Logging Events Crossing Architectural Boundaries

Gregory S. Hartman¹ and Len Bass²

¹ Carnegie Mellon University, Institute for Software Research International, USA
gghartma@cs.cmu.edu

² Carnegie Mellon University, Software Engineering Institute, USA
ljb@sei.cmu.edu

Abstract. We describe an approach to study the long-term use of GUI applications that supplements a log of low-level events with additional data gathered at the applications' architectural boundaries. We implement a preliminary system based on this approach and apply it to two applications. For the second application, we compare the data collected with our technique to data collected with manual instrumentation. We demonstrate that our technique is easy to apply to new applications and captures information missed by manual instrumentation. This additional information is helpful in answering questions about the use of the application. However, our technique generates large logs and does not yet capture all of the information needed to study the use of applications. We conclude with proposals for rectifying these deficiencies in future systems.

1 Introduction and Problem

It is difficult to build usable applications. This problem can be addressed by involving users in the design of applications, which has been shown to increase their level of satisfaction with the final product [2]. When applications serve a large population of users, it may not be possible to involve a representative sample of users. Detailed records of the interactions between users and prior versions of the application could serve as a substitute for direct user involvement. Logging is a compelling way to gather these data, since it scales to a large number of users, does not require specialized hardware, and can support automatic analysis.

Existing approaches to logging require considerable investment in the process of designing and implementing the logs and/or the process of interpreting the logs to answer specific questions about the use of the application (called queries in this paper). We propose a method that exploits knowledge of the structure of GUI applications to reduce the cost of implementing and interpreting logs. While the initial effort needed to design this logging system is comparable to the effort to write application-specific instrumentation, the resulting logging system can be reapplied to other applications using the same GUI toolkit with relatively little effort. We demonstrate this by writing instrumentation for jEdit [10] (an open source text editor), and reapplying the same instrumentation to a drawing tool implemented for a computerized whiteboard. We compare the information gathered by our logging system to the information gathered by manual instrumentation in the drawing tool.

2 Existing Logging Techniques

Existing logging techniques, as outlined in [7] and [9], either gather data with application-specific changes (manual instrumentation) or gather GUI events and analyze these events to reconstruct interactions between the user and the application. The relative strengths and weaknesses of manual instrumentation and GUI event loggers become apparent when they are assessed against the selection, context, abstraction, reduction, and evolution problems described in [8]:

- What should be logged? This question has two parts: the approach must decide when to add an entry to the log (the *selection* problem) and what part of the application's state should be included in the entry (the *context* problem).
- How can the data be translated into terms that are appropriate to answer queries? This corresponds to Hilbert's *abstraction* problem.
- How can the data be processed to reduce its volume and highlight events that answer queries? This corresponds to Hilbert's *reduction* problem.
- How should the logging capability be maintained? Changes in the application may require new data to be logged, and changes in the logging may require new releases of the application. This corresponds to Hilbert's *evolution* problem.

In manual instrumentation, programmers solve the selection problem by first translating the queries into specific interactions between the user and the application. They then use these interactions to identify specific locations in the application's code that will execute when these interactions occur. They resolve the context problem by finding the parts of the application's state that are needed to answer the queries. The programmers then insert statements into the application to log execution of these sections of code. This work must be done carefully, since mistakes in the analysis of the selection or the context problems will produce logs that lack information needed to answer the queries. This approach also suffers from the evolution problem, since changes in the application after the logging is implemented may invalidate the analysis of the selection and context problems. In addition, modifications to the logging force a re-release of the application. Some projects attempt to avoid the evolution problem by postponing the implementation of logging, increasing the risk that defects in the logs may not be noticed before the application is released.

While these risks are substantial, manual instrumentation can return valuable data. For example, Cook and Kay employed application-specific changes to study the use of a text editor over a three year period [4]. This paper also illustrates the advantage of manual instrumentation; since the log messages are carefully designed based on the queries, the messages need little processing to provide answers to the queries. Therefore, manual instrumentation rarely suffers from the abstraction problem.

Approaches that log GUI events attempt to avoid the evolution problem by capturing information at the operating system level [5] or through modified GUI toolkits [1, 6, 13]. These approaches can also avoid the selection problem; it is typically possible to log every event and then filter out the unwanted events during analysis. There are two disadvantages to these approaches. First, no GUI event loggers provide a fully automated solution to the abstraction problem. In part, this is caused because the techniques work without detailed knowledge of the application. To solve the abstraction problem, programmers must develop a model of the users'

expected behavior [6] or create a mapping between the low-level events and higher level concepts during the analysis [1].

A second weakness becomes apparent when we consider the context problem. Several of the tools provide a logging mechanism to capture the data needed to solve the context problem, but no GUI event loggers can capture this data without modifications to the application. In addition, the loggers do not specify what context would be useful in interpreting the logs. To work around these limitations, programmers must still analyze the queries to identify the context and make changes to the application to capture this context. As with manual instrumentation, these changes reintroduce the evolution problem.

To conclude, we are aware of several viable solutions to the selection problem, but we cannot find a solution to the context problem that does not involve application-specific modifications. In addition, there are few general guidelines that developers can use to identify the relevant context in their applications. The lack of automated support for capturing context also prevents a complete solution to the evolution problem; any change to the application may invalidate the analysis done by developers to collect context, creating new deficiencies in the logs. Finally, there are no fully automatic solutions to the abstraction problem: existing solutions rely on input from either programmers or analysts to create a mapping between low-level and high-level events.

3 Approach

We propose an approach that solves the selection problem by logging at four architectural boundaries present in applications implemented with modern GUI toolkits, such as MFC, Swing, and Motif. These boundaries are illustrated in figure 1. We solve the abstraction problem by associating the data captured at each boundary with the event being processed. Including data captured at the boundary crossed by marker 4 in the diagram partially addresses the context problem. Finally, we address our current approach to reduction. We discuss the evolution problem and readdress the context problem in the next section, which gives more details about our initial implementation of this approach in a system called LECAB.

In most GUI applications, events representing low-level user actions (such as key presses and mouse movements) travel through several well defined layers in the GUI toolkit and the application. In figure 1, events move from the lower layers toward the top layers, eventually reaching application-specific code. At each layer, the events are represented at a higher level of abstraction. Like many other GUI event loggers, we capture events at the lowest level of the architecture. Unlike other loggers, we also track these events as they travel to higher levels of the architecture, logging additional information about the code processing the events.

Given the importance of solving the abstraction and reduction problems, it seems that we should only capture data at the higher levels of the architecture, as is often done in manual instrumentation. In our experience, the data gathered at each level of the architecture are essential, since some of the events at the lower levels will not be dispatched to higher levels. For example, users may click on parts of the application

Fig. 1. The architecture of a typical Swing application. The numbered circles portray the path of a typical event through the application. Boxes below the bold line are part of the GUI toolkit; boxes above the line are part of the application.

windows where there are no controls to accept the event [12]. In this case, the GUI toolkit may silently drop the event. By capturing events in level 1, we record these events. Later, these events can be detected by searching logs for mouse clicks that did not translate to higher levels in the architecture. While users may occasionally click the mouse button without expecting a response from the application, a large cluster of these events in a particular location of the window may indicate a problem in the design of the user interface.

If the events are successfully dispatched to a control (reaching 2 in the diagram), we log the identity of the control that received the event. We identify the control by logging both the type of the control (scrollbar, button, etc), strings that are displayed on the screen, and the name of any icons on the screen. By associating this information with the low-level event, we solve part of the abstraction problem: the event is now described in the terms displayed in the user interface of the system.

This information also points to usability problems that are not visible at higher levels of the architecture. We encountered one of these problems when we observed actual use of a drawing application that provided a toolbar of buttons for selecting controls. Users occasionally attempted to insert shapes into their drawings by dragging these buttons into the drawing canvas. The buttons interpreted the drag as a canceled button press. Figure 2 indicates that all of these events were dispatched to a control; logging at level 1 would not have identified the problem. However, there are no activations at level 3 for the MouseToolBar, even though the user clicked in it at step A1. Once again, this pattern does not automatically point to a usability problem; users may make this gesture to cancel an unwanted button press. However, this sequence of events was often present when users were confused by the user interface of the drawing application.

At level 3, the event processing moves into application specific code. The GUI toolkit sends the event to the application by invoking one or more callbacks registered by the application (called listeners). When this happens, we log the name of the listener being invoked. The name of this listener is often closely associated with the name of one of the commands of the application. By logging the name of the command invoked by the low-level event, we complete our solution to the abstraction problem.

In general, we do not attempt to capture information in the application-specific code represented by layer 4 in the diagram. Capturing more information would require either modifications to the application or the use of application-specific

	EventQueue	Control	Listener
A1.	User clicks on the textbox button in the toolbar		
	MousePressed	MouseToolBar\$1	
A2.	User drags into the canvas		
	MouseDragged	MouseToolBar\$1	
	MouseDragged	JScrollPane	
	MouseDragged	DocumentCanvas	SelectTool
A3.	User releases the mouse button		
	MouseReleased	DocumentCanvas	

Fig. 2. Events showing a failed attempt to insert a new object in a drawing by dragging a button from the MouseToolBar into the drawing canvas. The EventQueue, Control, and Listener columns show what information we collect at these levels. The table is interrupted by lines explaining the user's action.

knowledge. Both of these approaches would necessitate additional work from developers, which we avoid.

However, we do identify information that will help to solve the context problem. The GUI applications that we have studied to date implement undo by representing changes to documents in a stack. This code is represented by the UndoManager layer on the architectural diagram. We provide an interface to allow changes to this state to be recorded. We discuss an application independent technique that allows this data to be captured for some Swing applications in the next section.

Unlike other approaches that solve the reduction problem by analyzing events as they are captured [6], we log every event and boundary crossing and rely on an application-specific post-processing script to reduce the volume of the data. Our approach minimizes the risk that useful information will be lost, and also allows a solution to the reduction problem to be tailored to specific queries. Unfortunately, this approach creates extremely large logs. However, application developers may choose to stop logging at levels 1 and 2 if they do not want to identify usability problems like the ones described in this section. Once we gain more understanding of the data that are needed to analyze typical queries, we hope to reduce the size of the logs by excluding common sequences of events that are not useful in analysis.

4 Implementing Our Approach in LECAB

Our initial implementation of this approach (LECAB) uses AspectJ [11] to automatically instrument Java applications implemented with the Swing GUI toolkit. AspectJ automates the instrumentation of the application, greatly reducing the evolution problem. The choice of Java and Swing simplifies the implementation of LECAB and reduces the programmer effort needed to gather information about changes in the undo state that LECAB uses to address the context problem.

LECAB avoids the evolution problem by relying on AspectJ to automate the instrumentation of the application. The AspectJ project provides a compiler that accepts standard Java source code and one or more aspects that describe changes to be

woven into the code. These aspects are made up of two parts: fragments of Java code to be inserted, called advice, and patterns that trigger the insertion of the code, called pointcuts. These pointcuts match one or more well-defined points in the execution of the program code. The architectural boundaries that LECAB uses to instrument Swing applications are easy to represent in terms of pointcuts. For example, the pointcut that captures calls to the listeners in the application is shown in figure 3. We have developed similar aspects to capture interactions with the undo system provided by Swing and to initialize the event capture system when the application starts.

```
public aspect AWTListeners {
    pointcut action(EventObject e, EventListener l):
        execution(void *.*(*)) &&
        args(e) && target(l);

    before(EventObject e, EventListener l): action(e,l){
        AOPLoggerExt.LogListener(CrossType.ENTER, e, l);
    }

    after(EventObject e, EventListener l): action(e,l) {
        AOPLoggerExt.LogListener(CrossType.EXIT, e, l);
    }
}
```

Fig. 3. An aspect that inserts logging code that will be triggered when Swing calls one of the EventListeners defined by the application

Automatic instrumentation has several advantages. First, it reduces the effort needed to apply LECAB to new systems. Second, it reduces the chances for omissions and other errors in the instrumentation. Finally, it eliminates most of the evolution problem, since the automated placement is repeated whenever programmers recompile their application.

LECAB also benefits from patterns defined in the Java language. For instance, LECAB calls the `toString` method on each new listener and control that it encounters to extract information that describes the control. This method typically returns both the name of the class and much of the internal state of the object, such as the text string being displayed on the control. The class names are often sufficient to describe the purpose of the control. Programmers typically follow one of two patterns when they implement EventListeners. Often, programmers do not name the classes. In this case, the EventListener is assigned a name derived from the enclosing class at compile time. In other cases, programmers name the class after the command being implemented. For example, a listener attached to an undo button may be called `UndoHandler`. In either of these cases, associating the class name with the low-level GUI event solves the abstraction problem: it translates the low-level event into the terms that the programmers use to reason about the application.

Swing also dictates that all GUI processing must happen on a single thread. Therefore, LECAB can assume that any crossings of the architectural boundaries that occur while the event thread is processing a GUI event are caused by the event.

LECAB logs both calls and the returns that cross architectural boundaries, and assigns any messages that happen between these crossings to the event being processed.

Finally, LECAB benefits from Swing's approach to implementing undo. Unlike many other GUI toolkits, Swing provides a centralized `UndoManager` that is well integrated with some of the more complex controls in the toolkit. While the use of this manager is optional, many developers choose to use it to reduce the effort of implementing undo in their applications. In LECAB, we use the knowledge of this service to track changes made to the document, eliminating the need for programmers to provide information about their implementation of undo.

An example of a LECAB log is shown in figure 4. The top section of the log shows the descriptions of various events, controls, and listeners that LECAB gathers with the `toString` method. Each of these descriptions is given a short name that is used in other log entries. In a real log, the descriptions are interleaved with the messages; in this sample they were moved to the beginning of the fragment to improve the readability of the log. The second section shows the architectural boundary crossings for a single event. The first line corresponds to level 1 of the architecture, and indicates that the application is processing event E626. The definition of E626 indicates that it is a press of button 1 of the mouse. The second line shows that the event was sent to control C2, a `TextAreaPainter`. The `TextAreaPainter` informs the application of the event by calling listener L15, a `MouseHandler` defined in the `JTextArea` class of the application. While this listener is processing the event, some part of the application adds the record U2 to the `UndoManager`. This record indicates that the caret moved to location 25 in the document. This chain of reasoning from the low-level event to the high-level change in the state of the document represents a solution to the abstraction problem.

```
DEFINE E626 MouseEvent[MOUSE_PRESSED, (43,109),button=1...
DEFINE C2 TextAreaPainter[,4,0,638x420,alignmentX=...
DEFINE L15 JEditTextArea$MouseHandler@23bdd1
DEFINE E627 MouseEvent[MOUSE_PRESSED, (33,10),button=1...
DEFINE U25 caret move[start=25,end=25]

ENTER EVENT E626
  ENTER CONTROL C2 EVENT E626
    ENTER LISTENER L15 EVENT E627
      CREATED UNDO U25
    EXIT LISTENER L15 EVENT E627
  EXIT CONTROL C2 EVENT E626
EXIT EVENT E626
```

Fig. 4. LECAB events for a click inside a `jEdit` document. This sample log has been reformatted: the timestamps and thread id on each entry have been removed, some of the statements have been abbreviated, and indentation has been added to illustrate the nesting that LECAB uses to solve the abstraction problem.

5 Applying LECAB to a Second Application

During the development of LECAB, we ran simple tests on jEdit to verify LECAB's solutions to the selection, abstraction, and context problems. To verify LECAB's application independence, we used the version of LECAB developed against jEdit to instrument a drawing application designed for an interactive whiteboard. This application was developed as part of a research effort by an outside group of developers. As part of their research, these developers added manual instrumentation to the application. Our prior experience studying these logs to answer queries highlighted several deficiencies in the data collected. Since the drawing application already contained a version of manual instrumentation and we already had a large list of queries, it was a good candidate for an initial test case.

When we examined the first set of logs created by LECAB, we realized that the logs lacked context information about the changes to the document being edited. After inspecting the code, we discovered that the developers did not make use of the undo functions provided by Swing. This new version of the undo system had a similar structure to the one provided by the Swing libraries, but used different class names. Therefore, we wrote an application-specific aspect to collect information from the application's undo system.

6 Comparing LECAB to Manual Instrumentation

We decided to compare LECAB to manual instrumentation by assessing their ability to count the number of times that users grouped multiple objects in the drawing application. We defined a simple user session that would simulate use of the drawing application. In this session, the user creates three simple objects, groups two of the objects, and then moves both the group and the ungrouped object. These actions and the resulting log messages are given in more detail in figure 5.

The events in these log messages correspond to five general categories. The messages marked with E1 and E5 represent simple clicks in the application's toolbar to select a tool. In both of these cases, the LECAB log is roughly identical to the manual log: both identify that a tool was selected (LECAB's log of the "MouseRelease heard by" is roughly equivalent to the manual log's "Selected Tool"), and both identify which tool was selected.

E2, E3, and E4 are logged as the user draws the shapes on the board. Both LECAB and the manual instrumentation identify the operation, but LECAB's instrumentation also captures unique identifiers for the objects being created (FH0, FH1, and FH2). While this is not valuable for this query, we have encountered other queries that could use the information. For instance, we wanted to count the number of times that a user immediately deleted a new object, since this is similar to an undo. This pattern can be detected by searching the log for create statements that are immediately followed by delete statements that reference the same features.

E6 marks a scenario where neither LECAB nor the manual log returns as much information as we would like. Here, the user has just completed a select operation. Ideally, we would like to know which objects were selected. Since this does not

User Action	LECAB Log	Manual Log
E1. User clicks on freehand drawing tool in the toolbar	MouseReleased heard by FreeHandTool	Whiteboard: Selected Tool: FreeHandTool
E2. User draws a line with the tool	MouseDragged heard by FreeHandTool that creates FH0	Whiteboard: Did Freehand draw
E3. User draws a second line	MouseDragged heard by FreeHandTool that creates FH1	Whiteboard: Did Freehand draw
E4. User draws a third line	MouseDragged heard by FreeHandTool that creates FH2	Whiteboard: Did Freehand draw
E5. User clicks on the select tool in the toolbar	MouseReleased heard by SelectTool	Whiteboard: Selected Tool: SelectTool
E6. User selects the first two lines	MouseReleased heard by SelectTool	Whiteboard: Used select tool
E7. User presses the “make group” button on the toolbar	MouseReleased heard by GroupHandler that deletes FH1, deletes FH0, creates PF0	Whiteboard: WB Action Button: Grouping figures
E8. User clicks on the grouped lines and begins to move them	MouseDragged heard by SelectTool that does nothing?	
E9. User releases the mouse button after moving the group	MouseReleased heard by SelectTool heard by SelectTool that changes PF0, ...	Whiteboard: Used select tool
E10. User clicks on the third, ungrouped line and begins to move it	MouseDragged heard by SelectTool that does nothing?	
E11. User releases the mouse button after moving the line	MouseReleased heard by SelectTool heard by SelectTool that changes FH2...	Whiteboard: Used select tool

Fig. 5. This figure compares the data collected by LECAB to the manual instrumentation of the drawing application for a simple drawing session

change the state of the document, LECAB's instrumentation of the undo system does not see this change in the application state.

E7 demonstrates that LECAB's instrumentation of undo captures information missed in the manual instrumentation. The LECAB log shows that the group button removed the first two features in the document (FH0 and FH1) and inserted PF0. The manual instrumentation only indicates that the grouping figures button was pressed, it does not describe the effect that this operation had on the document.

E8 and E10 mark places where LECAB detected a bug in the undo system implemented in the drawing application. We were aware of this bug from prior studies that observed actual use of the application with video recording, but were unable to reproduce it. With the detailed information provided by LECAB we were able to see that the bug was triggered by moving objects on the drawing canvas.

E9 and E11 also illustrate a case where LECAB's log is superior to the manual instrumentation. Here, the user clicks on the new group of objects (E9) or the remaining ungrouped object (E11) and drags it to a new location. This operation is handled by the same callback that handles selecting objects. Therefore, both the LECAB log and the manual log record the use of the select tool. However, the instrumentation that LECAB adds to the undo handling in the application detects a change to the document and identifies the changed object.

This example illustrates that LECAB can return more of the data needed to answer queries than manual instrumentation. However, it also points to two limitations of LECAB. First, even after aggressive reduction, the log messages returned by LECAB are considerably more verbose than messages generated by manual instrumentation. Second, LECAB misses changes to the state of the application, such as the state tracking the current selection, that do not affect the document being edited. While this missing state did not adversely affect our ability to analyze this query, it could prevent the analysis of other queries.

7 Future Work

There are several limitations in our evaluation of LECAB. Before designing LECAB, we attempted to analyze queries by examining the logs generated by the manual instrumentation in the drawing application. The design of LECAB was informed by our knowledge of what information was not included in these logs. In addition, the developers of the manual instrumentation did not know about our queries when they designed the instrumentation. Given knowledge of our queries, they would have been more likely to design instrumentation that would capture the relevant data.

This study would be enhanced by looking at a wider assortment of applications and more queries. We intend to do this as part of our future work. This collection of applications should also help us to understand how often developers using Swing do not use the undo manager provided by the toolkit. If this is a common pattern, our approach to automatically collecting context will not be workable.

Based on our initial experiences with LECAB, we now realize that we need to find a way to reduce the volume of the data collected by the system (the reduction problem). This is a difficult problem to solve, since overly aggressive reduction could discard data needed to answer queries. We plan to examine approaches used in prior work that identify and eliminate the events that can be reconstructed from other data [3].

Logging systems must protect the privacy of users [12]. We have not addressed this in LECAB. We may be able to address privacy by discarding data about the content of documents, such as the information in a text box on a drawing, while retaining information about the documents' structure, such as the position of the text box in the drawing and the amount of information that it contains.

If we assume that developers can invest a small amount of effort to improve logging, we may be able to direct them to write aspects that would log context that is missed by LECAB. For example, developers may be able to identify the state in their applications that tracks the current selection and write aspects to log changes to this state.

Our current approach to logging changes to the document relies on a correct implementation of undo. If developers forget to implement undo for a specific command, LECAB will not log the corresponding changes in the document. In future work we intend to provide a toolkit to support developers as they implement features that cannot be isolated to one part of the application, such as cancel. This toolkit would isolate much of the state of the application, providing another interface that could be targeted by LECAB. Discrepancies between the crossings of the cancel interface and crossings of the undo interface would reveal defects in the implementation of undo.

8 Conclusion

We described a new approach to capturing interactions between users and GUI applications. Our approach augments a log of GUI events with class names and descriptions of document changes captured at architectural boundaries of the application. These additional data facilitate the interpretation of the logs; class names often correspond to features in the application and changes in the document often describe the application's response to the user's action. We implemented this approach in LECAB, a logger designed to capture information about Java applications using the Swing GUI toolkit, and use LECAB with two different applications. Simple tests demonstrate that the resulting logs address the selection and abstraction problems described by Hilbert [7]. In some cases, the LECAB logs provide more information about interactions between users and applications than manual instrumentation. However, the same tests indicate that additional logging is needed to capture the context. In addition, the reduction problem must be addressed during data capture to control the size of the logs. We plan to address these issues in future work.

Acknowledgements

This work was supported by the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010. We thank Bonnie John for her help in forming the queries we used to evaluate the logging systems. We also thank the developers and other researchers working with the drawing application for their help in gathering data and reconstructing the development process.

References

1. Al-Qaimari, G., McRostie, D.: KALDI: a computer-aided usability engineering tool for supporting testing and analysis of human-computer interaction. In: Proceedings of the third international conference on Computer-aided design of user interfaces, Kluwer Academic Publishers. (1999) 337-355
2. Baroudi, J.J., Olson, M.H., Ives, B.: An empirical study of the impact of user involvement on system usage and information satisfaction. *Commun. ACM* 29 (1986) 232-238
3. Chung, G., Dewan, P.: A mechanism for supporting client migration in a shared window system. In: UIST '96: Proceedings of the 9th annual ACM symposium on User interface software and technology. (1996) 11-20
4. Cook, R., Kay, J., Ryan, G., Thomas, R.C.: A toolkit for appraising the long-term usability of a text editor. *Software Quality Journal* 4 (1995) 131-154
5. Gellner, M., Forbrig, F.: ObSys: a tool for visualizing usability evaluation patterns with mousemaps. In: HCI International. (2003)
6. Hilbert, D.M., Redmiles, D.F.: An approach to large-scale collection of application usage data over the internet. In: Proceedings of the 20th international conference on Software engineering, IEEE Computer Society. (1998) 136-145
7. Hilbert, D.M., Redmiles, D.F.: Extracting usability information from user interface events. *ACM Comput. Surv.* 32 (2000) 384-421
8. Hilbert, D.M., Redmiles, D.F.: Large-scale collection of usage data to inform design. In: Proceedings of the Eighth IFIP TC.13 Conference On Human-Computer Interaction (INTERACT 2001). (2001)
9. Ivory, M.Y., Hearst, M.A.: The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv.* 33 (2001) 470-516
10. *jEdit: Open source programmer's text editor.*
11. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Proceedings of the 15th European Conference on Object-Oriented Programming, Springer-Verlag. (2001) 327-353
12. Nielsen, J.: Usability Engineering. Morgan Kaufmann Publishers Inc. (1993)
13. Uehling, D.L., Wolf, K.: User action graphing effort (UsAGE). In: Conference companion on human factors in computing systems, ACM Press. (1995) 290-291