

# HERCULE: Monitoring Component-Based Application Activity and Enabling Post-Implementation Tailoring of Feedback

Karen Renaud

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

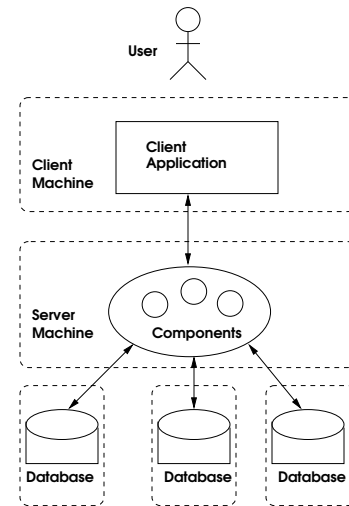
## Abstract

*This paper presents a novel approach to monitoring end-user application activity in a distributed, component-based application. Such monitoring can support the visualisation of user and application activity, system auditing, monitoring of system performance, and the provision of feedback. A framework is provided which, by means of reflection, allows the insertion of proxies into a three-tier component-based system. Proxies are inserted dynamically 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 monitoring is effected, how information produced by this monitoring can be used to provide a visualisation of application activity, and how this mechanism enables the enhancement and tailoring of feedback throughout the software development life cycle. The viability of this approach is demonstrated by means of a prototype implementation.*

## 1 Introduction

*Component-Based Systems (CBSs)* are composed of independently produced and tested components, offering an alternative between standard and customised software. CBSs are typically structured as a distributed three-tier architecture, as shown in Figure 1, with presentation software at the end-user level, server components in a component runtime environment forming the middle tier, and multiple databases making up the lowest tier. [17]

The use of independently produced components makes the provision of other essential facilities more complex. This paper will discuss problematic issues of component-based systems, with respect to providing



**Figure 1. The Three Tier Architecture**

for user feedback needs in a distributed CBS, and suggest a way of alleviating them. Feedback is even more important in a CBS because:

- CBSs will usually be distributed, which introduces an extra realm of unpredictability to the system. This is exacerbated by busy or partitioned networks and overloaded hosts. The end-user is often whether to regard the resulting delays as an error, or not.
- Many different people will be involved in the production of a component-based software system, and few of these people will ever consult with each other.

There is, however, the difficulty of developing a global strategy of any kind — this will apply to explanations of system actions, error reporting, recovery, feedback, help and systems support equally. Each developer/programmer will develop their server component in isolation, and decide, using their own judgement, how and when to report

errors to the end-user. They will also use their own discretion to decide what level of detail is put into the component documentation, and whether or not the component provides an online manual.

The user's experience with a CBS can be made far more rewarding by increasing the level of feedback so as to continually inform and advise about the true state of the system. Providing this level of feedback in systems composed of independently produced, probably distributed, components is difficult, and will not happen without a great deal of work.

Furthermore, feedback needs to be pitched towards the particular user's needs. Any CBS has different types of users during successive stages of the life cycle of the application. We identify three distinct categories, as differentiated by their different roles. The first user is the *application programmer*, who will be creating the end-user application. The next is the *end-user*, the client for whom the application has been created. The third is the *system support person* responsible for providing technical assistance and error intervention to end-users. Each type of user has very different feedback needs:

- The *application programmer* will need highly technical feedback. The goal of the programmer is to produce a working application and the feedback provided must therefore assist in the debugging process. The type of feedback required could be the parameters provided in a particular method call, or the return value supplied, or a stack trace of an exception thrown by a method call.
- The *end-user* needs to be given feedback relating to specific goals, linked directly to the task being carried out. The feedback must be on a much higher level than that required by the programmer.
- The *system support* staff will often be summoned when the end-user has received a message from an application which is indecipherable, or due to an error message indicating some sort of problem. The first question asked by system support staff will be: "What were you doing?" followed by, "What message did the system display?". This will assist them in tracking down the source of the problem.

It is extremely difficult for an application to provide for *all* these different user needs. One does not have to look far to find evidence of this failing in many applications currently in use today. In order to assist the developer of the application in meeting these needs,

I propose the use of a feedback enhancing framework within which an application can execute to enhance feedback. A framework, defined as "*a large design pattern capturing the essence of one specific kind of object system*" [7], should specify mechanisms for observing the activities of the application, and for making sense of this data to provide significant feedback during a processing session.

The creation of such a framework capitalises on built-in features of CBSs which support the discovery of important details about the server components being used, enabling the framework to give the appropriate feedback.

The remainder of this paper will elaborate on the need for, and logistics of, providing such a framework. Section 2 will make the case for having a feedback framework, Section 3 will explain our approach with respect to providing the framework. Section 4 will give details about the implementation and portrayal of the required feedback. Section 5 concludes the paper.

## 2 Feedback in Component-Based Systems

### 2.1 The User Perspective

Users, especially novice users of an application, often have no idea of how to use the application. They need to build up an internal model of how the application will allow them to carry out their tasks. However, they tend not to read manuals, wanting rather to find out for themselves how the system works. They also tend to be impatient to get on with their task, and don't want to spend hours being taught how to use a system. This tendency is less risky if the relevant information is easy to find. 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. [1, 3, 8] Users especially need feedback because they have severe limits with respect to [12]:

- perception — perceiving small differences 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
- 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 [12]:

- 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.

In addition to alleviating the effects of users' cognitive shortcomings, Chan, Wei and Siau [4] have also shown that an active feedback system greatly *improves* user performance.

## 2.2 The Component Perspective

There are several *independent* groups of stakeholders participating in the production and use of a CBS. The first group is the component developers who design and develop the component. For the purpose of this paper, it can be assumed that the systems analyst, another stakeholder, will have modeled the system and that the server components will be installed into the middle tier of the CBS.

The final step in producing the software system is the programming of the client program which interacts with the user via the *Graphical User Interface (GUI)*. The programmer is given the server component interface documents and API documentation, and will produce the client application.

The programmer need not have any intimate knowledge of the component implementation. In the process of developing the application, the programmer will no doubt develop a fuller understanding of how each component operates, but cannot hope to anticipate the potential for erroneous execution, or possible pitfalls in using this component. Many of these pitfalls will only become apparent once the application is in use.

The most important stakeholder in this exercise is the end-user, for whom all the preceding work has been done. If the system does not meet his or her needs all the work has been done in vain.

## 2.3 Consolidation of two Perspectives

To the end-user, the application seems to run on the computer in front of them — when it runs correctly. If the GUI displays an error message, or is subject to long delays, the end-user is often at a loss as to the cause of the problem. User feedback needs can be summarised as being three-fold:

1. *Confirmation*: The user needs to know whether their input has been accepted by the computer, and whether their request is being processed;
2. *Explanation*: The user must understand what the consequences of their actions are. This includes:
  - (a) understanding how the application interpreted the inputs that were entered, and what was done as a result;
  - (b) assistance in handling errors [14]:
    - i. awareness of error occurrence;
    - ii. understanding the nature of the error; and
    - iii. understanding how to recover from the error.
3. *Reassurance*: The user needs to know whether the application is in working order. Even though they are not aware of it, this means that the entire three-tier application system should be in working order. Since users often have no understanding of the true distributed nature of the system it is no wonder that they are stymied by messages about computers that they had no idea they were even using.

The application programmer is expected to provide for the feedback needs of at least three completely different types of users. Many applications in use today are evidence of the variability of this provision by different programmers. The reasons for this are complex and beyond the scope of this paper, so only two of them will be briefly discussed.

1. The programmer belongs to the world of technology, and finds it hard to conceive of users who do not have this understanding. It therefore is extremely difficult to provide feedback at the level required by the user. Since there is a shortage of programmers with specific training in human-computer interaction [9], it is realistic to expect that most applications will fall short of the ideal level of feedback.

2. Grudin [6] published a paper which looked at the issue of technologies where one person did work for which another person would reap the benefits. This was coined by Norman [10] as *Grudin's Law*:

“When those who benefit are not those who do the work, then the technology is likely to fail, or, at least, be subverted.”

The programmer achieves little benefit from providing the right level of reporting for other types of user. Indeed, some organisations actually *profit* from software which provides inadequate feedback — by requiring users to pay for advice on using their systems.

## 2.4 What Feedback, and How should it be Provided?

The inclusion of the following features would greatly enhance feedback levels in CBSs:

1. an indicator which shows whether the application is either waiting for input, busy servicing a request, or unable to handle requests [5]. Planas and Treurniet [13] have shown that continuous feedback reduces annoyance with respect to slow responses;
2. a facility which allows the user to step back through their actions to confirm inputs provided to the application [18];
3. a record of actions, and a history of the success or failure of these actions with respect to server method invocations [16];
4. an explanation of what user inputs mean to the system, to assist in building up a correct internal model of application functionality [5];
5. a mechanism for updating and augmenting explanations after the system has been delivered — if users have difficulties with a particular part of the system; and
6. provision of feedback pitched towards the end-user [16].

There are a number of ways in which these features can be provided:

1. Programmers could provide this feedback while implementing the application. This is the best solution, since the feedback can be perfectly pitched towards the user's interaction with the application. The previous discussion has argued that this

is unlikely, but even if it were provided, there is the issue of duplication since each distributed application needs a core of generic feedback-producing activities.

2. An online help system could be provided. This is unlikely to be tenable in CBSs since there is currently no standard for components to adhere to which requires that they provide an online help facility for their component. Even if provided, the diversity of the different component producers would not facilitate a coherent, understandable help facility. The issue of dynamic feedback is not catered for by online help.
3. The last option is to provide feedback in a generic manner not specific to the particular application. This enables reuse of feedback facilities and makes the application programmer's task much simpler. Since such a framework is independent of the application, it must employ observation techniques to gauge the operation of the application, and be able to detect user and system activity. A mapping between these two activities must be derived in order to provide meaningful feedback. Such a framework cannot provide feedback with respect to internal application execution, since it has no idea of the semantics of the application — it can only observe the application's interactions with the environment, and provide feedback based on the information gleaned from these observations.

Since the first and second options are clearly not viable, the third and last option — that of providing the feedback independently of the application — was developed, using Java, as a working prototype, called **HERCULE**<sup>1</sup>. This prototype will be discussed in the remainder of the paper.

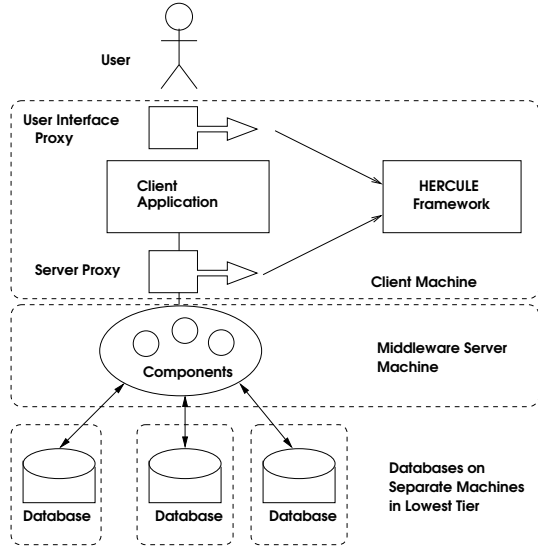
## 3 Approach to providing HERCULE

HERCULE is a feedback enhancer providing the user with helpful and information-rich confirmation of actions as well as explanations of system actions. Being a generic feature, it also frees the application programmer from providing those feedback features which are common to all distributed CBSs, leaving the programmer to concentrate on application specific feedback and error management.

HERCULE has two enabling features: application monitoring, and session visualisation. To provide the visualisation, it must get information about what the

<sup>1</sup>Named after *Hercule Poirot*, Agatha Christie's legendary detective.

application is doing at any time, and it must be able to assign some semantics to this activity. HERCULE therefore collects two distinct types of information in order to perform its function:



**Figure 2. CBS Application Architecture with Proxies**

1. *Observation Reports* — HERCULE needs to watch all application interaction with its environment — both with the user and with the server components. This activity must not interfere with the operation of the application, and should not require application participation. The method used is to insert *proxies* as shown in Figure 2:
  - (a) one between the application and the user interface. Reports generated by the proxy are used to build up a tree structure, depicting the appearance of the user interface. HERCULE keeps track of the user activity context by maintaining a history of windows which are shown at the user interface, and also keeps track of user interface activities, and user actions which cause a change in the user interface appearance [15].
  - (b) one between the application and the server components. All calls to the server, responses and times of completion are recorded.

These features are implemented in Java since this provides systematic mechanisms for intercepting method calls, and introspecting over the objects being transmitted. Having collected these two sets of information, the particular task of HERCULE is to relate them. This requires some effort, since

server proxies are totally unaware of the user interface proxy and visa versa. To provide feedback, server actions must be linked to the user actions which preceded them. The only way that HERCULE can link user actions to server method invocations is by using the time factor enclosed within the generated reports.

Since not all user interface activity will result in server component method invocations, there could be a number of user interface activities occurring before a method invocation. In the same way, a whole string of method invocations could be precipitated by a sequence of user interface activities. The sequence of user interface activities which precede one or more method invocations is called a *UI-sequence* (User Interface sequence), and the series of server calls thus precipitated is called an *MI-sequence* (Method Invocation Sequence). When a UI-sequence is matched to an MI-sequence, this mapping is called an *Episode*.

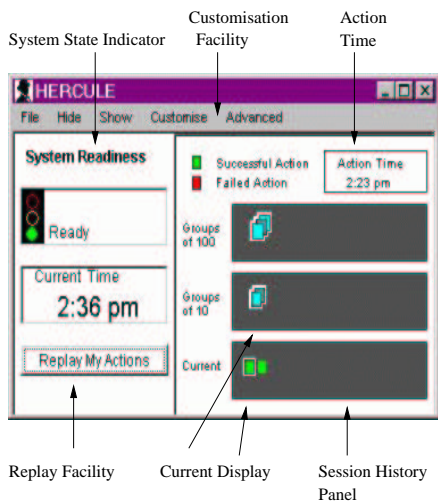
2. *Semantics* — HERCULE makes use of the server component documentation to customise the framework for a particular server component. The documentation is “mined” in order to extract *descriptor objects* which hold details about the methods used to access the server components. The programmer can, either immediately, or at any time in the future, augment the method explanations in these descriptor objects in order to provide better feedback with respect to system activities. Parameters used in method invocations can be inserted into the explanations of these methods invocations.

HERCULE's *discovery process* generates the **descriptors** and **proxies** automatically, thereby customising it for that particular application using the specified server components.

## 4 HERCULE's Functionality

### 4.1 Feedback Provided

The console designed for HERCULE, shown in Figure 3, satisfies feedback requirements by depicting all Episodes for the entire application session in one window; allowing detailed information about any Episode to be obtained at the click of a mouse; portraying a great deal of information in a small screen space; and not intruding — but always being available as an icon, offering the possibility of obtaining feedback at any time.



**Figure 3. The HERCULE Console**

At runtime, the HERCULE console 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. This could be a server breakdown, a network problem, because the application hasn't yet started executing, or because the application has terminated;
  - orange when the server is busy servicing a request; and
  - green when HERCULE is in feedback mode, and waiting for reports.

2. A **Replay My Actions** button will summon a playback facility which allows the user to view a screen replay of all UI-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.

To allow extra flexibility, the user can search for a particular window with some key phrase in it, or step back a certain number of windows, or simply replay all activity from beginning to end.

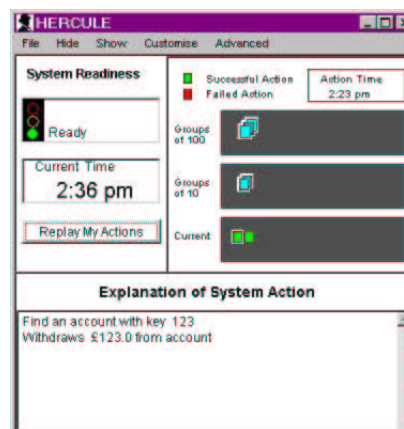
This serves to remind the user of past actions and window appearances. By providing this functionality, HERCULE supports users by alleviating their weaknesses (such as limited working memory), and capitalising on, and utilising their strengths (such

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 our 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 *UI-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.



**Figure 4. 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 identified so far are:
  - (a) explanation of an MI-sequence. In other words, what the system did as a result of a UI-sequence. This is primarily an end-user need, and is met as shown in Figure 4.

- (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, which 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 categories 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 Figure 3 that each panel displays some rectangles with one being highlighted. The highlighted rectangle indicates the Episode for which feedback is currently being given in the visible feedback components. The *Current Display* label in Figure 3 points to the highlighted rectangles in the history panels, indicating that the second last Episode's 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 Figure 4, the Episode actions are explained as `Withdraws 123.0 from account`. This is not the explanation of the most recent Episode's MI-sequence, since the highlight is in the last but one position, indicating that the explanation belongs to the second last Episode.

- (b) *How additional feedback components are added to 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 added to the HERCULE console at runtime. The top section of the console, as shown in Figure 3, will always be displayed, since it provides the core functionality of the console. The programmer simply informs HERCULE about the new feedback component, and it is incorporated into the system. The user can choose which components should be visible or invisible using the menus.

It is beneficial at this stage to discuss how HERCULE's console meets each user's needs.

1. The *programmer* will use the console to:
  - (a) get detailed information about server calls, including parameters passed, and return values obtained.
  - (b) augment the feedback provided by the application, by using HERCULE's customisation process. The initial explanations will be generated automatically, but they can be improved upon at any time while the application is being developed.
  - (c) provide a means for assisting testing procedures.
2. The *end-user* will use the console to:
  - (a) get dynamic continuous feedback about the state of the system.
  - (b) get explanations of system actions.
  - (c) get confirmation of the success or failure of their actions.
  - (d) play back the user interface activities so that they can track down errors made, or confirm inputs given to the system.
3. The *system support staff* will use the console, upon being summoned by the end-user, to:
  - (a) get information about inputs given to the system prior to a breakdown.
  - (b) get confirmation of the error message produced by the system.
  - (c) be able to gauge the performance of the system.
  - (d) be able to augment the feedback explanations.

## 5 Conclusions and Future Work

This paper has discussed the HERCULE feedback enhancing framework. The framework does a number of things at runtime which enable an application to work towards the level of feedback which is desirable in CBSs:

1. it automatically generates the wherewithal to be able to offer this level of feedback to the user;
  - (a) it dynamically keeps track of the presentation on the screen, and also user's actions at the user interface;
  - (b) it watches and keeps a history of all communication with the server.
2. it offers a tool which can be used by programmers to provide adequate feedback for all possible users of an application;
3. it maps the user's activity to the system server calls, and portrays this mapping in an information rich format. This allows the user to get feedback about their current session at any time.

There are many other uses of this technology yet to be explored. Investigations into possibilities like keeping a persistent record of sessions, and determining the negative impact of the framework on application performance are presently underway.

## Acknowledgment

I acknowledge the valuable contributions of Richard Cooper, James Begole and Huw Evans. 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](http://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] N. Borenstein. *Programming as if People Mattered*. Princeton University Press, Princeton, New Jersey, 1991.
- [2] J. M. Carroll, editor. *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*. MIT Press, Cambridge, MA, 1987.
- [3] J. M. Carroll and M. B. Rosson. The paradox of the active user. In [2], chapter 5, pages 80–111, 1987.
- [4] 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.
- [5] J. D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, Mass. London, 1982.
- [6] J. Grudin. Social evaluation of the user interface: who does the work and who gets the benefit. In H.-J. Bullinger and B. Shackel, editors, *INTERACT 1987. IFIP Conference on Human-Computer Interaction*, Stuttgart, Germany, 1987. IFIP, Elsevier Science Publishers B.V.
- [7] S. M. Lewandowski. Frameworks for component-based client/server computing. *ACM Computing Surveys*, 30(1):3–27, March 1998.
- [8] C. Lewis. Understanding what's happening in system interactions. In D. A. Norman and S. W. Draper, editors, [11], chapter 8, pages 171–186. Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1986.
- [9] T. Merridenard and J. Bird. Filling the gap. In *Management Today*. June 1999. Produced for Microsoft.
- [10] D. A. Norman. *Things That Make Us Smart : Defending Human Attributes in the Age of the Machine*. Addison Wesley Publishing Company, 1994.
- [11] D. A. Norman and S. W. Draper, editors. *User Centred System Design. New Perspectives on Human-Computer Interaction*. Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1986.
- [12] J. R. Olsen. Cognitive analysis of people's use of software. In [2], chapter 10, pages 260–293, 1987.
- [13] M. A. Planas and W. C. Treurniet. The Effects of Feedback During Delays in Simulated Teletext Reception. *Behaviour and Information Technology*, 7(2):183–191, 1988.
- [14] J. Reason. *Human Error*. Cambridge University Press, 1990.
- [15] 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.
- [16] A. Rizzo, O. Parlangeli, E. Marchigiani, and S. Bagnara. The management of human errors in user-centered design. *ACM SIGCHI Bulletin*, 28(3):114–119, 1996.
- [17] C. Szyperski. *Component Software. Beyond Object Oriented Programming*. Addison Wesley, Harlow, England, 1998.
- [18] L. Tweedie. Characterizing interactive externalizations. In *Proceedings of ACM CHI 97 Conference on Human Factors in Computing Systems*, volume 1 of *PAPERS: Information Structures*, pages 375–382, 1997.