Chapter 1 Introduction

1.1. An Overview

Software has become more complex and larger over the years. With the increasing complexity of software, developers have encountered lots of difficulties during development. In order to overcome these problems, several approaches have been developed in software engineering. These approaches include software analysis, software verification, software visualization, and etc.

The goal of software engineering is transferring the knowledge of software development into an engineering discipline, like civil engineering or mechanical engineering, rather than building software depending on personal skills or talent of developers. One of the approaches to achieve this goal is that requesting developers to use software engineering tools which can fill the gap between excellent and poor developers as much as possible. Good software engineering tools can analyze software, reduce the development time, and manifest the hidden problems effectively.

In general, integrated development environment (IDE) is the most frequently used software engineering tool. It often provides a convenient environment which integrates tools like compiler, linker, editor, debugger, and simple project manager. An IDE can assist in software development in an effective manner. For example, when users input a

string, the editor will show all the function names and parameters contain the string to save time from searching the library.

Although there are lots of advantages in a well-developed IDE, there hasn't been much progress in software quality. Software to people is still notorious for its poor reliability and quality. Writing a program is still a mostly done by typing statements in an editor. In other words, the progress in IDE merely provides conveniences.

A program consists of codes which are texts conforming to the syntax of a programming language. The descriptions of words could be powerful in some ways. Famous master piece, poems, or novels are all written in words. Some textual contents are sometimes difficult to present in movies. People can be touched by texts in a profound way.

On the other hand, the drawbacks of texts or words are obvious: *First*, readers must be literate to comprehend the meanings expressed by authors. To some obscure materials, it is apparently that only a few people have the capability to understand. *Second*, the speed of reading can be very slow. For example, it may take one or a half day without resting to read a novel that has hundreds of pages. To make it worse, if there is a complex theory included in the content, it will take more time to absorb because one has to read iteratively to understand. *Third*, texts are powerful, but they are sometimes ambiguous and easily to be misunderstood.

Unfortunately, program codes are written in the form of text. A program written by one programmer can be very difficult to be understood by another programmer. Furthermore, comprehending and reading programs is a very slow process, most of the time much more difficult than reading a novel. By the contrast described above, we believe the progress of software can not be made in an order-of-magnitude, as said in Brooks' book [1]. These fundamental difficulties of software engineering are sometimes referred to as "software crisis."

We believe once programs exist in the form of texts, the problems of software engineering are unavoidable. For example, it is a well-known problem in software maintenance that programmers have the difficulties to comprehend the codes written by others. To avoid these problems and give assistance for code reading, the importance of using additional documents, such as design documents, system architecture documents, and diagrams, has been stressed in software engineering. For example, the objective of advocating UML (Unified Modeling Language) is mainly to address the issues discussed here. However, when a programmer needs to understand the detail of the codes, texts are the ultimate medium – the codes themselves. In other words, the readability and expressiveness of codes are very poor. To address these problems, we need to fully understand the essence of software.

F. P. Brooks. Jr. compared software with werewolves in his famous paper "No

Silver Bullet: Essence and Accidents of Software Engineering [1].” Werewolves transform unexpectedly from the familiar into horrors and can only be hurt by bullets of silver. Software has something of this character that suddenly becomes a monster of missed schedules, blown budgets, and flawed products. However, Brooks believed that there is no silver bullet that can make software costs drop as rapidly as computer hardware costs do in a decade from that time because of the essential difficulties of software. He divided the tasks of software development into “*essence*, the difficulties inherent in the nature of software, and *accidents*, those difficulties that today attend its production but are not inherent [1].”

Brooks summarized the inherent properties of this irreducible essence of modern software systems as follows:

- *Complexity*: “Software entities are more complex for their size than perhaps any other human construct because no two parts are alike (at least above the statement level) [1].” Many of the classic problems of developing software products derive from this essential complexity and its nonlinear increases with size. From the complexity come not only *technical* problems, such as less usability, defects of products, unreliability, but also *management* problems, such as conceptual integrity, and personal turnover.
- *Conformity*: The software is often forced to conform to lots of human institutions

and systems without reasons. And in many of cases, much complexity of software comes from conformation to other interfaces, which can not be simplified out by redesign of the software alone. A slight move in one part may affect the whole situation.

- *Changeability*: “All successful software gets changed [1].” There are mainly two reasons. *First*, People tend to use a software product in new domain beyond the original design of it. *Second*, a success software product has longer life cycle than that of the original machine which it written for.
- *Invisibility*: “Software is invisible and unvisualizable. Geometric abstractions are powerful tools [1].” However, software has no geometric representation like that land has maps; silicon chips have diagrams, and computers have connectivity schematics. Once using diagrams such as directed graphs to show the software structure, they will have low usability while a huge amount of graphs are used. People are difficult to comprehend the relationship of these graphs.

In the past, there have been three breakthroughs in software technology development: *high-level languages*, *time-sharing*, and *unified programming environment*. However, each of them attacked a different major difficulty in building software, but these difficulties are accidental, not essential, difficulties. In Brooks’ point of view, only using bullets of silver - overcoming these problems of essential

difficulties in software, can make the software development have one or more order-of-magnitude improvement in productivity, reliability, and simplicity. In our research, we try to solve the essential difficulty about invisibility mentioned above. We will have more discussion in next section.

1.2. Objective

With the increasing complexity of software, developing bug-free software is much more difficult than before. Debugger is one of the solutions which can help developers to find bugs in software. While debugging a program with the debugger, developers have to repeat steps such as setting breakpoints, running the program to hit the breakpoint, and watch the value of variables until the cause of defect is found. Most debuggers in nowadays can solve the problems during debugging, however, what we concern most is that whether the debugger is convenient to use and whether it can locate the bugs among software effectively and accurately. An experienced developer must have encountered one of the scenarios as follows:

- To find the bugs among software, a developer often need to set breakpoints according to his experiences, and then observe the state of the program that hits the breakpoint. But the values of variables are refreshed again and again because of the “continuous” steps. Developers often need to reboot the program to observe the variation values with time due to leaving the previous states behind.

- While debugging a program which has complicated data structures, a developer often needs to observe the content of some variables from the debugger, and then construct the data structure in mind. When more steps are traced and the variations of data structure are changed along the trace, the debugging information may become very difficult to comprehend.
- Besides reading static codes, experienced developers often use debuggers as an important toolkit to understand a program. Most debuggers only show the variation of variables rather than high-level relationships between data structures. They can not provide sufficient information effectively and rapidly for program understanding.

From the examples above, we can learn that there is still room for improvement for debuggers and the process of debugging.

Traditional debuggers only provide textual debugging information for developers. Developers are difficult to get the whole picture of complex data structures. There have been some systems developed to visualize debugging information in 2D (see section 2.1.3). However, invisibility, which is one of the essential difficulties of software engineering, makes diagrams in 2D no sufficient room for a comfortable rending for program comprehension. Furthermore, the contents required to be visualized may significantly increase as the programs scale up, which make these systems hard to be

applied in real scenarios. In other words, 2D visualization has low usability. Our research is trying to overcome some of the problems and the limitations in 2D by building a 3D debugging visualization tool called DIVINE (**D**ebugging **I**nformation **VI**sualizatio**N** in 3D **E**nvironment).

In the past, DIVINE has been implemented and provided a visualization of linked-list structure in 3D environment and interactions to manipulate the elements in the scene [2]. DIVINE can also communicates with HILCADT (**HI**gh-**L**evel **C**omputer-**A**ided **D**ebugging **T**ool) [3], a java debugger that follows the JPDA (Java Platform Debugging Architecture). HILCADT provides a high-level debugging language that is more expressive and flexible than the primitive of debuggers. DIVINE can send requests to HICADT to get the states of a program that paused by a breakpoint. Then, DIVINE visualizes the debugging information received from HILCADT that help users to comprehend the structure of the program [4].

After visualizing the debugging information in 3D environment, we often require some interaction with visualized elements which are called "visualization metaphor (VM)". For example, a user may want to expand some or all the children nodes of one leaf node among the tree structure by picking it. Non-traditional devices developed in virtual environment (VR), i.e., head-mounted displays, data gloves, and trackers, provide more intuitive and effective interaction than traditional desktop input devices

used in nowadays, i.e., keyboards and mice.

In order to use various input devices to interact with VM transparently, we try to provide a device-independent framework which makes users can interact with VM without complicating the programming task of visualized objects. Separating the visualization programming from user interface programming also reduce the burden of programming task in designing visualization objects. VM programmers can concentrate on the design of visualization objects without the limitation of interaction devices.

Metaphor is a way to present a set of data. For example, we are familiar with pie chart or bar chart that provides different comprehension of the same statistical information. There is no single visualization metaphor that suits every data structure due to the essential complexity and the invisibility of software. So DIVINE has planed to use multi-visualization metaphors that provide various aspects for comprehending the data structures, even composite data structures. In other words, debugging information is absolutely abstract, and users can capture the information by choosing suitable visualization metaphor presenting the debugging information in DIVINE. This makes DIVINE have not only high-level visualization in debugging but also adaptability for different kinds of data structures.

The objective of the thesis is not only to provide a device-independent 3D user

interface which can make users interact with 3D visualization metaphors in DIVINE by using either desktop input devices or VR input devices, but also making VM programmers concentrate on the design of visualization objects without the limitation of interaction devices.

1.3. Thesis Organization

The remainder of the thesis is structured as follows: next chapter gives an overview of visualization, including software visualization and visualized debuggers. In chapter 3, we introduce the architecture of DIVINE and the device-independent 3D user interface in DIVINE system. And some implementation issues are described in chapter 4. Finally, chapter 5 summarizes the conclusions and future works.