars
12    for sPointers in frame.vars
13        visit(NULL, sPointers)
14
15     $\Delta G$  ← compareGraph(oldGraph, currentGraph)
16     $\Delta LVar$  ← compareStackPointer(oldLVar, currentLVar)
17    onChange( $\Delta G$ ,  $\Delta LVar$ )
18 }
19
20 void visit(IVar pVar, IVar cVar)
21 {
22     IVal cVal ← cVar.value
23     if shouldFilter(cVal)  $\wedge$  isVisited(cVal) then
24         Node node ← createNode(cVar)
25         graph.addNode(node)
26         setVisit(cVal)
27         foreach childVar in cVal.vars
28             visit(cVar, childVar)
29
30     if parent  $\neq$  NULL
31         Edge edge ← createEdge(parent, child)
32         graph.addEdge(edge)
33 }

```

Listing 4.1: Depth-first-search heap traversal algorithm

4. *Implementation*

5

Evaluation

Evaluation of the application will be separated into two parts. At first, we will test the backend and see how is the performance when it is dealing different sizes of heaps. Second, we need to perform user study to verify that HeapVision helps user understanding program behavior and also find potential bugs in the program.

5.1. Experiment: Performance of basic algorithms

Due to the complex structure in the debugger hierarchy, it is hard to measure the computation complexity preemptively. We evaluate the performance by slicing the application into backend and frontend. The purpose for this evaluation is to see the capacity of algorithms in heap data retrieval, graph data processing and abstraction. We do not evaluate the algorithms based on any specific test programs. Instead, we see how they perform on specific data structures, which are a single linked list and double linked list with different lengths.

Back-end At first, we use a depth-first-search(DFS) algorithm to traverse the heap from stack pointers. The computation complexity for a typical DFS algorithm is $O(|E|)$. However, in JDI, the memory information is retrieved on-demand. And the query process includes three main steps: 1. JDI method invoking to JDWP message translation; 2. JDWP message to native JVMTI functions translation; 3. virtual machine level operation. It is clear that heap traversal using JDI should consume more time than a normal graph traversal. Second, in the communication component, we transform graph data structures to XML format, and send the serialized XML data using socket. We evaluate the performance of this communication architecture.

5. Evaluation

Front-end At first, we examine the capacity for parsing XML files in C#. Second, we evaluate the abstraction algorithm using the predefined abstraction predicates in Table 3.6.

5.1.1. Test program and method

We adopt two basic data structures, which are a single linked list, which has a primitive integer as value, and a double linked list, which is provided in the `java.util` package. It is easy to calculate the amount of vertices and edges. Meanwhile, the length of a list is the maximum depth of the spanning tree. We need to find the maximum depth which will not cause a stack overflow error.

To avoid interference, we need to make sure the top stack frame empty except the test data. We can simply use a method as stated in 5.1. A breakpoint is put in line 3, which allows the backend to visit from the head of the linked list. Meanwhile, we set timestamps in the source code and collect durations of execution.

```
1 void foo(List list){
2   System.out.println();
3 }
```

Listing 5.1: Example of the test program

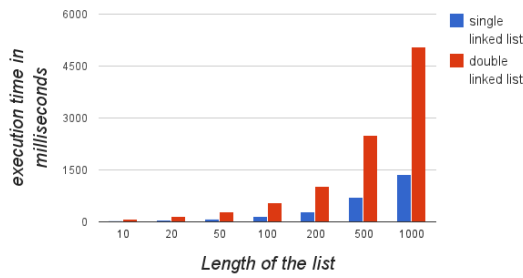
5.1.2. Results and discussion

length	backend				frontend			
	traversal		XML		XML		Abstraction	
	SLL	DLL	SLL	DLL	SLL	DLL	SLL	DLL
10	22.1	79.4	1.8	3	5.3	4	62.5	62.25
20	44.4	151.1	1.9	5.4	5	7.75	47	62.5
50	84.6	279	2.6	9.5	15.7	12	57.7	67.3
100	160.6	540.1	7	14.9	10.7	16	52.3	84.6
200	293.6	1013.2	11.43	31.4	15.7	23.75	73	172
500	704.6	2500.7	23.33	94.8	26.3	78	156.3	661.3
1000	1378	5063.6	61.42	179.6	93.7	285	495	2414

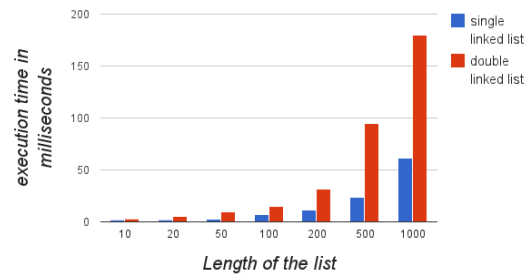
Table 5.1.: Execution time in millisecond of basic algorithms. SLL stands for single linked list. DLL stands for double linked list.

5.1. Experiment: Performance of basic algorithms

We have used different lengths of single and double linked list for the evaluation. A summary in Table 5.1 shows the execution time of different components. The heap traversal component takes the longest time in the backend side, while the abstraction component consumes the most in the front end side. Basically, we have three findings in the experiment. At first, a stack overflow error triggers when the list is bigger than 1300. To overcome this problem, we need to modify the recursive DFS algorithm to an iterative one. Second, the time scales exactly along with the amount of edges in the backend. The complexity of blur process in TVLA is based on the predicate set. Currently, the best case is $O(n)$, with the situation only a local variable is pointing to the linked list. The worse case is $O(n^2)$, with the situation that every node in the linked list is pointed by a local variable. The complexity of examining a $r_{n,x}(v)$ predicate is $O(n)$.

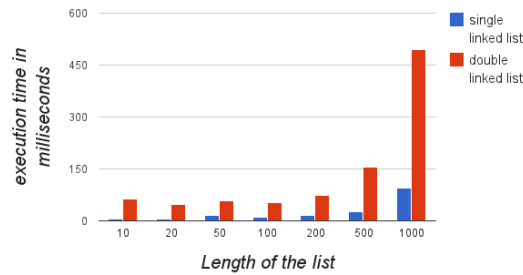


(a) Execution time of heap traversal

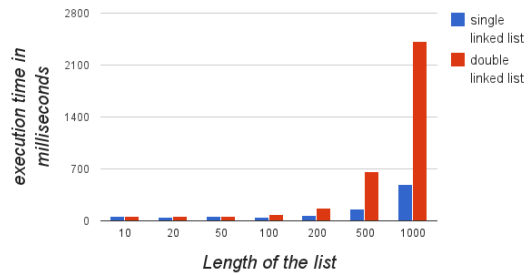


(b) Execution time of graph data serialization/transmission

Figure 5.1.: Execution time of heap traversal and graph data serialization/transmission in the backend



(a) Execution time of graph data deserialization



(b) Execution time of graph abstraction using TVLA's blur process

Figure 5.2.: Execution time of graph data deserialization and TVLA's blur process in the front end

Finally, we produce a summation of the time spent on the visualization and compare these results to three response time limits[Miller 1968], which are widely used in usability engineering. Three response time limits are used for the comparison, 1. 0.1 second limit for having user's feeling of latency; 2. 1 second limit for making user's flow of thought to stay uninterrupted; 3. 10 seconds limit for keeping user's patience . When the size of nodes in the linked list is less than

5. Evaluation

100, computations consume less than 1 second, which cost a sensible delay, but will not influence the user's flow of thought. For algorithm problems, the size of test data is not necessary to be large. So, we concluded that the processing time of the intermediate graph data does not cause a overhead. It is feasible to bridge the backend and frontend using the custom protocol.

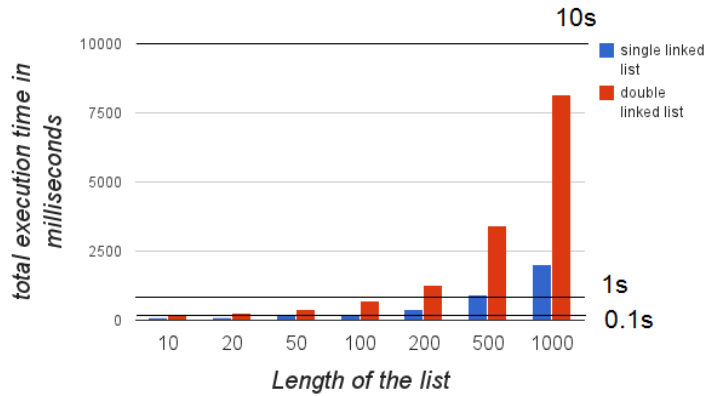


Figure 5.3.: Summation of the execution time used for producing a visualization for the linked list

5.2. User evaluation

Evaluating a tool for program comprehension and debugging is known as a systematic problem. At first, the efficiency in debugging is based on a user's understanding of the current program behavior. Skilled and novice programmers have various efficiency in debugging, which is caused by the difference in comprehension level of the program[Gugerty and Olson 1986]. That is to say, we need to first prove our tool facilitates program comprehension. In the second step, we can verify our hypothesis that user would be able to detect and correct logical bugs more accurately and efficiently.

5.2.1. Overview of program comprehension evaluation

Understanding how users comprehend a program helps us to build a evaluation for program comprehension. A mental model of programmers can abstract the program comprehension process in a high level. This research dates back 1970s, when Shneiderman mentioned that the syntax and semantic of a program can be separated in a program comprehension task. The syntax is programming language dependent, while the semantic is independent. We assume that the user have already had a good command of the syntax of the certain language, and what they learn from an algorithm is the semantic. Later, researchers have come up with more detailed models. They agree that comprehension happens either top-down, bottom-up, or using a combination of both. The top-down model from Soloway and Ehrlich[Soloway and Ehrlich 1984] typically applies when the code base is large. Programmers decompose the code into fragment of a similar type. Then programmers establish relations between the new code and the familiar code. The

bottom-up model from Pennington[Pennington 1987] describe how a user comprehends a program when he has no former experience of it. The User often subconsciously uses three program abstractions to represent programs and to assist program comprehension[Pennington 1987]. They are *data flow abstraction*, *control flow abstractions* and *function abstractions*. We concludes these three aspects as the answers three important questions during debugging process.

- **What** Users need to be aware of the current content or input in the program. These contents are data structures, which represent information of these contents in a different domain.
- **How** *How* questions are about the control-flow of a program. It reflects the execution sequence and allows a user to perceive how the program changes its content.
- **Why** Goal of a program or goals of subroutines should be clarified.

Referring to the engagement level by Naps[Naps et al. 2002], the ability of answering user's questions in the current context is one of the judge criterion of a good program visualization system. Our hypothesis is that our application can show data structures(*what*) and the changes(*how*) better than the traditional interface in IDE. To verify the former one, we examine two assumptions: 1. the graphical syntax is expressive; 2. the navigation is efficient. For the latter one, our assumption is that users can understand the pointer manipulation process from the graphical symbols and the animated transitions.

Our application is designed for heap-manipulation-programs, which mainly deals with linked-list and tree data structures. Since the prototype is not good enough to perform a test based on user's daily task, we need to choose specific programs fragments as debugees to perform the debugging evaluation. However, in a debugging process, programmer and debugee are two independent variables. To compare the efficiency when different debugging tools are used, we need to make sure the participants are fairly in a same level and the debugees are in a same level of difficulty. For the former part, it is not difficult to find novice users which are in a same knowledge level, while it is more difficult to find programmers who share a similar level. For the second part, it is hard to measure the comprehension difficulty or even code quality of a program. Since readability, efficiency as well the goals of different program fragments can not be the same, it is impossible to come up with programs with same difficulty. It is also agreed by Shneiderman in his evaluation about program quality and comprehension—"Measuring program complexity or quality is a difficult task for which there are no widely accepted techniques. Similarly, the measurement of a subject's comprehension of a given program is equally fraught with difficulties."[Shneiderman 1977]

To offer a test set which share similar properties, we choose three algorithms which are based on basic single-linked list manipulation techniques. They are a linked list traversal algorithm, a linked list reversal algorithm and a cycle-detection algorithm.

5.2.2. Initial questions

As we have mentioned in the previous two sections, the questions we want to verify can be summarized as below:

5. Evaluation

- Are the graphical symbols and animations easy to understand?
- Is the symbols and animations reflecting the execution process of the algorithm?
- Does the execution process show the trace of potential bugs?

5.2.3. Methodology

To verify the first question, we conducted a qualitative evaluation using an online survey. For the first question, we first showed a guide of the symbols and asked the participants whether the symbols were conventional and intuitive. For the second question, we demonstrated the visualization of a linked list reversal algorithm and asked the user what specific program behavior they could find from the execution process. For the last question, we showed the source code of a cycle detection algorithm and a key state of the running process. The test algorithm can verify a linked list has cycle, while it triggers a runtime exception when the linked list has no cycle. Since we need to eliminate the influence which was caused by the name of variables and method signature, we have renamed the variables, so that the participants could not easily find the bug by guessing. Finally, participants can express their attitudes towards our application and leave comments.

5.2.4. Results

We spreaded our online survey via Facebook in the group of students from master program of media informatics in RWTH Aachen University. The survey was open for 72 hours. We have received 16 replies. More than half of the participants have more than 2 years' experience of programming. And all of them are proficient in at least one object-oriented language. In the two methods about the strategy to comprehend a program, participants all agreed that it is easier to comprehend a program by running and debugging it(37.5% agree, 62.5 % strongly agree). However, currently they still comprehend a program by reading the source code and UML diagrams(25% neutral, 37.5% agree, 18.75% strongly agree). After we showed an introduction about the graphical symbols, half the participants agree that our representation is intuitive. It goes the same in the next two questions. More than half of participants think the animation can demonstrate the update of a pointer. Q.8 is based on a linked list reversal algorithm. We showed the animation instead of the source code and asked the participants to infer the intent of algorithm. In the animation, participants were supposed to see that the nodes in a chain were being moved to another chain. In the end, the head of the new list was returned. Only four of the participants have noticed that the last node was returned. And none of them found that the whole list has been reversed. Q.9 is based on a classic linked list's cycle detection algorithm [Floyd 1967]. The question is for testing whether the interface can help participant infer the bug by just reading the source code and see a snapshot of the program state(Fig.??). We renamed the method and variables and showed a snapshot of a running state of the algorithm as well as the source code. However, half of the participants chose *I didn't understand the question or the graph, so I didn't solve it*. Only two of them found bug of the program. These two participants said that they answer the question by combining the source code and the graph in the follow-up question.

Index	Question
Q.3	I understand the source code by reading the source code, UML diagrams statically.
Q.4	By running and debugging a program, I can comprehend the source code better.
Q.5	The graphical symbols are easy to understand.
Q.6	The animation above can represent the program state intuitively.
Q.7	The animation above can represent the program state intuitively.
Q.12	I find the animation is helpful.
Q.13	I find the application easy to understand.
Q.14	I think that I would like to use this application when debugging a linked list.

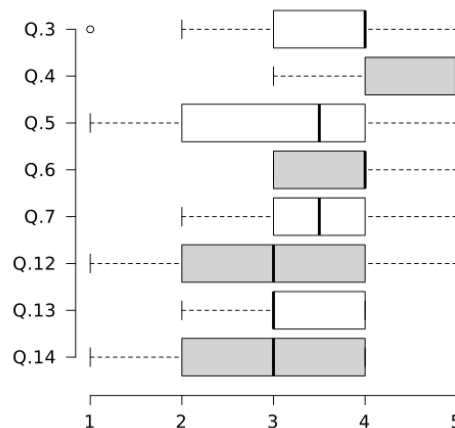


Figure 5.4.: Results of qualitative questions in the online survey

Participants have shown conservative feedback in Q.12 to Q.14. Less than half of them agree that the animation is helpful(Q.12) and the application is intuitive(Q.13). Eventually, they gave good feedback in Q.14. More than half of them will use this tool in the future. In the comment section, one participant expressed that the animation could not meaningfully convey the runtime status and the intent of a program. It could only act as an auxiliary view in a debugging process. One participant criticized the font size. It was too small, while the node size was comparatively big.

5.3. Discussion

In the user evaluation, participants generally liked the concept and consider the visualization and animation intuitive. However, the result shows that a user could not get the *intent* of the program by just watching its animation. It verifies the theory of Naps[Naps et al. 2002], that a good program visualization system should be highly interactive. In the survey, we found that participants were able to get the status of the program, but they would easily misinterpret the intent, which should be inferred from a series of status. We assume that it does not make sense to play the complete process of an algorithm. Instead, if a user can understand a few important

5. Evaluation

graphical frames, he is already getting meaningful information from the visualization.

In the user evaluation part, we did not manage to test the actual performance of the interface. We just managed to evaluate that our visualization is meaningful and intuitive, but we could not widely test the application. Since program comprehension involves complex cognitive process, it is imprecise to just the effectiveness by just using a few controlled experiments. It is necessary to conduct a qualitative evaluation based on a complete implementation. For example, the evaluation of Debugger Canvas[DeLine et al. 2012] has proved its utility as a debugger’s frontend.

Further evaluations should focus on the following aspects:

- *Efficiency of graph interactions.* We assume that our tool is efficient in visualizing recursive data structures. A controlled experiment can be conducted to compare the performance of value inspection using HeapVision and traditional IDE’s variable view.
- *Layout algorithms.* In hierarchical graph layout algorithm, the heuristics for optimization is based on the fact that hierarchical information is embedded in the graph. When recursive data structures are pervasive in a heap graph, it is not appropriate to use this type of algorithm. Therefore, conducting qualitative user evaluation to compare different layout algorithms is a efficient way to access user’s preference.
- *Usability of program traces.* Visualizing the trace of a running algorithm to improve program comprehension has been explored by Stasko and Kehoe[Stasko and Kraemer 1993] [KEHOE et al. 2001]. Trace visualization is built in HeapVision as well, we did not evaluate the usability of the trace view. This evaluation remains as a future study.

5.4. Limitations

Two aspects of limitations were found in the evaluation and in the design process of the evaluation. At first, in the visualization aspect, we could only effectively visualize limited amount of data structures, due to the stack overflow issue in the backend. Our experiments only focused on two types of data structures. In fact, they are just the tip of an iceberg. Many actual algorithms and data structures have not been tested. We have tried to visualize the in-order traversal algorithm in a tree. Nodes were rapidly concretized and abstracted, which confused the user. For the first problem, it is possible to leverage by modifying the heap traversal algorithm. For the second problem, we need to improve the abstraction technique and layout algorithm, which are the fundamental problems in the whole thesis.

6

Conclusion and Future Work

In this thesis, we have provided a solution for interactive debugging by combining pen and touch interaction in human computer interaction and shape analysis technique in program analysis. We delivered a tool, which can visualize the heap memory at runtime. In addition, by adopting the abstraction technique from TVLA[Sagiv et al. 2002] [Lev-Ami and Sagiv 2000], data structures can be abstracted as summaries. An user is able to interact with the visualization by inspecting values and objects' relations. To enable the interactions for graph visualization and manipulation, at first, we have designed a set of gestures which are used for changing the abstraction level as well as changing the view of the graph visualization. Moreover, we extend an existing graph visualization framework to perform interactions and transformations. In addition, we modified and applied an existing graph layout algorithm to give the heap graph an incremental layout. Finally, we have conducted evaluations and proved that, 1. the algorithms are feasible; 2. the abstraction for data structure is conventional; 3. our tool is good at assisting a programming to understand heap manipulation algorithms. We have contributions in three aspects:

- introduced static program analysis technique to help debugging;
- explored pen and touch interactions for graph visualization and manipulation;
- used heap graph to increase the awareness of heap manipulation algorithm.

However, many aspects can be improved. At first, in the backend, the heap traversal algorithm could only query a limited amount of recursive data structure, because of the recursive DFS algorithm. Secondly, user can only perform a fixed amount of gestures. Ideally, through combining different gestures, user can intuitively create new tools for graph visualization and manipulation. Finally, in the evaluation, we have only proved that the visualization and manipulation are suitable for the algorithm which manipulates linked list. Actually, the visualization performs not as expected in the tree data structure. Then it is needless to mention that we need

6. Conclusion and Future Work

to support graph, trie and red-black tree.

6.1. Future work

Program analysis technique helps abstracting the program status in a flexible manner. Currently, only visualizing the heap status using program analysis technique has been explored. We didn't manage to implement some features which allow programmer to change program status directly from the graph. Meanwhile, although we are already using a flexible layout engine to organize the data structure. The layout is still based on some fixed context-independent heuristics. The algorithm can not alter the layout based on meta information of the graph(label, type, etc). Furthermore, the way it maintains stability is by adding a compensation in the linear programming problem, which means that the original positions could still be modified. Finally, a new approach to evaluate the effect of debugger should be provided. A lot of previous works[Shneiderman 1993][Pennington 1987] have mentioned the difficulty of defining the conceptual model of program comprehension as well as measuring the comprehension of program. Without a meaningful model, a debugging system could be evaluated only by users' feedback and controlled experiments.

Based on the findings in this thesis, following directions are the future works.

Run-time program manipulation Edit-compile-test cycle is usually repeated a number of times in a software development process. To accelerate the process, a good approach is to merge these three steps, which either allows programmer to keep track of the current program status in edit mode, or allows programmer change the program in debug mode. The first approach is regarded as *Live Programming*(Section 2.5), which requires various changes in programming language and programming environment. The second approach is regarded as aspect-oriented programming(AOP)[Kiczales et al. 1997], which is initially designed for updating modules in a web server without pausing its execution. It allows modifications of application components without affecting the execution of it. Eclipse supports a similar feature called *Hot code replace(HCR)*[Jones and Thomas 2012], while it is called *Edit and Continue* in Visual Studio[Microsoft 2014]. So, it is technically possible to reduce the gap between edit and debug by allowing programming to directly change the program in runtime.

Based on the existing runtime code manipulation feature, we can extend the current system to allow changing the source code as well as alternating values at runtime.

Value modification. If a *value* has a primitive data type¹, it can be modified directly in variable view. The modification is simultaneously synchronized in the program. When the type is a reference type, programmer needs to either explicitly construct a new instance based the intrinsic constructors of the object, or assign an existing instance of an suitable type. To perform these two modifications in our system, user can change the value of a primitive data type by writing a new value on top of the old value, or assign a new reference by connecting two nodes together. The difficulty lies in how to change values and assign relations based on the constraints

¹We define primitive data type as the primitive data type in Java, namely int, double, float, byte, boolean, char, long.

in the program. For example, in a linked list (Figure 1.1), the next variable is only for connecting a node to the current list node. Alvarado's work [Alvarado and Davis 2004] has shown how domain-specific knowledge could be used for specifying constraints in sketching.

Code modification. The process can be regarded as a procedure of *program synthesis*, in which the source code is synthesized from the pen and touch interaction. For example, in a linked list, connecting node u_1 to node u_2 can be translated to source code $u_1.next = u_2$, while cutting the edge between u_1 and u_2 can be regarded as $u_1.next = null$. Furthermore, it would be more useful to combine different interactions for constructing a new algorithm, like if a user reverse a concrete linked list by hand, a linked list reversal algorithm should be therefore synthesized.

Visualization and interaction by example Although the abstraction can be customized through changing the predicate set, the operations regarding to the gestures could not be customized. It would be more useful that the operations can be parameterized. Moreover, by analysing user's behaviors, the pattern of interaction can be derived. These patterns can be used to associate a user's gesture to operations of heap navigation. On the other hand, the visualization is related to data structures and algorithms as well. It would be beneficial that the system can custom the visualization for different data structures and algorithms.

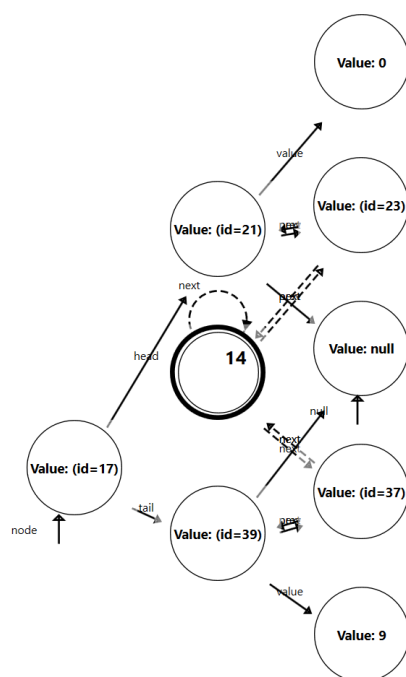


Figure 6.1.: Current visualization of a double linked list

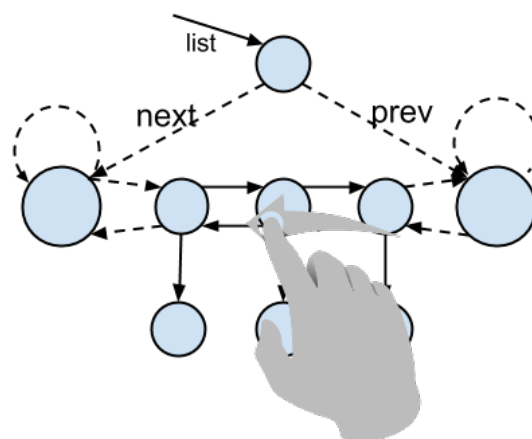


Figure 6.2.: Ideal visualization and interaction of a double linked list

Metadata-driven dynamic hierarchical layout algorithm *Visualization by example* resolves the question of *what to visualize*, while the layout of graph resolves the question of *how to visualize*. For example, all nodes in a double linked list should be aligned vertically or horizontally, since there should not be a hierarchy between nodes which are of the same type, in a double linked list. In Figure 6.2, we show the ideal visualization, while Figure 6.1 represents

6. Conclusion and Future Work

the current layout calculated by DynaDAG. The heuristics of the algorithm could not effectively capture the shape of a double linked list, so that the visualization becomes confusing. A simple method to solve this problem is to add a predicate($is_{DLL}(v)$) to evaluate whether a node is in a double linked list. When it evaluates to true, the certain node should be assigned to a same level as the other nodes. With the use of predicates, metadata of a heap graph(type, label) can be used to determine the layout.

A

Appendix

A.1. Definitions

<i>PrimVal</i>	::=	boolVal longVal intVal shortVal byteVal charVal floatVal doubleVal
<i>ArrayVal</i>	::=	(VarType (×VarType)*)
<i>RefVal</i>	::=	null ObjectVal ArrayVal
<i>Val</i>	::=	PrimVal RefVal this super
<i>LocalVar</i>	::=	Name
<i>FieldVar</i>	::=	Val.VarName
<i>ArrayEntry</i>	::=	AryVal.[intVal]
<i>Var</i>	::=	LocalVar FieldVar ArrayEntry

Table A.1.: Formal definition of variables and values in Java

A. Appendix

$RefType ::= ClassName | InterfaceName | NullType$
 $PrimType ::= boolean | long | int | short | byte | char | float | double$
 $ArrayType ::= (VarType (\times VarType)^*)$
 $Type ::= (RefType | PrimType | ArrayType)$

Table A.2.: Formal definition of types in Java

A.2. Figures

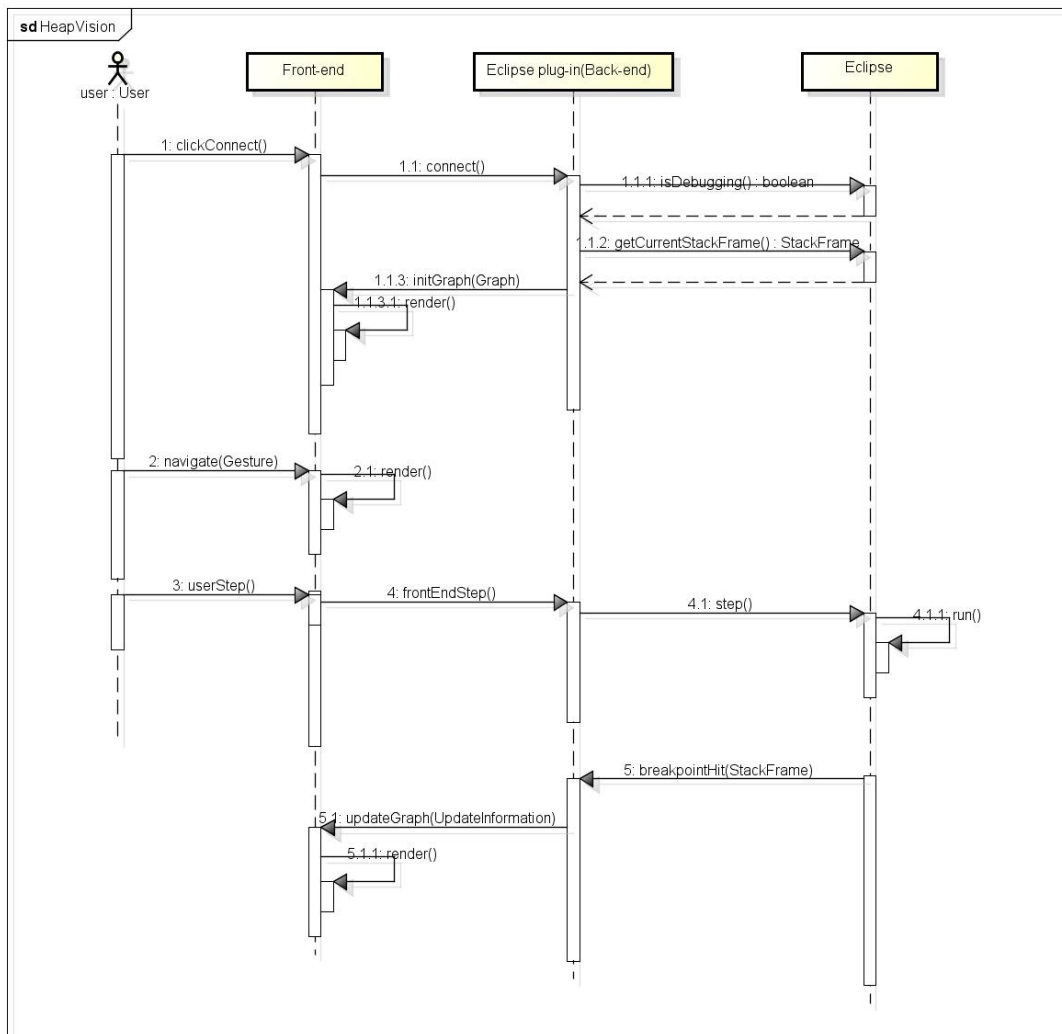


Figure A.1.: Sequence diagram of the whole HeapVision application, including the backend and front-end

A.3. Questionnaire

Survey of graph symbols and animations in HeapVision HeapVision is an interactive debugger for recursive data structures like linked list and tree. It can display structures in a graph and build the pointer manipulations in the program as animations. It is good for programmers, especially novice programmers to understand basic pointer-manipulation code.

Question 1 How many years of programming experience do you have?

- less than 6 months
- 6 months to 2 years
- 2 years to 4 years
- more than 4 years
- none

Question 2 I am proficient in the following programming languages.

- C\C++
- Java
- PHP
- JavaScript
- C#
- Objective-C
- Ruby
- Matlab
- Other : _____

Question 3 I understand the source code by reading the source code, UML diagrams statically.

Strongly disagree					Strongly agree
1	2	3	4	5	
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

A. Appendix

Question 4 By running and debugging a program, I can comprehend the source code better.

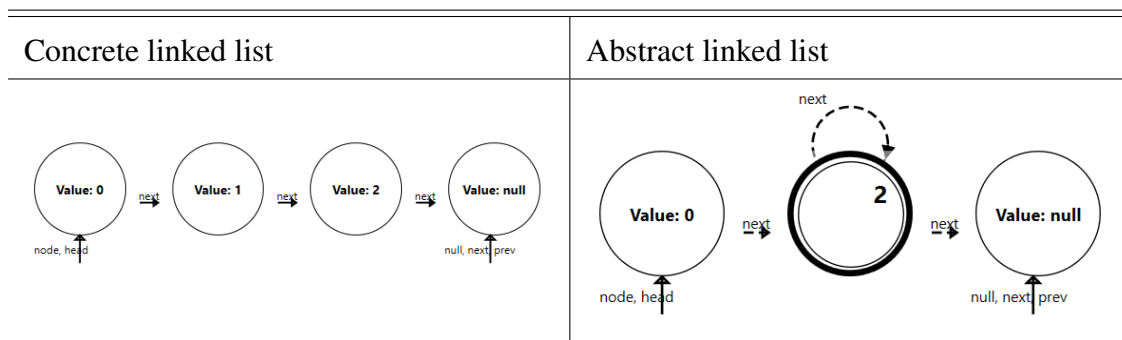
	Strongly disagree				Strongly agree
	1	2	3	4	5
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Introduction of graph symbols and transformation We are going to introduce different symbols, which we use in the visualization. The list node is defined in the following code in Java programming language.

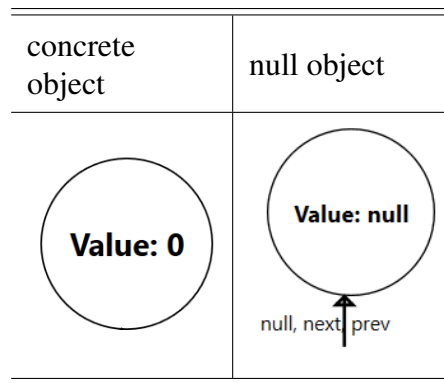
```


1 class ListNode{
2     public ListNode next;
3     public int value;
4 }
    
```

Example A singly-linked list can be represented as either a concrete linked list or an abstract linked list.

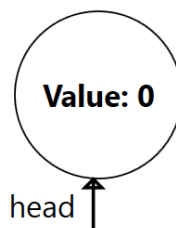


Concrete node A concrete node is an object in Java. Empty(null) is also an object, whose is null. The number in the center of the node indicates its value. In this data structure, the type of value is an integer primitive.

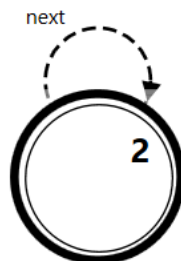


Field variable An edge between two nodes means a field named "next" of object o1 points to object o2. 

Local variable An edge without a source node means a variable in the source code(local variable) is pointing to that object.



Summary node A double circle node indicates it is a summary node, which contains more than one objects. A summary node with a self edge indicates that two or more concrete nodes inside this summary node is connected by this edge. The number in the top right corner indicates how many concrete objects are summarized in the node.



Question 5 The graphical symbols are easy to understand.

A. Appendix

Strongly disagree					Strongly agree
1	2	3	4	5	
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Question 6 Given a method

```
1 ListNode foo(ListNode head)
2 {
3     Node p = head;
4     p = p.next;
5     p = p.next;
6     return p;
7 }
```

We show the running of the program using our interactive debugger. The running process is shown in the video below.

(Video of Question 6¹)

The animation above can represent the program state intuitively.

Strongly disagree					Strongly agree
1	2	3	4	5	
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Question 7 Given a method

```
1 public ListNode search(int value, ListNode head) {
2     ListNode p = head;
3     while (p.next != null && p.value != value)
4         p = p.next;
5     return p;
6 }
```

head is the head of a linked list. value is equal to 6. The running process is shown in the video.

(Video of Question 7²)

The animation above can represent the program state intuitively.

¹Video of Question 6: <https://www.youtube.com/watch?v=LcTAFXhloCc>

²Video of Question 7: <https://www.youtube.com/watch?v=NwE0fCUQISo>

Strongly
disagreeStrongly
agree

1

2

3

4

5

Question 8 Given an anonymous program, and with method signature is `ListNode foo(ListNode head)` and return value is local variable `prev`.

(Video of Question 8¹)

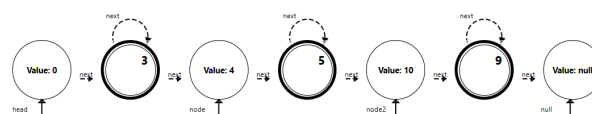
Could you describe or name the algorithm by watch it's running process above? _____

Question 9 Given a code fragment, and a state of the program.

```

1 public boolean foo(ListNode node) {
2     ListNode head = node;
3     ListNode node2 = node;
4     while (node2.next != null) {
5         node2 = node2.next.next;
6         node = node.next;
7
8         if (node == node2)
9             return true;
10    }
11    return false;
12 }

```



Is the program correct? If it is not correct, could you find a possible error? (We assume the minimum length of the linked list is 2.)

- Correct
- Incorrect
- I don't know

¹Video of Question 8: <https://www.youtube.com/watch?v=PkNSWx5SDiw>

A. Appendix

Question 9.1 If you have found a bug in the program, please describe it below.

Question 9.2 How did you solve the previous problem?

- By experience, I know the algorithm in advanced.
- By reading the source code
- By the graph
- By combining the source code and the graph
- I didn't understand the question or the graph, so I did not solve it.
- Other: _____

Question 10 I find the animated transitions helpful.

Strongly disagree					Strongly agree
1	2	3	4	5	
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Question 11 I find the application easy to understand.

Strongly disagree					Strongly agree
1	2	3	4	5	
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Question 12 I think that I would like to use this application when debugging a linked list.

Strongly disagree					Strongly agree
1	2	3	4	5	
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Question 13 Do you have any comments?

A. Appendix

Bibliography

- ABOUZIED, A., HELLERSTEIN, J., AND SILBERSCHATZ, A. 2012. Dataplay: Interactive tweaking and example-driven correction of graphical database queries. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, ACM, New York, NY, USA, UIST '12, 207–218.
- AFTANDILIAN, E. E., KELLEY, S., GRAMAZIO, C., RICCI, N., SU, S. L., AND GUYER, S. Z. 2010. Heapviz: interactive heap visualization for program understanding and debugging. In *Proc.SOFTVIS '10*, p.53–62.
- ALVARADO, C., AND DAVIS, R. 2004. Sketchread: a multi-domain sketch recognition engine. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, ACM, 23–32.
- ATWOOD, J., AND SPOLSKY, J., 2008. Stackoverflow, Aug.
- BEDERSON, B. B., AND BOLTMAN, A. 1999. Does animation help users build mental maps of spatial information? In *Information Visualization, 1999.(Info Vis' 99) Proceedings. 1999 IEEE Symposium on*, IEEE, 28–35.
- BOGDAN, J. L., CHEW, C. H., GUZAK, C. J., AND PITT III, G. H., 1999. Tree view control, Nov. 2. US Patent 5,977,971.
- BOSTOCK, M., OGIEVETSKY, V., AND HEER, J. 2011. D³ data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on* 17, 12, 2301–2309.
- BRANDL, P., FORLINES, C., WIGDOR, D., HALLER, M., AND SHEN, C. 2008. Combining and measuring the benefits of bimanual pen and direct-touch interaction on horizontal interfaces. In *AVI'08: Proceedings of the working conference on Advanced Visual Interfaces*, ACM, New York, NY, USA, 154–161.
- BRANKE, J. 2001. *Dynamic graph drawing*, vol. 2025. Springer.
- BREWSTER, S. 2002. Overcoming the lack of screen space on mobile computers. *Personal and Ubiquitous Computing* 6, 3, 188–205.
- BURCKHARDT, S., FAHNDRICH, M., DE HALLEUX, P., MCDIRIMID, S., MOSKAL, M., TILLMANN, N., AND KATO, J. 2013. It's alive! continuous feedback in ui programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, New York, NY, USA, PLDI '13, 95–104.
- BURNETT, M., ATWOOD, J.W., J., AND WELCH, Z. 1998. Implementing level 4 liveness in declarative visual programming languages. In *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*, 126–133.
- CARD, S. K., ENGLISH, W. K., AND BURR, B. J. 1978. Evaluation of mouse, rate-controlled isometric joystick, step keys, and text keys for text selection on a crt. *Ergonomics* 21, 8, 601–613.
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., STEIN, C., ET AL. 2001. *Introduction to algorithms*, vol. 2. MIT press Cambridge.

BIBLIOGRAPHY

- CORNELISSEN, B., ZAIDMAN, A., VAN DEURSEN, A., MOONEN, L., AND KOSCHKE, R. 2009. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on* 35, 5, 684–702.
- COUSOT, P., AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ACM, New York, NY, USA, POPL '77, 238–252.
- DELINE, R., BRAGDON, A., ROWAN, K., JACOBSEN, J., AND REISS, S. P. 2012. Debugger canvas: industrial experience with the code bubbles paradigm. In *Proc.ICSE 2012*, p.1064–1073.
- DIEHL, S., AND GÖRG, C. 2002. Graphs, they are changing. In *Graph drawing*, Springer, 23–31.
- DIETZ, P., AND LEIGH, D. 2001. Diamondtouch: a multi-user touch technology. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, ACM, 219–226.
- DWYER, T., MARRIOTT, K., AND STUCKEY, P. J. 2006. Fast node overlap removal. In *Graph Drawing*, Springer, 153–164.
- DWYER, T., LEE, B., FISHER, D., QUINN, K. I., ISENBERG, P., ROBERTSON, G., AND NORTH, C. 2009. A comparison of user-generated and automatic graph layouts. *Visualization and Computer Graphics, IEEE Transactions on* 15, 6, 961–968.
- ELLSON, J., GANSNER, E. R., KOUTSOFIOS, E., NORTH, S. C., AND WOODHULL, G. 2004. Graphviz and dynagraph—static and dynamic graph drawing tools. In *Graph drawing software*. Springer, 127–148.
- FLOYD, R. W. 1967. Nondeterministic algorithms. *J. ACM* 14, 4 (Oct.), 636–644.
- FORLINES, C., WIGDOR, D., SHEN, C., AND BALAKRISHNAN, R. 2007. Direct-touch vs. mouse input for tabletop displays. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, New York, NY, USA, CHI '07, 647–656.
- GUGERTY, L., AND OLSON, G. 1986. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, New York, NY, USA, CHI '86, 171–174.
- GUO, P. J. 2013. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, ACM, 579–584.
- HALLEUX, J. 2012. Quickgraph, graph data structures and algorithms for. net. *Last access: Mar*.
- HAN, J. Y. 2005. Low-cost multi-touch sensing through frustrated total internal reflection. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, ACM, 115–118.
- HEER, J., CARD, S. K., AND LANDAY, J. A. 2005. prefuse: a toolkit for interactive in-

- formation visualization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, p.421–430.
- HINCKLEY, K., YATANI, K., PAHUD, M., CODDINGTON, N., RODENHOUSE, J., WILSON, A., BENKO, H., AND BUXTON, B. 2010. Pen + touch = new tools. In *Proc. UIST '10*, p.27–36.
- HINCKLEY, K., BI, X., PAHUD, M., AND BUXTON, B. 2012. Informal information gathering techniques for active reading. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, ACM, 1893–1896.
- JONES, G., AND THOMAS, J., 2012. Runtime code replacement, Aug. US Patent App. 13/357,301.
- KEHOE, C., STASKO, J., AND TAYLOR, A. 2001. Rethinking the evaluation of algorithm animations as learning aids: an observational study. *International Journal of Human-Computer Studies* 54, 2, 265 – 284.
- KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. *Aspect-oriented programming*. Springer.
- KLEENE, S. C. 1952. *Introduction to Metamathematics*. North Holland.
- LEV-AMI, T., AND SAGIV, M. 2000. Tvla: A system for implementing static analyses. In *Static Analysis*. Springer, 280–301.
- LIEBERMAN, H., AND FRY, C. 1995. Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press/Addison-Wesley Publishing Co., 480–486.
- LOZANO-PÉREZ, T., AND WESLEY, M. A. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM* 22, 10 (Oct.), 560–570.
- MALONEY, J. H., AND SMITH, R. B. 1995. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, ACM, New York, NY, USA, UIST '95, 21–28.
- MARRON, M., SANCHEZ, C., SU, Z., AND FAHNDRICH, M. 2013. Abstracting runtime heaps for program understanding. *Software Engineering, IEEE Transactions on* 39, 6, p.774–786.
- MCDIRMIID, S. 2007. Living it up with a live programming language. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, ACM, New York, NY, USA, OOPSLA '07, 623–638.
- MCDIRMIID, S. 2013. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ACM, New York, NY, USA, Onward! '13, 53–62.
- MICROSOFT, 2014.
- MILLER, R. B. 1968. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, ACM, New York, NY, USA, AFIPS '68 (Fall, part I), 267–277.

BIBLIOGRAPHY

- MISUE, K., EADES, P., LAI, W., AND SUGIYAMA, K. 1995. Layout adjustment and the mental map. *Journal of visual languages and computing* 6, 2, 183–210.
- MURPHY, G. C., KERSTEN, M., AND FINDLATER, L. 2006. How are java software developers using the eclipse ide? *Software, IEEE* 23, 4, 76–83.
- MYERS, B. A. 1990. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing* 1, 1, 97–123.
- NAPS, T. L., RÖSSLING, G., ALMSTRUM, V., DANN, W., FLEISCHER, R., HUNDHAUSEN, C., KORHONEN, A., MALMI, L., MCNALLY, M., RODGER, S., ET AL. 2002. Exploring the role of visualization and engagement in computer science education. In *ACM SIGCSE Bulletin*, vol. 35, ACM, 131–152.
- NORTH, C., DWYER, T., LEE, B., FISHER, D., ISENBERG, P., ROBERTSON, G., AND INKPEN, K. 2009. Understanding multi-touch manipulation for surface computing. In *Human-Computer Interaction – INTERACT 2009*, T. Gross, J. Gulliksen, P. Kotz, L. Oestreicher, P. Palanque, R. Prates, and M. Winckler, Eds., vol. 5727 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 236–249.
- NORTH, C., DWYER, T., LEE, B., FISHER, D., ISENBERG, P., ROBERTSON, G., AND INKPEN, K. 2009. Understanding multi-touch manipulation for surface computing. In *Human-Computer Interaction – INTERACT 2009*, T. Gross, J. Gulliksen, P. Kotz, L. Oestreicher, P. Palanque, R. Prates, and M. Winckler, Eds., vol. 5727 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 236–249.
- OECHSLE, R., AND SCHMITT, T. 2002. Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (jdi). In *Software Visualization*, S. Diehl, Ed., vol. 2269 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 176–190.
- O'MADADHAIN, J., FISHER, D., SMYTH, P., WHITE, S., AND BOEY, Y.-B. 2005. Analysis and visualization of network data using jung. *Journal of Statistical Software* 10, 2, 1–35.
- PENNINGTON, N. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3, 295 – 341.
- PRESTON-WERNER, T., WANSTRATH, C., AND HYETT, P., 2008. Github.
- SAGIV, M., REPS, T., AND WILHELM, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (May), p.217–298.
- SHNEIDERMAN, B. 1977. Measuring computer program quality and comprehension. *International Journal of Man-Machine Studies* 9, 4, 465 – 478.
- SHNEIDERMAN, B. 1993. 4, 3 touchscreens now offer compelling uses. *Sparks of innovation in human-computer interaction*, 187.
- SOLOWAY, E., AND EHRLICH, K. 1984. Empirical studies of programming knowledge. *Software Engineering, IEEE Transactions on*, 5, 595–609.
- SORENSEN, A., AND GARDNER, H. 2010. Programming with time: Cyber-physical programming with impromptu. In *Proceedings of the ACM International Conference on Object*

- Oriented Programming Systems Languages and Applications*, ACM, New York, NY, USA, OOPSLA '10, 822–834.
- SORVA, J., KARAVIRTA, V., AND MALMI, L. 2013. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education (TOCE)* 13, 4, 15.
- STASKO, J. T., AND KRAEMER, E. 1993. A methodology for building application-specific visualizations of parallel programs. *Journal of parallel and distributed computing* 18, 2, 258–264.
- STASKO, J., BADRE, A., AND LEWIS, C. 1993. Do algorithm animations assist learning?: An empirical study and analysis. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, ACM, New York, NY, USA, CHI '93, 61–66.
- TANIMOTO, S. L. 1990. Viva: A visual language for image processing. *J. Vis. Lang. Comput.* 1, 2 (June), 127–139.
- VICTOR, B. 2012. Inventing on principle. In *Invited Talk at Canadian University Software Engineering Conference (CUSEC)*.
- VLAMING, L., COLLINS, C., HANCOCK, M., NACENTA, M., ISENBERG, T., AND CARPENDALE, S. 2010. Integrating 2d mouse emulation with 3d manipulation for visualizations on a multi-touch table. In *ACM International Conference on Interactive Tabletops and Surfaces*, ACM, 221–230.
- WALNY, J., LEE, B., JOHNS, P., RICHE, N. H., AND CARPENDALE, S. 2012. Understanding pen and touch interaction for data exploration on interactive whiteboards. *Visualization and Computer Graphics, IEEE Transactions on* 18, 12, 2779–2788.
- ZELLER, A., AND LÜTKEHAUS, D. 1996. Ddd—a free graphical front-end for unix debuggers. *SIGPLAN Not.* 31, 1 (Jan.), 22–27.
- ZIMMERMANN, T., AND ZELLER, A. 2002. Visualizing memory graphs. In *Revised Lectures on Software Visualization, International Seminar*, Springer-Verlag, London, UK, UK, p.191–204.