HERCULE: Non-invasively Tracking Java^{TM} Component-Based Application Activity

Karen Renaud

Department of Computing Science, University of Glasgow,
17 Lilybank Gardens, Glasgow, G12 8RZ, UK.
karen@dcs.gla.ac.uk

Abstract. This paper presents HERCULE, an approach to non-invasively tracking end-user application activity in a distributed, component-based system. Such tracking can support the visualisation of user and application activity, system auditing, monitoring of system performance and the provision of feedback. A framework is provided that allows the insertion of proxies, dynamically and transparently, into a component-based system. Proxies are inserted in between the user and the graphical user-interface and between the client application and the rest of the distributed, component-based system. The paper describes: how the code for the proxies is generated by mining component documentation; how they are inserted without affecting pre-existing code; and how information produced by the proxies can be used to model application activity. The viability of this approach is demonstrated by means of a prototype implementation.

1 Introduction

The motivations for tracking application activity are many and varied. Motivations could be a need to:

– understand the application execution process, especially if the application is distributed or runs on parallel processors;
– provide information needed to carry out system tuning;
– satisfy security requirements;
– provide an audit trail; or
– provide extra support for end-users.

It is the latter aspect which provides the motivation for the research outlined in this paper. The fact that feedback is required by the end-user of an application is no longer disputed [29]. However, the manner of providing this feedback, and standards for ensuring the quality thereof, are still open to debate. Feedback must, at present, be provided for during the development of an application, and it is extremely difficult to remedy applications which provide inadequate feedback, once they are in use. I propose to augment the application's end-user feedback provision by means of external application tracking.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 447–471, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Providing feedback independently of the application can only be done if we have a means for observing application activity. There are various ways to achieve this, and the approach chosen here is to engage proxies which will intercept communications and generate reports containing details about application activity. These reports are relayed to a framework, called HERCULE[1], which uses the information to provide the desired level of feedback.

Since portraying information about application activity in order to augment application feedback is a novel use of the information derived from application tracking, this paper also addresses the visualisation of the information thus obtained, and shows how the visualisation has been used to provide feedback tailored to the needs of users operating in different roles.

The following section will discuss current research into application tracking, feedback and separation of concerns. Section 1.2 will outline the proposed solution to the problem.

1.1 Related Work

This section discusses current research into application tracking, separation of concerns, and feedback.

Application Tracking. Applications can be tracked from two different perspectives, either tracking the user interaction with the application, or watching application interaction with the system. One example of user interface tracking is seen in the work of Trafton and Brock [39] whose system provides a layer between the user interface and the application to keep track of the user's actions, comparing them to an internal representation of various task models, to try to identify the task being done by the user. When a correspondence can be pinned down, the user is offered the option of the sequence being completed automatically. Fawcett and Provost [16] worked on finding ways to predict whether the user of a given account is not the authorised user. They profile each user by characterising their behaviour based on histories of previous sessions. Many researchers have studied the processes and patterns of user interaction with computer systems [5, 9, 22, 25], while Lin *et al* [24] have developed methods for visualising the masses of data collected about user search patterns in a variety of graphical formats, allowing human pattern recognition capabilities to be applied. Chalmers *et al* [12] have developed a methodology to build up Web usage histories for users in a particular community. The user search path is compared to paths of other users within the community and if a match is found, sites visited by the other users will be suggested as being of probable interest.

Other researchers have looked at tracking application use of system resources. Burton and Kelly [7, 8] have developed a tool which traces system calls and provides the ability to re-execute these calls to allow system tuning. Ball and Larus [4] have described algorithms for placing code within programs in order to record program behaviour and performance. Jeffery *et al* [18] introduce the

[1] Named after Hercule Poirot, Agatha Christie's legendary detective.

Alamo monitor program execution monitoring architecture which assists developers in bug-detection, profiling programs and visualisations. Siegle and Hofmann [35] have developed the SUPRENUM microprocessor which uses a hybrid combination of software and hardware monitoring to determine parallel program behaviour. This assists programmers in gaining insight to the execution of their parallel programs. Wybranietz and Haban [41] also use a hybrid approach to observe system behaviour, measure performance, and record system information. They make use of a special measurement processor which runs monitoring software for each distributed node in the system. The information thus derived is displayed graphically and used to improve understanding about run-time system behaviour. Joyce *et al* [19] monitor distributed systems by means of a distributed programming environment called Jade, which assists the programmer in debugging, testing and evaluating distributed systems.

To summarise, tracking can be done either invasively or non-invasively. Invasive tracking requires that changes be made to the application in order to support the tracking. This is risky, since it could be expensive in terms of time and effort to remove the reporting mechanism when there is no longer a need for it. It is also, by definition, application specific, and tracking must be added to each application type individually. Non-invasive tracking does not require an application to be changed in any way, is easily deactivated, and can seamlessly track a variety of applications, but is much harder to accomplish.

Why Feedback? Users, especially novice users of an application, often have no idea of how to use the application, and need to spend time and effort building up an internal model of how the application works. They seldom read manuals, wanting rather to find out for themselves how the system works [11], and they tend to be impatient to get on with their task [6], not wanting to spend hours being taught how to use a system.

Norman [29] argues that in any complex environment — like a new application — one should always expect the unexpected. To deal with the unexpected, Norman concludes that continuous and informative feedback is essential. Learning how a system works is made safer and less risky if the relevant information is easy to find [11]. Chan *et al* [13] have shown that an active feedback system greatly improves user performance. Therefore, *feedback* is far superior to user manuals for helping the user to build up a correct internal model. The role of clear explanations in this process is vital [23]. Users also need feedback because they have severe limits with respect to the following, taken from [31]:

- *perception* — perceiving small difference in detailed information;
- *memory* — the limitations of human memory is illustrated by the following examples:
  - users sometimes forget what they have done, especially if they are interrupted during a processing session;
  - users often do not detect their errors. Often the user is vaguely aware that something has gone wrong, but has no idea how this occurred.
  - difficulties are often experienced in holding recently experienced information until needed; and

  - users experience problems retaining information retrieved from long-term memory — such as remembering where they are in a plan of action.

On the other hand, users have particular strengths with respect to [31]:

- processing visual information rapidly, and coordinating multiple sources of information;
- making inferences about concepts or rules from past experiences;
- storing common patterns, like user action sequences, efficiently;
- retrieving relevant information quickly.

It is clear that the user requires feedback, to help them to discover how an application works, to help discover what effect their inputs have on the application, and to serve as a memory aid. The feedback is traditionally provided from within the application code, but this approach is flawed because programmers are seldom trained to provide adequate feedback, and it is almost impossible to augment the feedback once the application has been delivered. Users functioning in different roles often have completely different feedback needs, and it is difficult for an application to provide for all of them adequately.

Separation of Concerns. Programmers have to deal with a great deal of complexity as part of their task in developing software. They have to concentrate not only on the required functionality of the software, but also with important issues like replication, distribution, real-time configuration, synchronisation, perisistence and end-user needs. Much research has been done into providing programmers with tools which separate the behavioural features of the software from the functional features [17]. Some examples illustrate the approach with respect to:

- *Distribution* — since the failure of distributed systems has very different failure semantics from centralised systems, the separation of this concern is not a simple task. Guerraoui [17] describes Garf, a software development tool which provides a library of abstractions to simplify distributed programming by enabling the programmer to develop the functionality of the software first, and then add distribution by using Garf.
- *User Interface Code* — the Chiron-1 user interface development system [36] introduces a series of layers that separate user interface code from application code by using user interface agents called artists which are attached to the abstract data types. Operations on the data types then trigger user interface activities.
- *User Manuals* — Thimbleby [37] developed Hyperdoc, a system which allows a programmer to develop the user manual alongside the user interface, so that the user manual mirrors the structure of the user interface.
- *Exception Handling* — Dellarocas [15] makes a case for separating exception handling from normal system operation. An exception handling service is provided for use by component developers, which uses a knowledge base to describe the failure of the system to the user.

– Kiczales [21] introduces *aspect-oriented programming.* Aspects refer to location, communication or synchronisation and once specified, are automatically combined with the application program by using some tool, like AspectJ [3]. Kersten and Murphy [20] have built a web-based learning environment using this programming paradigm.

All these examples require the programmer to specify the basic functionality of the application, and special concerns, albeit separately. Kiczales argues that this helps to reduce complexity that the programmer has to cope with.

Use of Tracking Information. Whereas the results of user interface monitoring are sometimes utilised by the end-user of a system [12], it is often done primarily for the benefit of system developers and maintenance teams. System resource monitoring is done exclusively for the benefit of system development teams. One important stake-holder in application usage, the end-user, is seldom catered for. Since the focus of this research is the end-user, and their feedback needs, I propose to provide the programmer with a feedback enhancing tool which will enable them to concentrate on the functionality of the program without the complication of having to provide extensive feedback. This will be done by means of tracking the application in order to provide the feedback. The author is unaware of any work which currently addresses augmenting feedback needs in this particular way.

1.2 Proposed Approach: Disentangling Feedback by Using Tracking

Tracking application activity is simple if the need is anticipated before, or during, the development of the system. The programmer can explicitly code the necessary reporting activities along with the coding of the application logic, and this tracking will be detailed and provide excellent reporting facilities. However, it does place a significant load on the programmer, and it is difficult to anticipate all tracking needs that could be required. This also does not solve the problem of tracking an *existing* application. It would be very useful to have an add-on generic facility available to all applications. This type of application tracking must, perforce, treat the entire application as a *black box*, thus providing very little detail about the internal application operation. The application's interaction with its environment — the user and the rest of the distributed system — will have to be observed, and these interactions tracked in order to provide a log of application activity. This will be done by inserting proxies to report on this activity. While this obviously does not provide the intricate detail which could be provided by internal application reporting, it does have the advantage of being generic enough to be applied to any application, and the added attraction of not requiring any special effort from the application programmer.
The research discussed in this paper was done in the context of end-user applications within Component Based Systems (CBSs) (Fig. 1).

– CBSs were chosen because they have unique characteristics which make such application tracking not only possible, but extremely beneficial. These characteristics include the independence of constituent components; the fact

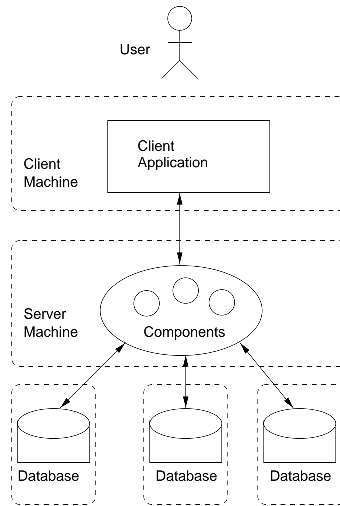

Fig. 1. CBS Application Architecture

that each component has its own documentation; and the use of interfaces to interact with these components. The need for such a facility is even greater in CBSs than in normal applications, due to this selfsame independent nature of the various component parts. This independence is a vital feature of CBSs since it allows the construction of a system from various parts to satisfy the user's exact requirements. An organisation can purchase different tailor-made components from various vendors, install them in a server runtime environment, and develop their end-user application to make use of these components.

However, the fact that the constituent parts are developed by largely unknown programmers means that the developer of the end-user application will have a limited understanding of the full functionality and expected behaviour of server components. An external tracking facility can therefore be extremely helpful to the programmer during both development and maintenance phases of an application, if informative feedback can be provided about application interaction with the CBS.
- The decision was made to concentrate on Java-based applications since Java has many features which suit tracking purposes admirably. Firstly, Java offers an introspective capability which allows us to dynamically explore details of Java classes. Secondly, the ability to dynamically substitute proxies is essential for external application tracking.

It should be stated at the outset that the aims in providing activity tracking are that it:

- should not interfere with the source code of the application;
- should not make any changes to existing package code, for any of the packages currently being used by the application;

– should require minimal application developer participation, and should even function adequately without it.

The mechanism for dynamically engaging the proxy between the user and the graphical user interface is explained in Sect. 2, while the mechanism for dynamically inserting the proxy between the client application and the rest of the CBS is explained in Sect. 3. The completed prototype is explained in Sect. 4. Section 5 discusses the advantages and disadvantages of the proposed approach. Section 6 discusses proposed future work on this project, and the paper is summarised in Sect. 7.

2 Interception of User Interface Communication

This section describes the mechanism used to insert a proxy, positioned as shown in Fig. 2, which tracks the appearance of, and interaction with, the user interface. The design of the user interface proxy is discussed in Sect. 2.2. How the proxy is used to satisfy our aims is discussed in Sect. 2.3. How information thus obtained is used to track user activity is discussed in Sect. 2.4.

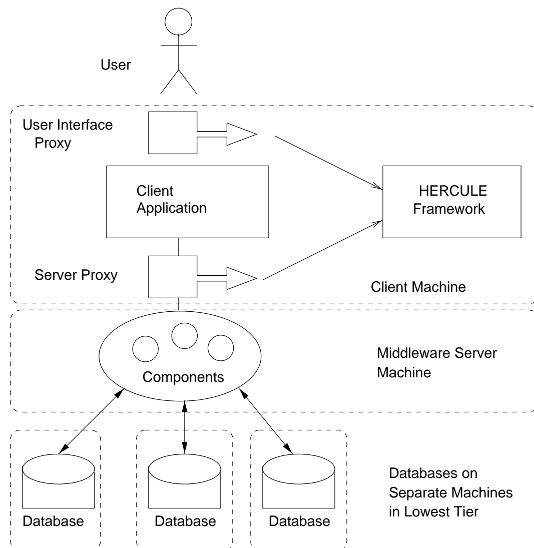

Fig. 2. CBS Application Architecture with Proxies

In order to track user activity, without, for the moment, being language specific, there are two requirements:

1. The first is to build an internal representation of the user interface structure. To do this, there is a need to know about each user interface component being created, and how the user interface is composed. For example, the window shown in Fig. 3 can be represented as a tree structure, as shown in Fig. 4.

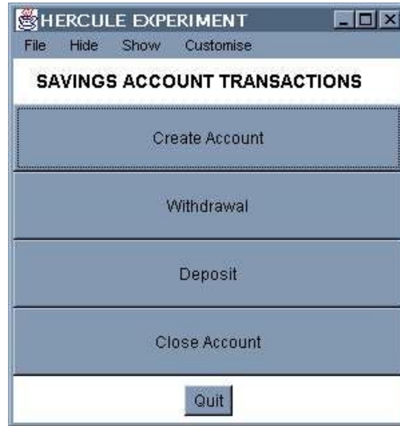

Fig. 3. The Client User Interface

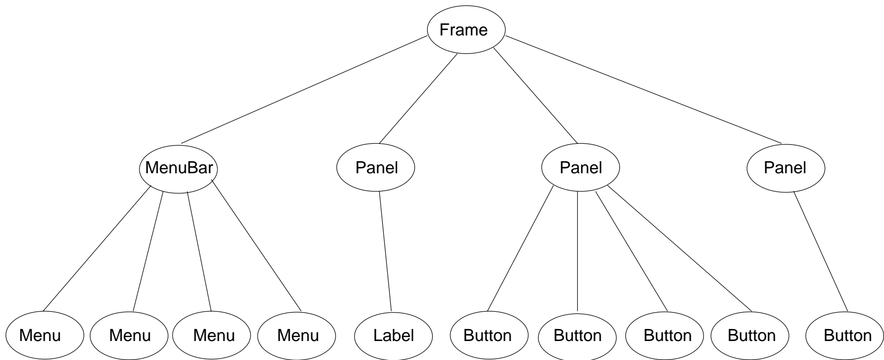

Fig. 4. The Internal User Interface Representation

2. The second is to keep track of user activities at the user interface, and to associate them with the parts of the interface being used. To watch user activities, a tracking facility needs to be notified whenever the user does something at the Graphical User Interface (GUI). Since GUIs are primarily event-based, we can reasonably expect to receive this information in the form of events. So, for example, in the `Window` shown in Fig. 3, the application obviously responds to button activations. In that case, HERCULE also needs to be apprised of button activations.

The user interface tree structure is required so that the user interface events make sense. Without such a structure, it would be impossible to identify windows containing components which have generated events, or to keep track of components within a window being added or removed, or to provide any sort of context sensitive feedback.

2.1 Inserting a Proxy — In Java

This section addresses Java specific issues with respect to inserting a proxy. Java is platform independent, so that a program written to run on one platform, using a particular Operating System (OS), can be executed on another platform using another OS, without alterations.

The way the Java Virtual Machine (JVM) provides this feature for the GUI is by means of a combination of the `java.awt.Toolkit` class and a library of platform dependent `Toolkit` classes. When the Java program instantiates user interface components in order to build a GUI, the actual component will use the `Toolkit` to establish a link to a platform dependent peer.

When the Java application interacts with these `java.awt` objects, the messages are relayed to platform dependent peers, in order to display the required GUI. The JVM handles these details so that the programmer is completely oblivious of the process. The programmer simply instantiates and invokes methods on the `java.awt` objects and subsequent calls to the peer objects are completely invisible. The `java.awt.Toolkit` class has the responsibility for loading these platform dependent classes. This `Toolkit` is loaded automatically by the `java.awt` classes when they are instantiated. A programmer will often never have to make direct use of this class at all. For example, they may do the following:

```
Button quit = new Button("Quit");
```

The `Button` class calls on the `Toolkit` to create a platform dependent peer object, `ButtonPeer`. This object is the actual platform specific object which is displayed on the user interface. If the programmer now calls:

```
quit.setLabel("Cancel");
```

then the `quit` object will call the `setLabel` method on `ButtonPeer` so that the label on the button on the GUI will change. The structure of this activity is shown in Fig. 5.

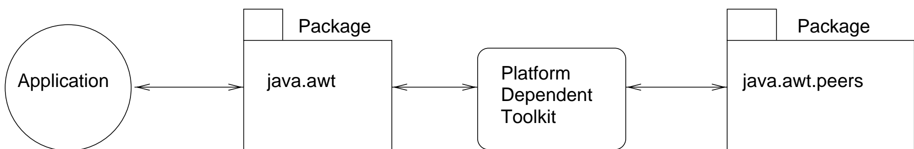

Fig. 5. The Use of the Toolkit to facilitate GUI platform independence

Our aim is to add activity tracking functionality to an application without making changes to either the application, or the `java` packages' source code. One approach would be to generate proxy classes for all the classes in the `java.awt` package, use an auxiliary class loader, and, by an additional level of indirection, substitute the proxy classes for the original classes. This satisfies our requirement that no part of the application should be altered, and it also does not

interfere with the `java.awt` package. Unfortunately we cannot create a proxy for the `java.awt` package, because of the `java.awt.Toolkit` class. A proxy cannot be created for this class since it is abstract, and so the platform dependent `java.awt.Toolkit` and `java.awt` peers are loaded by the *application* classloader. This confuses the original classes which are loaded by their own separate classloader, and they consequently cannot reference the `Toolkit` [32]. Since the `java.awt` package is essential for our purpose in tracking user interface activity, another mechanism must be used.

The previous approach attempted to intercept user interface communications for *each* user interface component. However, an alternative source of information can be found in the `Toolkit` class, since Java requires the creation of all user interface objects be done via this class. Two factors make this a viable proposal:

1. The first is that `Toolkit` is an abstract class. In general, to obtain an instance of an abstract class, you either have to use an instance of a class that extends the abstract class, or use a static method which returns an instance of the class. The `java.awt` package makes use of the abstract class `java.awt.Toolkit`, which provides a static `getDefaultToolkit()` method, which gets the name of the platform dependent `Toolkit` class from system properties, and obtains an instance of that class from the OS libraries, to be returned to the caller.
2. The second, which relies on the first, is that the static `getDefaultToolkit()` method allows the use of an environment variable to specify which `Toolkit` is to be loaded. The `java.awt.Toolkit` incorporates a mechanism to allow the developer to substitute another `Toolkit` for the one which would, by default, be loaded by the JVM.

So, suppose a *proxy* `Toolkit` is written which extends `java.awt.Toolkit`, called `java.awt.ProxyToolkit`. The `EssentialApp` application can be told to use this proxy `Toolkit` by starting the application with the following command line:

```
java -Dawt.toolkit=java.awt.ProxyToolkit EssentialApp
```

The `java.awt.ProxyToolkit` is now instantiated when the application needs an instance of a `Toolkit`, and the proxy is thereby dynamically activated.

2.2 The ProxyToolkit

The `java.awt.ProxyToolkit` class extends `java.awt.Toolkit`. When the application calls the static `getDefaultToolkit` method to get an instance of the toolkit, the `ProxyToolkit` is created, and this `ProxyToolkit` then loads the OS specific `Toolkit`, so that the `ProxyToolkit` acts as a channel, relaying all calls to the platform dependent toolkit, and relaying all return values back to the application. The resulting structure is shown in Fig. 6.

Each method in the `ProxyToolkit` generates *construction* reports which relay information about the structure and composition of the user interface, enabling the construction of an internal structure duplicating each window structure. This structure provides the basis for making sense of user activity reports.

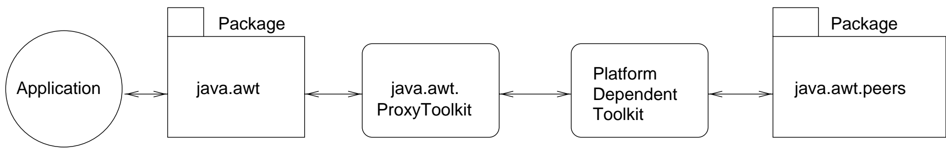

Fig. 6. The System using the ProxyToolkit

2.3 Watching User Activity

Once an internal structure has been created, the next requirement is to be able to keep track of user activities. This can only be done if we are informed when those actions occur. We could, upon learning that a component has been created, declare an interest in all events upon that component. This would mean that we would be interested in every button press, every mouse movement, every key press, window activation, and deactivation, and much more. This volume of reporting would slow the system unacceptably. The second best option is to register an interest in events which interest the application. These events would presumably precipitate some action on the part of the application, and are therefore meaningful activities from the point of view of the user when using that particular application.

All `java.awt` components allow other objects to declare an interest in events on the component by registering as a *listener*. The `ProxyToolkit` will be used to allow HERCULE to register an interest in events of interest, and thereby remain informed of all user activity at the user interface, by the receipt of *event* reports from the `ProxyToolkit`.

The event notifications received as a result of registering as a listener will serve to provide a tangible record of all user activity. Together with the previously defined internal structures representing these windows, the meaningful information can be provided about user interaction with the system.

2.4 Maintaining and Using the Internal Image of the GUI

The *construction* reports generated by the `ProxyToolkit` are used to build up a tree structure, depicting the appearance of the user interface, as shown in Fig. 4. *Event* reports will keep the tracking program informed of all activity, and it will know exactly what the user has been doing at any time, together with the effect on the user interface of that user activity [33]. A history of windows which are shown at the user interface is maintained, as well as a history of user actions which cause a change in the user interface appearance, including changes to "editable" entities like text fields.

3 Intercepting Server Communication

This section discusses the mechanism for inserting a proxy between the client application and the rest of the CBS, positioned as shown in Fig. 2. The figure depicts the CBS as a 3-tier system, which is common, but not essential

for HERCULE's operation. A decision was made to insert proxies to intercept communication with the server components, because this would cause minimal interference with the application. This can also be done transparently, so that it requires no effort from the end-user or programmer.

3.1 Insertion of Proxies

Each server component interface is a potential source of contact with the component, and so each interface requires a proxy. The JVM makes use of a `CLASSPATH` environment variable that can be exploited to ensure that the JVM loads a *proxy* class *instead* of the original class, simply by putting the location of the proxy class ahead of the location of the original class in the `CLASSPATH`. For the purpose of the HERCULE prototype, we choose to implement a proxy insertion mechanism which would work for client applications using the JNDI package to contact the application server. JNDI requires the client application to establish communication with the server housing the server components before any connection can be made with those components. This connection is made by means of the `InitialContext` class. The proxy for this class will be dynamically inserted by making use of the above-mentioned `CLASSPATH` mechanism.

Once the proxy is inserted at this level, it is relatively simple to introduce proxies for each component, since the `InitialContext` object is used to locate required components, and the proxy object can be loaded and substituted for the original component completely transparently.

3.2 Using the Reports Generated by the Proxies

When HERCULE receives the reports from the proxies, they have to be stored so that the information can be retrieved at any time for feedback purposes. It is important to realise that the server proxies are totally unaware of the user interface proxy, and that they therefore have no communication with one another. The only way that HERCULE can link user actions to server method invocations is by using the time factor enclosed within the generated reports. Therefore, when server reports are received, these actions need to be linked to the user actions which preceded them. At this stage, there is a need to clarify the strategy for mapping these separate activities to each other.

Since not all user activity will result in server component method invocations, there could be a number of user actions occurring before a method invocation. In the same way, a whole string of method invocations could be precipitated by a sequence of user actions. In each application thread, the sequence of user actions which precede one or more method invocations is called a *UA-sequence* (User Action sequence), and the series of server calls thus precipitated is called an *MI-sequence* (Method Invocation Sequence). When a UA-sequence is matched to an MI-sequence, we can call this mapping an *Episode*. This is illustrated in Fig. 7.

When storing the proxy information (both user interface and server), it is vital to store it in the form of Episodes, which can then be depicted in the

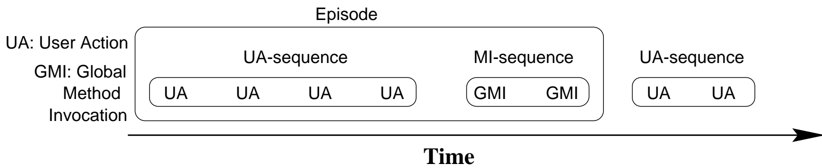

Fig. 7. UA-sequences, MI-sequences and an Episode

feedback. It is still necessary to keep them apart for some specialised feedback requirements, so HERCULE will store a list of UA-sequences, and link each UA-sequence to the MI-sequence precipitated by the UA-sequence. These two lists will be linked one to the other, forming a history of session Episodes.

4 HERCULE Implementation

In order to test the viability of this approach, a prototype of HERCULE was implemented. The prototype was tested on a three tier CBS with *Enterprise Java Beans* (EJBs) [27] fulfilling the role of the server components. The application server used was the Tengah server from Weblogic [38], an all-Java application server. The test system was composed of a client on an NT host running on an Intel, the Tengah server running on Solaris on an Intel, with the third level being made up by a Cloudscape database [40] containing a set of client accounts.

HERCULE tracks application activity by dynamically inserting proxies, and extracts information based on the reports generated by these proxies. The next section discusses the inputs that HERCULE receives from the two proxies. Section 4.2 explains how HERCULE is customised for any given EJB, and how HERCULE extracts the required inputs at runtime. Section 4.3 shows how HERCULE would be used with a running application, and Sect. 4.4 gives an example of how the information gleaned from the proxies can be presented to the user in order to provide a useful tool to programmers, end-users and support staff. Section 4.5 reports on a preliminary study of how the presence of HERCULE affects application performance.

4.1 Inputs into HERCULE

HERCULE basically operates based on two types of inputs. The first is made up of the documentation and Java class files delivered with the EJB. The second comprises the reports generated, at runtime, by the proxies. The following documents are the least we would expect to be delivered along with each server component, since they provide the basic minimum information required to use the component:

1. An *Application Programmer Interface* (API) document, explaining the purpose of the component, and giving details of method functionality, for example, `javadoc` [26] output.

2. One or more interfaces through which the component can be accessed.
3. A deployment document which specifies the context dependencies of the server component, and explains how the component should be deployed.

HERCULE must use the information from this documentation to customise itself. This customisation facilitates the operation of the proxies at runtime. HERCULE receives two types of reports from run-time invoked proxies:

1. *User Interface Reports*: signaling events and the user interface construction. These events enable HERCULE to keep a history of user interface appearance and user activity.
2. *Server Method Invocation Reports*: The reports received here indicate different stages of communication, including initial establishment of communication with the server and global method invocations following thereafter.

Once the UA-sequences have been linked to the MI-sequences, and the Episodes constructed, the results need to be depicted in a helpful manner on the screen. There are many aspects of this interaction that could be depicted, but for HERCULE, the decision was made to depict the success or failure of each Episode. This decision was made because our focus is to provide end-user feedback, and the success or failure of an Episode is of critical interest to the end-user. Since a particular application session could easily generate many Episodes, the display chosen has some important characteristics:

– it should be able to depict either one or many Episodes in a clear manner, so that the user can obtain as much information as possible at a glance;
– it should not intrude, but offer user assistance. Thus, it should use as little screen space as possible. Many feedback devices tend to become overpowering, and the last thing we want to do is to annoy; and
– it should allow the user to step backwards in time to view and confirm their previous actions.

4.2 HERCULE's Phases

HERCULE has two distinct phases of use, discovery and runtime. The discovery phase is a *customisation* phase, which serves to inform HERCULE, essentially a generic feedback mechanism, of the server components which will be used by an application. During the *runtime* phase, the results of the customisation will be used to facilitate the required feedback.

The customisation is done prior to HERCULE being used, and as often as necessary after that as the programmer becomes more familiar with the operation of the component. HERCULE makes use of the server component documentation to customise the framework for a particular server component. The component documentation is "mined" in order to extract *descriptor objects* which hold semantic details about the methods used to access the server components, and to generate proxies. If we consider these documents to be the *input* to the discovery process, then the output we produce is:

- *proxy classes* used in intercepting communication with the server.
- *descriptor objects*: which are essential to the visualisation of session activity. Tracking will only be meaningful if its results can be depicted in an information-rich and useful fashion. In order to provide the users with explanations of server activity, the method invocations should be described in terms easily understood by the user, rather than in language familiar to the programmer of the system. These explanations are all to be found in the server component API documentation, and the initial descriptor objects will thus be derived from these documents. Since Java class documentation is generally produced by `javadoc`, this makes the mining process simpler[2]. This mining process should produce at least an *adequate* descriptor object, since it contains the information as obtained from the API document. In order to improve this object, HERCULE provides a tool to allow the programmer to augment the descriptor object. With the programmer's assistance the descriptor object can be augmented from *merely adequate*, to *substantially helpful*.

4.3 HERCULE in Action

An example of how HERCULE would be used by the user of an applicationn will now be given. HERCULE runs in a separate process so that its execution and termination are not dependent on the application. The HERCULE console operates in the following modes:

- *waiting mode* as it waits for the application proxies to make contact.
- *dynamic feedback mode* after the application has made contact via the proxies, and while it executes.
- *static feedback mode* after HERCULE registers the termination of the application but stays active so that the user can use the console to provide post-execution feedback, if required.

We will consider the case of two different users — the programmer and the end-user.

The Programmer. The programmer will run the discovery process to customise HERCULE for the server components which will be used by the application. Descriptor objects and proxies will be generated for each interface of each component. These will automatically be compiled and made available to the JVM.

The programmer would then start execution of HERCULE, and the HERCULE console would appear on the rightmost side of the screen. An icon would appear at the base of the screen if it is being run on a Windows platform.

The programmer now starts execution of their program. HERCULE tracks the application activity and uses the generated descriptor objects to provide an

[2] If this is not done by `javadoc`, it becomes difficult to mine since we have no idea how the documentation would be structured. The next EJB specification requires the use of XML for this documentation, which would make the process even simpler.

explanation of method invocations. If the programmer needs to augment these, the discovery process can be executed again, and the explanations changed. If a method invocation could result in an exception being thrown, the programmer can also enter meaningful explanations for these exceptions.

Once the application program shuts down, the traffic lights display on the HERCULE console will display a message indicating that the application has finished executing. This is done so that the information about the session is still available even if the application crashes. The programmer will then have a tangible record of inputs given, together with details of method invocations and results returned by the server components.

Should the programmer want to track a new session, HERCULE can be reset by choosing a `reset` option from the console menus.

The End-User. The end-user starts execution of HERCULE, and the HERCULE console appears on the rightmost side of the screen.

HERCULE tracks the application and displays the Episodes dynamically. If the user chooses not to actively use HERCULE, it will remain unobtrusively in the background. If the user needs an explanation, HERCULE can be made active, and will provide the user with an explanation. Should this still be inadequate a support person can be summoned, and the record of activities and explanations can be used to solve the problem.

4.4 Feedback Provided

In Sect. 4.1 some criteria for the feedback display were mentioned. The console designed for HERCULE was created with those requirements in mind, and satisfies them as follows:

- it depicts the all Episodes for the entire application session in one window;
- it allows detailed information about Episodes to be obtained at the click of a mouse;
- it depicts a great deal of information in a small screen space;
- it does not intrude, but is always available as an icon, offering the possibility of obtaining feedback at any time; and
- it allows the user to obtain information about previous Episodes at the click of a mouse.

At runtime, the HERCULE console (Fig. 8) provides the following information, which is dynamically updated as the user works:

1. A traffic lights widget depicting the current system state. This will display:
   - red when the application cannot be tracked. The legend beside the traffic light will display the result of HERCULE's attempt to diagnose the cause;
   - orange when a component is busy servicing a request; and
   - green when HERCULE is in dynamic feedback mode.

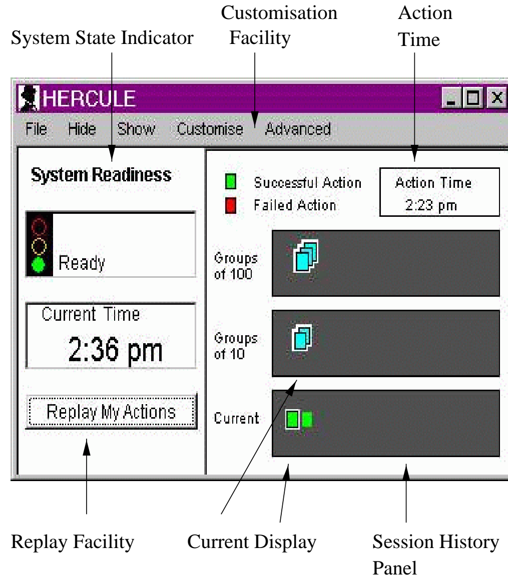

Fig. 8. The HERCULE Console

2. A `Replay My Actions` button will summon a playback facility which allows the user to view a screen replay of all UA-sequences as they took place. This is done in the form of the windows displayed by the application being shown to the user, one at a time, with a highlight on the action which caused the transition to the next window. For example, in the Window in Fig. 3, if the user clicked on the `Create Account` button, that button would be highlighted by setting the background colour to yellow in the replay window. This serves to remind the user of past actions. By providing this functionality, HERCULE supports users by alleviating their weaknesses (such as limited working memory), and capitalising on, and utilising their strengths (such as swift pattern recognition, and retrieving relevant information about the meaning of these patterns quickly). This replay has no effect on the application whatsoever, in accordance with the non-intrusion policy, and should be considered to be rather like an action replay used in television sports broadcasts.
3. A session history panel which presents all Episodes, displayed in three separate panels:
   - the bottom panel displaying the last ten Episodes,
   - the middle panel depicting groups of ten Episodes, and
   - the top panel depicting groups of hundreds of Episodes.

   Each distinct Episode is displayed as a coloured rectangle. This depicts the result of the *MI-sequence* resulting from the Episode UA-sequence as:
   - red if it failed — assumed if the server throws an exception,
   - yellow if the outcome is pending, and
   - green if it succeeded — assumed by the absence of an exception.

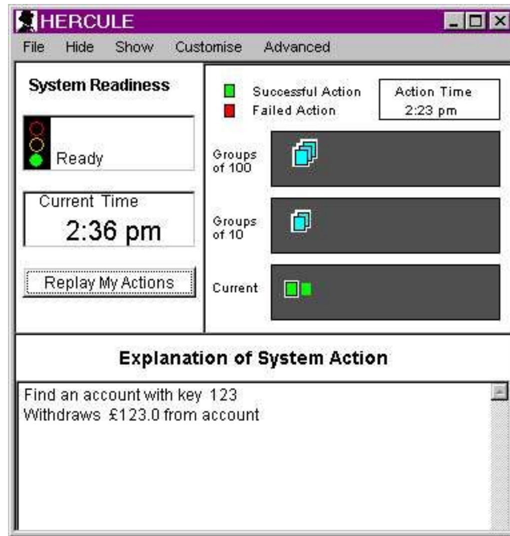

Fig. 9. The User Viewing an Explanation of a Previous Episode

4. In order to provide for different types of user needs, HERCULE can be customised to give extended feedback. The type of user feedback needs already identified are:
   a) explanation of an MI-sequence. In other words, what the system did as a result of a UA-sequence. This is primarily an end-user need, and is met as shown in Fig. 9.
   b) assistance in debugging. A detailed explanation of the method invocations making up the MI-sequence, together with the parameters for each invocation, and the return value or exception thrown.
   c) performance monitoring, as shown in Fig. 10. For example, a graphical display of response times is a good indicator of application performance with respect to communication with the rest of the CBS. This feedback component could be of use to system support.

   There are two aspects of these feedback components to be considered:
   a) *Which Episode the feedback applies to* — The first and second of the above-mentioned feedback needs should be met by displaying an explanation of the most recent Episode, or a previous one, as required by the user. Feedback should be provided for the most recent Episode by default, allowing the user to request the display of the explanation of a previous Episode by clicking on one of the previous blocks in the lower panel. Feedback can be obtained for Episodes not shown in the lower panel by clicking on one of the grouped symbols, as depicted by groups of blocks in the middle and upper panels. You will note from Fig. 8, that each panel displays some rectangles, and that one is highlighted. The highlighted rectangle indicates the Episode for which feedback is currently being given in the visible feedback components. The *Current*

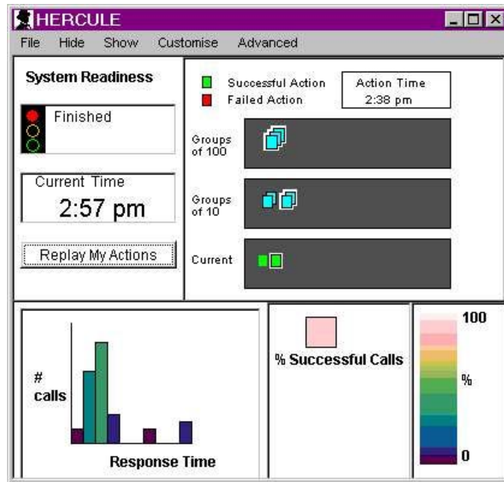

Fig. 10. The HERCULE Console showing the Support Panel

Display label in Fig. 8 points to the highlighted rectangles in the history panels, indicating that the second last Episode MI-sequence explanations would be displayed (if any feedback components were visible).
The user's immediate feedback requirements with respect to individual Episodes, as indicated by clicking on a block which represents a previous Episode, will be met by dynamically reflecting the feedback for that Episode in the information displayed by each of the visible feedback components. On the console shown in Fig. 9, the Episode actions are explained as `Withdraws 123.0 from account`. This is not the explanation of the most recent Episode MI-sequence, since the rectangle in the last but one position is highlighted, indicating that the explanation belongs to the second last Episode.

b) *How additional feedback components are plugged into the system* — The HERCULE console is dynamically extensible, so that the identification of a new user feedback need can be accommodated. New HERCULE feedback components can be coded, and plugged into the HERCULE console at runtime. The top section of the console, as shown in Fig. 8, will always be displayed, since it provides the core functionality of the console display. When a programmer wants to add a new feedback component, the `Advanced` menu is chosen, and the programmer clicks on `Plug new Component in`. The class name of the new component is entered, and HERCULE will register the existence of this component. The user can choose which components should be visible or invisible by using the `Show` and `Hide` menus, and the user can save their overall preferences with respect to visibility of tailored feedback components, so that the console is customised to satisfy their particular needs. The `Customise` menu is reserved for customisation such as user preferences with respect to feedback components displayed, and choice of whether a failed epi-

sode should be brought to the user's attention by an audio signal or not. To support this extensibility of the HERCULE console, the following are provided:

- An abstract class named `HerculeComponent` (which extends `java.awt.Panel`). This class must be extended by any feedback component to be incorporated into the HERCULE console.
- A `HistoryListener` interface. The feedback component implements this interface, and registers as a listener with the history panel. The feedback component will then be notified of user actions at the history panel, which will enable it to provide relevant feedback. The feedback component will implement this interface if it is going to provide dynamic feedback related to a specific Episode.
- An `OutcomeListener` interface. The feedback component implements this interface, and registers as a listener with the history panel. The feedback component will then be notified of the outcome of Episode MI-sequences. The feedback component will typically implement this interface if it wants to provide statistics about the number of successful Episodes, or performance.

4.5 Performance Reduction

It is very important that the presence of HERCULE should not affect application performance unacceptably. Since HERCULE inserts proxies between the user and the user interface, and between the user and the rest of the CBS, we can expect any performance degradation to take place:

1. when the `Toolkit` is being loaded, since an extra level of indirection is being introduced;
2. when the initial connection with the server is being forged, since this is where the server proxy will be introduced;
3. whenever a new window is being constructed; and
4. when global methods are invoked on distributed components. Two types of methods need to be considered independently, methods which will require action by the component container, and methods on the component itself. The former take longer than the latter to process.

A preliminary study of performance differences was undertaken, by running the example application twenty times both with and without HERCULE. Where effects were observed, the results are shown in the table below[3].

Action	Without Proxies	With Proxies
Display of Initial Application Window	1.44	2.45
Initial Contact with Server	5.92	8.73
Container Method Invocation	1.40	1.56
Component Method Invocation	0.25	0.54

[3] There was no discernable effect when new windows were constructed, with only the time taken for the initial window being affected.

It is clear that the user *will* have to pay a penalty for using HERCULE. It would be unreasonable to expect otherwise. The entire insurance industry is based on the "present pain, future gain" principle. Shneiderman [34] cites research which shows that modest variability in response times is deemed by users to be acceptable. If the user sees the benefits of using HERCULE, they will hopefully be prepared to pay the small penalty of slightly longer response times, for the future gain of having informative and extensive feedback available. Finally, it can be expected that the negative impact of HERCULE will soon become less apparent due to the increased efficiency and speed of hardware.

5 Pros and Cons of this Approach

End-user feedback needs are often not adequately catered for. Traditional approaches to providing feedback have required the application programmer to incorporate it in the end-user application, or to provide it by means of an add-on facility like an online manual. Manuals cannot hope to supply dynamic feedback, but at best can only provide static help and explanations. The only other way to provide feedback is by means of a set of libraries somewhat like those provided by the Garf tool [17], which a programmer could use to provide feedback. Feedback is quite unlike distribution, however, since it is far more demanding and pervasive than distribution. The guidelines for providing feedback advise that *every* operator action should have accompanying feedback [34], and that the user should be continuously informed [2, 28], which means that the load on the programmer is substantial. The advantages of the HERCULE approach are that:

- it requires minimal participation from the programmer;
- it frees the programmer from a heavy load of catering for feedback continuously;
- it can be disabled or enabled as required, since it does not require the addition of any code to the application;
- it provides a uniform, customisable feedback for different applications;
- the extensibility of the HERCULE console makes it easy to accommodate changing user needs; and
- the tendency of distributed systems to indeterminate failures makes it useful to have a standard way of indicating that an error has occurred, and for explaining the causes.

The disadvantages of the approach are that:

- it can only give feedback based on the external activities of the application. Thus the feedback that can be provided is limited to the interaction of the application with the user and the rest of the distributed system; and
- it requires the use of a language with introspective capabilities, since this is essential for the generation of proxies.

The HERCULE approach was chosen since the primary aim was to simplify the programmer's task of feedback provision. This had to be done with minimum disruption to the normal application development. The other important aim was that this facility should be easily disabled or enabled. HERCULE satisfies these requirements.

6 Future Work

An Episode has been defined as a UA-sequence followed by an MI-sequence, and HERCULE presently links the UA-sequences to the precipitated MI-sequences by using the time in each report. This will obviously cause problems in multi-threaded applications. Work is underway to design and implement a version which will cater for multi-threaded applications. We are currently further evaluating the negative impact of HERCULE on application performance. We are also planning user evaluation trials of HERCULE to measure user reactions to it. The querying capabilities of the console are presently limited to stepping through the Episodes one at a time. It would be beneficial to extend the system to group like Episodes to enhance querying facilities.

7 Conclusions

This paper has shown that it is indeed possible to augment the feedback provided by an application by tracking application activity and providing the feedback by means of a generic framework. The HERCULE framework uses the information gained from this tracking to provide different users of the application with a visualisation of their session activity. The scheme has been proved to be viable by means of the implementation of the HERCULE prototype.

The mechanism for dynamic insertion of proxies detailed in this paper are necessarily applicable only to Java systems using JNDI. That does not mean that it cannot be done in other application architectures. Insertion of the user interface proxy is done relatively simply in Windows systems, by means of hooks explicitly provided by the Microsoft OS. The mechanism for insertion of the server proxy is easily extended for other types of CBS architectures, since communication protocols in these systems are fairly standardised. For example, there are a few techniques which will cover insertion of server proxies for most CBSs:

- COM+, the Microsoft component model, and CORBA (the OMG model) allow the specification of interceptor components between a client application and server components [14].
- Browsers (often housing client applications) commonly make use of proxies, and many generic proxies, which can be tailored to specific needs, are widely available.
- Two widely used protocols would allow the insertion of a proxy using the `CLASSPATH` mechanism as follows:
  - The Java Naming and Directory Interface(JNDI) protocol — already explained in Section 3.1.

– The Remote Method Invocation (RMI) protocol — The `Naming` class is used to establish contact with services offered on any machine, and so a proxy `Naming` object can be used as the point of insertion.

There are some issues which need to be considered when advocating a scheme such as the one outlined in this paper. The first is whether the support structure for applications should allow the normal execution environment to be replaced by one which reports on user activities. The second issue raises the question of whether the end-user will resent HERCULE — seeing it as some sort of spying device — or whether it will be perceived as a worthwhile feedback provider. These issues are outside the scope of this paper, but nevertheless constitute an interesting research area.

Acknowledgements. Special thanks to my supervisor, Richard Cooper, for his help and guidance. Thanks too to Huw Evans and Ela Hunt for their extremely helpful comments on the draft of this paper. I also acknowledge the valuable contributions of James Begole and Susan Spence. This research is supported by a scholarship from the Association of Commonwealth Universities and a grant from the Foundation for Research and Development in South Africa, and the University of South Africa. I am currently on leave of absence from the University of South Africa and I would like to acknowledge their magnanimity in allowing me this extended period of absence.

I gratefully acknowledge the donation of the Tengah server from Weblogic/BEA (`URL: weblogic.beasys.com`), and express my appreciation for their prompt responses to my queries. This work could not have been carried out without their kind donation.

References

[1] ACM. *1998 International Workshop on Component-Based Software Engineering. Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Kyoto, Japan, April 25–26, 1998 1988. URL: http://www.sei.cmu.edu/cbs/icse98/papers/index.html

[2] *Human Interface Guildelines: The Apple Desktop Interface.* Addison-Wesley, Reading, Massachusetts, 1987. Apple Computer Inc.

[3] http://www.parc.xerox.com/spl/projects/aop/aspectj. AspectJ Web Page, 1998.

[4] T. Ball and J.R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, 1994.

[5] L. Blackshaw and B. Fishhoff. Decision making in online searching. *Journal of the American Society for Information Science*, 39:369–389, 1988.

[6] N. Borenstein. *Programming as if People Mattered.* Princeton Univeristy Press, Princeton, New Jersey, 1991.

[7] A.N. Burton and P.H.J. Kelly. Workload characterization and using lightweight system call tracing and re-execution. In *IEEE International Performance Computing and Communications Conference. IPCCC '98*, Phoenix/Tempe, Arizona, USA, February 16–18 1998. IEEE.

[8] A.N. Burton and P.H.J. Kelly. Tracing and reexecuting operating system calls for reproducible performance experiments. *Computers and Electrical Engineering: An International Journal*, May 1999.

[9] F.R. Campagnoni and K. Ehrlich. Retrieval using a hypertext-based help system. *ACM Transactions on Information Systems*, 7:271–291, 1989.

[10] J.M. Carroll, editor. *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction.* MIT Press, Cambridge, MA, 1987.

[11] J.M. Carroll and M.B. Rosson. The paradox of the active user. In *[10]*, chapter 5, pages 80–111. 1987.

[12] M. Chalmers, K. Rodden, and D. Brodbeck. The order of things: Activity-centred information access. In *Proceedings of the 7th International Conference on the World Wide Web*, pages 359–367, Brisbane, Australia, Oct 5-7 1998.

[13] H.C. Chan, K.K. Wei, and K.L. Siau. The effect of a database feedback system on user performance. *Behaviour and Information Technology*, 14(3):152–62, 1995.

[14] D. Chappell. Com+. WEB Document, April 1998. www.chappellassoc.com/articles.htm.

[15] C. Dellarocas. Toward exception handling infrastructures for component-based systems. In *[1]*, 1998.

[16] T. Fawcett and F. Provost. Activity monitoring: Noticing interesting changes in behaviour. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Diego, CA USA, August 15–18 1990. ACM.

[17] R. Guerraoui, B. Garbinato, and K.R. Mazouni. Garf: A tool for programming reliable distributed applications. *IEEE Concurrency*, 5(4):32–39, October/December 1997.

[18] C. Jeffery, W, Zhou, K. Templer, and M. Brazell. A lightweight architecture for pro- gram execution monitoring. In *ACM SIGPLAN/SIGSOFT workshop on Progra Analysis for Software Tools and Engineering*, Montreal, Canada, June 16 1998. ACM.

[19] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2):121–150, May 1987.

[20] M.A. Kersten and G.C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. Technical Report TR-99-04, Department of Computer Science, University of British Columbia, March 31 1999. Wed, 07 Apr 1999 21:31:26 GMT.

[21] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es):154–154, December 1996.

[22] P. Leibscher and G. Marchionini. Browse and analytical search strategies in a full-text cd-rom encyclopedia. *School Library Media Quarterly*, Summer:223–233, 1988.

[23] C. Lewis. Understanding what's happening in system interactions. In D.A. Norman and S.W. Draper, editors, *[30]*, chapter 8, pages 171–186. Lawrence Erlbaum Associates, Publishers, Hilldale, New Jersey, 1986.

[24] X. Lin, P. Liebscher, and G. Marchionini. Graphical Representations of Electronic search Patterns. *Journal of the American Society for Information Science*, 42(7):469–478, 1991.

[25] G. Marchionini. Information-seeking strategies of novices using a full-text electronic encyclopedia. *Journal of the American Society for Information Science*, 50:54–66, 1989.

[26] Sun Microsystems. javadoc - The Java API documentation Generator. Web Document. http://java.sun.com/products/jdk/1.3/docs/tooldocs/solaris/javadoc.html.

[27] Sun Microsystems. Enterprise Java Beans Specification. Web Document. URL: java.sun.com/products/ejb, March 1998.
[28] J. Nielsen. *Usability Engineering*. AP Professional, Boston, 1993.
[29] D. Norman. The “problem” of automation: Inappropriate feedback and interaction, not “overautomation”. Technical Report ICS Report 8904, Institute for Cognitive Science, University of California, San Diego, La Jolla, California, 92093, 1989.
[30] D.A. Norman and S.W. Draper, editors. User Centred System Design. *New Perspectives on Human-Computer Interaction*. Lawrence Erlbaum Associates, Publishers, Hilldale, New Jersey, 1986.
[31] J.R Olsen. Cognitive analysis of people’s use of software. In *[10]*, chapter 10, pages 260–293. 1987.
[32] K.V. Renaud. A Non-Invasive Mechanism for Monitoring Calls to Java Packages. Technical Report TR-1999-32, Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow, G12 8RZ, April 1999.
[33] K.V. Renaud. Tracking activity at the user interface in a Java application. Technical Report TR-1999-33, Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow, G12 8RZ, April 1999.
[34] B. Shneiderman. *Designing the User Interface*. Addison-Wesley, Reading, Massachusetts, 1998.
[35] M. Siegle and R. Hofmann. Monitoring program behaviour on SUPRENUM. In *International Conference on Computer Architecture. Proceedings of the 19th Annual International Symposium on Computer Architecture*, Queensland, Australia, May 19–21, 1992 1992. ACM.
[36] R.N. Taylor and G.F. Johnson. Separations of concerns in the Chiron-1 user interface development and management system. In Stacey Ashlund, Ken Mullet, Austin Henderson, Erik Hollnagel, and Ted White, editors, *Proceedings of the Conference on Human Factors in computing systems*, pages 367–374, New York, 24–29 April 1993. ACM Press.
[37] H. Thimblebey. Combining systems and manuals. In J.L. Alty, D. Diaper, and S. Draper, editors, *People and Computers VIII HCI’93*, pages 479–88, 1993.
[38] A. Thomas. Selecting Enterprise JavaBeans Technology. Prepared for WebLogic, Inc., July 1998. http://www.beasys.com/products/weblogic/server/papers.html.
[39] J.G. Trafton and D.P. Brock. Simplifying interactions with task model tracing. ACT-R Summer School, Psychology Department, Carnegie Mellon University, June 1996.
[40] S. Willett. Cloudscape Woos VARs for Java Database. *Computer Reseller News 6-99*, June 1999. http://www.crn.com/search/display.asp?ArticleID=7017.
[41] D. Wybranietz and D. Haban. Monitoring and performance measuring distributed systems during operation. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 197–206, Santa Fe, USA, May 1988. ACM.