Error Handling in Structured and Object-Oriented Programming Languages

University of Oulu
Department of Information Processing Science
Thomas Aglassinger
040972-903J
Master's Thesis
24.11.1999

Abstract

A program can face conditions where it is impossible to continue with the normal control flow. Common reasons are erroneous input data or lack of resources. Most structured and object-oriented programming languages provide mechanisms to deal with such errors. This thesis takes a look at some of the most common ones.

It examines theoretical issues how different kinds of errors can be detected, and which general possibilities there are to deal with them. Because eventually errors have to be treated by human beings, considerable concern is taken at proper wording of error messages. Consequently, it investigates how the language and its libraries support the programmer in creating such wording.

To see how practical implementations deal with this, a set of simple example programs is implemented in various languages. In particular, Basic, C, Eiffel and Java are studied. The tasks to solve are converting a text string to a number, copying a file and detecting a bug during runtime.

The findings are that none of the mechanisms allows simple generation of useful error messages. They are either overly complicated and ill-structured or leave most of the work to the programmer. Many of them just swallow errors on various occasions, allowing latent errors to remain in the program. This can be seen as precondition to disasters, especially for complex programs.

Preface

The main motivation for the topic resulted from my experiences with computers both as user and programmer. As the first, I was constantly annoyed by the generally useless error messages programs provide and the incomplete ways they can deal with errors. As the second, I never really figured myself how to create programs that behave properly under such conditions. I saw this thesis as a good chance to set the record straight for me.

Thomas Aglassinger

Oulu, 24.11.1999

1 Introduction

The ability of a program to deal with errors is an important part of it. The world is not perfect, and things can go wrong. Several books and papers collect stories in which computer programs cause funny or terrible incidents (depending on the view point) because certain error situations were not handled properly. Many computer magazines frequently publish the most useless error messages users encountered.

There seems to be a common believe that all this is the programmer's fault. If they were just a little more careful when creating the program, and a bit more ambitious when wording error messages, all this trouble could be avoided. Branches to engineer usability, fault tolerance or software in general have been established long ago, promising to target at these problems. Alas, things have not improved much. Some might even say, they have become worse.

1.1 Related Work

General means to detect and handle errors both in terms of hard- and software are discussed by Lee & Anderson (1990). The human contribution to errors is explained by Reason (1990), who is strongly influenced by work of Rasmussen. Norman (1983) reflects on how this should be considered in the program design. Dain (1991) evaluates the state of the art in dealing with input errors. Shneiderman (1997) gives general suggestions on wording of error messages. Brown (1988) provides more specific guidelines, together with many examples. Laprie & Kanoun (1996) explain how to apply the classical reliability theory.

All examined programming languages support some sort of error handling, which is consequently described in the standard documentation. In particular, this means special function results and status indicators in C (Harbison & Steele, 1991), traps in Basic (Sharp Corporation, 1989), exceptions in Java (Gosling et al., 1996) and assertions in C and Eiffel (Meyer, 1997). Influential material from other, earlier languages can be found in (Goodenough, 1975; Liskov & Snyder, 1979; ANSI & US Government Department of Defense, 1983).

In the recent years, research has mostly focused on distributed and concurrent error handling, with Campbell & Randell (1986) describing general issues. Several proposals and reflections on existing languages exist (Romanovsky, 1997; Meyer, 1997; DeRusso & Haggar, 1998). I decided to exclude these topics, as there are enough (unresolved) questions for comparably simple non-distributed, single-threaded programs.

Current work on error handling often centers on certain aspects, and more or less ignores others. Fault tolerance and reliability engineering is mostly concerned with keeping a system running - despite errors. Publications from this area rarely consider to come up with an error message. Usability literature on the other hand tends to end up in unspecific guidelines which are difficult to apply for a programmer - especially on errors that are not caused by the user input. Programming language designers are concerned about directing the control flow to error handlers, but don't give much advice what to do after that. Literature on algorithms, patterns, design methods and system architectures is so busy reflecting on the normal cases, that there is no place left for thinking about errors.

Another point rarely being considered is the notion of a latent error. Informally, it is an error that has no visible effect to the user, making him believe that everything is all right though it isn't. As it turns out, such errors can be introduced by the programming language, without the application programmer being able to do anything about it. They happen on a level lower than he can express. Language definitions either treat them as undefined cases, or bluntly make the program ignore them.

This thesis is not going to reveal any new facts on error handling. However, it tries to give a more complete view by trying to combine the above mentioned rather narrow perspectives. It attempts to stay simple and specific enough so that many considerations can be applied by programmers directly. Although the issues are examined from several different angles, error messages are a recurring theme. Latent errors are not seen as rare cases without practical relevance, but as serious flaws. But probably the only thing really new is the attitude the problem is approached with.

1.2 Research Method

The research problem is addressed by the following questions:

- What are errors, speaking in terms of a program?
- What different types of errors exist?
- How can they be detected?
- How can they be reported?
- How might they be avoided?
- How can a program respond to them?
- How do different programming languages allow to deal with all these issues?

The research works in a conceptual analytical and constructive way. It is mostly placed in the area of information technology, but obtains a lot of influence from other areas such as

cognitive science and sociology. Despite the title, reflections on programming languages make a relative small part of it.

The conceptional analysis is carried out by collecting theoretical considerations from literature. This gives a general picture of what error handling means and what are the concepts involved. It describes the different ways a program can respond to errors, and reflects in which situations they are appropriate. It presents a simple scheme to obtain good error messages that does not require much effort from the programmer.

In the constructive part, it looks at different existing error handling mechanisms actually implemented in certain programming languages. Roughly, they are: in C, routines can return special results to indicate errors, e.g. a function might return a pointer to an opened file handle or `NULL` if it failed. C also supports status indicators, e.g. an integer variable `errno` (for error number) holds `0` to indicate success, and other values to point out different errors. In Basic, a trap handler can be assigned with a `on error goto` statement, to which the control flow is changed automatically upon detection. In Java, exception objects refine this concept by allowing to attach more details to an error. In Eiffel, assertions are used to detect bugs during runtime and to document a routine interface.

All the mechanisms are evaluated for a set of criteria that are derived from the theoretical parts. This is supported by implementing a set of small example programs. In particular, the tasks to solve are: converting a text supposedly consisting of digits into a number, copying a file and detecting a bug. Finally, there is a critical discussion of possible inconsistencies between theory and practice that also outlines solutions to the severest problems.

1.3 Overview

Chapter 2 introduces the terminology used within the rest of the thesis. It gives an overview of the basic steps and concepts involved in error handling. Related terms and conflicting terminology from other areas are shortly discussed.

Chapter 3 discusses error handling from a user's point of view. It describes possible program responses to errors and considers when they are appropriate. It summarizes general guidelines on wording, sentence structure and highlighting mechanisms useful for error reporting. Finally, it illustrates a simple scheme to create good error messages derived from the contradiction used to detect the error.

Chapter 4 characterizes different kinds of defects and error situations. It discusses how to detect the various types of errors and what details to report to the user.

Chapter 5 analyzes various error handling mechanisms provided by widely used programming languages. It describes all these mechanisms, their underlying ideas and some implementation details. The analysis is done by validating every mechanism for a set of criteria and implementing some simple example routines.

Chapter 6 gives a discussion of questions raised during the analysis. For several issues, it is not easy to decide whether they are a "good thing" or not. Solutions to some of the most apparent problems are outlined.

Chapter 7 shortly summarizes the research results and points out possible future directions.

1.4 How It Really Happened

The above description would suggest that there was such a thing like a master plan from the beginning, and all I did was execute it. In fact, there wasn't. Initially, my naïve assumption was that somebody must have clarified the whole issue to some extent, and I just have to summarize a few theoretical considerations, see how they work with existing implementations and write about them.

As it turned out, nobody seems to have a real clue how computers can handle errors. Even better, people rarely speak about errors rather than exceptions, faults, defects and a couple of other terms. As a result, I had to dig through the various contradicting definitions and find a terminology that would at least work for me. The next problem then was the theoretical background of error handling. It took me a horrible long time to find out that people who talk about "fault tolerance" also discuss error detection and recovery.

Thus, for quite some time I worked on the analysis and comparison of different programming languages. From the beginning, I had planned to see how to convert a string to an integer and how to copy a file. Another idea was to write a little more complex program such as a command line pocket calculator and show how to deal with a bigger set of errors at the same time. But after my initial experiments, I soon found out that current implementations are far worse than I had noticed in the recent years. Consequently, I soon discarded the calculator idea.

After a couple of months I was convinced there is no useful error handling mechanism, nobody knows what is going on, and apparently nobody really cares. However, my advisor didn't seem to be very fond of my conclusion and suggested I should write something constructive, too. Even if I might be right.

At this point, things started to become difficult. How is a student supposed to fix the mess others (with many of them way smarter than me) didn't manage to cleanup within the last decades? Anyway, I turned away from information technology and computer science in favor of a couple of books that had no real relation to programming. These finally brought me on some more constructive track. I then started to compile a list of error situations and structure it. This slowly gave me a general understanding of the nature of errors. (Although I'm still not sure if my classification makes sense.) In a retrospect, most of the useful literature came from the seventies or from outside the plain computer context.

Eventually I managed to collect theoretical considerations from several areas for which I didn't have a deeper understanding. This also seems to be a reason why until yet nobody really tried to write something solid on error handling: because nobody can know it all. Anyway, not having any reputation to defend, I bluntly wrote about everything. In a few years, probably I will have a jolly good laugh reading all those that are wrong.

2 Terms

This chapter introduces the terminology used within the rest of the thesis. It gives an overview of the basic steps and concepts involved in error handling. Related terms and conflicting terminology from other areas are shortly discussed.

2.1 Program, System, Component

A computer can be seen as a system to execute a program. A simple view of the term *program* is to claim that it takes an input and creates an output from it:

Figure 1: A Program

The input is essentially provided by a user, who also has a certain desire in the output. The program is a sequence of instructions and rules that specify *how* the output should be created.

But a program is only an abstract software system and not enough to get the output. It also needs a hardware system, which provides the following services:

1. It *interprets* the program by executing the instructions (CPU)
2. It *stores* both input and output in resources (memory, disk, ...)
3. It acts as an *interface* between program and user, allowing to enter the input and read the output (keyboard, mouse, display, ...)

The term *system* is extremely general and has a meaning in nearly every discipline - also outside the computer context. Nevertheless, a possible definition is that a system consists of a set of components which interact under the control of a design.

A *component* can be seen as another system. This recursion eventually ends in *atomic systems*, for which any further internal structure cannot be discerned, or is not of interest and can be ignored.

The *design* also is a system, but is has some special characteristics: it controls which components interact, and how. It also determines the way in which interactions between a system (component) and its environment (containing system) influence the components. Thus it must ensure that each component receives as input an appropriate subset of the outputs from all other components.

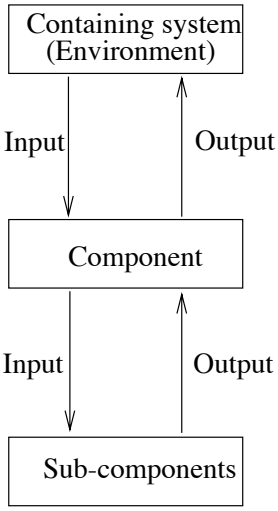

Figure 2: General system component

A more detailed discussion of these terms can be found in (Lee & Anderson, 1990).

Returning to programming languages, the component structure is achieved by mechanisms like routines, records, modules and classes. Routines are common for both structured and object-oriented languages. They differ mainly in how they allow to organize and access routines, and how routines are bound to datatypes.

2.2 User, Administrator, Programmer

Basically, there are three different kinds of human beings which have a special relation to the program:

1. The *user* is the person providing the input and also has some interest in the output. He is the reason why the program is executed at all.
2. The *administrator* is responsible to provide a working environment for the program to be executed in. This means both hardware components (e.g. memory, network cables, storage devices) and external software components (e.g. function libraries, device drivers)
3. The *programmer* creates the program. He is responsible for the design and code, and without him, the program would not exist at all.

This distinction does not necessarily mean persons rather than roles. For instance, many users of personal computer systems also act as administrators. And very few programs are actually created by one single programmer rather than by a whole team with tasks separated between all the members, aiming at a bigger market than one particular user. All programmers are also users in some sense as they use development tools such as compilers, editors and function libraries to create the new program. For some programs user, administrator and programmer

are the same person (in particular in-house tools).

2.3 Problem, Defect, Error

Generally, a *problem* is a subjectively unsatisfactory state. For this discussion this usually means that the user won't get the desired output from the program. Independent of the actual cause for the problem, the user is unlikely to obtain new versions of the program or recommend it in his circle of acquaintances. This in turn causes a problem to the programmer.

There are four ways for the programmer to avoid this situation: first, he can create a program that does not cause the problem to the user. Second, he can create one that helps the user to find out the cause for the problem and remove it. Third, the programmer can refocus on advertising and hope that the user is stupid enough to use the program anyway. And forth, he can use certain techniques to make the user so dependent on the program that he will not consider any alternatives. The last two methods are widely used and have proven successful for solving the problem from the programmer's point of view (Thimbleby, 1998a; Thimbleby, 1999). They are however not very interesting from the viewpoint of computer science rather than sociology and marketing. Thus I'm going to focus on the first two.

Problems are caused by *defects*, meaning a non-fulfillment of a requirement. Defects simply exist. However, most defects can be removed by performing changes on the item that caused them. For that, I introduce the term error. The error is a model of the underlying defect that makes it possible for the program to detect and handle it. These terms have a certain relation between each described in figure 3.

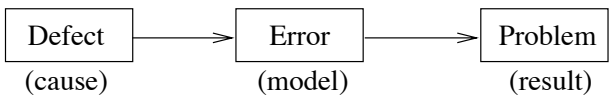

Figure 3: Relation between defect, error and problem

An error is caused by a *defect*. The problem would even be there if there wasn't any error. A program can happily crash or lose data without the programmer wasting a single thought on any possible defect at all. Putting it that way, errors are a model to deal with defects. In most cases, an error still imposes a *problem* for the user, as it interrupts the normal control flow of the program which would create the output. Therefore it is considered "subjective unsatisfactory".

2.4 Specification, Correctness, Acceptability

One of the myths of computer science is that *correctness* is the premium quality of a program. To be correct, a program simply has to act according to a *specification*. The specification must then be unambiguous, consistent, and complete. After that one has to fiddle a bit with design issues, maybe refine the specification iteratively, and eventually can do the implementation mostly automatically (Dromey, 1989).

In practice, this is rarely the case (Beizer, 1990). Lee & Anderson (1990, 33) give a simple example of a computer system that is struck by lightning. Nearly every program will behave in an arbitrary fashion. In particular, the nifty merge-sort routine for which some skillful mathematician proved two minutes ago that it is correct.

A less illusive goal than correctness is *acceptability*, which also refers to a set of conditions describing the current state of the program. However, the unrealistic need for completeness is dropped. Still it has to be unambiguous and consistent. Programmer and user then have to agree how close acceptability converges with correctness.

The main problem about acceptability in practice is that usually only the programmer decides about it. This has lead to the term "good enough software", meaning that it's good enough to be sold to users who are naïve enough to buy it and enter a never ending upgrade cycle. Or as the saying goes: Not good is good enough to release.

2.5 Robustness, Reliability, Applicability

Acceptability mainly aims at providing a correct output. This however is not the only requirement for a "good" program. Another one is *robustness*, meaning to behave "reasonably" under a wide range of circumstances, especially those which don't allow to produce an output for some reason (Liskov & Snyder, 1979). In simple words it means that a program should not just quit without any further diagnosis if it can't compute the output. This again depends on how much effort the programmer wants to invest.

A more specific term is *reliability*, commonly defined as a function $R(t)$ which expresses the probability that the system will be able to produce output according to its specification throughout a period of duration t. The well established theory of reliability modelling makes it possible to predict this time. But even more important, reliability can also simply be measured. The Mean Time Between Failures (MTBF) probably is the most well-known indicator (Laprie & Kanoun, 1996).

Still this is not enough if the program lacks *applicability* to the user's needs. For instance, if the user wants to compose a music tune, the best paint program won't help him. Finally, it is always the user himself who decides whether a program is useful or not. Although the program itself can't know that, it nevertheless should make the decision about applicability easy for the user.

2.6 Error Handling

If an error occurs during a routine, it has to do something about it - it has to *handle* it. First of all, the error has to be *detected*. Only after that, the routine can recover to some save state. Eventually, the error can be *reported*, either to the calling routine or the user.

The ultimate reason for error handling is to notify the user that there is a defect and provide him with information to fix the problem.

2.6.1 Error Detection

Before a routine can do anything about an error, it has to detect it. Essentially, there are two ways to detect an error: the first one is that a sub-routine detected an error and reported it back to the caller. The second is that there is a *contradiction*. This means that a value or state does not match an *expectation*. General ways to create expectations to provoke contradictions are (adopted from Lee & Anderson, 1990;):

- *Replication checks*, basically meaning a comparison with a complete backup.
- *Coding checks* don't need a complete backup but only give a "summary". Examples are various checksum algorithms like parity or CRCs.
- *Reasonable checks* validate conditions that must be maintained, for example array indices must be within a certain range, and a linked list must not contain any cycles.
- *Reversal checks* validate that the output can really be a result of the input. This only works if there is an inverse function. For example, a function to compute the square root of x could check if the square of the result yields x. (At least in theory; thanks to floating point math, this won't work. But the example outlines the idea.)
- *Timing checks*, where a certain operation did not finish within an expected period of time.

How and on which occasions these checks can be applied to identify specific errors is discussed later in detail.

2.6.2 Error Recovery

At the time a program encounters an error, in most cases one of its components is in an inconsistent state. Speaking less abstract, this usually means that resources have been allocated to facilitate the computation of the output, routines have been called and changed the program state, while other routines where only intended to be invoked.

In theory, error recovery is a general concept applicable to all systems that can easily be implemented, e.g. by taking a "snapshot" of the program and its state before invoking a routine. Provided that all recovery operations were successful and the snapshot was taken at a time when the system was not in an erroneous state, error recovery can remove all defects from the system (Lee & Anderson, 1990). In practice, the following problems arise:

1. Taking the snapshot is usually too expensive.
2. Recovery operations might fail.
3. Some components are unrecoverable, for instance there is no way for the program to snapshot and restore the user. (At least not to our current knowledge.)

In practice, program components instead choose one of the following ways to recovery from an error:

- *Halt* the whole program. This is the most violent way to deal with errors, and programmers should try to avoid it. Sometimes this is not possible, in particular when crucial hardware components fail or bugs become apparent.
- *Reset* the current component, meaning that the component looses all information about its current state, often including the user input.
- *Terminate* the current routine. In most cases, this is rather violent and can easily turn the calling component in an inconsistent state. Nevertheless, in some simple cases this is a proper way of recovery.
- *Cleanup* and then terminate the routine. This basically means to "undo" all effects of the operation interrupted by the error. In most practical cases, this is the way it should be done.

Halt and terminate are trivial to implement. All serious programming languages support functions named like abort or die to stop a running program. (The naming seems to reflect the violent nature of such an action.) Terminating a routine is generally also trivial. Reset can sometimes be implemented by setting state variables to defaults. But often, reset and cleanup requires a lot more effort, and is very cumbersome without any support of the programming language.

Ideally, recovery works hierarchically, meaning that a component can recover by recovering all its sub-components, together with its local information. Then, a program can be written so that it is able to recover from errors that do not require any change in its physical environment, in particular interpreting hardware and user. Still, recovery can imply data loss for the user. Another interesting point arising is how languages behave if the recovery actions fail.

2.6.3 Error Reporting

Error reporting can happen in two ways: first, the report can be kept "internal", where the failing routine just informs the calling routine and leaves further actions to it. Second, the routine can decide that the user has to be informed.

An internal error report usually consists of a couple of data that describe the problem, generally error codes and references to relevant objects. This is appropriate for a programmer who should write some error handling code for a calling routine.

If however the user should be informed, this internal representation somehow has to be turned into a human-readable, meaningful error message. Sometimes this can mean to consult the administrator or programmer. Generally, error messages should be specific and precise, giving exact information about the error and its cause, constructive, suggesting how to proceed and defensive, blaming the system rather than the user (Shneiderman, 1997). Proper error reporting may make the difference between being able to use a program and obtain results or being frustrated and unable to do anything with it.

2.6.4 Error Correction

The information of the error report is used to identify the underlying defect or the consequently apparent problem, and correct it. This can again happen in two ways: first, automatically by the program, and second, manually by the user, administrator or programmer (depending on the type of error).

Automatic error correction, though sounding seductive, imposes a lot of problems. Different to recovery or manual correction, there is no general scheme how to do it. It needs to be part of the design, and basically has to be implemented differently for every case. Often, there is no proper automatic corrective action. But what's maybe more important, it can significantly impair the user's understanding of the system, and might cause many subsequent errors or a major annoyance.

2.6.5 Defect Avoidance

Many defects can be avoided - and consequently, resulting errors and problems. But when toying around with programming constructs, implementation details and fancy data structures, programmers tend to forget that. Weinberg observes that "problem-avoiding behavior [...] is intelligent behavior at its highest, although not very intelligent if one is trying to attract the eye of a poorly trained manager. It will always be difficult to appreciate how much trouble we are *not* having" (Weinberg, 1971, 165).

Mechanisms to avoid defects are:

- *Simplicity*, meaning to avoid complexity. Simple systems are easier to comprehend, modify and adopt. There are several ways to achieve that, with the popular ones being: *Omission*, meaning to get rid of features that are unneeded. *Consistency*, basically meaning to do similar things in similar ways, thus reducing the cognitive memory load and increasing the hit-rate in finding existing "patterns". *Partitioning* decomposes a system (and the problem it is supposed to solve) into its constituent parts. This is done by establishing a vertical hierarchy to express increasing detail, and a horizontal hierarchy decomposing the problem (Pressman, 1997). (Lee & Anderson, 1990) call it the principle of "divide and hope to conquer", which avoids that system is constructed as a monolithic entity. Instead, it it is built as a coherent assembly of component sub-systems.
- *Conservatism*, meaning reluctance to change. It avoids that new defects are introduced into a system. Belady & Lehman argue that any system that is used undergoes continuing change, and that as a result its structure inevitably degenerates (quoted in Lee & Anderson, 1990, 46). In a more practice oriented view: "If it ain't broken, don't fix it. Stable systems are stable because they don't change much. (Duh.)" (Halfhill, 1998)

Mechanisms that avoid further defects after one already occurred, or at least reduce its impact are:

- *Independence*, to avoid that defects spread over a system and also infect other components that formerly had no relation to the defect. It can be measured using two qualitative criteria: *cohesion*, meaning that a cohesive component should (ideally) do just one thing, and *coupling*, measuring the interconnection among components (Pressman, 1997).
- *Redundancy*, meaning an exact copy of a component exists which can be used if the original fails due to physical defects.
- *Diversity*, meaning that a copy of a component exists using a different implementation with the same specification. The copy might be immune to design defects that can make the original fail.

- *Replicability*, to create an exact copy of a system simplifying redundancy. This is trivial for software.

Notably, defect avoidance has to be part of the design and can only be considered before the system becomes operational. Furthermore we should not expect all defects to be avoidable. But this is no excuse for not even trying.

2.7 Discussion

So far, terms were introduced and explained without reflecting much on them. This seemed appropriate for explaining the basic ideas of error handling without confusing details. Although the above terminology will be used throughout the rest of this thesis, there are some issues that require further clarification.

2.7.1 Conflicting Terminology

Many of the terms defined above have different meanings, not only in other areas of science, but also speaking in terms of computer jargon. This is a summary of other possible definitions and a rationale why they have not been used above.

Reason claims that the psychological study of human error does not require a precise definition due to its inductive nature and suggest as working definition to take error "as a generic term to encompass all those occasions in which a planned sequence of mental or physical activities fails to achieve its intended outcome, and when these failures can not be attributed to the intervention of some change agency." (Reason, 1990, 9) Still he concedes that this has merely proven useful in a psychological rather than a philosophical sense.

A popular definition used in many other sciences is

Error = True value - Approximation

Here, the *true value* is what the user wants to get, and the *approximation* is what he really got. Generally the true value is a number, and the reason for the approximation can be found in round-off errors, experimental error (probably arising from measurements) and truncation errors. As a result, the actual value of the error rarely can be evaluated exactly, but only be estimated. There is no such thing as a true value, but only an interval within which the true value is assumed to reside with a computable probability. Sophisticated methods exist to predict this interval, in particular when errors are propagated through several numerical expressions (Kreyszig, 1993). A classic example is the representation of floating point numbers in computers. For instance, with many formats the value 0.1 can not be expressed

exactly.

Discussing programming in a non-numerical context, this definition is not very helpful for several reasons:

1. If we would know the true value, in most cases there would be no need to create the program.
2. Programs process data as exact digital bit sequences, there is no approximation.
3. For most complex data structures, the minus operator (-) is not defined.

Clearly, these errors happen while converting analog real world data to a digital representation for the program. Evaluation of the impact on the accuracy of the output is not part of the program, but the programmer and the user. To be realistic, not many programmers and users are aware of these facts (Kahan & Darcy, 1998). Despite all this, the above definition essentially reassembles that of a contradiction as used earlier:

Contradiction = Computed value *does_not_match* Expectation

Contradiction is no fuzzy number but an explicit boolean value representable by a single bit. *Computed value* and *expectation* can both be complex data structures, and the *does_not_match* operator is a lot easier to define than the minus operator. The contradiction however is not the whole error, rather than the only way to detect it.

Pressman (1997) defines *error* as "some flaw in a software engineering work product or deliverable that is uncovered by software engineers before the software is delivered to the end user" (Pressman, 1997, 82) If the same flaw is uncovered after delivery to the end user, he speaks of a *defect*. While this distinction might be of importance for the commercial success of a program, it does not have much influence on the program code. Apart from that, he does not specify what exactly he means by "some flaw".

Meyer (1997) uses the following terms to denote "software woes":

- An *error* is a wrong decision made during the development of a software system.
- A *defect* is a property of a software system that may cause the system to depart from its intended behavior
- A *fault* is the event of a software system departing from its intended behavior during one of its executions.

He then identifies the a causal relation: errors cause defects, which in turn cause faults. This is in many ways similar to the use of defect/error/problem I'm using, but dismisses the impact of defects (or errors in his words) resulting from physical source. The fact that computers rust simply cannot be contributed to a programmer's decision. Next, "intended behavior" is a rather defuse issue, which is probably best commented on by the following user statement:

> "I know you believe you understood what you think I said, but I am not sure you realize that what you heard is not what I meant." (quoted in Pressman, 1997, 286).

Another term often used in the meaning of error is *exception*. There were however several reasons not to use it:

1. Exceptions don't necessarily indicate something "wrong", but something that just rarely happens. Thus it can easily be part of the "normal" control flow and result into an output.
2. Exception is widely used to refer to one particular technical concept (out of many) to handle errors. This would have been confusing for the later discussion.

I came to suspect that scientists and engineers feel uncomfortable when using such common words as error because even the man from the street could understand what they are talking about. An example for this claim can be found in (Scheuning, 1996). The author summarizes several definitions from official sources (ISO, IEEE, ANSI, ...) to substitute the word error in a sample text excerpt by words such as mistake, fault, failure, deviation and problem, while also discussing defect, nonconformity and omission. Personally, I can not endorse his claim that the example paragraph become more understandable, and would suggest to use a less terse wording. At best, his definitions might work within a rather small group of people with a quite similar background.

Strong & Miller (1995) give a more business oriented view of *exceptions* as "situations that cannot be correctly processed by computer systems alone, and thus require manual intervention to produce output that meet organizational goal". Their article is a report from the "real world", where it has been proven several times that a binary view of the world as correct or in error is too narrow. Still, I will advocate that a programming language should provide a deterministic set of binary mechanisms to deal with a non-binary world. Resolving this mismatch on a higher level is the job of the programmer and its creativity, not the language. One does not try to build a less skewed house by using awry nails.

Among other definitions, Raymond et al. (1999) first describe program as "A magic spell cast over a computer allowing to turn one's input into error messages". Although this has been written with a humorous intention, it is remarkable that programs created from people with a similar cultural background to the authors (*hackers*) are infamous for their notoriously bad

error handling code (if there is any at all), not to speak about the wording of their error messages. Further notably, *programmer* remains undefined in this publication. Opposed to *real programmer*, which deserved a considerable amount of space and devotion.

Meyer (1997) uses the term *software engineer* instead of programmer. In his discussion about multiple inheritance, he suggests a class SOFTWARE_ENGINEER, which inherits from ENGINEER, POET and PLUMBER. Although I found this a remarkable fitting description, I was slightly repelled by the word "engineer". This can be explained by my cultural background. In Austria, there are two levels of engineering: first, the academic level (*Diplomingenieur*), and second, the common one (*Ingenieur*), for which one doesn't need to visit any university, but only a special school. The cliche then goes something like this: the *Ingenieur* doesn't really know what he is doing but simply applies a set of rules he never reflected on. Most of the time, he somehow manages to make things work, but usually far worse then he is able to confess - especially to users. Nevertheless, in most cases he can get the desired amount of money. The *Diplomingenieur* on the other hand is all the time busy objecting the rules of the *Ingenieur*, pretending to look for better ones. Normally, without success. Different to the *Ingenieur*, he doesn't produce anything the user could apply on real world problems. Nevertheless, in most cases he can get the desired amount of respect in academic circles. Now there are people who observe that many programmers are a mixture of these two species: neither do they understand what they are doing, nor do they have any intention to solve real world problems. Although this might be true for several of them, it is not the kind of programmer I'm hinting at.

Another popular wording is to distinguish between *computer scientist* on the positive side and programmer on the negative. Until yet, I couldn't think of a set of criteria that makes you a computer scientist opposed to a programmer. My personal experience is that most people who call themselves computer scientist have no idea what it's like to create a program with more than 1000 lines of code. So they definitely have no relation to the programs discussed herein.

All this however yields to the observation that pretty much every one can write a program that produces an output that can satisfy the user. Still, it doesn't make him a programmer. Because of that, I came to see programmers as "people, who create programs" (as defined above). These are distinct from the "people who write programs". An easier to understand analogy could be a poet, who creates poems. But not everyone who writes a poem automatically becomes a poet.

As a final note, I'd like to mention that other people might prefer different terms for the same meanings. As a conclusion, it can be said that it's not the term, but the meaning that matters.

2.7.2 People vs. Programs

At this point it is appropriate to emphasize that the discussion is about programs, not people. Though there seems to be some consent that the human "program" is the mind, opinions differ about its specification.

Pinker gives an interesting view: he sees the mind as a "system of organs of computation, designed by natural selection to solve the kinds of problems our ancestors faced in their foraging way of life, in particular understanding and outmaneuvering objects, animals, plants and other people" (Pinker, 1997, 21). He observes that the mind is capable of solving certain illposed problems without a literal solution. It does this by providing assumptions for missing information. These assumptions can then be coded into genetic information and passed on to children. According to the logic of natural selection, the ultimate goal of the mind is to maximize the number of copies of the genes that created it. But he opposes common misunderstandings, like that spreading ones genes is the point of human life or that natural selection is a puppetmaster that pulls the strings of behavior directly. He also explains how concepts seemingly useless for this specification (like art, humor and friends without sexual interest) fit into this picture.

Other views have been proposed by philosophers throughout the time, with none of them being accepted by the human race in general. Zima (1997) presents many of them from a contemporary perspective but, like Pinker, opposes fashionable beliefs when demanding:

> "Ein Theoretiker zerstört seine Theorie nicht, wird nicht zum Ideologen, indem er einen besonderen - liberalen, konservativen, sozialistischen oder feministischen - Standpunkt einnimmt. Er zerstört sie im Diskurs, sobald er als für den Ablauf verantwortliches Aussagesubjekt den semantischen Unterschied zum manichäischen Gegensatz, zum Dualismus werden läßt, seine Rede als semantisch-narrative Konstruktion mit der Wirklichkeit identifiziert (verwechselt) und dadurch Gegenentwürfe und Gegenargumente monologisch ausgrenzt." (Zima, 1997, 369).

Another difference is how the human mind and programs change. For their whole life, people learn. They reflect upon the "input data" they get while living. They obtain new experiences and interaction patterns with the world, which are stored in the mind and modify its "program". Computer programs on the other hand don’t change depending in the input data. They just select from a limited amount of interaction patterns hard-coded by the programmer. The input has influence on the behavior, but the program will not try to modify its code when it experiences something outside its specification.

Nevertheless, certain parts of the human mind are similar: the aforementioned assumptions inherited from ancestor. Interestingly, they don't change during the life time of the individual being, but during reproduction. Pinker observes:

> "Natural selection is not the only process that changes organisms over time. But it is the only process that seemingly *designs* organisms over time" (Pinker, 1997, 159).

This "design" is based on random errors during copying the genetic information. While most copies will change to the worse, some will have minor improvements - out of plain dumb luck. When they compete with others over the limited resources in the world (which forbid them to reproduce endlessly), their lucky improvement will make it easier to reproduce, and thus to pass them on to the next generation.

Computer programs written in structured or object-oriented languages are quite different: it is trivial to create a byte-by-byte exact copy of a program. That's why software piracy is so common. But changes are only induced by the programmer, and hopefully only after careful considerations. Still, there is natural selection: it is the user, who decides whether a program is used or not. Without users, the programmer will soon lose interest in changing the program. The program can very well change data, but it doesn't change its own code.

For the sake of completeness, it's inevitable to mention that there are of course programs that modify their own specification: neural networks. But their "programming language" is usually distinct from structured and object-oriented languages. It is difficult to predict what's really going on. Probably worse, such networks are extremely difficult to change if their output does not match the user's requirements. For instance, Cohen & McCloskey trained a network to "add 1 to any number". When it later was trained to be capable to also "add 2 to any number", the new problem sucked the connections weights over to "add-2" and made it forget everything it had already learned about "add-1" (Pinker, 1997, 122). The application of such programs generally is not to reliably turn exact input into deterministic output, rather to come up with some "sort of" solution for which programmer's can't find a useful specification (e.g. reading handwriting, searching pictures after keywords etc.).

2.8 Summary

This chapter introduced the required terminology to further discuss error handling. As it turned out, many established terms are not very useful for the restricted context of programming languages. The following notions deserve a close look:

A program has the following atomic components: the *interpreter hardware*, which executes it, the *storage hardware* eventually used to obtain input and store output, the *user*, who specifies the input and has desire for the output, the *administrator*, who is responsible for hardware and software-components to be in proper shape, and the *programmer*, who created the program.

Realistic values to judge about the quality of a program are *acceptability*, meaning that it conforms to specification which in practice always is incomplete, *robustness*, meaning that it behaves reasonably under error conditions which the specification usually doesn't describe, and *applicability*, meaning that the user benefits from the output. In practice, their fusion can be measured as *reliability*.

3 The User's View

This chapter discusses error handling from a user's point of view. It describes possible program responses to errors and considers when they are appropriate. It summarizes general guidelines on wording, sentence structure and highlighting mechanisms useful for error reporting. Finally, it illustrates a simple scheme to create good error messages derived from the contradiction used to detect the error.

3.1 Program Responses to Error

General ways in which a system can respond to errors are discussed by Reason (1990), which are adopted from (Lewis & Norman, 1986). However, his elaboration only focuses on potentially desirable responses. Because this would be incomplete, I'm also describing some others rarely talked about.

3.1.1 "Let's Talk about It"

The program begins a modal dialogue where the user receives a message describing what's wrong. Common choices are to retry or cancel the operation, read the online help on the current operation, read the online help on the error message or obtain more details by e.g. activating a debugger environment. The actual choices of course depend on the error, and not all of them are useful or available for all kinds of errors.

In almost all cases, this is the most appropriate response. First, because it should give a fairly complete description of the error, and second, because it induces a *forcing function*, meaning that it "prevents the behavior from continuing until the problem has been corrected" (quoted in Reason, 1990, 161). A simple example taken from the real world is a locked door. Unless the "user" turns the appropriate key, the door won't open.

3.1.2 "Gagging"

A gag also is a forcing function. Different to the "let's talk about it" strategy, gags do not offer any explanation why they prevent the user from executing a certain operation.

This can be useful in cases like direct manipulation interfaces, where popping up dialogs all the time would quickly be considered an annoyance. An example is a mouse pointer that can not leave the visible screen area. If the user tries to move "outside", the pointer remains at the border. In this case, the error is "obvious". But for many applications, gags are insufficient error feedback.

3.1.3 Warning

Whereas gags present a block to anything but appropriate input, warnings simply inform about potentially dangerous situations. The user is left to decide the correct course of an action.

Meyer opposes this concept with the following observation: "Warnings may be viewed as an act of cowardice: you don't dare to refuse the submission because it "might" work, but you don't make any commitment either as to the effect of processing it, because it might just as well not work! This is really a way of passing on your responsibility to your poor users." (Meyer, 1998). Consequently, warnings can be seen as indicator for a defective design.

3.1.4 "Teach Me"

On detecting an unknown or inexact input the first time, the program asks the user what to do about it. After that, it knows what to do in future: accept it, or treat it as error, but don't ask again.

This is common for applications having to deal with concepts from the real world, which is often highly indeterministic and unpredictable. An example is a spell-checker, that considers the user's own name as spelling error. Because no dictionary will ever contain all the names human beings invent all the time all over the world, serious spell-checkers offer a function to extend the dictionary with new words. This declares a former typo to be perfectly correct.

3.1.5 Automatic Correction

The program tries to guess some action that corresponds to the user intention. This basically is, what all users seem to wish for. However, as the word "guess" suggests, the underlying mechanisms are rarely very deterministic.As this is a complex issue, it deserves a detailed discussion later.

3.1.6 "For some reason it doesn't work" Syndrome

Until yet, all responses have been more or less desirable. As practice shows, programs sometimes select from less popular choices.

It so not so uncommon that a program neither creates an output nor an error. This completely leaves the task of sorting out what went wrong to the user. Reason claims that this is only helpful when adequate feedback information is available, but is unable to give a specific example.

I thus rejected Reason's original term "Do nothing". I cannot think of a sensible design decision why a program should do that. Most of the time this happens because the programmer did not consider a certain error situation at all. Such a behavior can be considered a bug. Although it does not have any negative impact on the state of the program, it confuses the users.

3.1.7 Halt

Sometimes there is no sensible way to continue, though the program is still in charge. For example, when an internal consistency check revealed that the program has a bug. Here, it is often the best thing to stop all activities to avoid that the error spreads further.

A halt usually implies one of the following effects:

- *Data loss*, meaning that the user partially lost his data, typically the input of the current session or everything since he saved the last time.
- *Data devastation*, meaning that certain data were destroyed permanently. This can happen when the program was halted in the midst of an action like updating a database record.

However, as the program is still in charge, it can estimate the extent of data affected, and tell the user in an error message. Different to "Let's talk about it", there are no dialog choices like "Retry" to influence the control flow from the outside. They only option is to quit. (Online help might be provided, though).

3.1.8 Reset

Sometimes, a halt is not a proper response insofar as the system should remain available. In such a case, the program might decide to reset itself, meaning to start from the beginning. Although in most aspects similar to halt, most of the functionality and the data are available again. A reset doesn't necessarily have to affect the whole program. Sometimes it can be possible to reset only the routine context.

But like halt, this usually implies data loss and has to be reported to the user.

3.1.9 Crash

If a program does not know how to handle an error, it can easily crash. Halfhill (1998) discusses the anatomy of a crash, and essentially all reasons for a crash are missing or improper error handling.

A crash is very annoying because it does not give any clue to the user why the program doesn't work (unlike halt and reset). Furthermore, it is almost impossible to estimate the extent of the problem on the data. The only "nice" thing about a crash is that it is completely obvious, and the user is aware that there's something wrong, though he might have no idea, what.

3.1.10 Continue with Latent Error

Thimbleby (1998a) observes that computing systems can fail, but not stop working. If the program produces an output when the proper response would have been to report an error, it has a bug. But, different to a crash, this is not obvious to the user. Instead, it tricks him into believing that the operation was successful. He might use the output as input for further steps, which again might work out. When eventually the error becomes apparent, he has to face data devastation with a huge cause-effect chasm. Such errors can stay invisible for a long time and are thus called *latent errors*.

Often, the final exposure happens because of an accident, where several unusual situations combine. Reason explains the contribution of latent error to system disasters: "Their part is usually that of adding the final garnish to a lethal brew whose ingredients have already been long in the cooking." Reason (1990, 173). Consequently, latent errors are far worse than a crash.

3.2 Wording of Error Messages

Errors are usually reported using text information. The literature for human computer interaction (HCI) has a huge repository of guidelines on how to word error messages, for example:

- "Good error messages are defensive, precise and constructive" (Molich & Nielsen, 1990).
- "Messages should be appropriate to the user's level of knowledge" (Brown, 1988).
- Avoid words like illegal, fatal, abort, corrupt, bad, error (Shneiderman, 1997).

From a programmer's point of view, most of these guidelines are essentially useless, some of them even ridiculous. What a programmer needs is not a pile of informal guidelines and pages filled with "Don't, don't, don't". Instead, he needs data structures and algorithms. And what he really wants are ready implemented routines and libraries.

Despite this, it is to some extent possible to collect the more specific part of HCI literature and suggest a detailed path to design useful error messages without having to interpret around too much.

Sentences should be short and have a simple grammatical structure. This is generally achieved by:

- using present tense
- using active voice
- placing the main topic at the beginning
- telling what to do rather than what to avoid
- describing sequences of instructions in the order of the temporal sequence of events
- not being overly polite

Reconsidering that an error always is detected by a contraction of the form

Contradiction = Computed value *does_not_match* Expectation

an obvious way to derive an error message from it is to announce that

Computed value must match expectation

Less abstract, this usually means to have error messages of the form

Something must be something else
Something must have something else

The programmer's job then is to specify something and something else. Brown (1988) hints that something is the name of the fields or parameter in error, and something else is one of the following: the action required to correct the error, the correct format for the field, or a list of valid entries for the field. For instance

Month must be a number between 1 and 12
Size must be one of S, M, L or XL

However, if the message includes elements not related to the user input directly, it is necessary to put them into a context. This is done by preceeding the above message text with a description of the form

Cannot do something

Do something indicates the operation the program attempted to carry out. For instance,

Cannot save image to file "a:hugo.png"

If this still does not have any meaning to the user, more abstract descriptions of form Cannot do something form can preceed the message. Generally, one error can be described by a sequence of Cannot do something messages, with a Something must be something else message at the end suggesting what can be done in order to fix the problem.

Following the above scheme should quickly result into "good" error messages without a lot of consideration. However, this does not mean that there are no better messages. Thus, improving, revising and evaluating them in practice is necessary. Gould & Lewis (1985) and Shneiderman (1997) give some advises on how to proceed.

3.3 Where to Report

The optimal placement of error messages on the display is another undecided matter of HCI. Basically, there are three possibilities (Shneiderman, 1997):

1. Near the erroneous item, so it is easy to see what is wrong. Some claim that this clutters the display.
2. At the bottom of the display, at a consistent position. This avoids clutter, but might permanently waste space. Also, the human eye might have a long way between error message and cause.
3. Pop up a dialog box in the middle of the display. This is popular in practice because it is trivial to program. Often, the dialog covers the erroneous item, requiring the user to waste time with reorganizing his windows. Another quirk is that such dialogs are often modal. This implies that the user can see the information about the error, but not fix it in the blocked part of the program. When he cancels the dialog and can start fixing, all information about the error is gone.

3.4 Colors, Highlighting and Audio

Certain colors have meanings associated with them, and thus can be used to emphasize error related information. Table 1 lists codings relevant for error reporting (Brown, 1988).

Table 1: Color codings relevant to highlight errors

Color	Use
White/Black	Base color
Red	Errors or alarms; Stop
Yellow	Warnings or data that may require attention
Green	Normal or OK; Go
Pink (Magenta)	Secondary alarm color

However, uncritical addition of color to displays is not uniformly beneficial. Lots of different colors and highlighting make the screen "noisy" and slow down the extraction of information. Experience shows that six different colors for text on one screen seems to be an upper limit. And as many users are color-weak and some systems may have monochromatic displays, the programmer cannot rely on colors only.

Auditory signals such as bells and beeps can be useful to attract the user's attention. This is even true if he does not currently look at the display. However, overuse of auditory signals can defeat their purpose and may also annoy users. This is particularly true if several users share the same room. There such signals can also have an embarrassing effect if they are associated with an error. Therefore a mechanism should exist to turn off non-critical audio signals or control the volume.

Again, one cannot rely on audio signals only. Not all systems have the required hardware to play them. And a simple beep can only point out that something went wrong, but cannot "describe" further details - this is the task of the message text (Shneiderman, 1997).

3.5 An Example

The application of the above is probably best outlined when suggesting how to improve the error message of an incident that really happened: The user of a paint program selected the "Print" function from a menu, which caused an error message announcing "Disk full error".

Despite the fact that the message does not meet any requirements of a useful error message, it is not really clear how the printing process can cause a disk to be full. A closer investigation eventually revealed that the program does not send the data to the printer directly. Instead, it renders to a temporary file created on the disk to save memory. This file did not fit on the disk, and caused the error.

The first step of the improvement is to obtain the something must be something else form:

Volume T: must have more free space.

In this case, the program might even know how much space exactly is needed for the temporary file, as it depends on the resolution of the picture, which is already known before the file is written. A further improvement would be:

Volume T: must have at least 3.8 MB free space.

However, the user will still be puzzled by the fact that printing can make a disk run out of space. The context of the error has to be described more exactly by prepending a Cannot do something type message:

Cannot write temporary file:
Volume T: must have at least 3.8 MB free space.

A further refinement could be to make it clear which operation applied on which data failed:

Cannot print image "hugo.eps":
Cannot write temporary file:
Volume T: must have at least 3.8 MB free space.

The first line can be omitted, if the context is clear without it. For example, if the error message interrupts a printing progress indicator. Using a window title like "Printer problem" in the dialog for the error message might also be sufficient. Notably, the above lines have to be displayed at the same time, as they belong together.

One interesting fact is the usage of the rather technical term "temporary file". Inexperienced users might have no idea what it means. HCI people often demand that technical terms have no place in error messages. Alas, the dirty implementation details of the temporary file is why the error occurred. Would the printer buffer be in memory, this error couldn't happen. But then, without mentioning the temporary file, it is not clear why the printer runs out of disk space.

Clearly, the user cannot understand why a printer runs out of disk space without being told about the temporary file. Furthermore, he cannot understand the error message without knowing what a temporary file is.

A simple solution to this dilemma is to have an online help button accessible from the dialogue. The help text can explain that a temporary file is a common programming concept to store big amounts of data created during a complex step on the disk rather than in memory. They are called temporary because the program automatically removes them once the operation is finished.

4 The Program's View

This chapter characterizes different kinds of defects and error situations. It discusses how to detect the various types of errors and what details to report to the user.

4.1 Defect Classification

Laprie & Kanoun (1996) basically distinguishes between *physical defects*, caused e.g. by corrosion or material fatigue, and *design defects* introduced by the creator of the system. Applying this on programs, the following atomic components can have either hardware- or design defects:

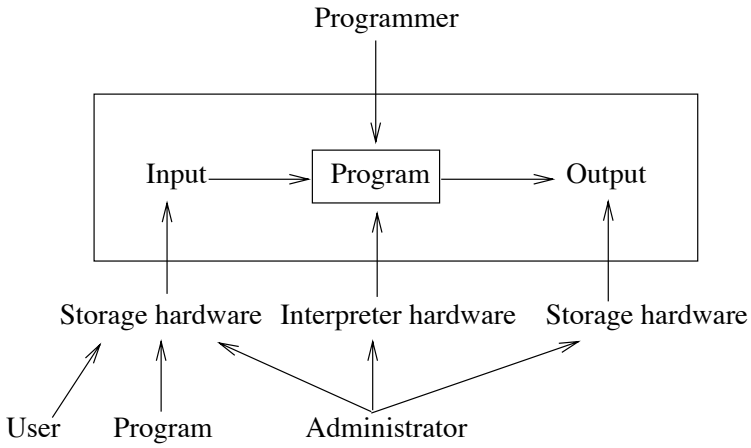

Figure 4: Atomic components of the program environment

In this simplified view, the program is an abstract entity designed and used by humans, which has a physical representation in the hardware.

4.1.1 Design Defects

Currently, all design defects are introduced by human beings. As already outlined, roughly there are three different kinds of people that can cause the program to run into trouble:

1. The *programmer* can fail to create an acceptable program.
2. The *user* can fail to provide a reasonable input.
3. The *administrator* can fail to provide a working environment for the program speaking in terms of hardware and software components.

Unfortunately, people are incredibly creative at generating problems, and incredibly fast. Furthermore, they prefer to respond with action, instead of pausing to sort things out (Carroll & Aaronson, 1988). Reason uses the term *unsafe acts* to describe the human contribution to a state that eventually results in a problem:

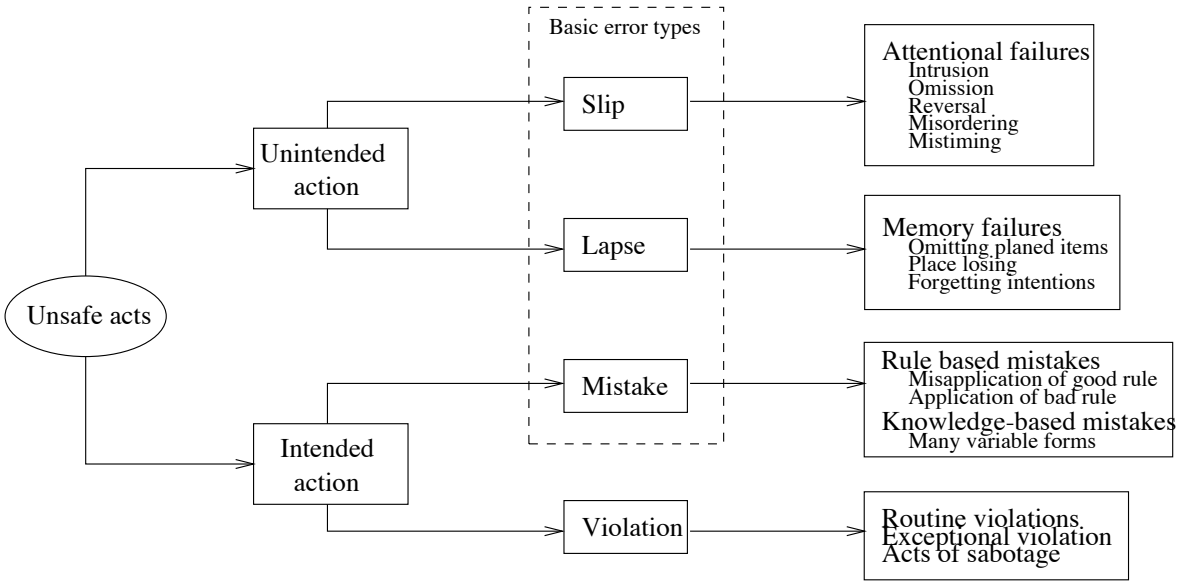

Figure 5: The psychological varieties of human unsafe acts (Reason, 1990, 207)

This raises some trouble with the terminology: are unsafe acts human defects or errors? From a program's point of view, it seems to be appropriate to describe people as potential defect source, but use the term "error" to talk about how people become "defective". The notion of *human error* instead of human defect has also been used by (Norman, 1983; Pinker, 1997; Reason, 1990; Schmidt & Bisang, 1998).

Applied on the context of using a program, unsafe acts can be described as:

- *Slips* happen when a person does something wrong, though he didn't intent to and actually would know better. For example, the user types "their" when intending to type "there".
- *Lapses* indicate that the user forgot to do something he wanted to. For example, he wants to save a text before quitting an editor. But then he has a sudden idea, makes some last changes and exits without saving.
- *Mistakes* happen when the intention of an action is wrong. Somebody "intentionally" carries out a sequence of actions that - maybe even possible and correct for the program - do not give him what he wants. For example, the user might start a plain text editor with the intension to write a text with fancy layout and multiple fonts. While he will be able to write text, he won't find any functions to change layout and style.
- *Violations* occur when somebody deliberately does something "wrong". Often this is the only way to work around a defect. Sometimes it is laziness because there is no reasonable "right" way. If the intention is to cause damage to the system, we speak of sabotage. Violations are no real errors. They often hint at defects in the program design or the environment.

The reason for committing unsafe acts can be contributed to basic properties of the human mind, like limited memory and a desire to simplify and generalize knowledge in patterns. Pinker gives an exhaustive discussion, though he admits that we still don't know exactly what is really going on (Pinker, 1997).

Norman further categorizes slips and lapses into a small set of classes based on the mechanisms that seem to be the most likely cause, focusing on the user (Norman, 1983). The most common ones are:

- *Mode error:* when the user believes that the system is in one state though it isn't. An example is the text editor vi, which has two modes: a command mode, where the user can do things like moving the cursor, and an insert mode, where he can enter text. Often users forget in which mode they are.
- *Description error:* when insufficient specification of an action is given and the user is not sure what to do. An example is the meaning of the d key in vi, which is different if pressed alone (d), with Shift (D) or Control (^D).
- *Capture error:* When there is an overlap in the sequence required for the performance of two different actions. For example, vi has a command ":w" to save the current text and then continue editing. Another command is ":wq", which is used to save and exit. The second command is very frequent, and often the user finds himself out of the editor back in the command line because by a capture error he typed ":wq" instead of ":w".
- *Activation error:* An appropriate action fails to be performed. This is usually caused by the fact that the user has a very limited memory. As a result, he might forget something he intended to do just seconds ago.

Norman points out that many user errors can be avoided by a change in the program design or the documentation. Common literature on human factors already describes these issues. Basic guidelines are: avoid modes (to prevent mode errors), use different command sequences for different classes of actions and avoid overlapping sequences (to minimize capture and description errors), try to make actions reversible ("undo") and keep the whole system consistent (to avoid activation errors).

However, this is easier said than done. Just try to specify what "consistent" means. Removing modes is not always possible for complex systems or just increases the frequency of description errors. And overly cautious or verbose program design can limit the usefulness for experienced users. It should not come as a surprise that with HCI a whole branch of computer science discusses these trade-offs.

Human error is no sole property of users. The role of the programmer is described in more detail later.

4.1.2 Physical Defects

For the program, physical defects refer to errors that lie within the physical representation of the program or the data it processes. Essentially, this means the hardware:

- A defective storage hardware is unable to read input or write output. For example, a floppy disk after spending some time in the sun.
- A defective interpreter can stop to interpret the program. For example, a power failure.

Hardware is distinct from software in many ways, which generally makes it more reliable. First, there is a long history during which many experiences were collected, allowing to predict physical decay quite accurately and consequently replace a component before it fails (Laprie & Kanoun, 1996). Second, hardware systems are generally simplier than software, have small interfaces and don't change much over the time.

The reasons for that are not that hardware people are generally smarter than software people, but that hardware errors are immediately expensive speaking in terms of money, resulting in a bigger pressure to be careful. One can't just upload an improved CPU to the Internet. Furthermore, because hardware wears off, users are forced to upgrade anyway, and do not shun a generation change from time to time (unless they depend on old programs running on the new hardware). Newer generations often include a redesign.

This avoids that a system degenerates over decades like it is the case with many programs, whose design doesn't "wear off" over the time. It just becomes more and more inapplicable, but it doesn't physically fall apart.

Physical defects can be avoided by providing multiple identical copies of a component. The canonical example for this is Triple Modular Redundancy (TMR). To make a component in Figure 6(a) tolerant to physical defects, it could be replaced by the TMR system in Figure 6(b):

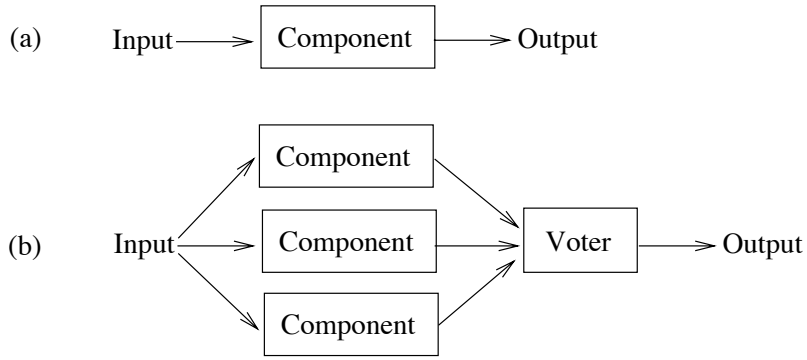

Figure 6: Triple Modular Redundancy

The system consists of three copies of the component of identical design and a *voter* which checks the value of the majority. If thus a single component fails but the remaining two agree in their output, it is possible to detect the broken component and continue using the output of the others (Lee & Anderson, 1990). However, eventually all copies will be broken. So someone has to replace the demolished components by working copies before this can happen.

4.2 Error Classification

A program cannot detect defects. All it can do is reflect upon its current state and point out inconsistencies. To do that, it has to provoke contradictions, which it does by comparing its state to expectations. These expectations have to be provided by the programmer. Once a contradiction turns out, the program detected an error. It proceeds by refusing to produce the output. Instead, it gives a description of the error, and, depending on the error and possible user requests, decides how to continue.

For every class of errors, the following issues are taken into account:

- Further classifications of the situation allowing a more detailed discussion. This thesis mainly aims at giving the big picture.
- How to perform the actual detection and what type of contradictions can be useful
- What to report about the error, so that the description can be considered precise enough for the user to figure what went wrong.

4.2.1 Input Error

Generally, programs translate input to output. An *input error* indicates inadequate input data which cannot lead to any output.

4.2.1.1 Classification

Actually, this class of error should be covered by one of the busiest disciplines of computer science: parsing and compiling. The expected input format is often described by means of a context-free grammar, where such errors are further categorized as lexical, semantic and syntactic errors (Sippu, 1981; Dain, 1991). This approach has the problem that it does not exactly describe what an error is. Various models attempt to fix this, with the most popular ones being:

- The *minimum distance error* measures the shortest way to transform a syntactically incorrect input into a correct one. The transformation can be done by basic edit operations, typically insert, delete or replace a single symbol.
- The *parser defined error* in an incorrect string, with respect to the language, marks the point at which a prefix of the string ceases to be a prefix of the language.

The main trouble with this seems to be the lack of a context. Dain observes: "The human who reads the source program [=the input] makes use of much more information than is available to a parser or syntax error handling scheme, including not only tokens from source characters, but also layout of the program, context-sensitive information such as types, and logical meaning" (Dain, 1991, 5).

Things become easier when data are structured into hierarchies and sequences of data items, as for recursive descent parsing (Aho et al., 1985). This allows every hierarchy to be seen as context on its own, also having an own grammar. Though this sounds very abstract, such structures are basically everywhere, for example: A sentence is a sequence of words, and every word is a sequence of letters. Object oriented programs consist of a sequence of classes, a class is a sequence of routines, routines are sequences of statements etc. Quite recently it has become fashionable to structure text data, like in XML and its predecessors (Bray et al., 1998). Morrison (1985) explains a less restricted and resource consuming way also applicable on general binary data, which can be used to store graphics, music, configuration files etc and lately influenced the PNG image format (Boutell, 1996).

A sequence of data items basically can contain the following simple errors:

- *Wrong items* are items that are simply not allowed to show up in a certain context. For example, the letter "ö" is not part of the context "ASCII character".
- *Missing required items* can only occur in a context that has required items. For example, an executable C program requires a `main` function.
- *Duplicate items* can only occur in a context that does not allow them. For example, a C

compiler rejects the two variable declarations `int x; int x` because there can be only one variable with the name `x`.

- An *incomplete item sequence* is also possible without required items within the context, for example the mathematical expression `((a+b)*c` lacks a closing `)`.

4.2.1.2 Detection

The detection of missing required and duplicate items should be straight forward for most programmers.

To detect an incomplete sequence, two approaches are possible:

1. The first item in the sequence describes the expected amount of items in the sequence. If the sequence ends earlier, it is apparently incomplete. For example, many programming languages implement a string datatype that way.
2. A special item describes the end of sequence. For example, a mathematical expression started with "(" must end with ")".

Wrong items are described by the rules in the context, thus every item can be validated against it. It is however not trivial to define these rules. Common concepts are:

- Possible ranges. For example, "ö" is not an ASCII character, but can be part of a sequence if it is stored in a resource that allows 8 bit values.
- Redundancy. This is commonly done by adding items that contain no new data but only a checksum computed over all other items within the context
- Order. Certain items are only allowed at a certain position, for example `(a+b)` is an expression, but `)ab(+` is not though it contains the same items
- Version information. Many operating systems for example have library items with a version number. If the version is too small, the library is considered "wrong", meaning outdated.

Not all these mechanisms are possible or feasible in all cases. The main difference is if input data items were entered directly by the user (such as text) or formerly stored by a program that interpreted more abstract commands (such as a paint program). For example, we cannot expect a programmer to compute an Adler32 checksum (Boutell, 1996) for every line of source code he enters. On the other hand, such a checksum can easily be computed by a program that writes a data hunk in an image.

4.2.1.3 Reporting

To create a descriptive error message, the context needs a name the user understands. The same goes for every item. For example, the item "a" can be in the context "letter", the item "duck" in the context "word", or the item "Work:user/hugo.png" in the context "file".

Some of the above errors are detected at a certain *position* in the sequence. Others can only be detected at the end of the sequence and should use the start of the sequence as error position. Duplicate items have two positions: the first where the original showed up, and the second where the reoccurrence happened. Both have to be given in an error message.

In binary data, the position is often of no use for the user and can be omitted. It does not help him if he knows that in the PLTE chunk at the file position 768 the checksum should have been 0x7632 instead of 0x8743.

In text data, there is a line and a column. Internally in the program the position is often stored as index, that is translated to line and column for reporting. A line number alone is not enough, although it is unfortunately common in practice.

4.2.2 Component Allocation Error

Programs need hard- and software components to process their data. These can be memory, disk space or a network connection, but also external libraries or bitmaps. Before a component can be used, it has to be *allocated*. If this for some reason is not possible, we speak about *component allocation error*.

4.2.2.1 Classification

A component allocation can fail due to the following reasons:

- *Exhaustion* if the capacity of the component is not sufficient for the amount of data to be stored in it. One main thing to observe is that all components can run out, no matter how numerous they seem. This can simply be explained by the fact that there is only a limited number of atoms in the universe.
- *Unavailability* occurs when there is no component matching the allocation parameters.
- *Denial* indicates that the system has a component, but does not allow the program to allocate it. Possible reasons are security restrictions or exclusive locks acquired by others.

4.2.2.2 Detection

Components are maintained by a *component manager*, for example by the operating system or the compiler (Stallings, 1998). This manager is supposed to know about the whereabouts and properties of his components, and thus can identify most impossible allocations by simply scanning the list of available components for a fitting one.

In some cases, the manager has to utilize time-checks. Examples are physically broken connections to hardware devices like printers or networks machine using certain operating systems and protocols.

4.2.2.3 Reporting

Many components act on a quite low level. Very often it is completely impossible for the programmer to design a message in terms of the user, avoiding tech-jargon. For instance

> The baud rate for serial.device, unit 3, must be at most 9600.

Or maybe the reason was as follows:

> The device driver for serial.device, unit 3, must be updated to at least version 3.7. A newer version can be obtained from the vendor homepage at http://www.hugo.com/.

Probably every first year student of computer science can outline the fields of a device datatype that would allow such messages. But it seems that the implementers of most common operating systems and APIs cannot.

In case of time-checks, no useful error message can be generated because no communication channel exists to exchange error information. This clearly is a design defect in the component management.

4.2.3 Component Access Error

Components can deny to work even after their successful allocation. For example, a network connection can get broken during data transfer because somebody stumbled over a cable.

From a programmer's point of view, these are probably the errors from which it is most difficult to recover because the program has already undergone several state changes, but it is not easy to decide which. Often, a reset of the current routine context is the only feasible thing to do.

4.2.3.1 Classification

No further classification is necessary. Either it occurs, or not.

4.2.3.2 Detection

Like allocation errors, access errors are usually detected by the component manager. Generally, it will use coding checks (like checksums of a disk block) or reverse checks (like a verify after a write operation).

4.2.3.3 Reporting

Component access errors can be detected quite exactly, and often a huge amount of information is available about it. However, most of this information like internal disk block numbers or network ID's is not helpful for the user, like in:

> Disk read error on block #1234.

In most cases, the only thing to report is the name under which the user addressed the component. For the above example, this would mean a filename:

> Cannot read "hugo.data": disk structure must be repaired.

Under most operating systems however all this is not possible because the internal resource access structures, like file handles, do not store the name under which the file was opened. The same goes for network connections, where the system soon knows nothing more than an internal handle ID. This results from the fact that the human readable name used to allocate a resource is not needed anymore for reading, writing and releasing it later. (Possible routines to reconvert the handle to a name often do not work reliably during error situations).

4.2.4 Program Bug

Different from all the above mentioned errors, with bugs the error is in the program code and can only be fixed by the programmer. All a user can do is "work around" it, avoiding to execute those parts of the program that cause the problem.

4.2.4.1 Classification

There are already several attempts to classify bugs in literature (Beizer, 1990; Chillarege, 1996; Eisenstadt, 1997; Knuth, 1989). Unfortunately none of them turned out to be useful for exact reproduction in this paper. Either the existing classifications were too detailed, or

certain important categories were omitted. The following rough classification nevertheless is based on the above references, with emphasis on the source of a bug.

- *Programmer mistake:* Requirements and the specifications developed from them can be incomplete, ambiguous, or self-contradictory. Examples are usage of inapplicable algorithms and data structures, overlooking of special cases - especially error handling, or wrong understanding of the system the program runs on like CPU byte order, linefeed character or integer range. Common reasons are that the programmer did not understand the system, certain algorithms or the requirements of the user. The latter is often a result of the fact that the user did not know himself what he wanted or couldn't explain it properly to the programmer. Sometimes it is due to conflicting goals of different users because of political reasons (Strong & Miller, 1995).
- *Programmer slip:* All of the above mentioned slips can happen to programmers when entering the program code if tools and languages have a flawed design. Common examples taken from the C language are:
  - Mode error: The expression `(x + (y << 16))` is written as `(x + y << 16)`. This can happen because the shift left operator (<<) has a lower priority (=unexpected mode) than the addition (+).
  - Description error: The function call `strstr(haystack,needle)`, used to search for a substring `needle` in a source string `haystack`, is called as `strstr(needle,haystack)`. As both parameters are of the same type, no compiler error is produced.
  - Capture error: The conditional `if (x == 3)` is written as `if (x = 3)`, acting as a fully legal assignment.
  - Activation error: After a `case`, the `break` is forgotten.

  Many slips are typos and can be caught by the compiler if it requires things like variables and routines to be declared. Nevertheless many dialects of Basic and several scripting languages do not enforce this. This makes it faster to type the program code, but often induces bugs that can only be detected during runtime.

- *Component clobbering* refers to memory clobbering, releasing resources and still access them later on or "forgetting" to release a resource after it is not needed anymore. (Programmers are often fully aware of the fact that their program contains resource leaks. But finding and fixing them is near to impossible with current tools.) It can also happen to components mostly maintained by the compiler like variables and the stack. One might write into an array outside its defined boundaries or use an uninitialized variable.
- *Reuse error:* The actual bug was within the compiler, the operating system or an off-the-shelf component provided by somebody outside the own team. According to

Eisenstadt (1997), this is the second most frequent root cause of "difficult" bugs.

Another version of this kind is when code "known to work" is reused in a new program under circumstances it was not designed for. Jézéquel & Meyer (1997) describe a famous example.

But most common, reuse bugs are introduced after performing a change to the code in order to fix a bug or add a feature. It can be said that the new version of the program "reuses" most parts of the old version. The change can have unexpected influences, causing new bugs in other places.

- *Documentation errors* are the most common kind of bugs - and often considered the least harmful. Although many of them are simple spelling mistakes, several are misleading and lead to incorrect maintenance, therefore causing the insertion of other bugs, especially reuse errors (Beizer, 1990).

4.2.4.2 Detection

The general mean for detecting bugs during runtime is called *assertion*. This is a boolean expressions defining the correct state of the program at a particular location in the code. If the expression does not resolve to true, the program detected a bug. Technical details are discussed later.

There are several ways to detect bugs without executing the program. One is to scan the source code for them, preferably by a different person because this increases the chance of detection (Reason, 1990, 165). In an organized way, this is called *software inspection* (Gilb et al., 1993). Many documentation bugs can be exposed using common spell checkers (surprise!). Furthermore, one can *test* the program by providing an input and comparing the actual output with the expected (Beizer, 1990). But Chillarege (1996) laments that such practices are largely human-intensive processes, which are qualitative, suffer poor repeatability, and have difficult introduction barriers. They are regularly subject to violations and carried out with less care if the deadline is approaching. Additionally, such tasks can only be performed before the program is shipped to the user, not giving any protection afterwards.

4.2.4.3 Reporting

When the program encounters a bug while running on the programmer's machine for development and testing purpose, it should attempt to provide exact information about its state when the bug was detected. Traditional ways to do that are showing a stack trace or automatically activating a special debugging environment. Still, Lieberman observes in his

elaborations on the "debugging scandal": "[Today's] debugging tools are little better than the tools that came with the programming environments 30 years ago. It is a sad commentary on state of the art that many programmers identify 'inserting print statements' as their debugging technique of choice." (Lieberman, 1997)

A completely different case is when the program detects the bug while executed on the user's machine. The user does not know anything about the meaning of internal procedure names and hexadecimal representations of pointer values. Providing the same information as before on the screen probably causes panic symptoms. Instead, the program should attempt to write these internal data to a file and urge the user to submit it as "automatic bug report" to the programmer. Figure 7 shows an example wording for such a message.

Figure 7: Example wording for reporting a bug to the user

> The application Hugo-Tool encountered a condition not anticipated by its programmer.
>
> Because the accuracy of your data can not be ensured anymore, the program is halted. This means that all data of your current session with it are lost.
>
> A document that contains a detailed technical description of the problem has been stored in "Work:Hugo-Tool.bug". You are not supposed to understand the contents of this document, but it will help the programmer to fix the problem in the next release of Hugo-Tool.
>
> Please submit the file "Work:Hugo-Tool.bug" to bugs@hugo.com.

But it is necessary to emphasize that bug detection is not a task that should be carried out by the user. Joyner (1996) criticizes current development practice: "The real inconsistencies are often removed by hacking until the program works, with a resultant dependency on testing to find the errors in the first place. Sometimes companies depend on the customers to actually do the testing and provide feedback about the problems. While [bug] reporting is an essential path of communication from the customer, it must be regarded as the last and most costly line of defence."

The advantage of the above behavior is that it represents a forcing function that does not leave the user many other choices than reporting the bug. Consequently, the programmer might fix it, or at least face a user that demands an explanation why not. For most applications this is preferable to automatic corrections.

However, sometimes this is not acceptable for the user: in systems where a software halt would cause loss of life or financial ruin, as with software in aero planes or medical equipment. Lee & Anderson (1990) discuss ways for damage assessment, so that the program

can reset itself to a state where there is a good change to prevent a disaster. However, these "software fault tolerance", as it is commonly called, is not without danger. In any case, an automatic bug report should be stored in a log for the administrator. In such critical systems, he can be expected to actually read such logs. The programmer can then be informed about the bug once the plane managed to land with one engine less on the nearest airport (or the like).

But such sophisticated mechanisms require a careful design, and administrators that actually care about their job. This is definitely not the case with most personal computer systems, simply because it would exceed all costs and be generally unrealistic. Here, halting the program is definitely more appropriate, as otherwise latent errors would be the most likely result.

5 The Role of the Language

This chapter analyzes various error handling mechanisms provided by widely used programming languages. It describes all these mechanisms, their underlying ideas and some implementation details. The analysis is done by validating every mechanism for a set of criteria and implementing some simple example routines.

5.1 Analysis Method

The following error handling mechanisms are analyzed:

- Return codes that report about success and error in C
- Status indicators that hold error information in C
- Traps that can be handled in Basic
- Exceptions that are thrown and caught in Java
- Assertions that can fail in C and Eiffel

The analysis is done by first giving a short outline about the scope of the various languages and their culture and history. It proceeds by explaining the usage in theory and attempts to implement a couple of small programs showing how to perform common example tasks with it. After that, implementation details and their implication on performance are investigated. A detailed discussion of the issues identified follows in the next chapter.

5.2 Criteria to Evaluate

The analysis evaluates the usefulness of the mechanisms in terms of control flow, error reporting and dealing with bugs. For that, it will examine the following set of criteria:

Is ignoring errors reported by sub-routines impossible? Of course, a programmer can always decide not to do anything about an error, but nevertheless, a programming language should not allow him to do so without some effort.

Can parameters be passed to error handlers? Errors are not only simple numbers, but often need other values to describe their nature. This is also crucial for useful error reports.

Can the mechanisms be used by the programmer for own errors? If only the language and the standard library can use a mechanism, things become far more complicated for the programmer: he has to write his own error handling mechanism, and error handling code requires an unnecessary distinction between programmer and language errors.

Does the standard library itself use the mechanisms consistently? Naturally, it is not desirable that badly designed libraries use other, generally more primitive, mechanisms when dealing with errors.

Can errors be handled in sequential and hierarchical groups? This is useful for writing error handlers that want to deal with more than one error, but not yet with all of them. Sequences allow to assign several distinct errors to a handler (like "read error" and "write error"). Hierarchies are useful if there is an abstract relationship between them (like "all I/O errors").

Is there no or only a very small performance overhead for the code if there is no error? If such an overhead is noticeable, programmers might be tempted to reduce the number of error handlers if they slow down the program. The overhead when an error occurs is not considered here because it has no influence on the normal control flow.

Can error handling code be separated from productive code? This avoids that the normal code is obfuscated by error handling code, which would make it a lot more difficult to understand and change for the programmer.

Are retry and resume supported? As discussed, these are two distinct ways to return from an error handler to the normal control flow.

Are errors delegated automatically to possible outer handlers, if no proper handler exists within the current routine? Error handlers should only deal with the errors they know how to cope with. Using handlers that can handle all errors compromises the component structure and leads to complexity.

Can the error message be generated automatically from an error reported from a routine? Ideally, only one line of code should be needed to turn an internal error event into a descriptive message that can be presented to the user. Everything else is torturing the programmer.

Does the above error message support Internationalization? In recent years it has become popular to deliver programs which support more than one language with dialogues and labels. Does this also work for error messages?

Is the error message quality considered in the standard documentation of the programming language, and do the examples use a wording basically conforming to the guidelines discussed before? This is important, because programmers tend to ape the style and wording of this documentation in their own code.

The rules change a bit when detecting bugs, especially concerning error reporting:

Is the exact position in the source code available, where the bug was detected? This is not particularly powerful, but a minimum requirement to allow the programmer tracing it back.

Is a detailed stack trace available? This includes a lot more information, in particular values of parameters for nested routine calls, which usually allow to reproduce the bug quite easily, even if it happened on the user's machine. Preferably, the trace should be accessible as e.g. string so that it can be used for automatic bug reports. As discussed, displaying the trace to the user is not helpful.

Can the detection of bugs be (partially) disabled to satisfy performance and space constraints?

5.3 The Languages

5.3.1 C

The C language was originally designed around 1972 at Bell Laboratories as a general purpose language, and its datatypes and operators have a close relation to the underlying hardware of most computers. This still makes the language popular for system programming if low overhead is important.

The first language reference was written by Brian Kernighan and Dennis Ritchie in 1978, usually referred as "K&R-C". Although the language and its standard library were considered to be portable, this was not really true. Consequently, in 1989 the American National Standards Institute (ANSI) introduced ANSI C. This did not extend the original K&R-C significantly, but mostly defined existing practice as standard and removed or changed some things that were known to cause trouble. Nowadays, ANSI C is supported by virtually all C compilers that are still in maintenance (Harbison & Steele, 1991). Therefore from now on when referring to C, it means ANSI C.

The language itself is very small and does not provide any language specific features for error handling, thus making it interesting to examine from a minimalists point of view. Error handling according to C can be done in nearly every programming language, including machine languages.

Joyner (1996) discusses many deficiencies of the C language and its successor, C++. Ritchie (1993) explains some of the obscure design considerations behind C, in particular concerning the syntax.

5.3.2 Basic

Basic was originally designed for an experimental timesharing system in the early 1960's. The name was chosen simply because it was a simple and basic programming language. People later on made up various "backronyms", with "beginner's all-purpose symbolic instruction code" being the one that is still in use today. During the 1980's, Basic was widely used on several low-end microcomputers because it is comparably simple to use and does not require many resources. These early dialects are often offended because of the bad programming habits they teach due to the total lack of control structures except if .. then goto. Later dialects have acquired most control structures from Pascal and similar languages and are popular among application programmers who are not keen on programming on the system level (Raymond et al., 1999).

Still, the dialect evaluated here is a quite early one. It is used on a Sharp Pocket Computer E500 developed in 1989. This system is equipped with 32KB RAM where both the running program and a RAM-disk reside. Data in the RAM-disk are preserved even if the computer is turned off. A common setup is to use about 10K as program memory and the rest for the RAM-disk. (Sharp Corporation, 1989)

At a first glance it might seem a bit odd to evaluate such a constrained system. But despite the progresses in computer hardware, even today not all programs run on machines with several megabytes of memory and CPUs clocked close to the GHz range. Apart from that, variations of the mechanism described here can still be found in several scripting and batch languages that are widely used to control bigger applications "from outside" or add macro capabilities to otherwise cumbersome to use systems.

To avoid being unfair to contemporary Basics, the capabilities of the modern VisualBasic dialect (Microsoft, 1996) are occasionally outlined in the shadow of severe disturbances of the E500. The main reason for not discussing this particular dialect in the first place was that it is a strange mixture of traps, exceptions and assertions, that has not been designed from the scratch but has incorporated many features over the time. Despite the usefulness in practice, scientific reasoning would have been difficult due to the resulting complexity and various interferences.

5.3.3 Java

Java was originally called Oak, and designed for use in embedded consumer-electronic applications by James Gosling. After several years of experience with the language, and significant contributions by several other people it was retargeted to the Internet (whatever that means) and renamed Java. It is a general-purpose concurrent class-based object-oriented

programming language, specifically designed to have as few implementation dependencies as possible. Java claims to allow application developers to write a program once and then be able to run it everywhere on the Internet (Gosling et al., 1996).

A critical discussion on certain Java features is available from (Thimbleby, 1998b).

5.3.4 Eiffel

Eiffel is an advanced object-oriented programming language that emphasizes the design and construction of high-quality and reusable software. It was created by Bertrand Meyer and evolved since its first introduction in 1986. The definition of the Eiffel language is in the public domain. It is controlled by the Non-profit International Consortium for Eiffel (NICE), which uses (Meyer, 1992) as the initial definition. Meyer (1997) describes a more up-to-date version in a less formal way. A set of standard classes is described in (Bezault et al., 1995). However, this standard is not supported by any of the compilers available today, partially because it is widely considered incomplete and outdated.

Meyer (1997) also includes critical discussions of most language features, in particular concerning the assertion mechanism. Kogtenkov (1998) maintains a collection of sample source codes that expose undefined and inconsistent cases in Eiffel. Liskov & Wing (1994) point out limitations of assertions in Eiffel and some other languages, while proposing solutions.

5.3.5 Other Languages

In my opinion, the above choice of programming languages gives a good mixture of different approaches to error handling. They also represent quite distinct cultures of programming. All of them have a certain practical relevance, and have actually been used by many people to create programs. Additionally, I could access them without further expenses, usually because Freeware compilers are available.

Although the main focus will be on those language, it doesn't mean I didn't consider other choices. Some possible popular alternatives and reasons not to include them are:

- Ada 95 (Taft & Duff, 1997) is complicated enough that Ichbiah, the creator of the earlier version, resigned publicly from the reviewing group after trying in vain for years to keep extensions simple. Many remarkable features like dealing with concurrent errors are of no relevance for my analysis.
- Two of C++'s (Stroustrup, 1997) error handling mechanisms are identical to C, and the exceptions handling is more considered and cleaner in Java. Although there is an ANSI

standard, it is very recent. It is difficult to find a compiler that really supports it. Apart from that, with C++ even more cultural redundancy would have been added to C and Java.

- Oberon (Wirth, 1988) never got an affirmative standard for its libraries. This would have made it difficult to reason about the examples. Attempts like Oakwood were incomplete and not implemented. Error handling in Oberon doesn't offer really significant improvements to C (which has a very explicit standard). Though the same can be said about Eiffel and its ELKS, it at least has a much more powerful assertion mechanism.

Generally, they all represent so called hybrid languages, which support both a structured and object-oriented style of programming. This might make them useful as transition technologies for programmers getting used to a new style. But it also makes them bigger and more complex, and consequently more difficult to analyze.

Nevertheless, some of the above languages contain interesting ideas that will occasionally be mentioned during the analysis and discussion.

5.4 Usage

5.4.1 Special Results in C

Usually, functions return values that describe the output. If there is no output because of an error, it is possible to indicate this with special values in the result. To deal with the errors, the programmer has to use conditionals (if...else...) that check for a result indicating an error. For example, an attempt to open a file in C can look like:

```
file = fopen(filename, mode);
if (file != NULL) {
   ... /* continue */
} else {
   ... /* handle error */
}
```

The function fopen() returns NULL if the file couldn't be opened. All other results can be interpreted as an internal handle to a successfully opened file.

There are several obvious problems with this approach:

- A special value might not exist (Stroustrup, 1997). An example for that is the soon to be discussed strtol(), where every returned long can indicate a proper result.
- A special value might be an incomplete way to describe the error or pass parameters to it. For instance, the NULL returned by fopen() does not contain any clues why it didn't work.
- It is almost impossible to preserve error information over multiple levels of calls as the

error has to be "converted" to the result type of the current function every time.

- Even plain procedures without an actual result have to be turned into functions. This makes it more difficult for the programmer to find out what a routine actually does when browsing the documentation.
- Every error has to be re-detected by the programmer over and over again, upon every routine call. This can blow up the source code to a magnitude (Stroustrup, 1997).

Additional problems of the C library and language are:

- Many results are completely useless. For instance, the routine printf() to format and print text returns the number of characters it printed. As the formating can be very complex, the programmer does not generally know how many characters should be printed before calling the routine, and can't compare the result to it.
- Function results don't have to be assigned to a variable (like in Pascal), but can be ignored. This makes it even easier to forget to handle errors.

5.4.2 Status Indicators in C

A separate variable or function can be used to indicate the status of a routine concerning possible errors. Meyer (1997) discusses two possible schemes for that:

5.4.2.1 A Priori

A routine that might cause an error has an accompanying function that tells if the routine will work with a given input. For example, one can attempt to invert a non-singular matrix like this:

```
if (!is_singular(matrix)) {
   invert(matrix)
   ... /* continue */
} else {
   ... /* handle error */
}
```

Possible errors are first checked by a boolean function is_singular() that returns TRUE if the matrix cannot be inverted. The call to invert() is only allowed after all requirements have been ensured.

5.4.2.2 A Posteriori

Another way is to try the operation first and then check how things worked out. In C, this requires to either declare a global variable or make it part of the result data structure. Here, the field matrix->inverted is used:

```
invert(matrix);
if (matrix->inverted) {
   ... /* continue */
} else {
   ... /* handle error */
}
```

5.4.2.3 The Actual C Implementation

The C library uses the a posteriori scheme based on the global integer variable errno declared in <errno.h>. It is initialized to 0, which represents "no error". Most functions set this variable to other values if detecting errors. However, library functions must not set errno to 0 by themselves. Thus a typical usage involves the following steps:

- Set errno = 0
- Call a library function
- Check errno to be 0. If not, handle error

The function strerror() declared in <string.h> allows to convert the numeric value of errno to a human-readable text. In more detail, it just returns the proper entry from an array of string constants holding the corresponding message. The language used in these constants can be changed using routines in <locale.h>.

Possible values for errno are not standardized and depend on the library implementor. Exceptions are the constants EDOM and ERANGE, which indicate "domain errors" (e.g. when attempting a log(-1)) and range errors in mathematical functions. Furthermore all error codes have values greater than 0.

Stroustrup (1997) points out that the use of a global variable is generally clumsy and doesn't work well in presence of concurrency. The latter problem however can be addressed by making the status indicator part of a data structure, like in matrix->inverted - at least as long as matrix is used by only one process at a time.

5.4.2.4 Comparison

The a priori scheme has several obvious problems addressed by a posteriori:

- Efficiency considerations might make it impractical to use. In the above example, both is_singular() and invert() would probably use the same rather expensive algorithm (Gaussian elimination) to compute its result.
- External events might render the result of the first function useless before the second function is called. With is_singular(matrix) this was not a problem. But as a counter example, consider opening a file in a multitasking system. An openable()/open() pair would be rather useless here.
- It introduces redundancy in expression. The programmer has to type in and learn how to use two routines to achieve one task.

Meyer (1997) reflects on status indicators in object-oriented languages. Obviously, they should be part of the class that provides the routine possibly causing an error. This solves the problems arising from using a global variable like in C. However, new trouble is introduced:

- It is impossible to describe errors that prevent the creation of a new object (commonly called creation routines or constructors).
- It causes a memory overhead as every object of the class carries most of the time useless error handling queries with it.
- The status indicators are part of the "normal" class interface, thus polluting it with features often irrelevant for the output.
- They often have various names like impossible, failed or connected, making it more difficult for the programmer to find out what constitutes an error.

And finally, same as special results, status indicators still rely on the programmer to manually check if things worked out.

5.4.3 Traps in Basic

The basic idea behind trapping is that once a routine detected an error, the control flow automatically changes to an error handler. Consequently, the handler can be separated from the normal control flow, unlike conditionals as used above. Goodenough (1975) gives an in-depth discussion of the general principle.

5.4.3.1 Traps in E500 Basic

In the E500 Basic dialect, the programmer can activate an error handler routine at the label *handler by using the statement on error goto *handler. As the statement suggests, the interpreter automatically performs a goto *handler if a routine detects an error.

The variable ern ("error number") contains a numeric code for the error that occurred. These codes are documented in the manual, for example 76 indicates that a file is write protected. The variable erl contains the line number from where control was passed to the error handler.

Apart from terminating the program, there are three things an error handler can do after attempting to fix the problem:

- resume returns to the statement that caused the error and re-executes it.
- resume next returns to the statements after the one that caused the error and continues executing the program.
- resume *label continues the program at the specified label.

In all three cases, the error handler remains active. Thus if the same error or a different one occurs after the resume, the same error handler is called again. This can easily result into endless loops. To avoid this, the error handler can be deactivated using the statement on error goto 0.

If the programmer installs an error handler while there is already one active, all information about the previous handler is discarded. In particular it is impossible to temporarily install an error handler and reactivate the former one by deactivating the new one using on error goto 0. Thus nesting of error handlers is not supported.

If an error occurs with no error handler provided by the programmer, the default error handler terminates the program and displays a message containing a description of the error and the line number where it occurred. One remarkable fact is that the programmer does not have any possibility to utilize the message text obviously depending on the value of ern - there is no erm$ ("error message string") or the like.

Basically, all functions of the standard library use this mechanism and reliably activate the error handler in case of trouble. However, many functions tend to ignore dangerous conditions that might be considered errors. For an example, see the val function below. Unfortunately, the programmer cannot use the mechanism for own errors.

5.4.3.2 Some Idiosyncracies

The E500 Basic dialect does not support routine parameters and local variables. Routines can be called using the gosub statement, return jumps back to the caller. Parameters and results can only be exchanged by means of global variables.

Many early Basic dialects required the programmer to mark every line of code with a line number. This was mostly done because resource limitations made more reasonable editors impossible. The E500 supports this style for backward compatibility with earlier pocket computers of the same series. Alternatively the programmer can also use labels, resulting in a better readable program. Labels have to be placed at the beginning of a line and must start with an asterisk (*). After that, the name of the label follows. For example, "*quit" declares a label that can be reached using goto *quit.

To make the examples more legible, some minor modifications have been applied before inserting them in this text. The numeric line numbers have been removed, normal code is indented to separate it from labels and the whole code is rendered in lower case instead of the original upper case.

5.4.3.3 Notes on VisualBasic

The modern VisualBasic dialect has addressed many of the problems pointed out above. It supports nested routines with local variables and error handlers, essentially allowing a similar control flow as soon to be explained with Java. Furthermore, the programmer can use the mechanism for own errors preparing a global Err "object" with an integer number to identify the error. There are optional fields for a descriptive error message and even a link to the help system. Once the Err is filled, it can be raised. This automatically causes the current position in the source code to be stored in Err, although only one level of the call stack. As Err is a global variable, nested errors cause all information about the initial error to be lost.

There is also a resume. Unfortunately, it also fails to work consistently: it always returns control to the statement in the current routine where the error was raised. If the raise happened on a lower level in the call stack, it actually performs a retry, restarting the complete lower level routine. Thus it is very difficult for the programmer to predict what is actually going to happen when issuing a resume.

Error handling can be disabled completely using on error resume next. This simply terminates a routine after an error is raised and continues with the next statement at the above level. This is rather violent, and it is almost impossible for the program to find out in which state it is as there is no alternative way to check for errors. The examples given in the

documentation don't seem to justify such an approach.

Remarkably, the documentation contains several examples with relatively useful error messages (at least when considering the others evaluated here). There are even a few paragraphs giving general guidelines on good wording. The only other documentation with similar qualities I'm aware of is the Ada 95 reference manual (Taft & Duff, 1997).

However, the mechanism has apparently been extended and patched for a long time without getting rid of several redundancies and inconsistencies. This made it unsuitable for a detailed evaluation.

5.4.4 Exceptions in Java

In Java, errors are reported to the calling routine via exceptions. In terms of control flow, this is comparable to trapping discussed above. Different to before, information about the error is not stored in global variables, but in exception objects.

5.4.4.1 Statements and Clauses

Exceptions are said to be *thrown* from the point where it occurred and are said to be *caught* at the point to which control is transferred. Error handlers are established by catch clauses of try statements, followed by an optional finally clause executing cleanup actions:

```
try {
   // Do something
} catch (Exception exception) {
   // Handle possible errors
} finally {
   // Cleanup actions
}
```

A try statement executes a block. If an exception is thrown and the try statement has one or more catch clauses that can catch it, control will be transferred to the first such catch clause. If the try statement has a finally clause, another block of code is executed, no matter whether an exception occurred or not.

The catch clause is comparable with the switch clause and essentially represents a multilevel if...else if.

5.4.4.2 The Throwable Class

Every exception is represented by an instance of the class Throwable or one of its subclasses. Such an object can be used to carry information from the point at which an exception occurs to the handler that catches it. Three basic subclasses of Throwable are used to implement the other, more specific exceptions. Figure 8 shows their relation in a BON diagram.

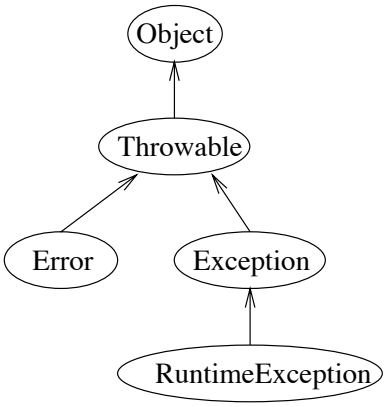

Figure 8: Java exception classes

Members of Error are usually thrown for serious problems, for instance VirtualMachineError, ClassFormatError and StackOverflowError. Members of RuntimeException indicate runtime errors of the program. Examples for such exceptions are NullPointerException, IndexOutOfBoundException and ArithmeticException. Often, they indicate bugs. Members of Exception, but not RuntimeException, are used to indicate abnormal conditions the programmer should decide how to handle while writing the program. Examples are FileNotFoundException, ParseException and SQLException. Usually, they indicate that something went wrong in the outside world.

5.4.4.3 Checked and Unchecked Exceptions

The compiler distinguishes between *checked* and *unchecked* exceptions. All members of Error and RuntimeException are unchecked, whereas all other members of Exception are checked.

When a routine calls another routine that might throw a checked exception, the compiler ensures that the developer either provides a catch clause for that particular exception (or one of its parent classes) or declares it in a throws clause within the routine header, for example:

```
void readSettings(String filename) throws IOException
{
  // Do something that might cause an IOException, but do not catch it.
}
```

5.4.4.4 Uncaught Exceptions

If a routine does not provide a catch clause for a particular exception, the exception is propagated to the first routine on the call stack that can handle it.

If no routine knows how to deal with it, all remaining finally clauses are executed and ThreadGroup.UncaughtException() is invoked. If a program does not implement this routine, a default handler depending on the Java environment is used. Most environments terminate the thread that caused the exception and print out its stack trace. If it was the main thread, the whole program quits.

5.4.4.5 Declaring and Using Own Errors

As already stated, exceptions are normal objects. Because of that, a developer can declare his own errors based on the class Throwable. Usually either the class Exception or RuntimeException act as base class:

```
class MyOwnException extends Exception {
}
```

An error can be detected and reported to the caller using the throw statement:

```
if (contradiction) {
  throw new MyOwnException(optional message);
}
```

Naturally, an exception class can have attributes and methods just like any other class. This allows to pass parameters by means of additional attributes extending the exception class. Preferably they are set by invoking the constructor, which should take them in its parameter list.

There is some confusion whether to use a checked or unchecked base class for own errors. This is discussed later.

5.4.4.6 Reporting Exceptions

The class Throwable has three functions to present its information in a human readable form:

1. toString() returns a short description of the object with most data only interesting for the programmer.
2. getMessage() returns the detailed message passed to the constructor when throwing it. In case of exceptions, this should be of a form that can be presented to the user without further considerations.

3. getLocalizedMessage() can be overloaded to return the message in the language preferred by the user. By default, it has the same result as getMessage().

A stack trace is available via printStackTrace(). This of course is again only of interest for the programmer.

5.4.4.7 Exceptions under Special Circumstances

There are several cases in the Java language where it is not initially obvious what will happen if an exception occurs:

If an exception is thrown inside a finally clause, it is terminated. The original catches are not evaluated again, but the exception is propagated to the encompassing try. All data about a possible initial exception thrown before the finally are discarded, thus all information about the original error is lost.

When a finalizer is called by the garbage collector, the programmer can not enclose its call in a try. If an exception is thrown at this point, it is ignored and the finalizer is terminated. Still the object is marked as finalized and will not be collected at a later point. This creates possible latent errors and resource leaks.

Exceptions thrown in a secondary thread can not easily be handled by the main thread. Instead, UncaughtException() of the secondary thread is invoked without the main thread getting notified. In many cases this is not what the programmer wants. DeRusso & Haggar (1998) suggest a workaround for this problem, where the main thread uses a listener class to be informed about exceptions in the secondary thread. But this compromises the original exception handling mechanism, forcing the programmer to think differently.

5.4.5 Assertions in Eiffel

Assertions are boolean expressions defining the correct state of the program. Simple examples are that an integer value must be greater than 0 or a pointer must not be Void (or NULL using C-terminology).

5.4.5.1 Assertions vs. Formal Methods

Assertions are based on work done by Floyd, Hoare and Dijkstra, which was also influential for several formal specification languages such as Z and VDM; Meyer (1997) gives detailed bibliographical notes. The underlying idea is that computer programs are mathematical expressions and their correctness can be proven by means of mathematical methods. Dromey (1989) discusses this from a formal and mathematical perspective.

However, things didn't work out as intended. Hoare gives the following reasons, why it is not easy to derive programs from specifications and proof their correctness: computers of the present day are poorly defined, mathematics has no tradition with dealing of such large scale expressions, programming languages are overly complicated and most programmers do not know how to handle the required mathematics properly (Hoare, 1986).

5.4.5.2 Types of Assertions

Eiffel addressed most of these problems in its Design by Contract. The idea is not any more to proof that a program is correct but to try very hard to expose it as incorrect. Though these are formally different things, the latter one can be done easier and suggests to be an improvement to "wishful thinking". There are several types of assertions, with the most important ones being:

- *Preconditions* must hold when calling a routine and are evaluate at the entry point. For example, a function to compute the square root of a given value `x` as non-complex floating point number only works for `x >= 0`.
- *Postconditions* check the outcome at the end of a method. For example, a function calculating the norm of a vector may check that the result is >= 0.
- *Class invariants* ensure the consistency of every object of a certain class. They are checked at the beginning and end of every routine invocation . For example, a class `LIST` may check that the element `count >= 0`.

These types are particular useful because they are part of the interface and thus also act as documentation. Further assertions dealing only with implementation specific details are *loop variants* and *loop invariants* to detect endless loops, simple *checks* for whatever conditions, validation for non-`Void` pointers when accessing attributes or calling routines and a few others.

Assertions replace cryptic mathematical notations by a quite readable English-like programming language. The proofing is not done by toying around on paper as with many formal methods, but by actually executing the assertion expressions in the computer during runtime. A nice side-effect of assertions is that they cannot only expose bugs in the own program, but within the compiler, standard library, operating system and sometimes even hardware components. Different to other languages, such events are quite apparent as soon as the programmer starts to examine the stack trace after a failed assertion, even if one of the own routines detected the error.

Another view of assertions is that they describe the *what* as executable code, not only the *how* as in more traditional programming languages. For the latter, the *what* often is only available as paper documents hardly anybody reads and maintains.

5.4.5.3 Assertions and Inheritance

Basically, in a descendant class, all ancestors' assertions still apply. However, some considerations are necessary when the new class wants to redeclare assertions and dynamic binding is involved:

For class invariants, the old invariant is still validated, but the descendant can have an own invariant with additional conditions.

For pre- and postconditions of routines, a new routine can

- Replace the precondition by a *weaker* one. This means accepting more inputs, which can cause no harm to a caller that satisfies the original precondition.
- Replace the postcondition by a *stronger* one. This means producing more than what was promised, which can cause no harm to a caller relying on the original postcondition being satisfied after the call.

Eiffel takes care of this by logically "or-ing" the old and the new precondition, and by logically "and-ing" the old and the new postcondition. To make this apparent for the programmer, it requires a slightly different notation (require else and ensure then instead of just require and ensure).

5.4.5.4 Developer Exceptions

Clearly, assertions are there to detect bugs in the program. For all other errors resulting from communication with the outside world, the standard libraries generally use status indicators. Additionally, Eiffel provides developer exceptions. Unfortunately, they are not remotely well thought out as the assertion mechanism. Facilities to raise developer exceptions and obtain error messages are different in (Meyer, 1992), (Bezault et al., 1995) and (Meyer, 1997). Actual implementations again differ in their possibilities.

Generally,the class EXCEPTIONS provides facilities to deal with exceptions and assertions. The trouble already starts when client programmers want to use this class. Some simply inherit from it, e.g.

```
class SOME_CLIENT
inherit EXCEPTIONS
...
```

Others prefer to declare a private expanded attribute for it to avoid polluting name space of the client class, e.g.

```
class SOME_CLIENT
feature {NONE}
  exceptions: expanded EXCEPTIONS
...
```

Again others declare this feature as a once routine which allows that several clients of EXCEPTIONS share the same attribute, e.g.

```
class SHARED_EXCEPTIONS
  Exceptions: EXCEPTIONS is
    once
      !! Result
    end
end -- class SHARED_EXCEPTIONS


class SOME_CLIENT
inherit SHARED_EXCEPTIONS
...
end -- class SOME_CLIENT
```

Most implementations allow to raise an exception using raise(name: STRING). The meaning of name is not further specified, and its contents are more or less arbitrary. In (Meyer, 1997), this routine does not exist anymore, and the whole EXCEPTIONS class looks different (Actually, almost identical to Ada 95 exceptions).

As neither the version described in (Meyer, 1992) nor (Meyer, 1997) contributes any remarkable new insights in handling of non-bug errors, I decided to not discuss the "convert a number" and "copy a file" example in Eiffel in depth. This decision was supported by the fact that the standard library does not use the exceptions mechanism to report I/O errors and impossible string to integer conversion, rather than class specific status indicators (similar to C's errno).

5.4.5.5 Recovering from Assertions

To handle assertions (and exceptions in general), a routine can provide a rescue clause. The purpose is to restore a consistent state that satisfies the class invariant. Additionally, the rescue clause can attempt to correct the error and issue a retry, which must also ensure the

precondition. If this is possible, the routine is considered to have failed, causing a Routine_failure exception (Meyer, 1997, 429).

The do...rescue idiom is basically the same as try...catch in Java. One major difference is the granularity: every routine can have only one exception handler. In practice, this is more an advantage than an obstacle: it avoids huge monster-routines, with the cost of a little more typing effort.

As assertions codes (and exceptions) are simple integer numbers, no new language constructs have to be introduced to write error handlers that can deal with more than one error. The build-in if...then and inspect...when already fully suffice, e.g.

```
some_routine is
  do
    ...
  rescue
    inspect
      exception
    when Routine_failure, Incorrect_inspect_value, ... then
      ...
    when ... then
      ...
    end
  end -- some_routine
```

Additionally, the EXCEPTIONS class supports a few queries checking if the last exception is of a certain type, e.g. is_assertion and is_developer_exception. Theoretically, the programmer could write own queries to build groups of exceptions belonging together, e.g.

```
is_io_error: BOOLEAN is
  do
    Result := developer_exception_name.has_prefix("Io_error")
  end -- is_io_error
```

In practice, this is jeopardized by the rather undefined (and probably soon obsolete) usage of name in raise. And because the string contents are not checked by the compiler, typos can easily result into bugs.

Every class can have a routine default_rescue that is called automatically if a routine doesn't have its own rescue clause. This makes is extremely simple to implement standard error handlers performing an object reset or other desperate measures. This is important because even a failed routine is not allowed to violate the class invariant.

5.4.5.6 Disabling Assertions

Assertion monitoring can be disabled by the programmer. Most Eiffel project tools allow to specify different, increasingly more complete levels of monitoring. For instance, it is possible to validate preconditions, but skip postconditions. However, it is not possible to skip preconditions when postconditions are validated. It is inappropriate to demand a routine to fulfill a postconditions if the routine cannot be sure if the caller fulfilled the precondition.

The compiler also disables assertions at one point: during routine calls in an assertion expression. There are two reasons for that: first, it is easy to construct examples where the called routine recursively validates the assertion where it was called from. And second, it would conceptionally put assertions on the implementation level, although they are on the "higher" specification level. However, Meyer does not give any considerations in how far this jeopardizes the possible detection of bugs in the specification.

5.4.5.7 Command-Query Separation

Naturally, assertion expressions must not contain any side-effects. Otherwise, the program would behave differently depending on assertion monitoring enabled or not. Ensuring this is left to the programmer. But this is not difficult:

Classes are supposed to strictly distinguish between commands (procedures) that can change the state of an object and queries (functions) that report about the current state of an object. There are two types of side effects: *concrete side-effects*, such as procedure calls or assignments, and *abstract side-effects*, which can change the value of a query accessible through the public class interface.

The *command-query separation* principle demands that a function must not have abstract side-effects. For instance, reading a character from a file in C is done using

```
character = fgetc(file);
```

Generally, the expression `2*fgetc(file)` and `fgetc(file)+fgetc(file)` will not have the same value because `fgetc()` implies an abstract side-effect: it moves the cursor of `file` to the next character. On the other hand, in Eiffel one has to use

```
file.read_character
character := file.last_character
```

In this case, `2*file.last_character` and `file.last_character+file.last_character` are always the same. In other words, a question does not change the answer.

Nevertheless, it is possible for a function to have side-effects while computing the result. It just has to make sure that they are undone upon exit. An example is a function that traverses a list advancing a cursor in order to find the maximum item stored in the list. Every cursor advance is a concrete side-effect. But as long the initial cursor position is restored upon exit, no abstract side-effects are noticeable.

If all functions of a class apply this principle (as they do in the standard libraries), the programmer does not have to bother whether a functions induces side-effects or not. He knows beforehand that none of them do. Other implications are easier maintenance and simplified reuse.

5.5 Example Programs

To avoid that the analysis is based on theoretical considerations only, a few example routines show how things actually work in practice. The example tasks have been chosen to have some practical relevance while still being easy to implement.

5.5.1 Convert a Number

A routine takes a text string as parameter and converts it to a number This is a very simple example for a parser. Because a low-level routine should not perform any automatic error correction, leading and trailing white space or a plus sign (+) are not allowed.

For the error report, the value is assigned to a field or parameter named size. Error conditions are reported to the user as shown in table 2.

Table 2: Errors and desired messages when converting a text to a number

Nr.	Error condition	Message reported
1	The value is not within thew acceptable range	SIZE must be between *minimum* and *maximum*
2	Text does not denote an integer value	SIZE must be a number [without "*character*" at position *index*]
3	Text does not contain any characters	SIZE must be a number
4	NULL or Void	(denotes a bug)

The exact values for *minimum* and *maximum* depend on the language and the implementation.

If the text contains non-digit character (case 2), the column of the first occurrence should be provided in the internal error report. This value is not used in the above error reports, but could be useful if for instance the value is obtained from a huge text in an editor, thus the cursor position can be changed to the in column of the offending character. Case 4 is not considered here as it generally denotes a bug. A user would not be able to enter NULL rather than an empty string as in case 3.

5.5.1.1 Special Results in C

The C standard library provides at least three functions that can be used to convert a string to a (long) integer number:

```
long atol(const char *text);
int sscanf(char *text, const char *format, ...);
long strtol(const char *text, char **parse_position, int base);
```

The one that gives most control to the programmer is strtol(). Additionally to the text to be converted, it takes a base parameter that allows to apply it to non-decimal numbers. For our purpose, base always is 10. The meaning of parse_position is of no relevance for result based error handling and is set to NULL. More details about this parameter are discussed with the next mechanism.

The use of atol() is discouraged and is only part of the standard for compatibility. It is merely a simplification of strtol() with an undefined result if the text can not be converted. The function sscanf() analyzes a input text according to a format specification and stores the extracted values in variables. However, it does not provide much error information except the number of successfully parsed characters. In particular, range errors in numeric values cause undefined behavior. Both functions are completely unsuitable here.

Thus, the conversion can be implemented as:

```
long convert_size(const char *text)
{
  return strtol(text, ((char **) NULL), 10);
}
```

On range errors strtol() returns the constant value LONG_MAX and LONG_MIN (depending on the sign). For all other errors the result is 0. Clearly, these results might also indicate a successful conversion. Consequently, it is impossible to detect errors at all.

5.5.1.2 Status Indicators in C

Alternatively, strtol() can report values out of range by setting errno to ERANGE. With non-numeric values, the case becomes a bit more tricky: First, some automatic error correction is involved because strtol() skips leading white space. Then all digits are considered to be part of the number, until a non-digit is detected or the string ends. The examples in table 3 illustrate this:

Table 3: Results and errors detected by strtol()

Input text	Error	Result	errno
"17"	-	17	-
LONG_MAX	-	LONG_MAX	-
"17.23"	Not an integer number	17	-
"17XY"	Not a number	17	-
"XY"	Not a number	0	-
""	Not a number	0	-
"12345678909876543 21"	Out of range	LONG_MAX	ERANGE
"-12345678909876543 21"	Out of range	LONG_MIN	ERANGE

This results into two problems: first, an empty string returns the value 0. Second, non-numeric characters after a couple of digits are not considered an error, like in "17XY". This might be practical if strtol() is used together with other string functions to parse a complex string containing several words. But here, it is inappropriate.

These two error cases can be distinguished by passing a pointer to a string as parse_position parameter to strtol() that afterwards will point to the character *after* the last successfully passed character in the input string. That way, a null byte at this position would indicate the end of string and a fully legal text. For reasons of symmetry, we also have to skip possible trailing white space before checking for the null byte.

```
long convert_size(const char *text)
{
  long size;
  char *parse_position;

  size = strtol(text, &parse_position, 10);
  if (errno == 0) {
    while (isspace(parse_position[0])) {
      parse_position += 1;
```

```
    }
    if (parse_position[0] != '\0') {
      errno = EDOM;
    }
  }

  return size;
}
```

For reporting errors, errno has to be compared with ERANGE and EDOM:

```
void report_convert_size(const char *text)
{
  long size;

  errno = 0;
  size = convert_size(text);

  if (errno == ERANGE) {
    printf("SIZE must be between %ld and %ld\n", LONG_MIN, LONG_MAX);
  } else if (errno == EDOM) {
    printf("SIZE must be an integer number");
  } else {
    printf("size = %ld\n", size);
  }
}
```

The index of a possible non-numeric character is not available directly. But as parse_position points to it, the expression (parse_position - text) yields it. But there is no way to transfer this value to the caller, as errno is just a plain integer value that cannot take any parameters.

5.5.1.3 Traps in Basic

The val function takes a single string argument as parameter and converts it to number. As there is no integer type in the E500 Basic, floating point numbers are also accepted.

```
*convertsize
  size=val (size$)
  return
```

When testing val with erroneous inputs, it turned out that it does not detect any kind of error except overflow. Table 4 summarizes this behavior.

Table 4: Results and errors detected by val

Input text	Error	Result	ern
"17"	-	17	-
"17.23"	-	17.23	-
"17XY"	Not a number	17	-
"XY"	Not a number	0	-
""	No value	0	-
"17e234"	Out of range	-	20

Thus, it is pretty superfluous to consider further error handling.

5.5.1.4 Exceptions in Java

The Java Core API provides several routines to convert a string to a number. One of them is Integer.parseInt:

```
public static int parseInt(String text) throws NumberFormatException
```

It does not have any automatic error correction (like C's strtol()) and only accepts numeric digits in the input, optionally preceeded by a minus sign (-). The allowed range for the type int is available in Integer.MIN_VALUE and Integer.MAX_VALUE. Errors are indicated by a NumberFormatException, which the below implementation delegates to the caller:

```
public static int convertSize(String sizeText)
{
   return Integer.parseInt(sizeText);
}
```

Reporting possible errors turns out to be difficult. First, the NumberFormatException does not allow to distinguish the different error cases discussed above. Next, getMessage() only seems to contain the string passed as parameter to parseInt(). Thus all one can do is:

```
public static void reportConvertSize(String sizeText)
{
  try {
    int size = convertSize(sizeText);
    System.out.println("size = " + size + "\n");
  }
  catch (NumberFormatException exception)
  {
    System.err.println("SIZE must be a numeric value between "
```

```
                + Integer.MIN_VALUE + " and "
                + Integer.MAX_VALUE + "\n");
    }
  }
```

Naturally, this is cumbersome for the programmer, and the resulting error messages are not precise enough for the user.

5.5.2 Copy a File

A routine takes two parameters: the name of the source file to copy, and the name of the target file under which the copy should be stored. This shows how to handle I/O errors. It also is a good example of an initialize/process/cleanup sequence. A good error handling mechanism should not split up these distinct three steps to keep the source code better readable. Another important point is how the language deals with closing the files in case of error.

In pseudo-code, this routine can be implemented as follows:

```
open source-file for input;
open target-file for output;
while not end of source-file do
  read byte from source-file
  write byte to target-file
close target-file
close source-file
```

All systems examined allow such an implementation, with minor variations in the loop syntax. Error conditions should be reported to the user as shown in table 5:

Table 5: Errors and desired messages when copying a file

Nr.	Error condition	Message reported
1	The source file cannot be opened.	Cannot read source file *filename: system message*
2	A read error occurs	
3	The source file cannot be closed	
4	The target file cannot be opened	Cannot write target file *filename: system message*
5	A write error occurs	
6	The target file cannot be closed	

There are several reasons why the source file could not be opened: the file simply does not exist, the name given by the user contained special characters not allowed (several systems reject "sepp:hugo$:!resi\/*hinz~?rödl" as filename), the file is exclusively locked by another process, the filename is too long, and several others. Most of them also apply for the target file, with additions like attempting to write to a write-protected disk.

Read and write errors are relatively rare and often the cause of physical damage to the storage media. Although such errors have a detailed physical position on the disk assigned, it does not make much sense to report these details to the user. The only reasonable thing to do is to use a disk-repair tool and attempt to save as many data as possible.

Although it might seem unlikely, such errors can even happen between the last successful write access and the finishing "close file". Most systems use buffered I/O, thus a "write" does not necessarily physically store data. Instead, the whole writing process may be deferred to the final "close file", where the buffer is flushed.

Ignoring errors while closing an output file would cause data loss without informing the user about that. This is far worse than "only" causing data loss but the user being aware of it. Although there is nothing the program can do about it, the user might take appropriate actions like attempting to copy again to a different target.

5.5.2.1 Special Results in C

To copy a file, the following functions of the standard library are involved:

```
FILE *fopen(const char *filename, const char *mode);
int fclose(FILE *file);
int fgetc(FILE *file);
int fputc(int character, FILE *file);
```

The function fopen() opens a file with a given filename for a certain mode. Among other values, "rb" indicates read access and "wb" write access to files containing binary data. The result is a pointer to an internal control structure that is not further documented and can only be used as parameter to all the other I/O functions. A NULL indicates that the file could not be opened for some reason.

To close a file opened with fopen(), the fclose() function has to be called. This releases all resources allocated for the internal FILE structure. In case of success, fclose() returns 0, otherwise the value of the constant EOF, which is traditionally -1. ANSI C allows EOF to be any negative value.

To read a single character from an input file, fgetc() is used. The file parameter is the result of the associated call to fopen(). The result is either the character code read in range from 0 to 255 or EOF in case the file ends or an error occurred. Similarly, fputc(character,file) writes character to file The result is EOF in case any errors occurred. In practice, the calls to fgetc() and fputc() would be replaced by fread() and fwrite() allowing faster blocked I/O, but the principle remains the same.

The following results can be used to handle all I/O errors:

- If the source file cannot be opened, fopen(source_name) returns NULL.
- If the target file cannot be opened, fopen(target_name) returns NULL.
- There is no way to detect a read error by examining the result of fgetc() because EOF can also indicate a normal end of file.
- If a write error occurs, fputc() returns EOF.
- If the target file can not be closed, fclose(target_file) returns EOF.
- If the source file can not be closed, fclose(source_file) returns EOF.

The following possible results of copy_file() reflect this:

```
#define CFE_NONE         0
#define CFE_READ_SOURCE   1
#define CFE_WRITE_TARGET  2
```

Using preprocessor defines is the way commonly taken for the like things because C does not really support constants in a way other languages do. The prefix CFE_ ("copy file error") is necessary as there is only one global name space for such constants. Typing names of constants uppercase is a silly C-convention. This does not make much sense because uppercase-only text is harder to read and has a highlighting effect not really appropriate. Enumerator types are rarely used in C due to several problems discussed by Wirth (1988) and Meyer (1997), who omitted enumerators from Oberon and Eiffel.

The behavior of fclose(NULL) is undefined. Therefore a fclose(source_file) has to be protected by an if (source_file != NULL). The same goes for fclose(target_file), which has to be placed even one nesting level deeper in the if-cascade, making it impossible to separate the cleanup-code.

```
int copy_file(const char *source_name, const char *target_name)
{
  FILE *source_file;
  FILE *target_file;
  int success;
```

```
  source_file = fopen(source_name, "rb");
  if (source_file != NULL) {
    target_file = fopen(target_name, "wb");
    if (target_file != NULL) {
      int byte = fgetc(source_file);

      success = CFE_NONE;
      while ((byte != EOF) && (success == CFE_NONE))
      {
        if (fputc(byte, target_file) != EOF) {
          byte = fgetc(source_file);
        }
        else {
          success = CFE_WRITE_TARGET;
        }
      }

      if (target_file != NULL) {
        if (fclose(target_file) == EOF) {
          if (success == CFE_NONE) {
            success = CFE_WRITE_TARGET;
          }
        }
      }
    }
    else {
      success = CFE_WRITE_TARGET;
    }
    if (source_file != NULL) {
      if (fclose(source_file) == EOF) {
        if (success == CFE_NONE) {
          success = CFE_READ_SOURCE;
        }
      }
    }
  }
  else {
    success = CFE_READ_SOURCE;
  }

  return success;
}
```

The following function checks the result and attempts to translate it to an error message.

```
void report_copy_file(const char *source_name, const char *target_name)
{
  int copy_file_status = copy_file(source_name, target_name);

  if (copy_file_status == CFE_NONE) {
```

```
    printf("Copied \"%s\" to \"%s\"\n", source_name, target_name);
  } else {
    switch (copy_file_status) {
      case CFE_READ_SOURCE:
        printf("Cannot read source file \"%s\"", source_name);
        break;
      case CFE_WRITE_TARGET:
        printf("Cannot write target file \"%s\"", target_name);
        break;
    }
  }
}
```

Apparently, copy_file() completely obfuscates the control flow and needs several precautions to avoid bugs like calling fclose(NULL). And the error messages producable in report_copy_file() are completely insufficient.

5.5.2.2 Status Indicators in C

All of fopen(), fgetc() and fpuc() set errno in case of trouble. This removes the ambiguity when fgetc() returns EOF: reading errno clarifies if the end of file is reached or an read error occurred.

On the other hand, fclose() does not set errno. This makes it impossible to detect data-loss in case of write errors during flushing the file buffers. Of course, this is not a problem of status indicator based error handling rather than of specification and implementation of the C standard library. Again, the second fopen() and the calls to fclose() have to be protected by if statements to avoid accessing NULL pointers in case of errors.

```
void copy_file(const char *source_name, const char *target_name)
{
  FILE *source_file;
  FILE *target_file;

  errno = 0;

  /* Open files */
  source_file = fopen(source_name, "rb");
  if (errno == 0) {
    target_file = fopen(target_name, "wb");
    if (errno == 0) {
      /* Copy data */
      int current_character = fgetc(source_file);

      while ((current_character != EOF)  && (errno == 0))
      {
        fputc(current_character, target_file);
```

```
        if (errno == 0) {
          current_character = fgetc(source_file);
        }
      }
      fclose(target_file);
    }
    fclose(source_file);
  }
}
```

When reporting errors, the strerror(errno) gives details on system level. However, because errno is just an integer variable, it is impossible to find out if the problem involves the source file or the target file. The best one can do is:

```
void report_copy_file(const char *source_name, const char *target_name)
{
  errno = 0;
  copy_file(source_name, target_name);
  if (errno == 0) {
    printf("Copied \"%s\" to \"%s\"\n", source_name, target_name);
  } else {
    printf("%s\n", strerror(errno));
  }
}
```

Due to the nature of strerror(), which only accesses a predefined set of string constants, it is not possible to include parameters like a filename in these strings. Thus the error message is again insufficient.

Experienced C programmers might now observe that it is possible to come up with a much better implementation combining special results and errno, while incorporating the printing of the error message into copy_file(). However, this violates the separation of user interface and functionality, and also does not use one mechanism consistently.

5.5.2.3 Traps in Basic

In Basic, files are access using numbers for file handles. All routines jump to the error handler specified with on error goto if something goes wrong.

```
*copyfile
  open source$ for input as #1
  open target$ for output as #2
*copyline
  if eof (1) then goto *endcopy
  input #1,lin$
  print #2,lin$
  goto *copyline
```

```
*endcopyfile
  close 2
  close 1
  return
```

To report errors, the numeric code ern has to be converted into a message manually, reassembling the original messages:

```
*copyerror
  message$=""
  if ern =76 then message$="file "+target$+" is write protected"
  if ern =75 then message$="bad drive name"
  if ern =77 then message$="disk full"
  if message$="" then message$="error "+str$ (ern )+" in line "+str$ (erl )
  print message$
  end
```

Clearly, the wording does not fulfill the requirements. But worse, *copyerror cannot be a subroutine. If an error occurs, the gosub stack is unwound, and the main context is restored. This means that turning ern into a descriptive message can only happen at this level. Clearly, this makes sensible error reporting completely impossible; even small programs usually have more than one level of routine nesting. Additionally, there is no sensible way to close possible open files.

5.5.2.4 Exceptions in Java

To read and write binary files, the Java Core API provides the classes FileInputStream and FileOuputStream. Both have constructors taking one parameter for the filename. The methods read() and write() work similar to fgetc() and fputc() in C. Both classes provide a close() method comparable to fclose() in C.

No catch clause is required because the routine is not going to do something about possible errors; they should only be reported to the caller. Uncaught exceptions are propagated along the call stack automatically, so the programmer doesn't have to provide explicit code for that.

However, the two files opened should be closed to avoid resource leaks. This can be achieved by putting the calls to close() inside a finally block. But this is only half the solution because we don't know if they could have been opened at all. Attempting to do a close on a null-file or a completely uninitialized one will cause a RuntimeException. To avoid this, the two files are first set to null, and the code inside the finally checks this before attempting to close the files.

```
public static void copyFile(String sourceName, String targetName)
      throws IOException
{
  FileInputStream sourceFile = null;
  FileOutputStream targetFile = null;

  try {
    // Open Files
    sourceFile = new FileInputStream(sourceName);
    targetFile = new FileOutputStream(targetName);

    // Copy data
    int currentByte = sourceFile.read();
    while (currentByte != -1) {
      targetFile.write(currentByte);
      currentByte = sourceFile.read();
    }
  }
  // Cleanup
  finally {
    if (targetFile != null) {
      targetFile.close();
    }
    if (sourceFile != null) {
      sourceFile.close();
    }
  }
}
```

A method calling copyFile and reporting errors or success basically only has to catch possible IOExceptions:

```
public static void reportCopyFile(String sourceName, String targetName)
{
  try {
    copyFile(sourceName, targetName);
    System.out.println("Copied \"" + sourceName + "\" to \"" + targetName);
  } catch (IOException exception) {
    System.err.println(exception.getMessage());
  }
}
```

But as it turns out, most values returned by getMessage() are near to useless. For example, FileNotFoundException only returns the filename that could not be found, but does not give any hints for the cause (like C's strerror(errno) does). The API documentation does not describe how getMessage() could be used otherwise to integrate its result into hand-crafted messages. This also questions the usefulness of getLocalizedMessage(): there is no point for it, as getMessage() does not contain any language dependant text anyway. And even if,

the documentation does not really describe how it should be used together with all the other routines of the internationalization package. Presumably, the programmer is supposed to pass the original message to an automatic translation service such as http://www.babelfish.com/.

The cleanup issue has not improved much compared to C: again, the calls to close() have to be protected by conditionals. But then, exceptions thrown during finally are ignored and terminate the block. Consequently, if targetFile.close() fails, sourceFile.close() will never be called.

In this example, this causes a resource leak. Even more disturbing is the case when FileOutputStream.close() would be called from the garbage collector (which cannot happen here): the exception is ignored. One might think that this does not matter because all data have already been written. But reconsidering the previous observations on buffered I/O streams, this is a potential case of unreported data loss - caused by the language, not the application programmer.

5.5.3 Handle a Bug

A sample condition for acceptability is validated: a variable sepp must not have the value 10. The name and the value are of course arbitrary. If the condition is not fulfilled, the program can consider itself to be buggy.

5.5.3.1 C

The standard header file <assert.h> contains the declaration of the macro or function void assert(int condition). If condition is 0, it displays a diagnostic message and immediately terminates the program without any cleanup actions using abort(). The example bug can be detected:

```
assert(sepp != 10);
```

The diagnostic message depends on the implementation but usually includes the name of the source file and the line number where the assertion failed. For instance, the widely used GNU CC compiler gives (Stallman et al., 1998):

```
Assertion (sepp != 10) failed in file internal-error.c at line 14
Abnormal program termination
```

If the preprocessor symbol NDEBUG is defined, assert() does not produce any code. This can be used to deactivate assertion monitoring. Obscure un/redefining NDEBUG and including <assert.h> multiple times can be used to activate only certain assertions with others being inactive. This however is not considered good coding style (Harbison & Steele, 1991).

Strangely, the C standard library is not required to use this routine. Generally, bugs that could easily be detected with assert() are specified as "implementation dependant behavior". For instance, a buggy routine call attempting to copy a string to a NULL pointer like

```
strcpy(NULL, source_string)
```

can have various effects, such as not copying anything (and maybe losing data if the source_string is discarded later on), no visible effect but trash the memory addresses after 0 causing a crash much later, making the operating system terminate the program, etc.

5.5.3.2 Skipping Basic and Java

The E500 Basic doesn't have any measures to deal with bugs. VisualBasic has an assertion mechanism comparable to Eiffel, but lacks most of its considerations.

In Java, the situation is similar. Various approaches have been suggested to work around the lack of assertions, usually based on throwing a programmer declared exception derived from RuntimeException, e.g.

```
if (sepp != 10) {
  throw PreconditionException("sepp != 10");
}
```

Several solutions are trying to reduce the typing effort, with the simplest one being to wrap the above conditional into a single routine which also contains a print statement. Essentially, this reassembles C's assert(). More sophisticated suggestions introduce preprocessors, inline scripting languages, collaboration patterns etc, which all apparently try to ape Eiffel assertions, for instance (Payne et al., 1998). Most of them are incomplete and cumbersome to use for the programmer. Because of that, I don't see any need to further discuss anything but original.

5.5.3.3 Eiffel

The simple example bug can be detected using

```
check sepp /= 10 end
```

However, the power and clearness of the notation used in Eiffel is probably best illustrated with some more examples:

```
class STRING
  ...
  prepend(other: STRING) is
    -- Prepend other to Current
```

```
    require
      other /= Void
    do
      ...
    ensure
      consistent_count: count = other.count + old count
    end
end -- class STRING
```

The routine prepend inserts a string other at the beginning of the invoking string object. The lines

```
text := "world"
text.prepend("Hello ")
```

would result in text containing "Hello world". *Require* asserts the precondition that there actually is a text stored in other, *ensure* asserts the postcondition that the new length (or character count) of the string has grown by the length of other.

The postcondition also has an optional label consistent_count, which is used in error messages by the compiler. This is supposed to be more descriptive for the programmer than the plain source code.

A simple class invariant for a generic class ARRAY could be:

```
class ARRAY[G]
  ...
  lower, upper: INTEGER
      -- minimum and maximum legal indeces
  count: INTEGER
      -- current number of elements
  ...
invariant
  consistent_count: count = upper - lower + 1
  non_negative_count: count >= 0
end -- class ARRAY
```

In Eiffel, array boundary checks are not a built-in language feature rather than an application of assertions.

5.6 Implementation

5.6.1 Special Results in C

The implementation is straight forward: if an error is detected, the result is set to a value documented as erroneous. All the work has to be done by the programmer, so no special compiler features have to be implemented.

5.6.2 Status Indicators in C

Almost the same goes for status indicators: a variable or function is declared to hold the error status. One value represents the state "no error", all other values represent some error. These values have to be well-defined at one central place. In C, manually by the programmer. (Some languages have mechanisms to automatize this, e.g. Ada and Eiffel).

The decision in C, to take one global variable of the type integer imposes several problems already discussed. However, there is no conceptional limit that prevents a status indicator to be assigned to a more flexible datatype and be used "locally", e.g. to make them part of a data structure that is needed anyway for transactions of the same kind. Most Eiffel classes of the standard library do this. But even the C library has an example for that: ferror(file) returns the last error associated with a FILE structure.

An error reporting function simply converts the status to a string. In case of a simply integer, this is trivial and can be performed by an array access. If the error datatype is more complex, this string could also contain parameters. However, the converter has to know all possible values beforehand.

5.6.3 Traps in Basic

There are no details available on how the E500 Basic implements error trapping. However, one obvious way would be to remember the location of the handler set with on error goto. If then an error occurs, the interpreter changes the control flow to this location. It demands only a single pointer and no stack for handlers.

The on error goto 0 idiom clears this pointer. If no handler is active, the interpreter uses the default handler described before, which halts the program. This is a very efficient way with hardly any influence on performance and memory usage.

The resume is also straight forward: the interpreter only has to remember a pointer to instruction that caused the error and its successor.

The ern variable to identify the error is set by the standard routines. The erl variable to remember the location in the source code can be kept up to date by the interpreter.

The simplicity for the implementor however is expensive for the programmer: return does not work inside an error handler, making it impossible to terminate a subroutine - only the whole program can quit. Resume has several inconsistencies. Error handlers cannot be nested. The whole mechanisms cannot be used for own errors.

5.6.4 Exceptions in Java

There are several ways to implement exceptions:

5.6.4.1 Range Tables

The compiler generates a table in which the start and end address of every try block as well as the address of the corresponding handlers are recorded. If an exception occurs, the program counter is looked up in the table. If the appropriate range is found, the stack is unwound and the corresponding handler is called. Otherwise, the program counter of the caller is tried, allowing to propagate unhandled exceptions.

The advantage is that no performance overhead is imposed if no exception occurs. Disadvantages are that the range tables have to be set up by the compiler and the loader, which is not possible in many existing environments. Furthermore the range table requires additional storage, which can be a problem in very small environments like many embedded systems provide.

This approach is used by the Java virtual machine. Venners (1997) outlines the happenings, Lindholm & Yellin (1996) describe them in more details.

5.6.4.2 Using Setjmp() and Longjmp()

This implementation is based on two functions provided by the ANSI C standard library:

```
int setjmp(jmp_buf environment);
void longjmp(jmp_buf environment, int status);
```

The function setjmp() stores the current machine environment, in particular the stack pointer and the program counter, in a buffer of the implementation dependant type jmp_buf and then returns 0. The function longjmp() restores the environment saved before. Thus the execution continues in the setjmp routine where this state was saved. After a longjmp(), however, setjmp() returns 1. This makes the following implementation of an exception handler possible:

Java source	C equivalent
try {	if (setjmp(environment) == 0) {
	Push(environment);
...*block*... ;	...*block*... ;
	Pop(environment);
} catch(...) {	} else {
	// Execution continues here after longjmp()
...*handler*...;	...*handler*...
}	}

If an exception is raised in *block*, the following code is executed:

```
Pop(environment);
longjmp(environment);
```

If the handler can not deal with the exception, it re-raises it so that a longjmp() to the previous setjmp() is executed, and so on (Koenig & Stroustrup, 1990).

Advantages are that the implementation can be mapped to standard C functions. Thus it is easy to do on most existing systems (though Harbison & Steele (1991) point out that longjmp() and setjmp() are quite difficult to implement and don't work correctly on all compilers if they are invoked in a signal or interrupt handler). Disadvantage are that a resume is difficult to implement and that setjmp(), Push() and Pop() impose a considerable runtime overhead even if no exception is thrown.

This approach is used by C++, most Eiffel and some Java compilers.

5.6.4.3 Metaprogramming

Metaprogramming means the ability to treat programs as data, for example to get information about the names and datatypes of their variables, types and routines. If a program can also acquire information about itself, this is called *reflection*. These mechanisms were pioneered by Lisp and Smalltalk and are now available in many modern languages such as Oberon and Java.

When an exception occurs, the system searches the call stack for a routine that can be used as handler. A handler is recognized by its signature: a routine which can have one parameter - the exception it knows how to deal with. Changing the control flow like resume and retry can then be done by modifying registers such as frame pointer and program counter.

Advantages are that no runtime overhead is imposed if no exceptions occur, and no special requirements are needed from the compiler (apart from metaprogramming facilities). It is

even applicable to languages that do not support exceptions beforehand.

Disadvantages are that actual handling of exceptions is several times slower than with the other implementations because a lot more code and data have to be executed and examined. For most practical cases however it should be "fast enough". But adding exception handling to languages that did not support it from the beginning rarely results in a consistent error handling from the programmers's point of view.

This approach has been used in an experimental implementation in Oberon (Hof et al., 1996).

5.6.4.4 Hidden Status Indicator

The status indicator concept described above can easily be automatized by the compiler. The programmer is relieved from manually writing the conditional after every function call that decides the new control flow. Still, all the advantages of the old scheme are retained.

The major disadvantage is a significant increase of the object code size. Krall & Probst (1998) used this for a Java "just in time" compiler. But the object code bloat jeopardized the "just in time" feature. Consequently, the next version used a different scheme based on range tables.

5.6.5 Assertions in C

In C, assert() is almost always implemented as macro. Depending on the value of the preprocessor define NDEBUG, it either does nothing or terminates the program with a diagnostic message containing the module name and line number. A possible declaration is:

```
#ifdef NDEBUG
  /* disable assertion checking */
  #define assert(condition) ((void) 0)
#else
  /* enable assertion checking */
  #define assert(condition)  \
    ((void) ((condition) ? 0 : __assert (#condition, __FILE__, __LINE__)))

  #define __assert(condition, file, line)  \
    (printf ("%s:%u: failed assertion ‘%s’\n", file, line, condition), \
     abort (), 0)
#endif
```

The position in the source code is obtained by two special preprocessor symbols, __FILE__ and __LINE__. As they are resolved during compilation, no performance overhead is induced during runtime. However, only one level of the call stack is accessible, and no information like values of variables are available.

5.6.6 Assertions in Eiffel

In Eiffel, more effort is needed as a complete stack trace is maintained. Furthermore, the compiler has to add code to validate the class invariant upon every routine entry and exit. But such issues should be straight forward.

However, assertion monitoring can result into a considerable overhead. Although it depends on code provided by the programmer to actually validate an assertion, Meyer (1997, 396) gives the following empirical observations: monitoring only preconditions imposes about 50% performance overhead. About 75% of this overhead are due to maintaining information about the call stack. Enabling all assertions commonly gives a 100% to 200% penalty.

Most Eiffel compilers create C code as output, thus utilizing the setjmp()/longjmp() scheme described with exceptions (with all its disadvantages). The additional retry is then straight forward to implement: it basically needs a label at the beginning of a routine and a goto to re-execute it. The various issues pointed out in respect of inheritance can also be implemented as simple "and-ing" and "or-ing" assertion expressions. The old keyword to remember the value of a variable upon routine entry can be mapped to an automatically created local variable.

6 Discussion

This chapter gives a discussion of questions raised during the analysis. For several issues, it is not easy to decide whether they are a "good thing" or not. Solutions to some of the most apparent problems are outlined.

6.1 Automatic Error Correction

Automatic error correction tries to guess an action that matches the intention of the user. This is not without trouble, and in literature there are many different opinions, some of which are discussed here.

Teitelman, the designer of "DWIM: Do What I Mean" for the InterLisp system explains the basic idea: "If you have made an error, you are going to have to correct it anyway. I might as well have DWIM try to correct it. In the best case, it gets right. In the worst case, it gets wrong and you have to undo it: but you would have had to make a correction anyway" (quoted in Reason, 1990, 164). However, this is not always easy to implement:

> "In one notorious incident, Warren [Teitelman] added a DWIM feature to the command interpreter used at Xerox PARC. One day another hacker there typed "delete *$" to free up some disk space. (The editor there named backup files by appending '$' to the original file name, so he was trying to delete any backup files left over from old editing sessions.) It happened that there weren't any editor backup files, so DWIM helpfully reported "*$ not found, assuming you meant 'delete *'." It then started to delete all the files on the disk!" (Raymond et al., 1999)

The problem here is the lack of undo for the "delete *" command. Although there are ways, existing implementations like the trashcan metaphor on many desktop interfaces severely hamper the user's understanding (Gentner & Nielsen, 1996).

Molich & Nielsen (1990) discuss ways to correct errors novice users might make with a hypothetic telephone index system. Basically, the user can enter a phone number and the system displays the subscriber. The authors suggest to perform the following corrections on user input:

1. Extra spaces and parenthesis around the area code are removed.
2. The letters "o" and "O" (lower and upper-case O) are replaced by "0" (zero).
3. The letter "l" (lower-case L) is replaced by "1" (one).

Here, the authors make it clear that the system is to be used by some people who may be totally new to computers, and are not going to use it regularly. The context is extremely restricted (phone number), and everything but a digit does not make sense. The few corrections are simple and well-defined replace/remove operations. For example, entering "A" would still cause an error message, and the paper devotes considerable space into the message design. Still it can be observed that the trouble of similar shapes in "O" and "0" resulted from mixing Roman letters with Arabian numbers several hundred years ago.

Rasmussen discusses "the fallacy of defence in depth", referring to the danger of a system's own defense mechanisms. Applied to the context of programming, this means the possibility to correct errors automatically. "Humans can operate with an extremely high level of reliability in dynamic environments when slips and mistakes have immediately visible effects and can be corrected." If this is not the case, latent effects of errors can be left in the system. He continues: "Analyses of major accidents typically show that the basic safety of the system has eroded due to latent errors. A more significant contribution to safety can be expected from efforts to decrease the duration of latent errors than from measures to decrease their basic frequency." (quoted in Reason, 1990, 179)

A popular application of automatic error correction can be found in most web browsers when parsing pages written in the Hypertext Markup Language (Raggett et al., 1997). Today it is an accepted fact that most pages in the Web are broken and do not use correct HTML. Many people think this doesn't really matter because the browser will fix it anyway. However, it is difficult to write other programs that read HTML input (like search engines or import filters) as they have to parse the document in non-deterministic ways. To make things even worse, these ways differ between programs, so it can happen that valid documents are not processed correctly. This severely hampers exchange of information.

This seems unnecessary, as HTML is an application of SGML, which was developed far earlier and can be considered to be its "big brother". Consequently, SGML parsers can check HTML documents for errors. However, these tools were not used, simply because most of them do not produce any error messages non-technophiles would understand (Bowers, 1996). For example, the faulty HTML code excerpt

```
<b><i>Hello</b></i>
```

results in the following error messages when checked with the popular Sgmls parser via "piping" from standard input (Clark, 1994):

```
sgmls: SGML error at -, line 7 at ">":
        I end-tag implied by B end-tag; not minimizable
sgmls: SGML error at -, line 7 at "":
        I end-tag ignored: doesn't end any open element (current is BODY)
```

The problem could be fixed by swapping and </i>.

Thimbleby (1998c) describes his experiences with various automatic error corrections in a popular word processor and grunts: "(despite having a PhD in computing science) I have no idea what is going on."

This suggests that automatic error correction on the long run causes more trouble than it solves. Although users might find it neat in the first place, it only defers problems. But later, there is no deterministic solution anymore. Additionally, it severely hampers the understanding of the user. Considering the above experiences, automatic error correction is commonly used to hide that the system is defective as a whole or unable to provide a helpful error message due to a defective implementation of error handling.

Nevertheless, there seem to be criteria where it might be useful:

- There is a good chance that the input came from a human user, like in window dialogues.
- The user is not expected to use the system regularly.
- The corrective actions only work within a very restricted context (usually one field or word)
- The corrective action is "simple", and the programmer can explain it in one short English sentence.

Automatic error correction in complex system demands more:

- There has to be some feedback that the system performed automatic correction.
- There is an undo, if the correction did not guess the user's intention correctly.

Shneiderman (1997) recommends a mixture of the response types "automatic correction" and "let's talk about it": the program opens a dialog with a descriptive error message, but also asks "Maybe you meant that?" and displays a corrected version of the input as part of the dialog. The user can then accept the corrected version, but still is fully aware of the erroneous state. If the hit-rate of the correction is high, this can help new users in learning how to use the program without having to look up the manual.

6.2 Garbage Collection

Many languages with exception handling also support garbage collection (GC). It is generally believed that GC is necessary to avoid resource leaks after exception. If implemented properly, GC does *not* cause memory leaks, slow the program down several times, make response times unacceptable, block all threads of a process or make the whole program perform horribly under low memory condition (though such things happen with many existing collectors). Joyner describes the reasoning behind garbage collection:

> "One of the hallmarks of high level languages is that programmers declare data without regard to how the data is allocated in memory. In block structured languages, local variables are automatically allocated on the stack, and automatically deallocated when the block exits. This relieves the programmer of the burden of allocating and deallocating memory. Garbage collection provides equivalent relief in languages with dynamic entity allocation." (Joyner, 1996)

Meyer also sees garbage collection as aid for programmers:

> "The spirit of object technology [...] suggests relying on compilers for tasks that are tedious and error-prone, if algorithms are available to handle them. On a large scale and in the long run, compilers always do a better job." (Meyer, 1997, 514)

Initially, GC was designed only for memory resources. In this context it works reasonably well, because every memory location is as good as any other. But quite recently, the concept has been extended to be applicable for all kinds of objects and resources. Java provides finalizers, Eiffel allows to inherit and redefine MEMORY.dispose.

The canonical example exposing GC to be dangerous is a simple output file: here, the finalizer executes a "close file" function. Due to performance considerations, files are usually buffered. Consequently, a final "close" can cause the buffer to be flushed, in other words: written. This final write operation might fail, and the user's data are lost. Though inevitable, the garbage collector does not have any means to report this, deluding the user into believing everything is all right.

Also, with GC one can never be sure when resources are actually released. In case of memory, this doesn't matter because the GC is called automatically when the system runs out of memory. This however does not work for other resources like files. In case one forgot to close a file, GC might not do so either for a long time, and another program might not be allowed to read the same file and therefore fail.

Another point is that GC is said to keep programs from crashing because it does not release resources that are still in use. This is also regarded a feature, and apparently targets a major problem of C++'s destructors, which are automatically called when the scope for an object ends - even if it is still referenced by others. However, GC does not fix the real bug (which is located in the program source code), but only hides and works around it. Instead of telling the programmer that he wrote a defective program, GC only applies automatic error correction and pretends that everything is all right.

Nevertheless, manual resource management is known to cause even more trouble. Reade describes programming as a "separation of concerns" consisting of basically three things: what has to be done, how it has to be done and the administration required for it. He says:

> "Ideally, the programmer should be able to concentrate on the first of the three tasks (describing what is to be computed) without being distracted by the other two, more administrative, tasks. Clearly, administration is important but by separating it from the main task we are likely to get more reliable results and we can ease the programming problem by automating much of the administration." (quoted in Joyner, 1996)

But if this automatization results into latent errors to be introduced into the program like with GC, this cannot be considered a solution.

Alternative implementations for general resource management have been described by e.g. (Bronnikov & Smirnov, 1998) or (Tasker, 1999). Generally, logically related resources are kept together in *holders* or a *cleanup stack*, where they can later be released with one routine call. As the releasing call is part of the application code, errors could be handled as usual (instead of being swallowed by an "invisible" concept as GC). Right now, none of the above implementations is without serious flaws. Nevertheless, they can be considered interesting starting points.

But all this does not mean GC is generally useless. First, it works for plain memory management issues. Second, it could act as a complementary debugging feature with the above holders. During the development and testing phase, it could report preliminary releases of resources to the programmer (with a stack trace etc). On the user's machine, it could create automatic bug reports.

6.3 Resumption

Resumption is one possible way for an error handler to continue the program after operations that hopefully corrected the defect. But there are some problems, both speaking in terms of design and implementation.

Liskov & Snyder (1979) point out that here the routine raising an error and its caller are mutually dependant: the caller invokes the raiser to perform some task, but the raiser depends on the error handler in the caller to fulfill it. This compromises the separation of levels of abstractions. Lee & Anderson (1990, 200) back this up in demanding that an error handler should not make any adjustments to the internal state of a component - in particular, when this state might be inconsistent, as it might well be the case during handling an error. They also observe that if the raiser expects a resume, the error handler might decide not to perform it. For this case, Goodenough introduced a special CLEANUP handler in the raiser, which in turn caused Liskov & Snyder to describe his proposal as overly complicated.

Stroustrup (1997, 370) describes how to implement resumption by means of function calls but discourages its use for similar reasons, also remarking that to his experience successful systems evolve in the opposite direction.

Another practical indicator of the trouble is the implementation problem of VisualBasic described before. Here, execution resumes in the procedure where the error handling code is found. If the error was propagated from another routine, this is not the routine where the error occurred. Then resume merely acts as retry, confusing the programmer and compromising consistency.

Speaking in terms of object-oriented programming, the issue should be clear: When the error occurs, the object is in an inconsistent state. If the error handler now starts to call "do something about my problem" handlers of other objects, it must avoid to address objects whose own consistency depends on the inconsistent caller. But in many cases, the whole program "object" is in an inconsistent state, so the handler can not call any objects of the current program.

The lesson seems to be: consider resume harmful. Instead, cleanup and retry.

6.4 Automatic Error Propagation

Many error handling mechanisms automatically propagate an unhandled error to the caller if no handler deals with it in the current routine. This is distinct from manually handling an error and deciding to "re-raise" it again. In particular, the mechanisms in Eiffel and Java support this. However, this is not without problems:

Goodenough (1975) observes two advantages in such a behavior: first, it makes it easier to add errors without having to modify all possible callers, and second, errors will ultimately be passed to the main environment, which might be a debugging environment or a default handler that reports the error to the user. But he then reconsiders that there are definite

advantages in examining how a new error affects callers, although without specifying which exactly. In turn, he excludes automatic propagation from his proposal.

Liskov & Snyder (1979) partially back up Goodenough's opinion, but go into more detail: a programmer should not need to examine implementations of routines he is about to call in order to understand what they do. Instead, this understanding should be obtained by reading the routine interfaces. Thus a routine signature should not only list parameters and results, but also all errors that might be raised while executing it. The same accounts for errors possibly raised in lower level subroutines. Different to Goodenough's proposal, the CLU language also supports special *failure exceptions*, which are not part of the signature and can be raised by every routine.

The problem arises when considering object consistency. Necessary cleanup actions in CLU require the programmer to handle even failure exceptions, cleanup and then raise the same error again. Java slightly reduces this work because finally automatically re-raises the same error. However, the language never validates consistency, and all sorts of evil things might happen if the cleanup did not restore consistency.

Eiffel on the other hand allows to specify consistency with the class invariant. It is even validated by the language in case developer exceptions are raised or (unless monitoring has been turned off) assertions fail.

As a result, automatic propagation can be considered acceptable only with a reasonable mechanism to ensure consistency before propagation.

6.5 Robustness via Compile-time Checking

A related issue to the above discussion is compile-time checking for error handlers. The compiler can ensure that the programmer provided a handler for a certain possible error reported from a sub-routine. Goodenough (1975) proposed a rigorous checking, where the compiler refuses to accept possibly uncaught exceptions. Liskov & Snyder (1979) felt this was unrealistic for "situations where no meaningful action can be taken". In turn, they distinguished between *exceptions* and *failures*, which do not require a handler.

Gosling et al. (1996) have adopted this for Java, which distinguishes between *checked* and *unchecked* exceptions. They claim that certain errors should be unchecked "because, in the judgement of the designers of Java, having to declare such exceptions would not aid significantly in establishing the correctness of Java programs". (Apparently, they mean robustness). In practice, this causes some confusion: Campione & Walrath (1996) speak of a "controversy" when trying to explain the different Java exception classes in a way that infant

programmers could understand it. They suggest to raise checked exceptions in the own code, and discourage rasing unchecked exceptions just because one does not want to be bothered by those annoying compiler errors. Venners (1998) gives only slightly more detailed guidelines what to do when.

The other extreme in this discussion is Eiffel, where a lot of concern is devoted to acceptability (to avoid the term correctness): assertions occur in numerous flavours, have well-thought out rules concerning inheritance, are easy to use, become part of the documentation, etc, etc. Opposed to developer exceptions: they are not part of the routine signature, there is no compile-time checking for handlers, the programmer has to figure out himself which errors can occur via trial-and-error, and the EXCEPTIONS class seems to change with every second new book from Meyer. Consequently, there are no data available to judge the robustness of an Eiffel program.

But the whole discussion seems to miss the point. From the programmer's point of view, there are two types of errors:

1. Errors the programmer cares about, and for which he wants to provide a proper handler.
2. Errors he doesn't care about.

An important observation is that the programmer does not necessarily care about every error that is part of the routine interface. CLU and Java do not really support this view of the world, and want to force the programmer to care about the same errors the library designers cared about. A programmer's common answer to this can be found below (coded in Java):

```
try {
   ...
} catch (Exception *exception) {
   // James Gosling, go home!
}
```

The wording of the comment varies. But generally, this represents an error handler that deals with *every* error but does *nothing* about it. This has two effects:

1. The compiler stops vomiting error messages.
2. The program obtains a family package of potential latent errors.

The first one is desired, the second one probably not. Popular consequences in practice have already been discussed, they are most likely "for some reason it does not work" or unreported data devastation. The problem here is not with the programmer, but with the designers of the language and the libraries. Often no error handler is better than an error handler like the

above: when the programmer assumes that an error is unlikely to occur in practice. If despite this attitude the error occurs anyway, the program will still detect it, but can regard it a bug (and deal with it as suggested before). Possibly the detection will happen slightly later, and on a lower level. But the stack trace should reveal the real culprit.

The discussion here is not if this is good practice or not. It also is not about the cliche that programmers are lazy, self-satisfied and overpaid. It is about the fact that programmers don't care about certain errors, and that it should be up to them which errors these are. Attempts to force them to care about errors according to arbitrary rules only result in latent errors - which are even worse.

Of course, the decision which errors to care about should not be up to the programmer alone. They have an inherent tendency to not care about any error. But the solution should be obvious: the errors to care about are part of the specification. In practice, this would mean that the compiler by default makes *all* possible errors part of the interface, and requires to deal with them. The programmer and the user then decide which are unimportant, and make them unchecked errors. In turn, the compiler stops complaining about them.

Provided proper tools, data about checked and unchecked errors can be viewed at one glance, and thus be used to judge of the robustness of the program. Essentially, this is similar to the classic Software Fault Tree Analysis (Leveson et al., 1991), but has a higher degree of automatization. This is nice for the user because he does not get a black box of possible false promises. The programmer on the other hand has something solid to advertise his program with. (Of course, it is still possible for the programmer to cheat with swallowing error handlers. But one major motivation for that, convenience when compiling, is gone.)

6.6 Dynamic Binding and Compile-time Checking

Another apparent problem with compile-time checking for error handlers is when it comes to dynamic binding. The trouble is that a virtual routine can have other errors in the signature than its ancestor.

For Liskov & Snyder (1979), this is a non-issue because CLU is a procedural language.

Stroustrup (1997, 377) goes into detail when reflecting on C++:

> "A virtual function may be overridden only by a function that has an *exception-specification* at least as restrictive as its own [...] *exception-specification*. [...] This rule is really only common sense. If a derived class threw an exception that the original function didn't advertise, a caller couldn't be expected to catch it."

In other words: the routine in the heir must not raise more exceptions than the ancestor. This guarantees that an error handler never has to face errors it can not deal with. Apparently, this is very restrictive, and it can be doubted that in practice it will always be possible to design a class to satisfy this constraint. After all, the client programmer doesn't "design" the errors of library routines.

But what's more alarming is the attitude exposed in the following excerpt:

> "Importantly, *exception-specification*s are not required to be checked exactly across compilation-unit boundaries. [...] The point is to ensure that adding an exception somewhere doesn't force a complete update of related exception specifications and a recompilation of all potentially affected code. A system can then function in a partially updated state relying on the dynamic (run-time) detection of unexpected exceptions. This is essential for the maintenance of large systems in which major updates are expensive and not all source code is accessible." (Stroustrup, 1997, 376-377)

In other words: a running system with latent errors is preferable to one that does not even compile due to active errors.

Gosling et al. (1996) basically state the same for Java, though without any remarkable considerations. Different to C++, Java rigorously checks *all* checked exceptions during compile-time. But it also introduces more problems, for example when loading classes during runtime: this can easily fail due to new exceptions in a throws clauses. In worst case, even before the actual program code gets executed. Then, the programmer is unable to provide a non-technophil error message, and the user is left to the mercy of Java's internal default handler (usually dumping the stack etc).

Eiffel, as already noted, cowardly dismisses the whole issue by making only assertions part of the routine signature. As assertions are validated during runtime, the assertion expression simply can be treated like a virtual function. But possible developer exceptions are subject to trial-and-error. Unfortunately, Meyer (1997) does not seem to justify this anywhere.

Apparently, there is something wrong here. But it is difficult to say what exactly.

6.7 Should Errors be Objects?

On the implementation level, Basic, C and Eiffel represent errors by numbers, possibly accompanied by a couple of variables and functions to further describe them. Java and many others use dedicated objects to represent errors. Why would it be sensible to use objects?

Apparent advantages of errors as objects derived from Throwable in Java are: it is simple to pass parameters to a handler, and hierarchical handlers for multiple errors are easy to implement.

Meyer (1997) opposes this by observing that the EXCEPTIONS class in Eiffel does not have any commands to change the state. It has of course several queries to check the state, such as exception and is_assertion. But they are set "in the background" by raise and retry. They are not really under the control of the program; they are rather caused by events beyond its reach. But it can be observed that Eiffel's developer exceptions didn't turn out to be particular useful: passing parameters and messages to them is not really possible. (A possibly interesting mixture can be found in Ada 95: errors are integer numbers, but have an optional "context", that can be an object of whatever type).

It is also not very difficult to think of possible commands that can actively change the state of an error. For instance, VisualBasic allows to assign a message and an optional reference to the online help.

Apart from that, all error handling mechanisms evaluated here are built on the misconceptions that one routine can cause only one error. As the example of copying a file showed, this is not true. Especially during cleanup actions, several errors can occur in the same statement block. None of the evaluated languages addresses this in a sensible way, either losing error information or showing undefined behaviour. So more commands would be needed to associate multiple errors with each other. Interestingly, Java already has a class SQLException, which allows to build a list of database errors using setNextException().

While it cannot be said that objects are the only or best way to express errors, Meyer's counter-arguments seem to be built on a rather weak foundation.

6.8 Exception = Error?

Many languages such as Ada, C++, Eiffel and Java can use a mechanism called *exceptions* to deal with errors. There are two questions arising:

1. Should exceptions be used for something else but errors?
2. Should errors be expressed by something else but exceptions?

There are several examples where exceptions are used to express non-error issues: Goodenough (1975) points out some applications that might have been useful in the 1970s lacking alternative approaches. From today's point of view, all of them can be implemented more elegantly using techniques such as inheritance, genericity, inter process communication

or at least callback routines. Stroustrup (1997) describes how to flush queues or find a node in a binary tree using exceptions. The point of this is not really clear to me because a simple while loop would suffice here. Java uses exceptions to indicate that a thread has been terminated with hilarious consequences for programmers who want to handle errors in multithreaded programs (DeRusso & Haggar, 1998). Another application in Java is *narrowing*, which is one way to find out whether a reference value is a legitimate value of the new type during runtime (instanceof is another). This is criticised by Meyer (1997), who compares it with Eiffel's *assignment attempts* doing without exceptions.

Sensible and working examples of using exceptions for something but errors have yet to be found.

But exceptions are just one way of expressing errors in a language that supports them. Special results and status indicators are others, that generally can be used by any language. So when should a programmer choose the others?

Venners (1998) discusses an interesting example in respect to Java, that apparently causes some confusion in literature: is an "end of file reached" an error? He says, for a "normal" binary file, it is not, as every such file will eventually end. Thus, FileInputStream.read() returns the special result -1 but does not throw an exception. DataInputStream.readInt() on the other hand throws an EOFException if it cannot find 4 bytes in the stream required to build a 32 bit integer value. This clearly represents an input error. (Reconsidering my analysis, it can be added that the special result of read() should be replaced by a boolean function FileInputStream.endReached() or the like.)

To summarize: if a language supports exceptions, it should not use them for anything but error handling, and errors should not be expressed by anything but exceptions. So called "exceptional cases" like an "end of file reached" should be dealt with by means of status indicators. I thus suggest to rename the technique called "exceptions" to "errors", so that this lengthy explanations can be omitted.

6.9 Disabling Assertions

Assertions in C and Eiffel can be disabled by the programmer. As this (partially) removes the capability of a program to detect bugs, there is an obvious question arising: is it appropriate?

The answer is a straight "yes" if the program otherwise would be too slow, and thus inapplicable for the user. A popular example is a program to compute tomorrow's weather predictions that needs 48 hours with assertion monitoring, instead of 12 hours without. Simply taking a faster computer is not an option here, as such applications already utilize the

fastest hardware on the market. Still, it makes sense to have monitoring on during development and testing, when data sets are smaller and time is not that crucial.

But often, performance penalties imposed by assertions are not that dramatic. Shneiderman (1997) describes appropriate response times for interactive tasks:

- 0.1 seconds for typing, cursor motions, mouse movement etc.
- 1 second for simple frequent tasks
- 2-4 seconds for common tasks
- 8-12 seconds for complex tasks

If the program is significantly slower, the user error rate increases, while productivity and satisfaction decreases. These experiences also seem to justify a "yes", if the program would otherwise exceed these limits with monitoring enabled.

However, the answer is difficult if it takes 12 hours with monitoring and 6 hours without to compute tomorrow's weather, or 0.07 seconds instead of 0.05 seconds to move the cursor.

Hoare compares disabling assertions in production code with "a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea" (Hoare, 1981). In the spirit of Rasmussen and Reason, it can also be observed that it causes otherwise active errors to turn into latent ones. Thus it is not acceptable.

Meyer basically agrees with that, but still objects that enabled assertions might have negative effects: "it can encourage among developers, even unconsciously, a happy-go-lucky attitude towards correctness, justified by the knowledge that if [a bug] remains, it will be caught by the user through an assertion violation, reported to the supplier, and fixed for the following release. So can't we just stop testing right now and start shipping?" (Meyer, 1997, 396) He suggests to consider delivering two versions of the program: a fast one without monitoring, used most of the time, and a slower one with assertions enabled the user can resort to if he becomes suspicious.

It seems futile to further discuss this on a rational-technical level. What is needed are data that would allow to estimate the impact of enabled assertions on the programmer's attitude.

6.10 Towards Automatic Error Reporting

The major problem with error handling by far seems to be the generation of error messages. None of the evaluated mechanisms has anything reasonable to offer. Operating systems concepts and current parser generators only make things worse. Understandably,

programmers rarely take the pain to turn this mess into a proper message. If they do, it requires an unreasonable amount of code.

The error messages discussed as a minimum requirement usually have the form

Cannot do something:
Something must be something else

[Do] something [else] represent all concepts, operations and items that can cause errors. To refer to them in an error message, they must have *names*. Finding these names and using them consistently is probably the biggest challenge. The importance of naming conventions and their consistent application seems to be rarely recognized among programmers (Laitinen, 1995).

This would also allow to address the internationalization problems: phrases like "cannot" and "must be" probably exist in every human language. All possible operations and concepts must have equivalents in other languages, otherwise it would be impossible to write a manual for the program. Consequently, the program should be capable of translating error messages during runtime.

If errors are represented by objects that carry the message already with them and allow optional attributes such as help and retryable, so the ultimate goal to display errors with one line should eventually be reached. Most of the effort is transferred to library programmers and translators, so that the application programmer can focus on the important tasks.

6.11 Out of Memory

One particular "funny" error is when a system runs out of memory. Despite the fact that this is a very common problem in the real world, there are hardly any scientific reflections about it.

Basically, this represents a component allocation error. Consequently it is within the responsibility of the user or administrator to fix or avoid it. This is not very difficult: the user can quit some other applications running in the background or provide less memory consuming input. Or the administrator can add more memory to the machine. So all that is needed is a little error message announcing the problem. But in practice, this is not always easy:

- Some operating system violently start to terminate programs if the system is low on memory - before the program can report anything to the user.
- In many GUI-driven environments, it is impossible to open a new dialogue announcing that the system ran out of memory because this would require memory.

- In some languages certain considerations are needed to raise an error without memory. For instance, in Java a `throw new OutOfMemoryError` can fail due to the lack of memory.

Consequently, it is popular among programs to just crash in such a case. Especially, if their error handling is based on special results or status indicators. Or does anyone seriously expect a programmer to check every memory allocation for success?

One general misconception is that this cannot happen on systems with virtual memory. This wrong interpretation is not only common among programmers, but also among people who should know better: for instance, Silberschatz & Galvin (1994, 301) suggest that virtual memory "frees programmers from concern over memory storage limitations". Stallings (1998, 258) describes a specific example, where insufficient memory leads to a deadlock, which in turn makes him observe that "The best way to deal with this particular problem is to, in effect, eliminate the possibility by using virtual memory" - as if it's not always trivial to modify the example so that it deadlocks again, no matter how much virtual (or physical) memory the system has.

7 Conclusion

This chapter shortly summarizes the research results and points out possible future directions.

7.1 Research Results

The questions asked during the description of the research method can now be answered - at least to some extent:

The notion of error has a different meaning pretty much everywhere. The one useful in this context is to see the error as a model to deal with defects. Defects are conditions that make it impossible for the program to produce proper output.

Defects are either in the design or the physical representation of the program. A program cannot detect defects. All it can do is reflect upon the data its own current state and point out inconsistencies. To do that, it has to provoke contradictions, which it does by comparing data and state with expectations. These expectations have to be provided by the programmer and are based on a few basic principles which were also outlined. Errors in the program's environment can be detected in the input, during allocating or accessing components. They can be fixed by the user or administrator. Errors in the program itself are commonly referred to as bugs. It can only detect them by rigorously validating the internal state against its specification. Different to all other errors, bugs can only be fixed by the programmer. Generally, it is important to attempt to detect errors as soon as possible to avoid latent errors to remain in the system.

A program can respond to errors in several ways. Basically, an error dialogue or "gagging" are appropriate for near to all cases. A need for warnings hints at a defective design. Automatic error correction by the program often introduces more problems than it solves. But there are cases where it is preferable to halt the program or reset the routine context. Though this causes data loss, it is preferably to continuing and devastating data. In practice, programs also respond without output nor error message ("for some reason it doesn't work"), or continue with latent errors. Clearly, those are bugs.

A simple scheme to design good error messages is to put them into the form "Something must be something else", which already states what must be done in order to fix the problem. Optionally, one or more "Cannot do something"s can describe the context in which it occurred. This is especially important if the error depends on implementation details not apparent to the user.

There are some general principles to avoid defects such as simplicity and conservatism. Others at least reduce the impact of them, like independence, redundancy and diversity. Unfortunately, they are rarely applied in a systematic way, in particular when looking at data formats and syntaxes of programming languages.

In this light, the current state of programming languages gives a rather sad picture. Error reporting apparently has never been considered a serious issue, and is in fact becoming worse comparing the quality of messages producable with C to Java or Eiffel. Though Basic and Java manage to delegate the control flow to error handlers quite nicely, they are not sensible about cleanup and errors during handling errors. Apart from that, none of the languages supports both hierarchical and sequential handling of multiple errors. Eiffel at least provides a simple, but effective framework to detect many bugs. But this doesn't seem to be recognized and used by many people. Many languages have subtile interferences with resource management, commonly resulting into resource leaks, undefined behavior, loss of information about already detected errors or inconsistent states.

A critical discussion gave a closer look to some of the most striking problems and concluded:

- Automatic error correction hardly ever works. Often it seems to be used to hide that the system is defective in its design, or unable to provide a helpful error message due to a defective implementation of error handling.
- Garbage collection and finalizers are a major source of latent errors, and need to be replaced by something more useful.
- Resumption does not work. It is more sensible to cleanup and retry.
- Automatic error propagation is only acceptable if object consistency is ensured.
- Current implementations of compile-time checking for the existence of error handlers only encourage the programmer to introduce latent errors by providing "swallowing" error handlers. Additionally, they don't seem to work well with dynamic binding.
- The automatic generation of error messages is challenging, but should be possible. An approach has been outlined.
- Programs running out of memory might have trouble to report this.

For most of these points, possible solutions were examined. However, some issues are still left to be resolved by others.

7.2 Future Directions

This thesis tried to describe the big picture of error handling, taking different views into account: the psychological background that leads to human error; the usability issues involved, in particular wording of error messages; programming languages and libraries which eventually decide what really can be done. During this, many useful existing ideas and concepts have been identified. But only few of them have been widely recognized.

What mainly needs to be done is incorporating them into programming language, libraries and operating systems. This might sound preposterous considering the current mood in computer science. Such dirty, low-level issues don't fit well to the current emphasis on design patterns, architectural styles and more and more abstractions. Alas, these things alone never result in an executable program rather than papers full of arrows and bubbles and a reduced understanding of the user and programmer what is really going on. It is often claimed that this understanding is only distracting from the real task, and thus not required to utilize the output of a black box. However, if the box cannot produce an output, this understanding is necessary in order to fix the problem.

Clearly, the main trouble is not on a technical level, but within the current attitude towards errors. We should stop to see them as annoyance, pretend that they do not exist or blame them to others. Instead, the error is a useful tool to gain a deeper understanding of matters, useful to learn from negative experiences and judging the overall quality of a concept.

8 References

Aho, A.V. et al. 1985. Compilers: Principles, Techniques, and Tools. Addison-Wesley.

ANSI & US Government Department of Defense. 1983. Military Standard: Ada Programming Language, ANSI/MIL-STD-1815A-1983.

Beizer, B. 1990. Software Testing Techniques, 2nd ed.. New York, NY: Van Nostrand Reinhold.

Bezault, E. et al. 1995. The Eiffel Library Kernel Standard, Vintage 95. Santa Barbara, CA: Nonprofit International Consortium for Eiffel (NICE).

Boutell, T. (ed.) 1996. PNG (Portable Network Graphics) Specification, Version 1.0. http://www.w3.org/TR/REC-png.html.

Bowers, N. 1996. Weblint: Quality Assurance for the World-Wide Web, Computer Networks and ISDN Systems. Vol. 28, no. 11, 1283-1290.

Bray, T. et al. (eds.) 1998. Extensible Markup Language (XML). http://www.w3.org/TR/1998/REC-xml-19980210.html.

Brown, C.M. 1988. Human Computer Interface Design Guidelines. Norwood, NJ: Ablex Publishing Corporation.

Bronnikov, G.K. & Smirnov, A.A. 1998. Shaman Library. http://starling.rinet.ru/shaman/.

Campione, M. & Walrath, K. 1996. The Java Tutorial. Addison-Wesley.

Campbell, R.H. & Randell, B. 1986. Error Recovery in Asynchronous Systems, IEEE Transactions on Software Engineering. Vol. 12, no. 8, 811-826.

Carroll, J.M. & Aaronson, A.P. 1988. Learning by Doing with Simulated Intelligent Help, Communications of the ACM. Vol. 31, no. 9, 1064-1079.

Chillarege, R. 1996. Orthogonal Defect Classification. In Lye, M.R. (ed.) Handbook of Software Reliability Engineering. New York, NY: McGraw-Hill, 359-400.

Clark, J. 1994. Sgmls SGML Parser.

Dain, J.A. 1991. Syntax Error Handling in Language Translation Systems. Research Report 188. Coventry, UK: University of Warwick.

DeRusso, J. & Haggar, P. 1998. Multithreaded Exception Handling in Java, Java Report, August 1998, 13-26.

Dromey, G. 1989. Program Derivation: The Development of Programs from Specifications. Addison-Wesley.

Eisenstadt, M. 1997. My Hairiest Bug War Stories, Communications of the ACM. Vol. 40, no. 4, 30-37.

Gentner, D. & Nielsen, J. 1996. The Anti-Mac Interface, Communications of the ACM. Vol. 39, no. 8, 70-82.

Gilb, T. et al. (eds.) 1993. Software Inspection. Addison-Wesley.

Goodenough, J.B. 1975. Exception Handling: Issues and a Proposed Notation, Communications of the ACM. Vol. 18, no. 12, 683-696.

Gosling, J. et al. 1996. The Java Language Specification. Addison-Wesley.

Gould, J.D. & Lewis, C. 1985. Designing for Usability: Key Principles and What Designers Think, Communications of the ACM. Vol. 28, no. 3, 300-311.

Halfhill, T.R. 1998. Crash Proof Computing, Byte, April 1998, 60-74.

Harbison, S.P. & Steele, G.L.Jr. 1991. C - A Reference Manual. Prentice-Hall.

Hoare, C.A.R. 1981. The Emperor's old Cloths, Communications of the ACM. Vol. 24, no. 2, 75-83.

Hoare, C.A.R. 1986. Mathematics of Programming, Byte, August 1986, 115-124.

Hof, M. et al. 1996. Zero-Overhead Exception Handling Using Metaprogramming. Technical report. Linz, Austria: Johannes Kepler University.

Jézéquel, J. & Meyer, B. 1997. Design by Contract: The Lesson of Ariane, IEEE Computer. Vol. 30, no. 2, 129-130.

Joyner, I. 1996. C++??: A Critique of C++, 3rd ed.. http://www.elj.com/cppcv3/.

Kahan, W. & Darcy, J.D. 1998. How Java's Floating Point Hurts Everyone Everywhere. ACM Workshop on Java for High-Performance Network Computing. Palo Alto, CA: Stanford University.

Knuth, D.E. 1989. The Errors of TeX, Software Practice Experience. Vol. 19, no. 7, 607-685.

Koenig, A. & Stroustrup, B. 1990. Exception Handling for C++ (revised). USENIX C++ Conference, 149-176.

Kogtenkov, A. 1998. Unrecognized Eiffel.
http://www.eiffel-forum.org/archive/kogtenkov/unrecognized.htm.

Krall, A. & Probst, M. 1998. Monitors and Exceptions: How to Implement Java Efficiently. ACM Workshop on Java for High-Performance Network Computing. Palo Alto, CA: Stanford University.

Kreyszig, E. 1993. Advanced Engineering Mathematics, 7th ed.. John Wiley & Sons.

Laprie, J.-C. & Kanoun, K. 1996. Software Reliability and System Reliability. In Lye, M.R. (ed.) Handbook of Software Reliability Engineering. New York, NY: McGraw-Hill, 27-69.

Laitinen, K. 1995. Natural Naming in Software Development and Maintenence. Espoo: Technical Research Center of Finland.

Lee, P.A. & Anderson, T. 1990. Fault Tolerance - Principles and Practice, 2nd ed.. Vienna: Springer Verlag.

Leveson, N.G. et al. 1991. Safety Verification of Ada Programs using Software Fault Trees, IEEE Software, July 1991, 48-59.

Lewis, C. & Norman, D.A. 1986. Designing for Errors. In Norman, D.A. & Draper, S. (eds.) User-Centered System Design. Hillsdale, NJ: Erlbaum.

Lieberman, H. 1997. The Debugging Scandal and What to Do About It, Communications of the ACM. Vol. 40, no. 4, 27-29.

Lindholm, T. & Yellin, F. 1996. The Java Virtual Machine Specification. Addison Wesley.

Liskov, B.H. & Snyder, A. 1979. Exception Handling in CLU, IEEE Transactions on Software Engineering. Vol. 5, no. 6, 546-558.

Liskov, B.H. & Wing, J.M. 1994. A Behavioral Notation of Subtyping, ACM Transactions on Programming Languages and Systems. Vol. 16, no. 6, November 1994, 1811-1841.

Meyer, B. 1992. Eiffel: The Language. Prentice-Hall.

Meyer, B. 1997. Object-Oriented Software Construction, 2nd ed.. Upper Saddle River, NJ: Prentice-Hall.

Meyer, B. 1998. Warnings are a Cop Out.
http://www.elj.com/eiffel/bm/warnings-are-a-cop-out/.

Microsoft. 1996. Building Applications with Microsoft Access 97.

Molich, R. & Nielsen, J. 1990. Improving a Human-Computer Dialogue, Communications of the ACM. Vol. 33, no. 3, 338-348.

Morrison, J. 1985. "EA IFF 85" Standard for Interchange Format Files. In Commodore Amiga Inc. Amiga ROM Kernel Reference Manual Devices, 3rd ed.. Addison Wesley, 355-379.

Norman, D.A. 1983. Design Rules Based on Analysis of Human Error, Communications of the ACM. Vol. 26, no. 4, 254-258.

Payne, J.E. et al. 1998. Implementing Assertions for Java, Dr. Dobb's Journal, January 1998, 40-44 101-102.

Pinker, S. 1997. How the Mind Works. New York, NY: W. W. Norton & Company Inc.

Pressman, R.S. 1997. Software Engineering - A Practitioner's Approach (European Adaption), 4th ed.. New York, NY: McGraw-Hill.

Raggett, D. et al. (eds.) 1997. HTML 4.0 Specification (revised). World Wide Web Consortium. http://www.w3.org/TR/REC-html40-971218.

Raymond, E.S. et al. 1999. The On-Line Hacker Jargon File, Version 4.1.0. http://www.tuxedo.org/~esr/jargon/.

Reason, J. 1990. Human Error. Cambridge, UK: Cambridge University Press.

Ritchie, D.M. 1993. The Development of the C Language. Second History of Programming Languages Conference. Cambridge, MA.

Romanovsky, A.B. 1997. Practical Exception Handling and Resolution in Concurrent Programs, Computer Languages. Vol. 23, no. 1, 43-58.

Schmidt, A.P. & Bisang, P. 1998. Kreativität als Innovation, c't. No. 18, 204-206.

Sharp Corporation. 1989. Taschencomputer PC-E500 Bedienungsanleitung (Pocket Computer PC-E500 User Manual). Osaka, Japan.

Shneiderman, B. 1997. Designing the User Interface: Strategies for Effective Human-Computer Interaction, 3rd ed.. Addison-Wesley.

Scheuning, A.J. 1996. Error Classification in the Light of ISO 9001. Conference Proceedings of Fifth European Conference on Software Quality. European Organisation for Quality, 255-266.

Silberschatz, A. & Galvin, P.B. 1994. Operating System Concepts, 4th ed.. Reading, MA: Addison-Wesley.

Sippu, S. 1981. Syntax Error Handling in Compilers. Report A-1981-1. Finland: Department of Computer Science, University of Helsinki.

Stallings, W. 1998. Operating Systems: Internals and Design Principles, 3rd ed.. Upper Saddle River, NJ: Prentice-Hall.

Stallman, R. et al. 1998. The GNU C Compiler. Boston, MA: Free Software Foundation.

Strong, D.M. & Miller, S.M. 1995. Exceptions and Exception Handling in Computerized Information Processes, ACM Transactions on Information Systems. Vol. 13, no. 2, April 1995, 206-233.

Stroustrup, B. 1997. The C++ Programming Language, 3rd ed.. Addison-Wesley.

Taft, T. & Duff, R.A. (eds.) 1997. Ada 95 Reference Manual: Language and Standard Libraries. International Standard ISO/IEC 8652:1995. Springer Verlag.

Tasker, M. 1999. Memory Management and Cleanup. EPOC Technical Paper. London, UK: Symbian Ltd.

Thimbleby, H. 1998a. The Detection and Elimination of Spurious Complexity. In Backhouse, R.C. (ed.) Proceedings of Workshop on User Interfaces for Theorem Provers. Eindhoven: University of Technology, 15-22.

Thimbleby, H. 1998b. A Critique of Java. http://www.cs.mdx.ac.uk/harold/.

Thimbleby, H. 1998c. Spare the rod, spoil the computer?. http://www.cs.mdx.ac.uk/harold/.

Thimbleby, H. 1999. Creating Discerning Users. http://www.cs.mdx.ac.uk/harold/srf/.

Venners, B. 1997. How the Java Virtual Machine Handles Exceptions, Javaworld, January 1997. http://www.javaworld.com/jw-01-1997/jw-01-hood.html.

Venners, B. 1998. Design Techniques: Designing with Exceptions, Javaworld, July 1998. http://www.javaworld.com/jw-07-1998/jw-07-techniques.html.

Weinberg, G. 1971. The Psychology of Computer Programming. New York, NY: Van Nostrand Reinhold.

Wirth, N. 1988. From Modula to Oberon, Software Practice and Experience. Vol. 18, no. 7, 661-670.

Zima, P.V. 1997. Moderne/Postmoderne. Tübingen, Germany: Francke Verlag.