

An Error Reporting and Feedback Component for Component-Based Transaction Processing Systems

Karen Renaud & Richard Cooper
University of Glasgow
17 Lilybank Gardens
Glasgow
G12 8RZ
{karen,rich}@dcs.gla.ac.uk

Abstract

This paper presents a novel approach to providing error feedback for distributed, component-based applications. We describe an error reporting component which can be seamlessly added to an application. This will collect data from user actions and server requests and provide context dependent feedback on errors.

Users of software systems often spend a great deal of time trying to work out how to use the system and, in particular, how to deal with errors. Sometimes users are completely unaware of errors, and when an error is detected, it is often difficult to locate the cause of the error, and the means of recovery. Component-based systems are composed of independently developed components, and consequently traditional methods for implementing global feedback mechanisms are impossible.

We therefore propose the use of an error reporting component which will enhance the level of feedback in component-based systems, and assist the user in recovering from errors.

1 Introduction

Component-based systems (CBSs) [14] are composed of independently produced and tested components. In addition to their desirable propensity for re-usability, they also offer an alternative to the traditional choice between standard and customised software. CBSs allow organisations to combine software components to create a system which is tailored to their specific needs, without having to devote extensive software development time. Desktop components, such as buttons or text fields, have been used for some time. Server-side components have recently emerged as a viable way of implementing business logic in a *middleware runtime environment* — often called an application server. CBSs are typically structured as a distributed three-tier architecture, as shown in Figure 1, with presentation at the end-user level, server components forming the middle tier, and multiple databases making up the lowest tier. [14]

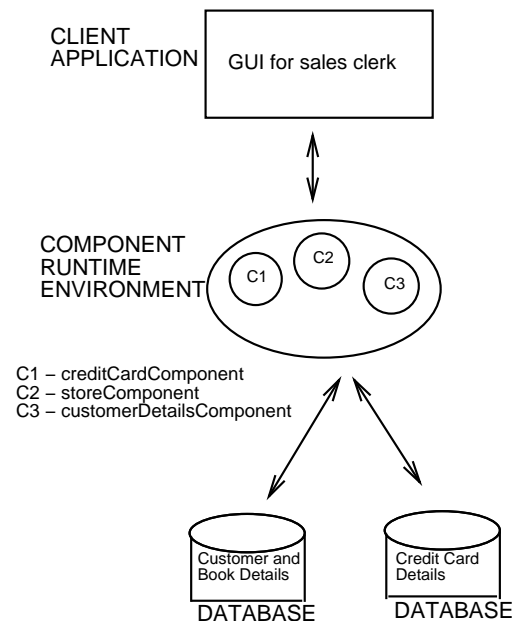


Figure 1. The Three Tier Architecture

The component runtime environment imparts many advantages including load balancing, automatic distributed transaction control, security and scalability. While this structure effectively separates the business logic from the presentation logic, which is essentially good, it also makes error diagnosis and recovery significantly more complex. It is often difficult to pin down the cause of an error, since different root causes of error could produce the same error message. An error could be caused by user actions, a prob-

lem at the server component being used, an error in one of the databases being accessed, or a failure of the communication mechanism linking the levels. Norman [9] argues that in any complex environment, one should always expect the unexpected. To deal with the unexpected, feedback is deemed to be essential.

Chan *et al* [1] have shown that an active feedback system greatly improves user performance. Since CBSs are composed of independently produced components, a comprehensive global feedback mechanism is probably going to prove to be an impossible dream. In the absence of this, we propose an add-on feedback component which will gather data about the state of the system, and provide significant feedback during a transaction processing session.

To create such a component, we rely on two features of emerging component-based systems. Firstly, component-based systems allow us to discover details about component interfaces at runtime. Secondly, desktop and server components are typically delivered with a descriptive aspect, which gives comprehensive information about the functions carried out by the component.

An Error Reporter component will implement two vital facilities. Firstly, it will gain access to a description of the user interface and log user actions in terms of screen components being accessed. Secondly, it will intercept server calls and log them, together with the server component description and the server feedback and, in particular, errors. The error feedback will be based on this information, facilitating an appropriate and timely feedback mechanism.

2 Example and Motivation

At this stage we will introduce an example to explain concepts. Best Books Inc is a catalogue book store that needs to build a system which will be used by their sales force to handle book purchases. Customers wishing to purchase books phone in their orders and receive their books by post. After some deliberation, it is decided that the organisation needs three server components. One component, the *creditCardComponent*, should handle credit card purchases. Another component, the *storeComponent*, should handle the details of the client order. Yet another component, the *customerDetailsComponent*, needs to keep track of clients, and purchases made. Best Books Inc chooses an application server which will provide a component runtime environment, and a database system to store stock information and customer details.

In a CBS, different groups, or stakeholders, are involved in the production of the complete system. Unlike customised or standardised software, these groups will have little, if any, contact with each other.

- The first group of people are the *developers* of the server components. They will ensure that the com-

ponents have a certain prescribed functionality. It is in their best interests that the component be as reliable as possible since they are presumably going to make a profit from selling the component. To this end, each component will be exhaustively tested *in isolation*. The developer of the particular component has no idea which other components will be working in conjunction with their component in the eventual system. This means that components have the very desirable property, from a reuse point of view, of being self-contained with explicit context dependencies. The developer will provide the purchaser with the runtime version of the component, together with an interface through which the component will be accessed.

- The second group is composed of the *buyers*, who select and buy components for the particular application. The Best Books Inc buyer might choose a *creditCardComponent* which can interface with all the major credit cards from a vendor called Credit Card Components Inc, who specialise in that type of component. The buyer might then choose to buy the *storeComponent* from another vendor, Book Seller Components Ltd, who have produced a component which is specially written to handle ISBN numbers, and which automatically re-orders specially marked books when they are out of stock. The *customerDetailsComponent* could be bought from yet a third vendor, who specialises in producing components for keeping track of customer details for all types of organisations.
- The third group are the developers of the graphical user interface (GUI) which uses server components to accomplish the business logic purposes for which they were designed. We will call this group the *programmers*. The GUI programmer will not, and indeed need not, have any intimate knowledge of the inner workings of the component. The programmer simply writes the presentation software and invokes the component where and when required. The programmer uses the interface document, which is packaged together with the component, to determine how to use the component. The resulting GUIs will vary from programmer to programmer.
- The final group are the *end-users* who will use the eventual system.

The end-user often does not have, and indeed need not have, any concept of the true structure of the system. To the end-user, the application seems to run on the computer in front of them — when it runs correctly. If a server component returns an error message which is then displayed by the GUI of the client application, the end-user is often at

a loss as to the cause of the error. There are three different situations where the user needs additional feedback and assistance.

- The first situation occurs when the user has recognised that there is an error, and needs help recovering from the error.
- The second situation occurs when the user does not get the required reaction within the expected time, and is therefore puzzled. The user needs to be able to determine the reason for the delay.
- The third is caused by the overconfidence bias, in which users simply do not realise that an error has occurred. Lewis and Norman [7] explain that people tend to look for confirmatory evidence. People are adept at filtering out and discarding “irrelevant information” and using their intelligence to provide logical explanations for events. This skill can sometimes lead them to discard evidence which, if considered, would lead to earlier discovery of an error condition.

Zakay [16] has shown that computerised feedback significantly reduces overconfidence. More specifically, Humble *et al* [3] have shown that *task-specific* feedback produced a reduction in the overconfidence bias.

This paper proposes a scheme for providing an add-on error reporting component. Section 3 discusses aspects of representing error in CBSs. Section 4 describes the error reporting component. Section 5 gives details of our implementation of this component. Section 6 gives some information about preliminary trials of the use of this error reporting component. Section 7 summarises the work done and gives details of ongoing work on this project.

3 Issues in Error Management

The types of errors that could occur in a CBS are:

1. failure of the middleware server;
2. failure of the communication mechanism linking the end-user to the application server, or linking the application server to the databases;
3. hardware failure of the system the application server is running on;
4. failure of the databases being accessed;
5. transaction failure due to a constraints problem;
6. application specific errors.

There are two primary aspects of effective error reporting to be considered. The first is user awareness of error occurrence. The user must be apprised of error with as little delay as possible, in order to minimise the effects of the error. Following that, the error must be explained in such a way as to enable the user to recover from the error.

3.1 Detecting Error

Detecting error is the first step towards recovery [7]. Reason [11] cites three ways in which errors are brought to someone’s attention. The first is that they find out themselves, the second is if something in the environment draws their attention to it, and the third is if another person discovers it and tells them.

The first case is obvious, so we briefly look at examples of the other two cases. A user could easily miss error details reported by an application. This could happen if the user is distracted when the application reports the error. For example, the customer could choose to order a book that is out of print. This might not be reported as clearly as it could be, and the clerk fails to notice the error message. Later, when the order is being finalised, the clerk notices that the book is not on the list of books to be dispatched, and consequently tries to discover what went wrong.

If the error is not detected by the clerk, either immediately or later due to some clue in their environment, someone else will have to pick up the error and alert the clerk. If this case, the incorrect order is processed, and the incorrect book is sent to the customer. Only then does the error become apparent, when the dissatisfied customer informs the sales clerk of the error.

3.2 Error Reporting

It is important that the errors be reported on the user’s level. Machine code is useless to almost everybody! Some researchers feel that systems should provide help based on the user’s skill level [5]. Others show that providing help according to the skill of the user is difficult and impractical [2].

Help is most useful if tailored to the context of user activity since any system which aids users must focus on their goals [2]. One of the few user assistance schemes which attempts to render situation-dependent options, dynamically interprets users’ inputs, and using that interpretation, offers the user situation dependent options for proceeding [13]. Their MIRACLE system uses this dialogue analysis in the framework of an information retrieval system.

In tailoring user assistance to their activity it should be borne in mind that the user may have progressed through various windows to arrive at the one in which an error

has occurred. The error reporting mechanism should include information to remind the user about their progression through these windows, to foster a comprehensive understanding of the true state of the system.

If an error is detected, how should it be handled? Lewis and Norman [7] identify six possible responses: Gag, Warn, Do Nothing, Self Correct, Teach Me and Let's talk about it. In an *add-on component* in a CBS, we realistically have only one option: Warn. This is because we are essentially providing a reporting mechanism, which does not interfere with the application's handling of an error. How can we best go about warning the user? For example, the error thrown back from the application server could be: `SQL Error code 40`. This will have no meaning to the end-user whereas to a programmer this might immediately suggest the root cause of the error. The problem is that the error is presented at the lowest level of program execution and not at the level of the user's intentions [7]. The user, in our example the sales clerk, probably has no idea what SQL is, and why it is involved in what they are trying to do. The clerk is left with no choice but to attempt to link the error to something at his or her level of intention. The user may simply submit the request again, or attempt to change the inputs given, or try to activate a different option at the user interface, in an attempt to get past the error. If the error was due to a concurrency problem in the underlying database, which caused the transaction to abort, the user's efforts are useless, and a waste of their time.

Finally, since server components are essentially transactional, it is worth considering how to report the success or failure of transactions to users. An informal survey (see Section 6) indicates that users with no formal computing science training do not necessarily have the same understanding of transactions as database specialists, and that it is unrealistic to expect it. Therefore great care should be taken in reporting that a transaction has committed or aborted. We suggest that the reporting should be done in terms of the user's *actions*, and that the user should be told whether their *action* achieved the desired result or not.

3.3 Configuring the Application to Report Errors

In developing an add-on error reporting component, the following features are important:

- The Error Reporter should be an *optional* addition. If the user decides not to use it, it should not intrude on the system.
- The failure of the reporting component should not in any way cause the failure of the application. This is important, because if the error reporting component causes the application to fail, the system has been impaired by the existence of the error reporter, which is

exactly the opposite of what we wanted to achieve.

- The Error Reporter should be non-invasive. In other words, no part of the application should be changed to accommodate the Error Reporter.
- The Error Reporter console should always be present on the desktop, but because of the aforementioned points, it will not intrude. It will continuously have up-to-date information about session activities, and can be used by the user as a feedback mechanism at any time.

4 The Model

So far we have presented a case for a feedback and error reporting component which can be added to a component-based application. This component must have several functions, and needs to keep track of diverse system events in order to provide the required feedback. The next section outlines our model of the Error Reporter component.

The Error Reporter component's purpose is to collect data about the state of the system, to provide continuous comprehensive feedback, and context-sensitive assistance when an error occurs. To do this, the Error Reporter will:

1. build up an internal representation of the structure of the presentation description. This provides us with a syntactic description of the user interface;
2. keep track of the user's actions at the user interface, enabling us to report an error in terms of the context. Trafton and Brock [15] have worked on a system where a layer between the user interface and the application keeps track of the users actions, compares them to an internal representation of various task models, and tries to identify the task being done by the user. When they can pin down a correspondence, they then offer the user the option of completing the sequence for them. We do not have the advantage of a task model, and so we are going to replay the user's actions at the interface, upon request. This will entail displaying each window the user progressed through, and highlighting their actions on that window. This reminder service will serve as confirmatory evidence of their actions;
3. keep a history of all calls made to the underlying middleware tier. This gives us an insight into the scope of an error, and allows us to derive an interpretation of the cause of the error. This history will be displayed so that the user can get a playback of their actions as mentioned in the previous section, followed by an explanation of what the server did as a result of the action.

It must be stressed that this component is not attempting to add any measure of fault tolerance to the system. It merely serves as an interpreter of system errors into a more helpful and information-rich format.

4.1 Requirements

Since one can only expect a system to give back what you put in, we propose the incorporation of a *Component Information Descriptor (CID)* which will incorporate semantic details about the interface to the server component being used. We propose that, in time, this descriptor component should be delivered, as a matter of course, with components which are offered for sale by vendors. If the CID is not provided, the system will generate it from the documentation provided with the component. This will be done by using the interface document, and the accompanying component documentation. The CID holds information about:

1. the semantics of each method invocation. This is usually a free text description of the meaning of the call.
2. the possible errors which could be produced by each invocation, and an explanation for that particular error. For instance, this could be in the format of an exception which could be generated by a method invocation.
3. the transaction attribute of each method call. Methods either execute within the context of an existing transaction, or create a new transaction, or are non-transactional.

These requirements are not unreasonable. In one popular component model they are already provided. (Sun’s Enterprise Java Beans)

4.2 Architecture

The Error Reporter, shown in Figure 2, while being an add-on component, is itself composed of various components. Each will be explained below:

A Reporting module tracks and reports on all activity at the GUI.

A Proxy Reporting module tracks and reports on all server calls made by the application.

An Error Reporter Console reports on errors and possible reasons for those errors. This console will be active, but will not intrude. If the user wants to verify any actions, or get explanations of errors, the console can be consulted. A first prototype of the console window can be seen in Figure 3. The *Last Action* button will indicate, using colour, the success or failure of the last user action. If the user’s last action succeeded, the

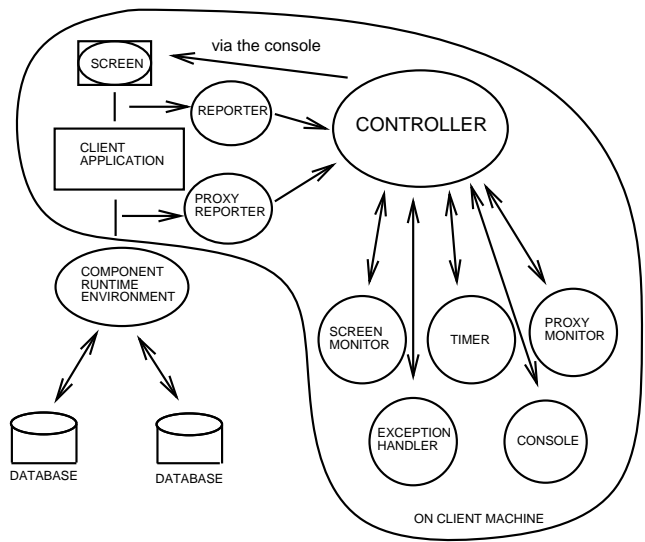


Figure 2. The Error Reporter Architecture

button will be green and display the message: “OK”. If the last action failed, the button will be red, and display the message “Error Explanation”, as shown in Figure 4. If the user activates the “Last Action” button, a window, as shown in Figure 5, is displayed, which can be used to get more information.

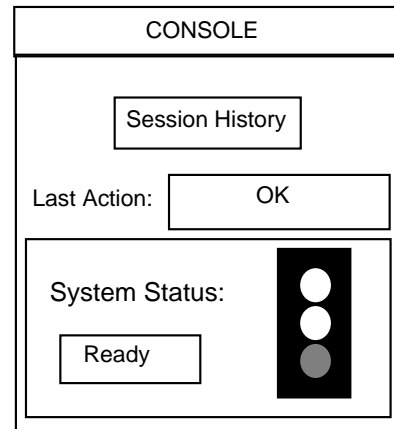


Figure 3. The Error Reporter Console - no errors

The *Session History* button, when activated, gives a list of all user actions during the session, as shown in Figure 6, and the buttons displayed under headings *Action* and *Effect* can be activated to get more information.

The traffic lights will be green when the system is ready for use, orange when the system is busy servicing a request, and red when the communications with the server have broken down. This is primarily a feedback device, which keeps the user informed about the

readiness of the system.

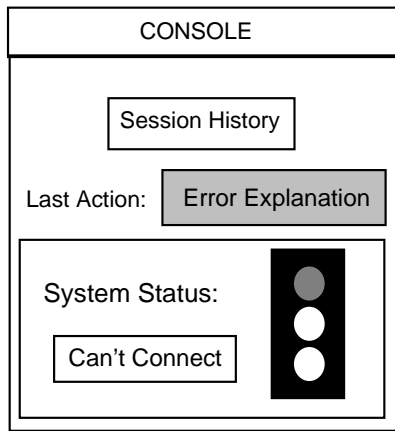


Figure 4. The Error Reporter Console - error

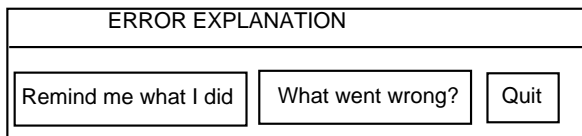


Figure 5. The Error Explanation

SESSION HISTORY			
TIME	ACTION	EFFECT	SUCCESS/FAILURE
3:01	Close Account	More...	SUCCESS
3:05	Withdraw	More...	SUCCESS
3:15	Deposit	More...	SUCCESS
3:16	Deposit	More...	FAILURE

Figure 6. Session History

A Screen Monitor receives all reports about the presentation details and activities at the user interface. It has a dual function. Its first function is to build up an internal structure in memory to represent the user interface. This structure will be augmented each time a new component is added to the user interface, or a new window is generated. The second function is to keep track of users' actions at the GUI. If a user action at the user interface directly precedes a call to the server, we can assign a *purpose* to a user action. This must serve as a substitute for an understanding of the user's intention when the action was taken.

A Timer keeps track of the time taken for each call to be serviced by the server. If a server fails to respond, the timer will attempt to ascertain why. The timer can differentiate between a dead server, a dead host machine, and a network problem. This information would be useful to the system administrator to whom the problem is reported.

A Proxy Monitor keeps a history of all calls to application server objects, and the return values or error messages returned by the server. This is done by using a proxy object which dynamically intercepts all communications between the client application and the server. This proxy is automatically generated from the server component interface definitions. The use of this proxy is made possible because components all advertise their functionality by means of interfaces, effectively separating the interface from the implementation. There are three possible consequences of a call to the server:

1. no response;
2. an exception, signalling that an error occurred; or
3. correct execution, signalled by the return of a return value or values to the user.

In the first case, the lack of a response within the expected time will trigger investigation into the source of the delay. In the second case, the exception handler will be activated to investigate the source of the error. In the third case, the return values will be stored together with the details of the call, to augment the history of the session.

An Exception Handler accesses the details of a call to the server, as well as the resulting exception. The exception handler will use the session history and component descriptors to derive the cause and scope of the error.

4.3 Representing the error

We have said that we need to tailor our help to the context of user activity, and focus on user goals. This is not at all easy to provide in an add-on component. To provide context-dependent assistance, we would have to try to deduce both the context of what the user is doing, and their intentions. To provide us with clues to lead us to this analysis, we need data about the appearance of the user interface, and a history of the user's actions at the GUI.

To get information about the user interface *construction*, a proxy is inserted between the application and the user interface, which filters off information about the user interface as it is built up. To get information about the user *actions*, we register an interest in application specific events at the user interface, and we will thus be notified of all events. The mechanism used is described comprehensively in [12].

It is impossible to deduce a user's intentions given only this syntactic and behavioural information, so we use another approach. We use the information we have collected about the structure of the user interface, and about the user actions, and when the user needs feedback, we replay the user's interaction with the user interface. This essentially provides the user with a reminder of what they did. The user's intentions will become apparent, as they are reminded of them. There is a real need for this playback facility in the case of complex applications where a user may have passed through many windows to arrive at the one where the error occurred. There is often no way to backtrack and check which options were chosen in a previous window.

Since we cannot deduce the user's intentions, the next best thing is to assign *purposes* to actions at the GUI [6]. Our Error Reporter assigns a purpose to each user action, if that action precipitates a call to the server. We can use the CID component (see section 4.1) to get an explanation for that call, and then link a textual description to the action at the GUI. This feedback will essentially inform the user as to what their action achieved or did not achieve.

This, together with the context reporting, will record the information flow which led to a server call, which in turn led to an error.

5 Implementation

5.1 Component Models

There are currently three main component technologies: COM/DCOM from Microsoft, Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG), and JavaBeans and Enterprise Java Beans (EJBs) from Sun Microsystems.

5.1.1 COM/DCOM

COM is the foundation upon which Microsoft's component software is based [14]. DCOM is Microsoft's component middleware service which provides the infrastructure for providing a runtime environment for distributed server-side COM objects. COM is a binary standard for software components, and although COM is available on many platforms, DCOM (Distributed Component Oriented Model) is presently only available on Windows NT. Microsoft's desktop components are called ActiveX (COM desktop components) objects and they have been in use for some years. ActiveX components are typically delivered with enough information to allow a purchaser to use the component, together with the binary code of the component, and the interface documents.

5.1.2 Enterprise Java Beans

JavaBeans is Sun's desktop component model, while Enterprise Java Beans (EJBs) are the server software components. EJBs were specifically designed to support transactional business systems using distributed objects. These objects will be built using the Java language. The EJBs are designed to run within a container, with the container running within any server container that adheres to the EJB specification. [4] The container is expected to provide support for multi-threading, security, transaction coordination, state management, resource pooling and global naming services. EJBs interact with clients via interfaces. According to the EJB specification [8], the EJB must be delivered with a home interface, a remote interface, the programmer API, and a deployment descriptor which gives information about the transaction attribute of each method, the security requirements for accessing the bean, and details about the deployment of the bean within the container.

5.1.3 CORBA

CORBA is a distributed object technology, not a middleware component runtime environment. At present CORBA objects are used mostly within organisations and so no outside market for these objects has developed. Because of this, it is difficult to talk about the types of documents that will typically be provided with these components, since this is presently done in-house and each organisation will have their own standards for communicating the information. The OMG is presently working on a Component Middleware standard, but at the time of writing it had not yet been released. Presumably, this standard must also provide a framework within which descriptive information can be placed.

5.1.4 Availability of third-party server components

Since this is a relatively new field, there are only a few vendors who currently market server components. There is a substantial market for presentation level ActiveX (COM desktop components) objects already, and a growing market for JavaBeans desktop components. This would suggest that as the technology matures the availability of server-side components will grow accordingly.

5.2 Choice of EJB for our implementation

We chose to implement our system using EJBs for several reasons. The platform independent nature of Java applications was a strong feature, as well as the fact that the EJBs could be placed on different vendors application servers and still run without any changes. The application server used was the Tengah server from Weblogic. This server was donated for a period of 9 months specifically for this research, and gave us the opportunity of using an all-Java application server. Another reason that EJBs were chosen was because they have the most flexible options with respect to transactions. Transactions on CORBA objects must always be initiated and managed directly by the client, using the CORBA transaction service. EJB transactions can be either managed by the client, or by the EJB itself, or by the container which houses the bean in the application server. COM/DCOM has many of the same characteristics that EJBs have with respect to transactions, but DCOM objects are not platform independent and this would tie us down to a Windows NT architecture. It was felt that this would be too much of a restriction for a scientific research project. The final reason was that Java has a very powerful introspection mechanism which allows us to exercise a discovery process on Java components, and this is an invaluable property for our project.

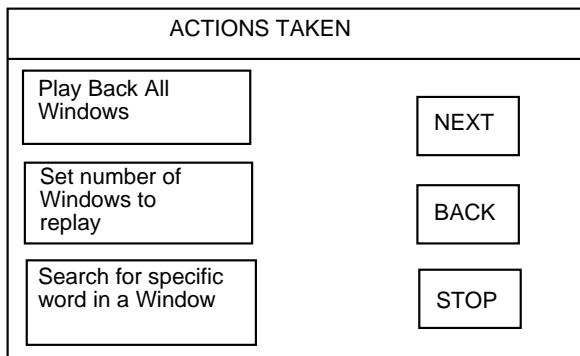


Figure 7. Replay your Actions

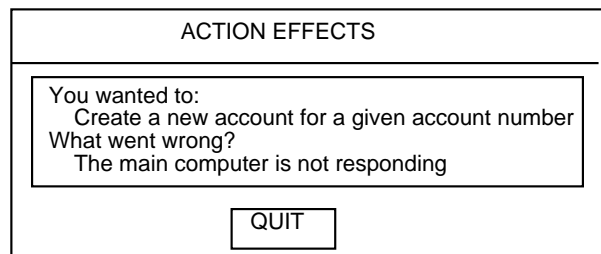


Figure 8. What went wrong?

5.3 Feedback Provided by the Error Reporter

The Error Reporter console is shown in Figure 3. Basically two types of feedback are provided:

- A replay of all windows the user has progressed through during the session. This serves as a confirmatory tool in case the user needs to check previous actions, or be reminded of previous messages. The replay control window is shown in Figure 7.
- An explanation facility, where a message will be given which explains how the user's action was understood by the computer, and what went wrong, if anything. This is shown in Figure 8.

This feedback is available directly from the console. In the beginning of this paper, we mentioned three situations in which the user needed some extra feedback and assistance.

- If the user needs help in recovering from an error, we give the user information which will enable them to understand the system state, and recover from it.
- If there is no timely response, the user will be able to ascertain from the displayed traffic lights whether the system is busy, or whether there is some more serious problem. If the user can see that the system is busy processing their request, they will be more likely to wait for the application to respond.
- If the user does not know that an error has been made, the level of feedback has the potential to reduce the overconfidence bias. Enforcing the use of the feedback mechanism is a management issue, but since the technology is available, it can become a tool in the hands of managers to reduce overconfidence errors.

6 Discussion of User Trials

To determine the efficacy of this technology, we need to get end-user feedback. We decided to test two aspects: firstly we wanted to find out how users understood the concept of a transaction; secondly we wanted to see whether users found the Error Reporter helpful.

To test the perception of a transaction, we chose eight respondents who had had no formal computing science training. The respondents were simply asked what they understood by the concept of a transaction. The answers can be grouped as follows:

- Six of the respondents linked the concept of a transaction to some sort of exchange of money, or goods, between two parties. When asked about failure of the transaction, some said that they would take legal action, others said that they themselves would take action to ensure that the transaction was declared null and void, and the money or goods returned. They displayed no concept of atomicity since they all referred to taking steps to correct the situation *themselves* if the transaction were to fail. Only one mentioned trying to redo the failed transaction.
- The other two respondents had a different understanding.
 - One likened it to a form of contract between two parties, and
 - the other to the process of “getting over an action of any kind”.

This survey seems to suggest that reporting “transaction failure” to an end-user could possibly leave them with a feeling of incompleteness. They might feel the need to *do something* to achieve a sense of closure, to restore the situation to what it was before the transaction executed. Since the nature of database transactions ensures that aborted transactions have no effect on the database, it is important that users do not try to duplicate this function. Additionally, they really need to know whether they can re-try a transaction, or to realise that it is fruitless to try again. However, it would be premature to draw conclusions based on such a small survey, and we intend to follow this up by carrying out a far more extensive study of end-user perceptions.

Preliminary tests with the Error Reporter have produced encouraging results. Of the eight subjects who have used the error handler, seven found it helpful, while the eighth was not hindered by it. The application was a simple savings account system, which allowed for the creation and closure of accounts, and withdrawal and depositing of funds.

The usage of Error Reporter features during the trials is shown in Figure 9. As can be seen from the chart, users did not make much use of the replay mechanism. This could be because the application they were using was a very simple application, and they had no difficulty remembering what they had done. The colour indicator which shows the presence of an error was also not found to be helpful. It is interesting to note that many of the subjects did not notice error

conditions, even though the error was reported by the application, *and* the console indicators were clearly indicating that an error had occurred. We clearly need to find some better way of bringing errors to users’ attention.

Users were asked for comments about the Error Reporter. While all were positive about the concept, the majority asked that the error explanation be displayed directly on the screen, without the need for navigating through some windows to get it. This is obviously valuable feedback, and will be taken into account.

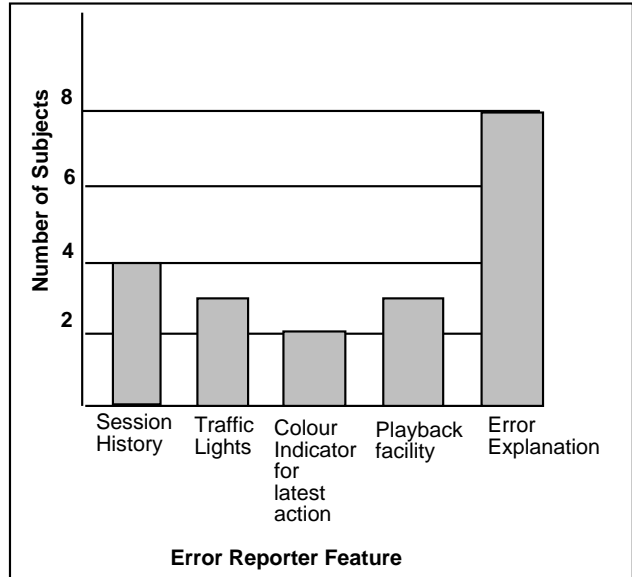


Figure 9. Error Reporter Feature Usage

7 Conclusions and Further Work

In software systems the help provided to an end-user has traditionally been provided in the form of online or offline manuals. Manuals provide assistance to the user, but this assistance is seldom contextual. Manuals, even online, cannot provide the feedback required to maintain user confidence in a system. Much research has been done into the use of these manuals, but in the case of CBSs there is a need for a new manual construction technology due to the late composition of CBSs, and due to the possibly distributed nature of the participants in the creation of a CBS. A new mechanism for generating a manual from the individual components’ manuals is an interesting research problem, but not one that we have addressed at this stage.

In the absence of a manual, we have proposed the use of an add-on error reporting component. The need for such a component will become increasingly evident with the growth of commercially available server components. We have implemented a prototype of an Error Reporter component. The component does two things which enable us to

work towards the level of fault management we would like to be able to provide in CBSs:

1. it dynamically keeps track of the presentation on the user interface, and also of user's actions at the user interface;
2. it watches and keeps a history of all communication with the server.

We have a working prototype, and we are in the process of evaluating it. This should give us an idea of how to improve the Error Reporter, so that we can provide a better feedback mechanism for users. In our trials we noted that users did not always detect errors, even though the console indicators were clearly indicating the presence of an error. We will be investigating the use of some attention-grabbing mechanism to draw a user's attention to the presence of an error, since the Error Reporter can only perform its function if the user becomes aware of the error.

Since the Error Reporter only presents errors at one level of detail, we are also interested in pursuing ways of pitching the error reporting to the level required by each individual user. As users become more proficient, it is conceivable that their error feedback needs might change, as they build up an internal model of the functioning of the application. We will investigate the possibility of accommodating this need.

We would also like to do a more extensive survey of users' understanding of transactions, in order to gain an insight into how best to portray the success or failure of transactions at the user interface.

8 Acknowledgements

This work was done after consultation and discussion with many people. We would like to acknowledge the contributions of: Malcolm Atkinson, Huw Evans, Susan Spence, Steve Draper, Meurig Sage, Mark Dunlop and Phil Gray. We would also like to acknowledge the donation of the Tengah server from Weblogic (URL: weblogic.beasys.com), and express our appreciation for their prompt responses to our queries. This work could not have been carried out without their kind donation.

This research was done in the context of Karen's PhD. She is supported by a scholarship from the Association of Commonwealth Universities. This work is also funded in part by a grant from the Foundation for Research and Development in South Africa, and the University of South Africa. Karen is currently on leave of absence from the University of South Africa and she would like to acknowledge their magnanimity in allowing her this extended period of absence.

References

- [1] 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.
- [2] T. M. Duffy, J. E. Palmer, and B. Mehlenbacher. *Online Help. Design and Evaluation*. Ablex Publishing Company, 1992.
- [3] J. E. Humble, R. T. Keim, and J. C. Hershauer. Information systems design: an empirical study of feedback effects. *Behavioural & Information Technology*, 11(4):237–44, 1992.
- [4] D. Kara. The Enterprise Java Beans Component Model. *Component Strategies*, 1(7):18–25, January 1999.
- [5] G. Kearsley. *Online Help Systems. Design and Implementation*. Ablex Publishing Corporation, Norwood, New Jersey, 1988.
- [6] C. Lewis. Understanding what's happening in system interactions. In D. A. Norman and S. W. Draper, editors, [10], chapter 8, pages 171–186. Lawrence Erlbaum Associates, Publishers, Hilldale, New Jersey, 1986.
- [7] C. Lewis and D. A. Norman. Designing for Error. In D. A. Norman and S. W. Draper, editors, *User Centred System Design. New Perspectives on Human-Computer Interaction*, chapter 20, pages 411–432. Lawrence Erlbaum Associates, Publishers, Hilldale, New Jersey, 1986.
- [8] S. Microsystems. Enterprise Java Beans Specification. Web Document. URL:java.sun.com/products/ejb, March 1998.
- [9] 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.
- [10] 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.
- [11] J. Reason. *Human Error*. Cambridge University Press, 1990.
- [12] 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.
- [13] A. Stein, J. A. Gulla, and U. Thiel. Making sense of users' mouse clicks: Abductive reasoning and conversational dialogue modeling. In A. Jameson, C. Paris, and C. Tasso, editors, *Proceedings of the Sixth International Conference, UM97*, pages 89–100. Springer Wien New York, Vienna, 1997.
- [14] C. Szyperski. *Component Software. Beyond Object Oriented Programming*. Addison Wesley, Harlow, England, 1998.
- [15] J. G. Trafton and D. P. Brock. Simplifying interactions with task model tracing. ACT-R Summer School, Psychology Department, Carnegie Mellon University, June 1996.
- [16] D. Zakay. The influence of computerized feedback on overconfidence in knowledge. *Behaviour & Information Technology*, 11(6):329–33, 1992.