An Examination of the Run-time Performance of GUI Creation Frameworks

Christopher J. Howell, Gregory M. Kapfhammer, Robert S. Roos
Department of Computer Science
Allegheny College
Meadville, PA 16335
{howellc,gkapfham,rroos}@allegheny.edu

ABSTRACT

The graphical user interface (GUI) is an important component of many software systems. Past surveys indicate that the development of a GUI is a significant undertaking and that the GUI's source code often comprises a substantial portion of the program's overall source base. Graphical user interface creation frameworks for popular object-oriented programming languages enable the rapid construction of simple and complex GUIs. In this paper, we examine the run-time performance of two GUI creation frameworks, Swing and Thinlet, that are tailored for the Java programming language. Using a simple model of a Java GUI, we formally define the difficulty of a GUI manipulation event. After implementing a case study application, we conducted experiments to measure the event handling latency for GUI manipulation events of varying difficulties. During our investigation of the run-time performance of the Swing and Thinlet GUI creation frameworks, we also measured the CPU and memory consumption of our candidate application during the selected GUI manipulation events. Our experimental results indicate that Thinlet often outperformed Swing in terms of both event handling latency and memory consumption. However, Swing appears to be better suited, in terms of event handling latency and CPU consumption, for the construction of GUIs that require manipulations of high difficulty levels.

1. INTRODUCTION

The performance of interactive systems is a significant part of a user's perception of overall system performance. For an application to be successful, a user must possess the ability to interact with the application easily. A graphical user interface (GUI) is one mechanism to allow easy interaction between the user and the application. While past estimates indicated that an average of 48% of an application's source code was devoted to the interface [18], current reports reveal that the GUI represents 60% of the overall source of a program [13]. Graphical user interface toolkits (or, alternatively, GUI creation frameworks) facilitate the construction of interfaces by providing a set of graphical widgets and a mechanism for combining these widgets into a complete GUI [17, 18].

GUI creation frameworks do not solve all of the problems that are associated with the construction of software systems that have interfaces. Indeed, additional techniques have been proposed and implemented to address the issues of GUI correctness [14, 16], GUI usability and widget layout [19], interactive system performance [5], and the analysis of the correctness and performance of the underlying application [8, 21]. Even though the performance of interactive systems is clearly important [7, 20], relatively little research has specifically focused on the measurement and comparison of the run-time performance of GUI creation frameworks. Since a GUI is an important component of most software applications and a GUI toolkit is often used to construct the interface, it is clearly important to develop an understanding of the performance of current GUI creation frameworks.

According to Horgan et al. [9], the analysis of Java programs can be broken into at least four levels, namely (1) statically, at the source code level; (2) statically, at the bytecode level; (3) dynamically, at the bytecode level; and (4) dynamically, on a specific virtual machine and architecture. In this paper, we focus on the fourth level by analyzing the performance of GUI creation frameworks on a specific Java Virtual Machine (JVM) and selected computer architectures. Using a case-study application, we experiment with two frameworks from the perspective of the user by comparing the event handling latency and CPU and memory consumption for events of increasing levels of difficulty. The contributions of this paper are:

1. The definition of the difficulty of a GUI manipulation event.
2. A detailed empirical analysis and comparison of the run-time performance of two Java GUI creation frameworks, Swing [12] and Thinlet [1].
3. Recommendations concerning the selection of a suitable GUI creation framework for classes of interactive applications.

The remainder of this paper is organized in the following manner. In Section 2 we use a simple definition of a graphical user interface to define the difficulty of a GUI manipulation event and review the concept of event handling

2nd International Conference on the Principles and Practice of Programming in Java 2003 Kilkenny City, Ireland

latency. Section 3 provides an overview of Java GUI frameworks and examines the two frameworks that we compare in this paper. Section 4 describes the case study application and states why it facilitates the investigation of the run-time performance of GUI creation frameworks. In Sections 5 and 6, we describe the experimental design and the tools used for measuring GUI event handling latency, CPU usage, and memory consumption. In Section 7, we analyze the data resulting from our experimentation with the Swing and Thinlet frameworks. Section 8 reviews related work. Finally, in Section 9 we summarize the performance of the selected GUI frameworks and make recommendations concerning the selection of a GUI creation framework for different classes of interactive applications.

2. GRAPHICAL USER INTERFACES

It is often difficult to define the term *user interface*. In this paper, we adhere to the intuitive definition provided by Myers and Rosson [18]:

> [...] we intend it to mean the software component of an application that *translates a user action into one or more requests for application functionality*, and that provides to the user *feedback about the consequences of his or her action*. This software component (or components) would be distinguished from the underlying computation that goes on in support of the application functionality.

For the purposes of this paper, we define a graphical user interface G as a set of widgets, so that $G = \{W_1, W_2, \ldots, W_n\}$. Furthermore, we use the function $S_C(W_i)$ to compute the current state of widget W_i and we have $S_C(W_i) = \{w_1, w_2, \ldots, w_m\}$. Thus, we see that the state of graphical user interface G is defined as $S(G) = \bigcup_{i=1}^{n} S_C(W_i)$. While interface toolkits often include the ability to describe the layout of the widgets in the interface, our definition of a GUI ignores layout and constraint issues because they do not directly impact the investigation of the performance of GUI creation frameworks.

There are a variety of definitions associated with the complexity and usability of graphical user interfaces. For example, Jeffries et al. examine four techniques for evaluating user interfaces through usability testing and guidelines [10], Comber et al. discuss information theoretic measures of interface complexity [3], and Sears proposes the notion of layout appropriateness by measuring the cost associated with the manipulation of the widgets within a GUI [19]. Since this paper focuses on the run-time performance of GUI creation frameworks, we are not concerned with the usability or complexity of a specific interface. Instead, we focus on the measurement of the event handling latency of GUI manipulation events of varying levels of difficulty.

According to Endo et al., the throughput of a system cannot adequately characterize the performance of interactive systems and they use *event handling latency*, or the time it takes for a system to respond to a specific event, as a more appropriate measure of performance [5]. In this paper, we are specifically interested in the event handling latency of GUI manipulation events. An example of a GUI manipulation event might be the clicking of a button that executes a structured query language (SQL) statement and then subsequently displays the resulting tuples in the interface. However, some GUI manipulation events might place a greater demand upon the GUI toolkit code than others.

Endo et al. observe that "measuring latency for an arbitrary task and an arbitrary application remains a difficult problem" [5]. During the measurement of the event handling latency of GUI creation frameworks it is important to distinguish between the latency that is associated with the computation in the underlying application and the actual latency resulting from the GUI manipulation code itself. We use $L(E)$ to denote the event handling latency for event E and we require that $L(E) = L_A(E) + L_G(E)$ with $L_A(E)$ and $L_G(E)$ denoting the latency attributed to the underlying application and the latency associated with the code for graphical user interface G, respectively. Any methodology for measuring the event handling latency of an event E must be able to measure the impact that $L_A(E)$ has on the overall latency $L(E)$. Our approach for handling this issue is discussed in Sections 6 and 7.

Since the measurement of $L(E)$ requires the execution of the application under analysis, we desire a static metric that is correlated with the latency that a GUI-based application will exhibit when it handles GUI manipulation event E. To this end, we define the worst-case *difficulty* of a GUI manipulation event in Equation (1). We use $D_A(E)$ to denote the difficulty of the computation associated with the underlying application and we use $D_G(E)$ to represent the difficulty that is directly associated with the manipulation of the graphical user interface. A formulation of $D_A(E)$ would require an analysis of the algorithms used in a candidate application and the Java virtual machine that executes a program. Since the focus of this paper is an empirical investigation of the run-time performance of GUI creation frameworks, we omit an analytical characterization of $D_A(E)$ and we intuitively describe this function for a selected candidate application in Section 7.

Equation (2) defines the worst-case difficulty of the manipulation of the GUI in terms of the number of widgets that must be updated and added to graphical user interface G. In Equation (2), we use $Update(E)$ to compute the set of widget indices that are updated by GUI manipulation event E. An update event E might correspond to the changing of the state of a checkbox widget from "checked" to "unchecked" or the addition of a new row to a textarea widget. Furthermore, we use $Add(E)$ to denote the set of the new widgets that are going to be added to G and we use $S_N(W_i)$ to compute the initial state of the newly added widget W_i. A GUI manipulation event might add a new textarea to G and then populate this textarea with several initial rows of text. It is important to note that $D_G(E)$ computes a worst-case measurement of the difficulty associated with GUI updates and additions because it assumes that the event will update every $w_j \in S_C(W_i)$ for each $i \in Update(E)$.

$$D(E) = D_A(E) + D_G(E) \tag{1}$$

$$D_G(E) = \sum_{i \in Update(E)} \sum_{j=1}^{S_C(W_i)} j \; + \sum_{i \in Add(E)} \sum_{j=1}^{S_N(W_i)} j \tag{2}$$

3. JAVA GUI FRAMEWORKS

The Java language has given rise to many frameworks that enable GUI development. These range from simple classes that can be imported from the Abstract Window Toolkit

(AWT) to full toolkits that are created without the use of the AWT. Some of the most commonly used GUI toolkits for Java are the Eclipse Standard Widget Toolkit (SWT), Swing, and Thinlet. Eclipse and Thinlet are both open-source projects that allow the user to extend the framework, if needed. In this paper, we evaluate and compare Swing from Sun Microsystem's Java Development Kit (JDK) and the fifth beta release of Thinlet.

3.1 Java Swing

The Java Swing framework was developed as an extension to the AWT which Sun developed for use beginning with JDK 1.1. Swing is an application program interface (API) of approximately 50 components that allows the developer to create a more interactive interface. According to Eckstein et al., the lightweight Swing components allow for a more efficient use of resources [12]. Furthermore, since Swing is written entirely in Java, cross-platform use is very consistent and the look and feel across the entire application is the same [12]. Recently, concerns have been raised about the inherent abstraction level and object creation patterns associated with the Swing toolkit and the impact that these facets of Swing might have on the run-time performance of applications that use the framework [22].

3.2 Thinlet

The Thinlet framework was developed by Robert Bajzat and the first version was released on June 9th, 2002. Thinlet takes a description of a GUI in the form of an eXtensible Markup Language (XML) file and parses it using the Thinlet API. The parsed file is then rendered on the screen as the interface of the application. To parse a file in Thinlet, the method `parse(<file>.xml)` is called, followed by `add(<parsedFileObject>)`, as shown in Figure 1.

```
1 public void outputfile {
2     Object file;
3     try {
4         file = parse("file.xml");
5         add(file);
6     }
7     catch(Exception e)
8         e.printStacktrace();
9 }
```

Figure 1: Java Class to Construct a Thinlet GUI.

Figure 2 contains the XML code for the interface. Currently, there are 22 widgets provided by the Thinlet toolkit. Bajzat is in the process of developing more interactive components that can assist in the development of GUI applications.

3.3 High Level Comparison

The Swing toolkit currently contains more components than the Thinlet toolkit. In Swing, a dialog box is easily created with a simple line of code, whereas in Thinlet the developer must create an entirely new XML file and the Java code must parse the file to display the contents to the screen. In Swing, the components are not centered, while in Thinlet all GUI components are automatically centered.

Since one of the motivations for the development of Thinlet was to create a GUI toolkit that had a minimal memory footprint (our measurements indicate that the `jar` compression tool produces a Thinlet archive whose size is approximately 44 KB), it is expected that a Thinlet-based application will consume less memory than a Swing-based application. However, a high level comparison of the two GUI creation frameworks does not reveal any concrete information about the CPU usage patterns and event handling latencies associated with each toolkit. Section 7 uses our empirical analysis to provide a more detailed comparison of these frameworks.

```
1 <dialog columns="1" height="75" width="250">
2     <panel columns="1" gap="3">
3         <label text="This is from the file"/>
4         <button action"Okay_clicked" text="Okay"/>
5     </panel>
6 </dialog>
```

Figure 2: XML to Describe a Thinlet GUI.

4. CASE STUDY APPLICATION: VISUAL DATABASE QUERYING TOOL

Our experiments rely upon a visual database querying application that allows the user to interactively create queries by selecting the tables, attributes, and comparison operators through a graphical interface. The program uses the information that the user selects to build a query and display the results in the form of a table. We created the application to retrieve information from a PostgreSQL database [6], although the particular choice of database management system is irrelevant to the experiments.

Our choice of a database querying application enables us to control the difficulty of a GUI manipulation event by changing the tuples stored within our relational database. For example, the candidate application can be used to construct a SQL query and the data within the chosen relation(s) can be modified to control $D_G(E)$, the difficulty of the GUI manipulation event directly associated with the update of existing widgets and the addition of new widgets. If the SQL query returns a significant number of tuples, this will cause the textarea widget, say W_i, to be associated with a $S_N(W_i)$ of a high cardinality. If a GUI manipulation event E corresponds to the clicking on a button, the submission of this SQL query to the database, and the display of the final results, a large $S_N(W_i)$ will increase $D(E)$, the overall difficulty of this event. An experiment that appropriately characterizes the $D_A(E)$ and measures the $L_A(E)$ associated with an event E can determine the impact that a change in $D(E)$ has on $L(E)$ for different GUI creation frameworks.

5. EXPERIMENTAL DESIGN

Three tools were used to analyze the GUI frameworks. The first was the standard resource monitoring tool, `top`. The second was Microsoft Performance Monitoring version 4.0 and the third was an API by Mike Clark of Clarkware Inc., called Profiler [2]. The measurements taken were based on three different table sizes. For example, the size of one of our database tables ranged from 25 tuples to 2500 tuples. Using a `select *` query while varying the size of the table, we produced result sets of sizes 25, 250, and 2500 tuples. As discussed in Section 4, the creation of varied result set sizes allowed us to control the $D_G(E)$ associated with the GUI manipulation event E. The varied sizes of the result sets

were only utilized when performing event handling latency experiments; all experiments for CPU usage and memory consumption used the largest table size and are thus associated with the highest $D_G(E)$ values.

All experiments were performed five times on each application. After conducting each experiment, we restarted the computer system to clear the memory and CPU of all interferences. This ensured that we had valid results, given limited experimentation. To avoid interference from any background activity, we also made sure that there was only one user on the system. We used a Pentium III, 533 Mhz with 128 megabytes of RAM to perform one set of experiments under both Debian/GNU Linux using the JVM 1.4.1 and Microsoft Windows NT with JVM 1.4.0. For the Solaris experiments, we used an UltraSPARC-5 Sun4u 360 Mhz with 128 megabytes of RAM running Solaris 8 and using JVM 1.4.1.

6. EXPERIMENTS

In order to examine GUI event handling latency we designed five distinct experiments: initial startup of the application (E_1), loading the first screen for database table selection (E_2), loading the second screen for attribute selection (E_3), loading the third screen for relational operator selection (E_4), and viewing the final query results with three different table sizes (E_5).[1] The second, third and fifth experiments include a database latency within the event handling latency. This is due to an unavoidable constraint set by Thinlet; to have comparable results, we incorporated the same constraint into the Swing application. However, these measurements do show the user-perceived performance to obtain the final query and also the latency for GUI components in both Swing and Thinlet. Experiment four is similar to experiments two, three, and five, but without the database latency. The experiments include the measurement of event handling latency and the CPU and memory usage associated with handling the events. The latency and memory measurements were taken by Profiler and the measurements of CPU usage relied upon `top` and Microsoft Performance Monitoring version 4.0.

7. EXPERIMENTAL RESULTS

The experiments performed on the two applications allowed us to see which GUI creation framework would perform better when used on different operating systems. The results of these experiments show that the Thinlet GUI creation framework often outperforms the Swing framework at run-time in both the latency and CPU experiments. Although Thinlet surpasses Swing in the majority of experiments, Swing is still more useful in certain areas. For example, the Swing-based version of the candidate application demonstrates lower event handling latencies for difficult GUI manipulation events.

7.1 Latency Results

Initially, we conducted a simple experiment to determine the event handling latencies and memory consumption characteristics associated with a GUI manipulation event that added a single row to one textarea. We performed this GUI manipulation five times and we report the average event handling latency times in Table 1(a) and the average memory consumption in Table 1(b). The difference in event handling latencies is noticeable in the Solaris operating system, with Thinlet reporting a lower event handling latency by approximately 2 ms. However, the difference in event handling latency for the two toolkits on the Windows and Linux OSes is negligible. As expected, the Swing-based application consumes on average 516 more bytes than the Thinlet-based one across the three selected operating systems.

	Latency Time (ms)	
OS	Thinlet	Swing
Solaris	4	6.33
Linux	3.16	3.66
Windows	3.33	3.33

(a)

	Memory Used (bytes)	
OS	Thinlet	Swing
Solaris	392	992
Linux	312	748
Windows	334.66	846.66

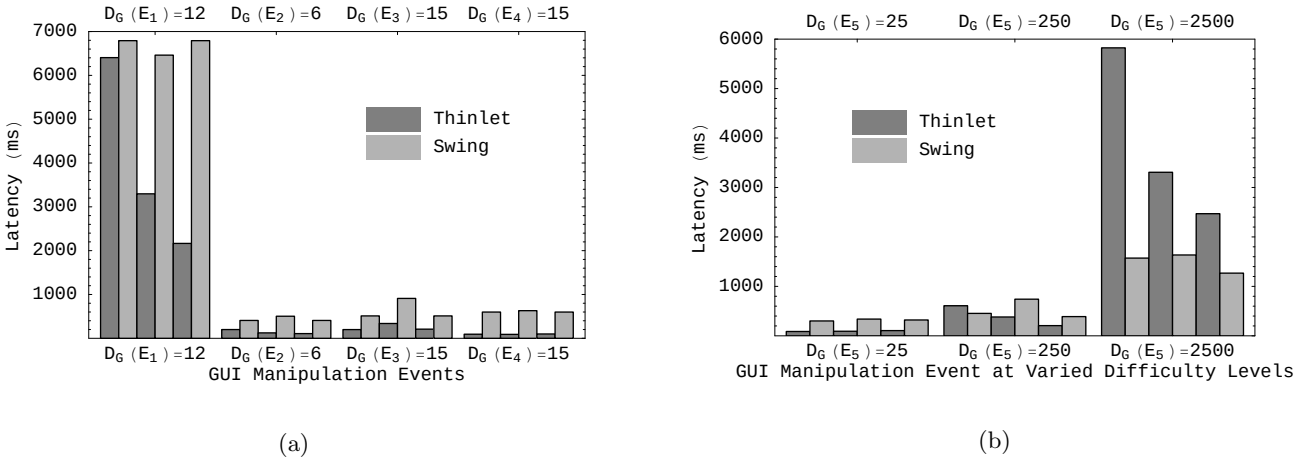
(b)

Table 1: Event Handling Latency and Memory Consumption for Single Addition to a Textarea.

The results of the latency experiments on the two frameworks show that, in most cases, the Thinlet application completed the GUI manipulation event with less latency. Figure 3(a) shows the first four experiments, labeled with the $D_G(E)$ values for each of the GUI manipulation events. Figure 3(b) shows experiment five, (time needed to obtain the final queried results) for three different table sizes. Each group of six bars shows Solaris/Thinlet, Solaris/Swing, Linux/Thinlet, Linux/Swing, Windows/Thinlet, and Windows/Swing. We see that the startup time of the Swing application was often almost double that of the Thinlet. Furthermore, it is important to observe that $L(E_1)$ is roughly equivalent to $L(E_5)$ even though $D_G(E_1) = 12$ and $D_G(E_5) = 2500$. Since E_1 incurs the computation costs associated with starting the JVM and initially creating a significant number of object instances associated with the GUI toolkit, we believe that $D_A(E_1)$ is significantly greater than $D_A(E_5)$. However, in all other experiments, the value of $D_G(E)$ is an appropriate predictor of the overall latency associated with the GUI manipulation event.

Experiment four does not contain database latency, unlike experiments two and three. This shows that, with or without querying a database, Swing takes more than twice as much time as Thinlet to complete the GUI manipulation event. Experiment five shows that there is very little difference between Thinlet and Swing when dealing with GUI manipulation events of small or moderate difficulty levels. However, when dealing with difficult GUI manipulation events, Thinlet often has a much higher latency than Swing. On the Windows operating system, Thinlet and Swing perform more equally when conducting strenuous GUI manipulations, but for the other two experiments, Swing clearly out-

[1]To ensure conformance to our previously established notation, we use E_l to refer to the GUI manipulation event that was used during experiment l.



(a) (b)

Figure 3: Event Handling Latency Results on Solaris, Linux, and Windows.

performs Thinlet. These measurements suggest that Swing is preferable if the application does large scale GUI manipulations. However, for less difficult GUI manipulation events, Thinlet should be selected because the framework quickly starts up and displays the results of less difficult GUI manipulation events with less latency.

7.2 CPU and Memory results

We measured the average CPU usage during the time required to complete the selected GUI manipulation events. According to Figure 4, experiment one, the CPU usage fluctuates between operating systems and JVMs. In this first experiment, the Thinlet application uses less CPU than the Swing application on Linux and Windows. For the Solaris OS, the Thinlet application nominally uses more CPU than the Swing application. Experiments two, three, and four all show that the Thinlet application often uses less CPU than the Swing application. Figure 4 also shows that there is a downward trend in CPU usage from the initial startup to the transition from one GUI screen to another. Figure 4, experiment five, shows an upward turn in CPU usage, which is expected since this is where the application heavily manipulates the GUI. Experiment five also shows that the Swing application uses less CPU than the Thinlet application when rendering a large amount of data to the GUI interface. Interestingly, the Swing-based application uses less of the CPU and still provides lower event handling latencies when the toolkit is responsible for handling a difficult GUI manipulation event.

For both the Thinlet and Swing applications, the memory usage increases at initial startup, stays constant over the course of the application, and then decreases once the application is exited. Both of the frameworks maintain similar memory usage levels during experiments two, three, and four. For example, when these experiments were executed on a Solaris machine, memory usage remained constant at 83 and 84 megabytes for Thinlet and Swing, respectively.

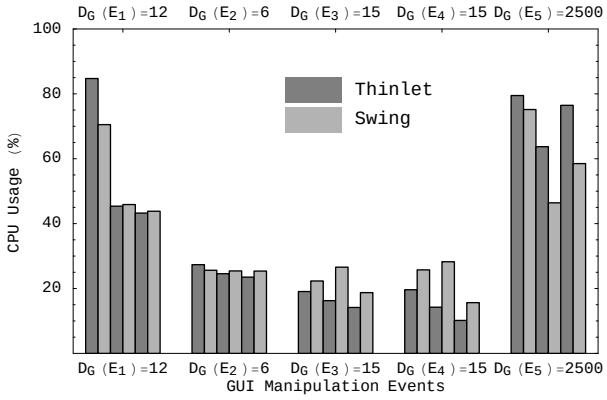


Figure 4: CPU Usage Results on Solaris, Linux, and Windows.

8. RELATED WORK

Our analysis of the run-time performance of GUI creation frameworks is related to three avenues of research: GUI correctness testing, analysis of interactive system performance, and the evaluation of Java program performance. GUI testing research conducted by Memon et al. has focused on the proposal of novel GUI coverage criteria [16], the automatic creation of test oracles [15], and the implementation and analysis of automated test data generation algorithms [14]. While Memon et al. have focused on the creation of test suites that attempt to cover GUI event sequences, Kasik et al. have addressed the issues related to the generation of test suites that mimic the behavior of novice users [11]. Yet, none of these research efforts have concentrated on the testing of applications with respect to performance or the explicit testing of the frameworks that facilitate the construction of GUIs.

Guynes et al. and Shneiderman have motivated the investigation of interactive system performance by observing that user satisfaction is tightly coupled to the performance of the user interface [7, 20]. Endo et al. have examined the issues surrounding the measurement of interactive system performance and the construction of toolkits to obtain user-perceived performance metrics [4, 5]. However, Endo et al. focused on the usage of latency to measure the performance of Windows NT and Windows applications and did not address the issues associated with Java-centric performance analysis. Horgan et al. did examine the platform-

independent analysis of Java program performance at the bytecode level [9] and Harkema et al. implemented an event-driven monitoring system for Java applications [8]. To the best of our knowledge, no research has specifically focused on the performance analysis of GUI creation frameworks for the Java programming language.

9. CONCLUSION

The graphical user interface is an important component of many applications and recent surveys indicate that the GUI can represent 60% of the overall source of a program [13]. GUI creation frameworks present an exciting alternative to the manual coding of a graphical interface. Since the run-time performance of a GUI can have a significant impact on the user-perceived performance for an entire application, it is clearly important to investigate the performance characteristics of GUI toolkits. In this paper, we examine the run-time performance of Swing and Thinlet, two GUI creation frameworks that can be used to construct graphical interfaces for programs written in the Java programming language.

The latency, CPU, and memory results gathered during these experiments can help a developer to understand different facets of the performance of each GUI creation framework. The experiments showed that the Thinlet framework often outperformed the Swing framework in terms of event handling latency and memory consumption. We have demonstrated that Thinlet should be selected if an application only performs GUI manipulation events of low or moderate levels of difficulty. However, we have also shown that Swing is better suited for the construction of GUIs that require significantly more difficult GUI manipulation events. Since there has been very little research in the field of GUI performance analysis, we are unable to compare the results of our experiments to prior research. Ultimately, the selection of a GUI creation framework depends upon the application to be developed and the manner in which the program uses the graphical interface.

10. REFERENCES

[1] Robert Bajzat. Thinlet. `http://www.thinlet.com`.

[2] Mike Clark. Profiler. `http://www.clarkware.com`.

[3] Tim Comber and John Maltby. Investigating layout complexity. In *Proceedings of the International Workshop of Computer-Aided Design of User Interfaces*, Namur, Belgium, 1996. Namur University Press.

[4] Yasuhiro Endo and Margo Seltzer. Improving interactive performance using TIPME. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 240–251. ACM Press, 2000.

[5] Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer. Using latency to evaluate interactive system performance. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pages 185–199. ACM Press, 1996.

[6] Ewald Geschwinde and Hans-Jürgen Schönig. *PostgreSQL Developers Handbook*. O'Reilly, first edition, 2002.

[7] Jan L. Guynes. Impact of system response time on state anxiety. *Communications of the ACM*, 31(3):342–347, 1988.

[8] M. Harkema, D. Quartel, B. M. M. Gijsen, and R. D. van der Mei. Performance monitoring of Java applications. In *Proceedings of the Third International Workshop on Software and Performance*, pages 114–127. ACM Press, 2002.

[9] Jane Horgan, James Power, and John Waldron. Measurement and analysis of runtime profiling data for Java programs. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 124–132. IEEE Computer Society Press, November, 2001.

[10] Robin Jeffries, James R. Miller, Cathleen Warton, and Kathy Uyeda. User interface evaluation in the real world: a comparison of four techniques. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 119–124. ACM Press, 1991.

[11] David J. Kasik and Harry G. George. Toward automatic generation of novice user test scripts. In *Conference proceedings on Human Factors in Computing Systems*, pages 244–251. ACM Press, 1996.

[12] Marc Loy, Robert Eckstein, Dave Wood, James Elliott, and Brian Cole. *Java Swing*. O'Reilly, second edition, 2002.

[13] Atif M. Memon. GUI testing: Pitfalls and process. *IEEE Computer*, 35(8):90–91, August 2002.

[14] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Using a goal-driven approach to generate test cases for GUIs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 257–266. IEEE Computer Society Press, 1999.

[15] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated test oracles for GUIs. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 30–39. ACM Press, 2000.

[16] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for GUI testing. In *Proceedings of the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 256–267. ACM Press, 2001.

[17] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28, 2000.

[18] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 195–202. ACM Press, 1992.

[19] Andrew Sears. Layout appropriateness: A metric for evaluating user interface widget layout. *IEEE Transactions on Software Engineering*, 19(7):707–719, 1993.

[20] Ben Shneiderman. Response time and display rate in human performance with computers. *ACM Computing Surveys*, 16(3):265–285, 1984.

[21] James A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 17(1):70–76, January/February 2000.

[22] Alan Williamson. Swing is swinging Java out of the desktop. *Java Developer's Journal*, 7(10), October 2002.