How should error handling be constructed?

Or, What should exceptions look like?

Mika Raento

Helsinki Institute for Information Technology
Department of Computer Science, University of Helsinki
mraento@cs.helsinki.fi

1 Introduction

There exists a tradition of criticizing the state of error handling and error messages in programs, dating from the 70's [1], through 80's [2,3], 90's [4] and to the 2000's [5]. The same, usually HCI-oriented but sometimes technical, literature also provides guidelines for better error handling from the user's perspective. The guidelines include providing advise to the user, uniform reporting strategies, attaching error messages to corresponding input fields, automatic recovery and graceful degradation.

Why aren't these guidelines followed by now? From experience in building medium-sized software with run-of-the-mill developers I claim that the reason is lack of good design practice and architectural principles when it comes to error-handling.

There is a wealth of literature in the low-level details of exceptions and their effects on control-flow, starting with the original proposal by Goodenough [6], through Cristian's comprehensive treatment on the theory of exceptions and fault-tolerance [7] to being included in any current textbook on C++, Java and .Net. What is missing are practical, implementable guidelines, design patterns and frameworks for turning the control-flow mechanisms into user-oriented error-handling techniques. This article provides a conceptual framework for arguing about error-handling, a concrete pattern based on the concepts, and experience in implementing it in a medium-size business information system.

The framework proposed here is designed to support a program built from several independent and re-usable components. This has a large impact on how easily we can reach the goals stated by the HCI literature: different subsystems cannot know the user's intentions and domain, and it mostly precludes the use of side-channels (like stderr) to communicate error information.

The article is structured as follows: I begin with a synopsis of the philosophy behind the framework. Before detailed reasoning, some assumptions and border conditions as to which kind of systems I can claim to be able to reason about are presented. I then introduce the basis of the reasoning: entities and strategies, deriving from their combination a number of informational requirements on exceptions. These information requirements and their use in error handling are discussed in the main part of the text, concluding with a design for a generic exception class and handling logic, fulfilling these requirements. Some of the assumptions are re-evaluated, implementation issues and further research are discussed in the conclusions.

2 Synopsis

This synopsis provides an introduction to the philosophy behind the framework proposed. It should not be taken too seriously, and the serious-minded reader is urged to continue straight to the next section (but maybe come back here in the end).

There are three kinds of really different error handling in programs:

ImmediateCaller A subroutine `foo` calls subroutine `bar` which detects an error. The possible errors detected by `bar` should be documented and `foo` may be able to recover or ignore some of them, based on the specific kind of error. If it can't, it'll pass the failure upwards.

Conclusion 1 You may need to document specific errors and supply additional information to the immediate caller.

Since by definition exceptions are exceptional, in *most* cases ImmediateCaller will propagate the exception.

Conclusion 2 In most cases, exceptions are handled somewhere else than ImmediateCaller.

UpperLayer Subroutine `baz` gets the error passed up by `foo`. It no longer knows about `bar`, so any documentation or specific error information produced by `bar` is no longer useful. *But since the error didn't originate in `foo`, neither is any additional information supplied by `foo`* (see section 4.4 for some relaxations of this claim).

Conclusion 3 The error information must support generic handling strategies, like a blind retry.

Conclusion 4 Low-level exceptions should not be remapped by upper layers.

Human The error cannot be corrected by the program, but must be corrected by a human. Since the error was detected by `bar`, `bar` has the most specific knowledge of the error and must describe it.

Conclusion 5 The routine detecting an error is responsible for describing it in terms understandable to a human, often the user.

It is not always the user who can fix the error, but sometimes it is a system administrator or even the programmer.

Conclusion 6 The system must be able to tell the user if they should contact another human.

Now the user is told 'Contact your system administrator'. The error description given by the low-level routine is not shown to the user, but must be forwarded to the administrator instead.

Conclusion 7 The system must produce different messages for consumption by different humans.

If these three radically different ways of dealing with errors are not kept separate, confusion ensues, and most current arguments in the developer community can be seen as resulting from this confusion: The discussion of the relative merits of checked exceptions [8] can be seen as mixing cases ImmediateCaller and UpperLayer: for the cases when ImmediateCaller can do something, checked exceptions are great. It's just that this is *normally* not the case, and so most effort should be spent on something else. Error codes as error information (as in most system calls, Unix, VMS or Win32) can be adequate for ImmediateCaller, but often insufficient for UpperLayer and completely useless for the Human.

The claim of this paper is then that by keeping these separate and providing facilities for programmers to take advantage of the separation, error handling will both be easier to code and produce better results.

3 Preliminaries

The proposed framework stems from experience with certain kinds of systems, and cannot be assumed to be directly applicable to all kinds of programming. The kinds of systems are detailed here, as well as the notation introduced.

3.1 Border conditions and assumptions

Throughout, I will assume that the program does automatic backward error recovery. Backward error recovery means that after an exception occurs, the program (block, component or system) returns to a known safe state, from which the operation may be safely retried or abandoned. (for a more detailed introduction, see for example; NASA's 'Software Fault Tolerance: A Tutorial' [9, section 4.1.4].) Note that not all levels must implement perfect recovery (or make the *strong exception safety guarantee* [10]), as long as the system as a whole knows which levels are safe.

Backward error recovery also presupposes a "safe" programming environment: an environment where errors cannot corrupt arbitrary state, e.g., buffer overflows. It is often necessary to assume safety even in environments which aren't strictly safe (such as C++), since otherwise we would not be able to do anything: the program state may have been corrupted even if we do not detect any errors.

Implementing backward error recovery is orthogonal to almost all of the discussion. Whether you use transactions, `finally`-blocks, or Resource-Acquisition-Is-Initialization [11, 129–130] to achieve rollback, rolling back should never depend on what went wrong — just do it if something does. This treatment can be seen as a form of forward-error recovery: how to continue after an error, towards filling the goal — but from the point of view of the user, not just the program.

The argumentation will mostly be based on the assumption that there is a human user interacting with the program, via some kind of keyboard-mouse-display modality. I shall not consider batch-oriented or autonomous systems nor tactile or voice-only modalities. Neither is the focus on high-reliability or real-time systems.

This paper is based strongly on implementation experience with a single medium-size (500 KLOC) system with a developer population of about 15. It would seem that the framework I propose will yield increasing returns as the system size grows, but I do not have empirical proof of that. Additionally the framework has been applied to a smaller (150 KLOC) system with two developers.

3.2 Example environments

In discussing the requirements and program frameworks I will use two example systems:

***sys*⟨*CCF*⟩** , a 500 KLOC web-based form-driven database-backed business system that I was the system architect for, and

***sys*⟨*EMail*⟩** (not a real system, just a fictional example) an e-mail application

Figure 1's rough system diagrams show the major independent subsystems in both examples. The subsystems tend to have their own kinds of possible errors.

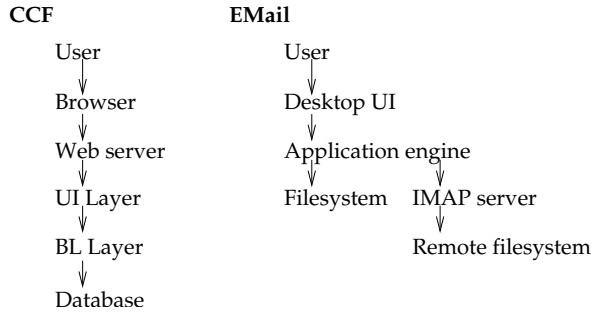

Fig. 1. Rough system diagrams for example programs, identifying separate subsystems and failure points.

sys⟨*CCF*⟩is selected since it represents real experience and an input-heavy, typical business system with multiple layers and possible errors. It has no direct manipulation and no local system component that the lower levels would communicate with. The implementation uses a typical three-tier architecture.

An e-mail app has a very different user-interface style - direct manipulation and a large local system component, which put different emphasis on error handling. Although I have no experience on building one, it should serve as a good example since most of the tasks and error-conditions should be well known to the readers.

3.3 On Notation

The following sections will discuss four kinds of constructs, each identified by a specific convention:

Entities like *ent*⟨*User*⟩. These are actors that may react to exceptions.
Strategies for continuing after an exception, such as *str*⟨*GiveUp*⟩. Strategies categorize possible reactions to exceptions after backward recovery.
Informational requirements for handling errors like *info*⟨*Inputs*⟩— what information must be propagated up when an exception occurs, in one way or another— and
Kinds of exceptions such as *exc*⟨*Bug*⟩. The kinds are selected so that they influence strategies, entities or informational requirements.

4 Requirements and guidelines for an exception handling framework

I first introduce a framework for reasoning about exception handling: *entities* that handle exceptions, and *strategies* they can use. From these a number of *informational requirements* for exceptions are derived.

4.1 What is exception handling: a framework

Definition: *Exceptions are used to signal that a block cannot carry out it's assigned task.* This includes Siedersleben's *Emergencies* [12] as well as violated pre- or post-conditions [7] (emergencies can be seen as a specific reason for

post-condition violations). I do not consider using exceptions for generic co-routine -like control-flow manipulation (NOTIFY, and SIGNAL exceptions in Goodenough's original proposal [6]).

Exception *handling* is about what to do next. The goal is to succeed in carrying out the task (or to give up). There are two orthogonal aspects in this:

- Who handles the exception:
  ***ent*⟨*System*⟩** the program or supporting system
  ***ent*⟨*User*⟩** the user
  ***ent*⟨*SystemAdministrator*⟩** a system administrator
  ***ent*⟨*Programmer*⟩** the programmer
- What do they do, which *strategy* to use:
  ***str*⟨*Retry*⟩** retry without changing anything
  ***str*⟨*ChangeSomething*⟩** change something and retry
    ***str*⟨*ChangeInput*⟩** change the input
    ***str*⟨*FixEnvironment*⟩** change environmental condition
    ***str*⟨*FixProgram*⟩** change the program
    ***str*⟨*Reset*⟩** reset to a known state
    ***str*⟨*ChangeState*⟩** change the subsystem's internal state
  ***str*⟨*Alternative*⟩** use alternative implementation or instance
  ***str*⟨*GiveUp*⟩** give up (and propagate exception)
  ***str*⟨*Ignore*⟩** skip the action (ignore)

The entities are chosen to represent most of the real-world entities, but the granularity has been chosen to support the logical structure of the argumentation: the different (We could easily divide the *ent*⟨*System*⟩either more along the lines of the Synopsis, section 2, or into the application program, run-time environment, agents etc. From the users point of view there may not exist an administrator and a programmer, just "support") entities can use different strategies. Although much of current discussion does revolve around differences in *ImmediateCaller* and *UpperLayer*, *ImmediateCaller* should seldom matter: exceptions should be handled centrally and `catch`-blocks should be rare [13].

The strategies are supposed to be a fairly comprehensive union of forward-recovery mechanisms proposed in the literature [14, p. 92] [9, p. 12–16][11, p. 130], and encountered in real life, modulo the constraints given in section 3.1 (we are not considering, for example, multi-version programming). Again, the granularity is chosen to fit the argumentation, but reflects real-world differences in ease, cost and results of using the strategy.

To carry out different strategies, the entities need *information* as to what went wrong. These are the informational requirements exceptions should fulfill (some of them are later expanded):

***str*⟨*Retry*⟩** : is the problem retriable (requirement: *info*⟨*Retriable*⟩)
***str*⟨*ChangeInput*⟩** : what input is wrong and how should it be changed (requirement: *info*⟨*Inputs*⟩, *info*⟨*Message*⟩)
***str*⟨*FixEnvironment*⟩** : what is wrong with the environment, and which environment (requirement: *info*⟨*Environmental*⟩, *info*⟨*Message*⟩)
***str*⟨*FixProgram*⟩** : what went wrong with the program and where (requirement: *info*⟨*Program*⟩)
***str*⟨*Reset*⟩, *str*⟨*Alternative*⟩** : which subsystem failed and should be reset (requirement: *info*⟨*Subsystem*⟩)
***str*⟨*ChangeState*⟩** : exactly what is the problem with the subsystem (requirement: *info*⟨*ErrorCode*⟩)

str⟨Ignore⟩ : whether it is safe to continue (requirement: *info⟨Corrupt⟩*)

Not all alternatives are available to all entities. The *ent⟨System⟩* can mostly do *str⟨Retry⟩*, *str⟨Reset⟩*, *str⟨ChangeState⟩*, *str⟨Alternative⟩*, *str⟨GiveUp⟩* and *str⟨Ignore⟩*— sometimes *str⟨FixEnvironment⟩*. The system should not normally do *str⟨ChangeInput⟩* on its own (although see [9, section 4.1.6]). Blind retries without replication/multi-version programming (*str⟨Retry⟩*, *str⟨Reset⟩*) are surprisingly effective: Grey describes such strategies extending Mean-Time-Between-Failures by a factor of 5–100 [15].

The *ent⟨User⟩* can carry out *str⟨Retry⟩*, *str⟨ChangeInput⟩*, some of *str⟨FixEnvironment⟩*and maybe *str⟨Alternative⟩*. If none of these are applicable the user must be told who to contact (*ent⟨SystemAdministrator⟩* or *ent⟨Programmer⟩*), in the *info⟨Message⟩*. Since in some cases the user will be told how to fix the problem, and sometimes they will be asked to contact somebody else, we need several error messages. We need one message for each entity (*info⟨UserMessage⟩*, *info⟨AdministratorMessage⟩*, *info⟨ProgrammerMessage⟩*). From the user's point of view *str⟨GiveUp⟩*and *str⟨Ignore⟩*tend to be same for single actions (like saving a file), but they may be able to skip some parts of multi-step actions (like sending mail to several recipients: if one of the addresses is invalid, the user may well want to sent the message to the rest).

The *ent⟨SystemAdministrator⟩*can do all except *str⟨FixProgram⟩*, and *ent⟨Programmer⟩*is the only one who can do *str⟨FixProgram⟩*.

Note that the question is not only about strict ability, but about efficiency. It is (within reasonable time limits) much more efficient for the system to carry out tasks than the user, the user rather than the administrator, and the administrator rather than the programmer. So this hierarchy of cost should be respected in any real system—but not to the extent of masking the actual source of error: if there is a bug in the program, the user should be notified that this is the case so that the program can be fixed.

There is also a very important subset of (*str⟨GiveUp⟩*): corrupt system state. If the system notices that internal state has been corrupted in a way that requires a restart or (worst case) manual intervention it should communicate this in clear terms to the user so that data-loss can be minimized (*info⟨Corrupt⟩*).

The kinds of information noted above: *info⟨Retriable⟩*, *info⟨Inputs⟩*, *info⟨Message⟩*, *info⟨Environmental⟩*, *info⟨Program⟩*, *info⟨Subsystem⟩*and *info⟨ErrorCode⟩*, are the requirements for our exception objects. They will be discussed in detail in the following sections.

4.2 Informational requirement *info⟨Inputs⟩*: what input was erroneous, and how it should be fixed

The user's primary (and you could say only) contact surface with a program is the user interface. If the program asks the user to change the input, it should indicate it *on the user interface*—not just via the text of an error message. E.g., when creating a new product in a product database with a name that already exists, the name field on the form should be highlighted and the error message printed next to that field. The user should not be made to guess which field to correct (especially crucial, if it possible to key in multiple new products at once). This is stated in most usability guidelines, and covered in detail by Alan Cooper [4, p. 398–402].

In the era of dumb terminals connected to mainframes, a single special-purpose program often accepted the input from a form and handled it all the way through to the data file. It *knew* what the user's domain was and what the user interface looked like. This is still true of e.g., compilers: the whole compiler knows that input comes in the form programming language constructs in files, and even the back-end can construct error messages to take this into account. A truly reusable component cannot do this—but it does not mean that we have to abandon the goal of relating errors to the user's domain.

This means that even a low-level component must be able to attach to the exception a reference, that can be ultimately mapped to a user-interface element. The actual values of the inputs are often useful as well, so that complete information is easily serialized for logging, and in case the original fields are somehow lost or modified.

There are two primary arguments against the need to identify user-interface elements from lower-level code:

- Validation can be done at the user-interface level, like checking that only numbers are typed in a number field. This is patently false in many cases:
  - *sys*⟨*CCF*⟩: a uniqueness constraint for a named entity in the database can be reliably checked for only by inserting into the database
  - *sys*⟨*EMail*⟩: password for the IMAP server can only be checked by the server; ability to copy messages to a folder only by copying
- the user interface should not allow the user to make errors (cf. Cooper [4, ch. 28]), e.g., by selecting a date via a calendar instead of manually keying in the date. This is false pretty much for the same reasons as the previous claim: the user interface cannot reliably check all possible errors.

(I do not claim that Cooper isn't right about error prevention being most efficient, I claim that not all errors can be prevented in a multitasking, distributed, multiple security domain environment).

In addition to noting which input field is to be corrected, the user must know *what* is wrong with it (e.g., that a widget named "baz" already exists) to be able to correct it. This message must be understandable to the user (*info*⟨*UserMessage*⟩), and provide hints as to how the input could be fixed [16].

But not only invalid inputs need to be connected to user-interface elements. Say an 'Out of disk space' error occurs when writing to a local mail folder in *sys*⟨*EMail*⟩. The operation is probably asynchronous, the user is already doing something else when the exception is raised. It is much more tangible to the user if the folder can be highlighted with the error message, instead of getting a generic error notification.

User inputs may of course come from other sources than currently active user interface elements. Settings should be identifiable in the same way, the location of processing errors in files passed by the user as well (as a compiler does).

4.3 Something is wrong with the environment *info*⟨*Environmental*⟩

Environmental problems signal that things outside the software system in use are in a state that make the action requested impossible. Examples are a full disk, lack of network connectivity or lack of rights to write to a file/directory. They are most often outside the scope of the system to resolve. Environmental problems thus can be simply flagged as such to the program, regardless of the exact error, and the program just needs to get the information to the right human.

Some of the environment in which the program operates may be controlled by the user. This is true, for example, of the local file-system of a desktop application (*sys⟨EMail⟩*). Some of the environment is *not* controlled by the user, even if the problem manifests locally — like the inability to obtain an IP address via DHCP. And parts of the program may rely on components running on other systems like IMAP servers and databases, whose environments are definitely outside the user's domain.

If the user is in control of the problematic environment, then the problem should be communicated to them in *info⟨UserMessage⟩*. If not, they should be directed to contact the system administration, and helped to send them the appropriate information. This information should be separate from the *info⟨UserMessage⟩*, since it is not useful to the user and will probably contain language and concepts alien and even frightening (*info⟨AdministratorMessage⟩*).

How does the program know whether the environment is under the user's control? In the general case, it cannot know for sure - but it can get it right in many typical cases:

- an environment-exception from a remote system should be mapped to a remote-environment-exception (*exc⟨RemoteEnvironment⟩*) at the communication boundary
- the remote system should be able to flag special *exc⟨UserEnvironment⟩*exceptions when it knows that the former is not the case (like out-of-quota on the IMAP server)
- local environment-exceptions can be assumed to be under the user's control, unless stated otherwise (the system component noting that there is no contact with the DHCP server although the network is connected can flag this as *exc⟨AdministratorEnvironment⟩*)

We can also try to err on the side of caution: unclear cases will be treated as *exc⟨AdministratorEnvironment⟩* exceptions. The advanced user can peek at the *info⟨AdministratorMessage⟩* to correct them.

4.4 Messages to humans *info⟨Message⟩*: *info⟨UserMessage⟩*, *info⟨AdministratorMessage⟩*, *info⟨ProgrammerMessage⟩* and levels

Conventional wisdom [17] goes something like this:

- each subsystem documents/exception-declares a number of error codes or exception classes that it may raise
- these should cover all cases when upper-level code can take some action
- layers must remap lower-level exceptions unless they have been declared to be thrown by the subsystem
- the UI layer manages how errors are presented to the user and with what messages, so that the lower layers can stay general and do not have to know about the user

This might work with infinite resources to document, declare, correctly map and handle all error conditions at all levels. Anybody who has used actual applications knows that this is not the case:

- The November 2004 ACM Queue [5] has several articles on time lost to debugging when error information is wrongly mapped and hidden

– Even a recent (Office XP) version of Microsoft Word gives this error message when trying to save a document to a write-only directory: 'Make sure the disk is not full, that the remote system can be accessed, and that the disk is not write-protected.'
– Trying to connect Mozilla Thunderbird to a misconfigured IMAP server:
  • on connection, IMAP server notices it is too misconfigured to run and drops the connection
  • Thunderbird fails silently, since it assumes all connection errors to be retriable

The detector creates the error message The solution is for low-level subsystems to provide reasonable error messages in language understandable to those that deal with the environment of that subsystem or with inputs to that subsystem. E.g.: a database constraint (since coded by the application developer) can already state in user-understandable language "A gizmo with the specified name 'foo' exists already in the database", or a filesystem can state 'you do not have permissions to write to directory /bar/baz', or that 'Disk D: is full'. This also removes the problem observed in many cases where only error-codes are used for exception information: although there is a string table somewhere that maps the codes to descriptions, it may be hard to find (lives in a third-party DLL) or not retrieved by the program.

The crucial step in the reasoning is to notice that *levels of abstraction* and *domains* are not the same: although filesystem manipulation is very different from the functioning of an e-mail client, errors with the local filesystem are often in the user's domain. Blindly translating lower-level exceptions to the assumed user level can produce very bad results: transforming 'Quota exceeded on /home/mraento' to 'Cannot copy messages to folder' where the latter helps the user none whatsoever.

When can we show the low-level message? These messages can be shown to the user (*info*⟨*UserMessage*⟩) if the conditions below apply:

1. for environmental exceptions, the environment is within the domain of the user (discussed in the *info*⟨*Environmental*⟩section above)
2. the inputs related to non-environmental exceptions are within the domain of the user

If the program knows that 2 holds, it can happily communicate the low-level message upwards, eventually to the user with no need for mapping. If 2 doesn't hold, then the exception should be handled as an *exc*⟨*Bug*⟩exception and a generic 'Contact support' message put into *info*⟨*UserMessage*⟩(and the original *info*⟨*UserMessage*⟩to *info*⟨*ProgrammerMessage*⟩).

How can the program then know that the input came from the user? In the *info*⟨*Inputs*⟩section I argued that the exception should be able to refer (in a yet unspecified way) to fields in the user interface when appropriate. If the program notes that the *info*⟨*Inputs*⟩information in the exception *does* refer to something in the user interface, then 2 holds. Let's look at an example:

> The *sys*⟨*EMail*⟩ application uses a library to handle mbox-format [18] mail folders (files). This same library could be used from a command-line program to modify mbox-files.

Now consider a 'File not found' error when the library tries to open the mbox file. If this is the result of the user selecting a drag-and-drop target in the *sys⟨EMail⟩* app, then it is the application's fault: it should not show non-existent folders and so the error is probably a bug (e.g., an UTF-8 encoding error in the filename handling). If, on the other hand, the filename was typed by the user as a command line, then the 'File not found: quuz' should be shown to the user.

In the first case a corresponding input field cannot be found, and so the *exc⟨InvalidInput⟩*exception is mapped to a *exc⟨Bug⟩*exception - in the second case the input does exist and the exception is used as-is.

There is a slight conflict here: to do this mapping from *exc⟨InvalidInput⟩*to *exc⟨Bug⟩*, we want to limit inputs to those directly available from the user – but in general we should try to map errors to the user-interface much as possible. In the example, we would like the mbox file name to be linked to the folder in the user interface. To fix this, we need to have an additional flag in the input mapping: whether the input has been *directly* given by the user, or *indirectly* derived from user input (or, in other words, *related* to user input).

What if we can't? What if neither of the conditions 1 or 2 hold? Can we do better than tell the user 'Internal error, contact support'? Again, traditional wisdom seems to tell us to describe the whole chain of actions, so that the user gets an error message:

Could not copy messages to folder X,
(because) could not open folder X
(because) the specified file does not exist

It requires all the larger steps to be named in the program, and the stack of context put together either by making a context stack that is used when constructing the exception object (like *info⟨StackTrace⟩*), or by each larger change of abstraction-level to catch errors and add their context to the message.

Now the question is: how useful is this? (I.e., does the user gain something from such messages). The message will not enable them to fix the problem any better, but it can still be useful *if the user can use another strategy based on this message*, instead of *str⟨GiveUp⟩*do *str⟨ChangeInput⟩ str⟨Ignore⟩*:

str⟨ChangeInput⟩ If there is an alternative course of action the user can take, and the message helps in finding that course (copy to folder Y instead of X).

str⟨Ignore⟩ the action that failed can be skipped: the user can direct the program to not do that step of a multi-step process (e.g., creating and sending an invoice, where the sending fails but the user may specify that the invoice should not be sent now but left in an pending-send state)

Can the program know that the current user action is made of several independent steps that may be skipped or whose inputs changed independently? A subroutine *does* know that: if the routine's steps rely on different inputs, or a flag may cause it to skip some steps then that routine is made of independent steps. The Controller in the Model-View-Controller paradigm [19, p. 4–6] should know how the user's actions are translated into steps in the program. With *sys⟨CCF⟩*we initially had quite a few modules add their current context to error messages, but it turned out it didn't help the user at all. I think this problem is hard to solve generically. When in doubt, don't add context.

Localization Since the low-level components are producing end-user messages, they need to be localized and aware of the end-user's language. This may require some changes to current call chains, but nothing insurmountable. Although requiring a service to know the client's interface language demands may seem backward at first, it can be save huge amounts of effort: there may be tens of thousands of applications that run on a specific operating system. If the operating system has localized messages, they only have to be written once.

On the subject of localization: it is perfectly normal not to localize *info⟨ProgrammerMessage⟩*and it may be acceptable not to localize *info⟨AdministratorMessage⟩*. Not localizing them will probably make developers feel more free in providing more information, since they do not have to take into account the cost and effort of localization and the code needed to support it.

Never lose information! If the program *does* decide to remap an exception, it should never throw away the original one (much of system and program debugging time tends to be spent in trying to figure out what originally triggered the fault). The original exception with no modifications should be stored in a *info⟨InnerException⟩*field, and it's messages be made available on request.

The idea of an *info⟨InnerException⟩*is not my invention, but consistent support for it is surprisingly new. The feature has been in Java from 1.4 and in .Net from the beginning. In both it must be specified by the programmer, however. In my opinion, the language should always supply the *info⟨InnerException⟩*when constructing a new exception in an exception handler. There must be a way to override this so that sensitive information is not leaked, but *it must be the default.*

4.5 Just retry *info⟨Retriable⟩*

The simplest strategy for overcoming an exception is just to retry the action. Simple retries can also be very effective [15]. This is suitable in many cases, like:

1. temporary resource exhaustion (no more database connections available)
2. contention for shared objects (deadlock in the database)
3. network errors (connection to IMAP server lost)

but on the other hand the same actions that trigger the above-mentioned conditions may easily result in non-temporary errors as well

1. the database is down
2. the table referenced does not exist
3. the local network interfaces are down

The retriability/temporariness is often a matter of degree, rather than an absolute— the database will probably come up again, and so will the local network. The decision whether to retry automatically has to take time spans into account. Retries can cascade, each layer doing it's own retrying. It is not atypical to see five layers, each doing five retries. If then the bottom-most action has a fifteen second timeout the user may have to wait for six minutes (add two more layers and it becomes 2.5 hours!). Routines that retry must both document their retry behaviour and take a timeout parameter, which specifies how long they may take to complete their work.

Note that for the retry strategy no further information on the kind of error is needed—just a flag that states that the exception is retriable. Quite often already the code that raises the lowest-level error (like the database deadlock) knows whether the error is retriable, and if not it should be wrapped with a Facade [19, p. 185–193] or aspects [20] to provide the information based on *info⟨ErrorCode⟩*.

4.6 Where's the bug in the program *info⟨Program⟩*

If there is an error in the program, it has to be unambiguously flagged as such, since no amount of effort on the user's part can *fix* the problem, although it can be worthwhile to try to provide workarounds. Sometimes the program can easily notice bugs (e.g., violations of pre- or post-conditions), but not nearly always—sometimes of course bugs will not result in exceptions at all.

If the exception *is* the result of a bug, then the programmer needs as much information as possible as to what the cause of the bug is. Typical desired information is a stack trace with

- current function
- source code file and line,
- file version
- input variable values (when feasible)

These can collectively be called the *info⟨StackTrace⟩*. The point in the program that detects the fault can often add a human-readable message for the programmer who has to debug the code (*info⟨ProgrammerMessage⟩*). Low-level routines may want to add current CPU register contents to this.

This is of course not always complete: a single stray pointer in a C/C++ program may cause a totally unrelated part of the program to fail at an unrelated time. To capture the original cause would mean to trace the total execution of the problem, which is not normally feasible. The actual amount of information has to be something less. Since this information does not necessarily have to be machine readable, any additional bits can be put in *info⟨ProgrammerMessage⟩*.

4.7 Tell me *exactly* what went wrong *info⟨ErrorCode⟩*

The most general source for error information a programmer has to deal with is the operating system or language platform. These tend to be mostly concerned with error codes and/or exception classes [21], [22, ch. 20], [23], [24]. Many language-oriented textbooks also seem to think that knowing the exact type of an exception is what you need to handle errors correctly [25, ch. 13], [26, ch. 5] [27, p. 138–139]. The analysis in this paper casts doubt on the usefulness of error-codes (even when redressed as exception types). Neither do the constructs displayed in these books support any strategies beyond those in this analysis, or support these strategies well.

As seen in previous sections there are many important (simple and efficient) automatic error-handling strategies that do not rely on what exactly went wrong—instead they need some general characteristics of the error (whether it is *exc⟨Retriable⟩*, *exc⟨InvalidInput⟩*, *exc⟨Corrupt⟩*, *exc⟨Bug⟩*, or *exc⟨Environment⟩*).

Assuming that low-level components may generate *info⟨UserMessage⟩*, there is no benefit to error codes from the user's immediate point of view (and they may the single most criticized [4,16] aspect of error messages).

That said, there are at least two situations where error codes are definitely useful:

1. Sometimes the layer above a certain subsystem *can* do error-specific actions: mostly related to *str⟨ChangeState⟩*Real examples I know include:
   - a build system resetting binary compatibility (BC) of a DLL when the linker gives BC error (while in development cycle)
   - a database client dropping indexes (and later recreating them) when the database states that the desired column definition changes would break the indexes
   - And there is the canonical 'File not found' exception example, where the upper layer may create the file and retry.

   It could be argued (and for the last case it is easy to do so) that these cases are the result of poor design. Instead of the client of an API having to catch an exception, change some of the state of the subsystem (or provide an empty resource) the API should provide a function that does all this (both Unix and Win32 provide means to open-or-create a file). Such an API would enable the upper layer to more clearly state its intentions and probably be less brittle.
2. Although from the user's *immediate* point of view an error code cannot help them, it can be convenient to have such an *exact* piece of information that specifies the error. It can then be put to Google to find others suffering from the same problem and maybe solutions. The localized error message does not appear in posts half as often, and it may take time for developers to decipher it.

On a related note, we are starting to see automatic bug resolution databases. I have experienced a Windows XP installation reporting a driver error to Microsoft and getting the prompted to install the correct updated driver that fixed the problem. This is clearly useful, and requires that most of the *info⟨Program⟩*information to be available in a machine-readable format.

All in all, I would say that we do need unique error codes, *at the point of detection*. They just should not be confused with unique *info⟨UserMessage⟩*s. And to make large systems maintainable [28, p. 347–348], the error codes should be in hierarchical namespaces. This *does not* directly mean that they must be classes, though - any namespacable identifiers will do.

4.8 SIDEBAR: Exceptions and error codes

Although exceptions were proposed already 30 years ago [6], there is still an ongoing argument about the merits of exceptions vs. error codes as witnessed by the amount of comments created by blogs discussing the issue [29,30].

The issue is muddled by the fact that exceptions can be both a control-flow mechanism and an error-information mechanism. First a simple example. Consider a (C) function (without error handling):

```
int foo(char* fn1, char* fn2) {
    int fd1, fd2;
    fd1=open(fn1, FLAGS);
    fd2=open(fn2, FLAGS);
    /* do something with fd1 and fd2 */
    close(fd1);
    close(fd2);
}
```

Now assume that this function uses a single integer code (whether as a return value or via an exception mechanism) to communicate what went wrong if something does. Each of the `open()` calls may fail in, say, 50 ways. To communicate exactly what went wrong, including with which of the two files, you need $50 \cdot 2 = 100$ error codes. In general you need (low-level error codes)·(low-level calls) error codes from any higher-level function. It should be obvious that this leads to a need for a combinatorial number of error codes—instead we should use a combinatorial mechanism: exception objects with a number of fields.

The traditional fix has been to use a side-channel to communicate the details of the error—often printing to a log or to stderr. But if the layer above needs to actually use the extra information it has to parse this—often imprecise—side channel. It should be noted that this kind of behaviour can be fully adequate in monolithic systems where there is a well-defined logging channel, such as command-line tools, database engines (for internal or environmental errors) or operating systems; and many examples [31, ch. 4.7] and guidelines [17] actually refer to such situations, obscuring the needs of systems built from fully reusable components.

The non-locality of exception-based control-flow affects the informational needs. It is all right for `open(char* fn, int mode)` to return a simple error code, since it is always related to the inputs it got and the error code can only be observed on the same line those inputs are specified. Now if the same error is communicated with an exception that escapes to another scope the inputs are not known anymore. Exceptions not only *can* carry more information, they *must* do so. This fact seems to escape some developers, a notable case in point being the smart-phone operating system Symbian: it uses exception-based flow-control but only allows for one 32-bit word as information [32, ch. 6].

The *greatest* benefit of exceptions is (of course) the non-locality: you don't have to check the return value of all calls. Error-handling can not only be separated from the normal flow, but it can also be *massively reduced*. Especially in the face of run-of-the-mill programmers and a dependable backward error-recovery mechanism this makes it much easier to construct reliable programs.

4.9 Which part failed *info⟨Subsystem⟩*

If the exception is not caused by something under the user's control, then they need to contact somebody else. A large system may consist of many independent parts—*info⟨Subsystem⟩*s. To find the right person who can fix the problem, the correct subsystem should be identified. (Note that it may not be necessary for the user to directly know the correct person, as long as their helpdesk can easily figure it out).

To increase fault-tolerance parts of systems may be replicated: several disks, servers, instances of processes. If one of the replicas fail, it is reasonable to retry the action on another one. It does not help, however, to connect to another web-server, if the database is down. (In deciding whether to use another replica, also the kind of error is important: it is no use trying to connect to another web server if the HTTP request is invalid — a *exc⟨InvalidInput⟩*error — but this is not always decidable).

The JavaGrid [33] provides a good example of the need to automatically identify which subsystem failed. JavaGrid is an automated batch-processing environment, which allocates jobs to remote machines. To be able to effectively handle processing errors, it needs to know whether they stem from a problem at the remote machine (which can be fixed by resubmitting to another machine),

from the network environment (which must be communicated to the administrator) or from erroneous input (which the submitter of the job must be told about).

The *info⟨Subsystem⟩*here is the *physical* subsystem: a run-time structural property, not logical. It means an object instance, process, or node - not library or class. Having such an information item will enable the fairly generic fault-tolerance mechanism of micro-reboots: restarting subsystems instead of whole systems, as proposed by Candea et al. [34].

4.10 Many exceptions? *info⟨NextException⟩*

It is often naive to stop the execution of the action at the first error in user input. The user may provide many pieces of input (in *sys⟨CCF⟩*a large, multi-item web form; in *sys⟨EMail⟩*when sending several mail messages; or as a source code file to a compiler), and it is most efficient if we can validate all or most of the input in one go.

Both .Net and Java assume that only one exception can be raised at a time. SQL-92 and ODBC on the other hand support an arbitrary (up to 32000) exceptions from a single database operation. I would say that the database approach is more efficient and generic. The program should be able to raise several exceptions (*info⟨NextException⟩*).

The problem with several exceptions is that they become harder to handle. It is very easy for the developer to just look at the first one and decide base on that which strategy to choose. To fix this the first exception in a chain must be modified, or in its placed added an exception, so that the first exception's:

- *info⟨Severity⟩* is the maximum of all *info⟨Severity⟩*s in the chain
- *info⟨ErrorType⟩* is the union of all *info⟨ErrorType⟩*s in the chain
- *info⟨Inputs⟩* is the union of all *info⟨Inputs⟩*s in the chain
- *info⟨Subsystem⟩* is the union of all *info⟨Subsystem⟩*s in the chain

The implementation may well hide this construction in access functions to the exception, and lazily construct such maximums and unions.

The user interface layer (and any logging layers) must of course loop through all the exceptions, but such handlers should be very few in number and so added effort can be spent in verifying their correctness.

4.11 Warnings *info⟨Severity⟩*

Often a program may decide on strategy *str⟨Ignore⟩* or even *str⟨ChangeInput⟩* [35, section 3.1]. A very common example is the web browser, which will silently try its best on the face of invalid HTML [36]. Other typical examples include a document-server that converts XML to PDF: it may encounter some unknown formatting constructs or unavailable fonts - it should probably do it's best to produce output, while warning that the output may not be what was expected; or a compiler that flags problematic but valid constructs.

A user facing warnings has basically the same choices as one faced with errors, and so all of the reasoning above holds—warnings should have the same information as exceptions. These results in a the need to specify a *info⟨Severity⟩*for exceptions.

5 Conclusions, implementation notes and future research topics

The information requirements derived in the previous section are pulled together to a design for a generic exception class. Examples of how the class with its information can be used in fully generic exception handling is shown. The implementation in *sys*⟨*CCF*⟩is discussed, as are possible difficulties in implementation.

5.1 So what should exceptions look like, then

Combining all of the requirements of previous sections, I propose that exception classes should provide the fields given in Table 1:

***info*⟨*ErrorType*⟩** a bit-field of {INPUT, RETRIABLE, BUG, ENVIRONMENT, USERENVIRONMENT, REMOTEENVIRONMENT, ADMINENVIRONMENT}. Indication of what strategies are applicable
***info*⟨*Severity*⟩:** one of {INFO, WARNING, ERROR, CORRUPT}. Did the operation succeed, can you continue.
***info*⟨*UserMessage*⟩:** string. User-friendly error message that states what is wrong and what can be done about it
***info*⟨*AdministratorMessage*⟩:** string. The same as above, but in language and concepts the user may not understand but the administrator does, or for an error the user cannot do anything about
***info*⟨*ProgrammerMessage*⟩:** string. Any internal details of the error that may help in debugging
***info*⟨*Inputs*⟩:** array of [field ref, field position, value, direct-flag]. The input field(s) that are the cause of the error, position within field and value of field
***info*⟨*StackTrace*⟩:** list of [function, arguments, source filename, line no, version].
***info*⟨*ErrorCode*⟩:** hierarchical namespaced symbol. Exactly what error was detected
***info*⟨*Subsystem*⟩:** tuple [namespaced symbol, string]. In which subsystem did the error occur
***info*⟨*InnerException*⟩:** Exception. What caused this exception in the rare cases when we *do* need to map exceptions
***info*⟨*NextException*⟩:** Exception. What other things should be fixed

Table 1. Exception class members

It may well be naive to enumerate all the error-types here. It might be better to organize them into a hierarchical symbol namespace as well. Objects of this class are on the large side, but not significantly larger than Exception objects in Java and .Net.

There are some changes in comparison to Java and .Net exceptions.

- I feel that the simple change of splitting error messages into three categories is an important one, and it provided us with good results when developing *sys*⟨*CCF*⟩. It might not be unthinkable to only use two: user-friendly and low-level.
- The severity is really only useful if we allow for warnings, otherwise *info*⟨*Corrupt*⟩could be done differently (e.g., as one of the types)

– Requiring exceptions to identify *info⟨Inputs⟩*s is a heavy challenge with substantial benefits. It allows for much better user-interfaces and mapping of one module's *exc⟨InvalidInput⟩* to another's *exc⟨Bug⟩* automatically and reliably. We did implement it in *sys⟨CCF⟩*, and the functionality is normally supported (at least in part) in database interfaces and compilers, so it is not unheard of.
– I do not have experience of the *info⟨Subsystem⟩*field, but it does seem necessary for several real use-cases.
– Several exceptions from one action.

5.2 Should we allow for derivation?

It is not quite clear whether this single exception class would be sufficient, or whether to allow for extension. Already Stroustrup claimed that exception classes can be used to group related exceptions into hierarchies like `Network_err` [37, p. 358–361]. This sounds like a nice idea, but I have seen *no real world benefit from it*. The aspects of exceptions I have identified above, that are needed for deciding what handling strategy to use, have nothing to do with such groupings—the aspects crosscut different *functional* failures. Concrete examples of useless exception hierarchies abound in Java: there is no commonality to `IOException`'s that would allow the program to handle them intelligently together, and most of the time new exceptions are defined it is to satisfy the idea of exception declarations.

There is a need for hierarchical, namespaced symbols in exceptions for the *info⟨ErrorCode⟩*. This should not be confused by a need for the actual exception to be extensible. In fact there are a couple of good arguments against:

– to make generic exception handling possible, not knowing the exact type of the exception thrown should not deny access to any information (this may rule out new members from derived classes)
– it is beneficial to be able to transport exceptions between language environments - so all the information should be language-neutral
– derived exception classes are problematic in distributed computing since the client may not have the implementation available for the exception it receives over the wire

On the other hand there might be a need for additional machine-readable information for *str⟨ChangeState⟩*, and so for additional members. This sentiment is shared by Siedersleben [12].

Of the 130 exception classes most closely derived from `Throwable` in Java [23] 21 extend the base class in some way (adding attributes). 13 of those do it to support *info⟨Inputs⟩*, 8 of these with *typed* arguments: it is assumed that the catcher will need to have a machine-readable *info⟨Inputs⟩*. Of the remaining 8, 5 encapsulate foreign exceptions (SQL, XA, Corba) and 3 have some specific attributes. All in all it seems that extension is not needed often, but that there may be a real need for it in some cases.

There is one clear axiom to exception class hierarchies though: **Any exception attributes that are needed for generic handling need to exist in the base-most exception class.**

5.3 SIDEBAR: Error logs, error dumps and security

The mechanism proposed assumes that we want to bring up all the error information to the interactive user. This may not always be suitable.

On the web the user may simply go away to another service provider. They are very loosely connected to the system and do not have good incentives to report errors. In such systems the same information that is given to the user should also be logged by the program to a log file which can be periodically checked so that the system quality can be improved.

The information proposed may contain sensitive data: local file names, database connection details etc. It is not suitable to give these to untrusted subsystems or users. In these cases only *info*⟨*UserMessage*⟩should be propagated, the rest logged to a local store with an id, and that id (and the store identifier) propagated up. If the user decides to report the error, the store identifier and error id can be used to look up (hopefully automatically) the details.

Sometimes you may want to dump a large amount of *info*⟨*Program*⟩information, e.g., with *exc*⟨*Corrupt*⟩faults (Bruce Lindsay from IBM tells of the DB2 database dumping huge amounts of internal state when suspecting a bug [17]). It is not practical to propagate that. Again, using a store name and dump id the dump can be later identified from the error report from the user.

This *does not* mean that errors should normally be logged to a side channel. Even if we assume that the program cannot take corrective action but the error is propagated all the way to the user, it is very important that the user can easily report all the details of the error. Much effort is currently spent in trying to correlate the different error logs of different subsystems when users experience errors or undesired behaviour [38].

5.4 Handling the exceptions

The whole idea of the proposed framework is to allow exceptions to be *handled* in as few places as possible. (On the other hand *detection* may occur all over the system). The most important handling point is the user interface event loop (this sentiment is shared by C#'s chief architect Anders Hejlsberg [13]), for processes without user interfaces the corresponding scope is the incoming message loop. The exception handler should attach the error message to a suitable area of the user interface, relating it to the corresponding user-interface elements via *info*⟨*Inputs*⟩. The message shown can be selected as shown in Figure 2 (the actual texts are meant to be illustrative, not realistic, and the code assumes INNEREXCEPTIONs to be mapped to NEXTEXCEPTION):

Additionally the handler would probably select the location of the message as well as it's depiction (icons, colours) based on *info*⟨*Severity*⟩. Note how simple the handler is.

Another place where you may want to place a handler is in the point where you access a major separate subsystem. The handler will look like (the subsystem reset is again not realistic, but illustrates the idea) the one in Figure 3.

You may additionally have the odd handler at *ImmediateCaller* where you need to use *str*⟨*ChangeState*⟩.

To get the system to this point of simplicity, all the fault detection code must put together exceptions as detailed above. In code under your control you must provide primitives that make it easy to attach input references, localization of *info*⟨*UserMessage*⟩and for throwing specific kinds of errors (e.g., *exc*⟨*Bug*⟩). For code not under your control you will need a Facade through which all access to such libraries occurs. This is not atypical of accessing third-party libraries in real systems anyway.

```
catch(Exception firstExc) {
  for (Exc=firstExc; Exc; Exc=Exc.NEXTEXCEPTION) {
    switch(Exc.ERRORTYPE) {
    case INPUT:
       if ( one of the fields in Exc.INPUT appears
          in the user interface and has direct-flag set )
       {
          message=Exc.USERMSG
       } else {
          message='The program encountered a defect.
                   Contact product support.'
       }
    case USERENVIRONMENT, ENVIRONMENT:
       message=Exc.USERMSG
    case BUG:
       message='The program encountered a defect.
                Contact product support.'
    case RETRIABLE:
       message='Your request could not be carried
                out due to a temporary shortage of resources.
                Please try again. If the error persists, contact
                your system administrator.'
    default:
       message='Contact your system administrator'
    }
    if (Exc.SEVERITY==CORRUPT) {
       message.prepend 'The system has detected an internal
                        inconsistency and it is not capable of
                        continuing safe operation. Contact your
                        system administrator immediately.'
    }

    attach(message, Exc.INPUT)
  }
}
```

Fig. 2. User interface exception handler

5.5 Implementation in *sys*⟨*CCF*⟩

sys⟨*CCF*⟩is based on the message-passing paradigm: the contents of web forms are transformed into XML-based messages (which map 1–1 to the creation of the forms), the UI layer communicates with the BL layer with messages, as do different modules in the UI layer. All system state is kept in the database, so backward error recovery can always be accomplished by rolling back the database transaction. Implementation languages include Visual Basic 6.0, C++, PL/SQL and Java. Support for the error handling paradigm is available in all language environments.

Of the requirements presented here the following are supported:

- *info*⟨*ErrorType*⟩
- *info*⟨*Severity*⟩
- *info*⟨*UserMessage*⟩
- *info*⟨*ProgrammerMessage*⟩

```
catch(Exception firstExc) {
  if (! time_limit_exceeded() ) {
      switch(firstExc.ERRORTYPE) {
      case RETRIABLE:
         goto retry
      default:
         if ( firstExc.SEVERITY != CORRUPT  &&
              can_reset(firstExc.SUBSYSTEM)) {
            reset firstExc.SUBSYSTEM
            goto retry
         }
  } else {
    for (Exc=firstExc; Exc; Exc=Exc.NEXTEXCEPTION) {
      switch(Exc.ERRORTYPE) {
      case ENVIRONMENT, ADMINENVIRONMENT:
         if ( subsystem_is_remote() ) {
            Exc.ERRORTYPE=REMOTEENVIRONMENT
         }
    }
  }
  rethrow firstExc
}
```

Fig. 3. Major subsystem interface exception handler

- *info*⟨*Inputs*⟩
- *info*⟨*StackTrace*⟩
- *info*⟨*ErrorCode*⟩
- *info*⟨*NextException*⟩
- Maximum of *info*⟨*Severity*⟩and union of *info*⟨*ErrorType*⟩for chained exceptions.

Notable omissions are *info*⟨*Subsystem*⟩ and *info*⟨*InnerException*⟩. *sys*⟨*CCF*⟩ does not include any internally developed separate run-time subsystems. It was assumed that any restarts of the web server or database will be managed by the system administrator, not automated error handling. *info*⟨*InnerException*⟩ were originally supported, but had so few uses (due to not mapping exceptions) that the few cases where changed to use *info*⟨*NextException*⟩(from the user's point of view the difference is very small). *info*⟨*AdministratorMessage*⟩and *info*⟨*ProgrammerMessage*⟩were merged into one field.

Input references, *info*⟨*Inputs*⟩, are accomplished by always using messages as input. Even a module's internal functions get references to parts of the messages instead of plain programming-language datatypes. (This is not strictly the case with database access and PL/SQL code. There input mapping is restricted to giving the row of array binds and the upper layer using database constraint names to map to bind variables. The bind variables themselves have again a 1–1 mapping to messages fields, maintained by the database Facade.)

Implementation-wise, the language-level exceptions were not used to transport the information, as they were not sufficient in the main implementation languages, Visual Basic and PL/SQL. Instead subroutines had access to a *response message object* in which they put error information. A special language-level exception was then used to indicate that the operation failed and that details were put in the response message. The mechanism is reminiscent of Unix/C `errno`,

but with no chance of losing information. The response message paradigm supported also warnings and informational notes from any level of code. There was some programmer-discipline needed to transfer errors from lower-level responses to upper-level ones, but this soon became idiomatic (and only needed a single function call).

The major deficiency in the framework implementation was the need to manually map input references if data was to be manipulated. This was supposed to be handled in the subroutine that called another module, after getting back an error message. Sometimes the mappings were not performed by developers or were not correct. The deficiency had surprisingly little actual effects: normally user input was just re-ordered, re-structured or split into different messages. These operations were reference-preserving. Another concrete problem was that Visual Basic necessitated duplication of module-level error-handlers, since the language does not support inheritance of implementation.

The resulting system's error messages were very successful. Consistent visual indicators allowed users to quickly see whether operations succeeded or not. Errors that required changing input were attached to the corresponding input fields allowing the user to easily find what to correct even with large forms. All error information was brought to the user's browser, where it could be forwarded to local support (subsuming from the user's point of view the $ent\langle SystemAdministrator\rangle$and $ent\langle Programmer\rangle$). Most of the time system administrator's were able to correct server-side problems without having to consult external logs and developers tended to have quite a good idea of where to start debugging if there was a program defect (additionally, the whole input message was available to the developer). The generic exception handlers were basic implementations of the handlers in the previous section.

5.6 Notes on implementing $info\langle Inputs\rangle$

The framework proposed is mostly quite easy to implement (Although $info\langle StackTrace\rangle$can be problematic in older environments with optimized programs, e.g., if no stack frames are generated for leaf functions). The only real problem implementation-wise is $info\langle Inputs\rangle$. It would seem to require a lot of effort from the programmer, language support, or an unusual datatyping model. There are, however, several implementations of similar constructs.

Perl's Taint mode tracks a single-bit flag from external inputs, propagating the flag through data-flow [39, p. 355–636]. It has simplistic rules (only assignments to expressions propagate the flag, not if/switch), but the mechanism does work over arbitrary function calls.

Parsers and compilers keep track of what tokens come from which file, row and column of input. This includes programming language compilers, query parsers and XML processors. These programs of course receive a highly structured input: a file or text buffer, where input can always be identified in the same way. It should be noted though that propagating the input file position throughout compiler passes is still a non-trivial problem, but is deemed to be necessary by the (good) compiler community.

There is thirty-years of literature on tracking the confidentiality of information in programs, *secure information flow* [40]. For keeping track of input references tracking only explicit information flow would be suitable, with probably additional primitives for semantics-preserving mapping of data (such as character-set mapping, 1–1 code lookups etc.) which will specify the direct/indirect flag. The information flow theory has been unsuccessful in gaining acceptance with

practitioners despite implementations [41]. Careful attention must be paid to that failure and its reasons if similar techniques are to be used.

The *sys*⟨*CCF*⟩experience shows that message-passing systems are highly suitable for *info*⟨*Inputs*⟩. The message-passing paradigm has other favorable qualities (freedom of implementation language, flexible run-time structure, batching support), but has drawbacks as well (it is harder to see what the control flow is, which modules are invoked and the need to either map message structures to programming-language structures to allow for static typing or to use run-time type checking).

Finally, tracking references to the user-interface throughout information flow need not be prohibitively expensive. All the references will relate directly to input by the user, which is always constrained to a fairly small size by the fact that it has to be keyed in by a human and shown on a single computer screen (file input is a special case, and it can be tracked easily on a filename-line-column basis). So the amount of overhead is not extreme in most cases. (This is obviously not true for certain classes of systems, such as numerical computation or data mining.)

5.7 Conclusions and future work

This proposal suggests several fundamental changes to the way error information is constructed and errors handled in most literature. There is good reason for this, though, as the current way of thinking definitely isn't producing good results. The proposal has been evaluated with a real-life system with good results.

There exist earlier error-handling proposals that have died because they were too hard to use. The X.500 error objects died together with the whole X.500 directory service (in favor of things like LDAP). The proposed framework is supposed to make it *easier* to handle errors while getting better results: both throwing exceptions and handling them are easier. Throwing since the programmer knows how to do it, can rely on all of his effort reaping benefits since no information is lost, and being able to distinguish between user-friendly and programmer-friendly information. Handling since it can be almost fully generic and so appears in only a few places.

Several aspects of the framework need to refined by other implementations: are derived exception classes really necessary and when, how do we know when to add upper-level (contextual) messages to errors, and what error types are needed. Efficient implementation of the input data references in different environments is a serious research problem.

Although the answers are not fully specified yet, the experience with a real implementation suggests that the framework *is* usable. It does not have to be uniformly supported by all libraries, subsystems and applications either: the principles can be used to get good error handling and error messages out of particular parts of a system as well.

Maybe most importantly, this paper should serve as a starting point for an ongoing discussion of how errors should be handled in typical real-world programs and projects. Such discussion has been sorely lacking.

References

1. Horning, J.J.: What the compiler should tell the user. In: Compiler Construction, An Advanced Course, 2nd ed., London, UK, Springer-Verlag (1976) 525–548
2. Brown, P.J.: Error messages: the neglected area of the man/machine interface. Commun. ACM **26** (1983) 246–249

3. Lewis, C., Norman, D.A.: Designing for error. (1987) 627–638
4. Cooper, A.: About Face. The Essentials of User Interface Design. IDG Books Worldwide, Inc., Foster City, CA, USA (1995)
5. Edward Grossman (ed.): ACM Queue. **2** (2004)
6. Goodenough, J.B.: Exception handling: issues and a proposed notation. Commun. ACM **18** (1975) 683–696
7. Cristian, F.: Exception handling and software fault tolerance. IEEE Transactions on Computers **31** (1982)
8. Kiniry, J.R.: Exceptions in java and eiffel: Two extremes in exception design and application. In Romanovsky, A., Dony, C., Knudsen, J.L., Tripathi, A., eds.: Proceedings of the ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems. Number 03-028 in Computer Science & Engineering Technical Reports, University of Minnesota (2003)
9. Torres-Pomales, W.: Software Fault Tolerance: A Tutorial. Technical Report NASA/TM-2000-210616 (2000)
10. Sutter, H.: More exception-safe generic containers. C++ Report **9** (1997)
11. Martin D. Carroll, M.A.E.: Designing and Coding Reusable C++. Addison-Wesley Publishing Company, Inc. (1995)
12. Siedersleben, J.: Errors and exceptions. rights and responsibilities. In Romanovsky, A., Dony, C., Knudsen, J.L., Tripathi, A., eds.: Proceedings of the ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems. Number 03-028 in Computer Science & Engineering Technical Reports, University of Minnesota (2003)
13. Venners, B.: The Trouble with Checked Exceptions, A Conversation with Anders Hejlsberg, Part II (2003). `http://www.artima.com/intv/handcuffs2.html`
14. Howell, C., Mularz, D.: Exception handling in large ada systems. In: WADAS '91: Proceedings of the eighth annual Washington Ada symposium & summer SIGAda meeting on Ada, ACM Press (1991) 90–101
15. Grey, J.: Why Do Computers Stop and What Can Be Done About It? Technical Report 85.7 (1985)
16. Molich, R., Nielsen, J.: Improving a human-computer dialogue. Commun. ACM **33** (1990) 338–348
17. Bourne, S.: A conversation with Bruce Lindsay. ACM Queue **2** (2004) 22–33
18. Hall, E.A.: IETF. RFC 4155. the application/mbox media type (2005)
19. Gamma, E., Helm, R., Johnson, R., Vissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman, Inc., Reading, Massachusetts, USA (1995)
20. Kiczales, G.: Aspect-oriented programming. ACM Comput. Surv. **28** (1996) 154
21. Microsoft: Platform sdk: Debugging and error handling: System error codes. online (2005). `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/system_error_codes.asp`
22. Raymond, E.S.: The Art of Unix Programming. Addison-Wesley Longman Inc., Reading, Massachusetts, USA (2003)
23. Sun Microsystems, Inc: JavaTM 2 SDK, Standard Edition Documentation. online (2003). `http://java.sun.com/j2se/1.4.2/docs/`
24. Liberty, J.: .net exceptions make the transition from traditional visual basic error handling to the object-oriented model in .net. MSDN Magazine (2002)
25. Deitel, H., Deitel, P.: C++ How to Program, 2nd edition. Prentice Hall, New Jersey, USA (1997)
26. Gehani, N.: Ada, an advanced introduction. Prentice Hall, New Jersey, USA (1983)
27. Arnold, K., Gosling, J.: The Java Programming Language. Addison-Wesley Longman Inc., Reading, Massachusetts, USA (1996)
28. Lakos, J.: Large-Scale C++ Software Design. Addison-Wesley Longman, Inc., Reading, Massachusetts, USA (1996)
29. Spolsky, J.: Joel on software: Exceptions. online (2003). `http://www.joelonsoftware.com/items/2003/10/13.html`

30. Chen, R.: The old new thing: Cleaner, more elegant, and wrong. online (2004). http://blogs.msdn.com/oldnewthing/archive/2004/04/22/118161.aspx
31. Kernighan, B.W., Pike, R.: The Practice of Programming. Addison-Wesley Longman Inc., Reading, Massachusetts, USA (1999)
32. Harrison, R.: Symbian OS C++ for Mobile Phones. John Wiley & Sons Ltd, West Sussex, England (2003)
33. Thain, D., Livny, M.: Error Scope on a Computational Grid: Theory and Practice. In: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 '02), IEEE Computer Society (2002) 199–209
34. Candea, G., Brown, A.B., Fox, A., Patterson, D.: Recovery-oriented computing: Building multitier dependability. Computer **37** (2004) 60–67
35. Wielemaker, J.: An overview of the swi-prolog programming environment. In: Proceedings of the 13th International Workshop on Logic Programming Environments. Volume CW371 of Report., Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee (Belgium) (2003)
36. Horning, J.J.: Risks of lenient parsing. online (2005). http://horning.blogspot.com/2005/01/risks-of-lenient-parsing.html
37. Stroustrup, B.: C++ Programming Language, 3rd edition. Addison-Wesley Publishing Company, Reading, Massachusetts, USA (1997)
38. Maglio, P.P., Kandogan, E.: Error messages: what's the problem? Queue **2** (2004) 50–55
39. Wall, L., Christiansen, T., Schwartz, R.L.: Programming Perl. 2nd edn. O'Reilly and Associates, Inc., Sebastopol, CA, USA (1996)
40. Sabelfeld, A., Myers, A.C.: Language-Based Information-Flow Security. IEEE Journal on Selected Areas In Communications **21** (2003)
41. Zdancewic, S.: Challenges for information-flow security. In: Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04). (2004)