

Extended Exceptions for Contingencies and their Implications for the Engineering Process

Thorsten van Ellen
BTC AG
Business Unit Software Solutions
26121 Oldenburg, Germany
thorsten.van.ellen@btc-ag.com

Wilhelm Hasselbring
University of Kiel
Software Engineering Group
24118 Kiel, Germany
wha@informatik.uni-kiel.de

ABSTRACT

We observed a general problem of sequential programs, which often results in design and programming errors in industrial software engineering projects, and propose a solution approach. Telephone lines may be busy, banking accounts may be overdrawn and disks may be full. These things happen in the real world, causing the disruption and non-fulfillment of an expected service. Ignoring these problems leads to violations of the postconditions of the caller that depends on the service. The conditions are exactly known and cannot always be avoided, but measures could be taken afterwards. A good program should handle them as part of the specification. As such they are not specification violations and should not be regarded as errors. Unfortunately, they usually can or shall not be handled immediately within the direct caller, e.g., for information hiding reasons. The problem is similar to the problem of error code handling and handling them with exception mechanisms seems reasonable, but the problem is even more complex. These situations must not terminate the system suddenly, because that also violates postconditions. Consequently, exceptions for these situations must be distinguished from exceptions for errors and are worth handling separately. Therefore, we introduce the new concept *contingency* for such situations. Since the conditions are defined, they are candidates for forward recovery, but conventional exception mechanisms are not appropriate for that purpose. Appropriate mechanisms are presented in this paper. A systematic inspection and handling of contingencies with these mechanisms before runtime can diagnose and avoid subsets of specification violations effectively. This implies some consequences for the engineering process.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features—*control structures, Procedures, functions, and subroutines*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WEH '08, November 14, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-60558-229-0 ...\$5.00.

General Terms

Reliability, Languages

Keywords

contingency, exception, forward recovery, resumption, reliability

1. INTRODUCTION

Telephone lines may be busy, banking accounts may be overdrawn and disks may be full. These things happen in the real world and cause the disruption and non-fulfillment of an expected service. Ignoring them leads to violations of the postconditions of the caller, since the caller depends on the service. You know exactly what happened if you are familiar with that implementation level. The conditions are exactly known and cannot always be avoided, but measures could be taken afterwards. A good program should handle them as part the specification. As such they are not specification violations and should not be regarded as errors.

A common recommendation is to use exceptions only for specification violations [4], to declare them explicitly within the interface [2, 8], redeclare them within the interface of the caller if they have not been handled [2], and to adjust their abstraction to the current interface abstraction [11]. Furthermore, the termination model for handling exceptions is preferred [3] over the challenged resumption model [3]. All these recommendations are challenged in this work.

Section 2 explains some terms and introduces the term contingency for these situations. Section 3 discusses contingencies extensively and shows that they are very difficult to handle and handling them within conventional programming languages with exceptions is reasonable. Section 4 presents the objectives of this work. Section 5 clarifies important properties of contingencies. Section 6 pinpoints some significant deficiencies of conventional exception handling mechanisms for handling contingencies. Accordingly, Section 7 proposes extended exception mechanisms. Finally, Section 8 shows some implications of systematically handling contingencies for the engineering process.

2. TERMS

This section describes some essential terms.

Often, errors are defined as states, e.g., by [1], for example, if a traffic light shows red and green at the same time, but such a definition does not include erroneous state *transitions*, e.g., if a traffic light changes directly from green to red where

no erroneous state is involved. Therefore, the term situation is used instead of state here and is defined as follows:

Definition 1. Situation: A situation is a sequence of n states with $n > 0$.

This definition includes states and transitions. A situation can be a sequence consisting of a single state and for the sake of simplicity situations can be illustrated as states.

Definition 2. Specification: A specification is a complete, consistent description of all situations that are allowed for a system.

Example 1. A specification of a function can be composed of a pre- and postcondition. The precondition for a traffic light transition function describes the valid states:

$$pre : state = green \vee state = yellow \vee state = red,$$

the postcondition describes the valid transitions, starting from the valid states:

$$\begin{aligned} post : & (state = green \wedge state' = yellow) \\ & \vee (state = yellow \wedge state' = red) \\ & \vee (state = red \wedge state' = green). \end{aligned}$$

The length of the regarded sequences of states (situations) is arbitrarily determined by the specification, e.g., the length of the regarded situations of the postcondition within example 1 is 2.

A specification is not only a mathematical set of states or situations, because it has a structure, conditions and names. A specification of a non-trivial system, e.g., with the specification languages B or Z, is usually modular, similar to program code, and usually contains modules or functions building upon or calling each other. A specification contradicts itself and is not consistent if it allows a situation as a normal result within a postcondition of a callee module, e.g., result `LineBusy` of module `sendFax` within example 3, and forbids the same simultaneous situation at another calling module, e.g., by preconditions of the following operation.

Definition 3. Error: An error is a situation, the conditions of which contradict the specification.

In other words, an error is always the same as a specification violation, i.e., if the situation does not comply with the preconditions or postconditions, as [4] states, e.g., if a traffic light changes directly from green to red.

The new term contingency (coined by [8]) is defined (differing from [8]) as follows:

Definition 4. Contingency: A contingency is a situation that is described within the specification of a module, and represents a module result where the task or function, which calling modules depend on, was not performed.

Contingencies are exactly described within the specification and no errors. They are results of modules indicating that the module could or should not fulfill its usual work. They are unavoidable or intentional behaviors. Usually contingencies are directly perceived as if they were errors, e.g., violations of preconditions [4], but that is not a correct perception.

Example 2. A method `allocateMemory` might expect a parameter `requiredAmount` of type `integer` that should be in the range of positive values. If the invoker passes negative values, it violates the precondition. Such a situation is an error, but if the invoker passes positive values and the condition `availableMemory < requiredAmount` evaluates to true, the method returns `OutOfMemory` as specified and both the invoker and the method are not doing anything wrong, nevertheless both cannot perform their usual work. `OutOfMemory` can be formally defined, often cannot be avoided and should be specified. Such a situation is a contingency.

Example 3. If a fax should be sent via a modem controlled by software, the telephone line can be busy (`LineBusy`) or the number might be gone (`SIP_ERROR_410_NumberGone`). Usually, both are not avoidable, not even by changing the program code of the software. Nobody can buy and control the whole telephone network and even a simple technical solution like a dedicated line might be too expensive for a small application.

At the time the contingency is detected no damage occurred, yet, the system still behaves as specified. From the perspective of the dependant module, contingencies are work refusals whose potential appearance is known in advance, independently of the abstraction level they represent. If no specific additional measures are taken, the expectations of dependant modules that do not regard the possible work refusal will not be fulfilled and they cannot reach their postconditions. As a result, an error will appear. Only if the contingency is ignored and not handled specifically, will damage inevitably occur. As will be shown in the discussion in Section 3, there is no practicable solution to regard every contingency at every call.

Contingencies differ from normal situations in that normal situations do not represent work refusals and without additional specific measures do not necessarily run into errors.

3. DISCUSSION

This section discusses language constructs to communicate and handle contingencies. You might think that there are only few such contingencies, but quite the contrary is true if you are looking for them.

Example 4. A simple routine like `openFile` might have up to 20 or more of them: `Drive/Dir/FileNotFound/Locked/NameInvalid`, `DiskNotInDrive`, `DiskNotFormatted`, `DiskFull`, `EndOfFile`, `NoAvailableFileHandles`, `NetworkDisconnected` etc.

Usually, they are far from being systematically and completely documented or even declared within the interface, but they are nevertheless always present. Such situations are omnipresent. Their number is finite as is the implementation, but Oracle database routines and other complex systems might return thousands of them (see Section 5).

Unfortunately, mostly contingencies can or should not immediately be handled within the direct caller as easily as other normal specified situations. The reason is that not all required information is available or not all required components for measures are accessible, because the required elements are kept encapsulated in other call levels. For example, the graphical user interface is usually not available

within lower levels to inform or ask the user. Even if immediate handling were possible, full disks and other contingencies may occur at several places and their resolution should not be implemented redundantly at several places. If the contingency is not solved immediately, it must be communicated to the dependant caller by one means or another.

3.1 Avoiding Exceptions?

How should these contingencies be communicated to the caller? Should contingencies be communicated to the caller not as exceptions as recommended by [4], rather as special values declared in the interface or its documentation? That is reasonable if they do not disclose implementation details, but declare about 20 or more contingencies in the interface of the caller of `openFile` because the direct caller cannot handle it immediately? And additionally, do it in the interface of the caller of the caller, and so on, along the whole caller chain until they are handled? The contingencies of `openFile` are not the only contingencies that might occur in the call graph of the call chain. All unhandled contingencies would occur and accumulate within all interfaces of all callers of the call graph until they are handled. This accumulation of unhandled contingencies is unavoidable. At the top level routine, e.g., `main`, all unhandled contingencies of the whole system accumulate. Manually declaring them all explicitly is not feasible.

Should these implementation details be disclosed within the interfaces on all levels? If, one day, someone decides to change the implementation and the corresponding contingencies, should all intermediary interfaces be changed? This leads to an intolerable maintainability problem. Should the maintainability problem be solved by abstracting the implementation details of contingencies as usually recommended, e.g., abstracting the implementation details `FileLocked` and `TableLocked` of two different implementations by `ResourceLocked`? This might be even more work. Furthermore, the primary argument against abstracting the implementation details is that indispensable implementation information gets lost along with options for forward recovery, i.e., a specific handling, repairing and finally resumption after successful repairing. The different contingencies can no longer be distinguished, and handled separately and specifically if they are abstracted. Therefore, implementation-specific measures cannot be taken, e.g., measures for `FileLocked` probably operate on the operating system and measures for `TableLocked` operate on the database. The valuable implementation information should never be lost [7].

Should an abstract return value class or exception class on the interface be declared for several contingencies and more distinguishable information be embedded at runtime (nested or derived objects or exceptions) like it is usually done in Java? Then you would never know before runtime what really can happen. If a method throws an `IOException`, is it a `DiskFull`, `FileReadOnly` or something else? The complete information of the interface must be available at development time, before runtime! If the information is not available within the interface at development time, but at runtime, it is too late to develop a handler.

3.2 Worse than Error Code Handling?

This problem is similar to the problem of error code handling that has been found impracticable and has been replaced by exception handling. Exceptions are surely a part

of the solution, but the problem is even more complex than error code handling. Nobody is amazed if an error terminates the system, but everybody is annoyed if the line is busy or the disk is full and additionally the system crashes.

If an error has already occurred, it is usually reasonable to terminate the system with an exception, but if the system is still in defined circumstances of the specification, sudden termination is one of the specification violations that should be avoided, therefore, not a good option. Forward recovery is the best option. At least in these cases, all corresponding exceptions must be distinguished and handled separately from exceptions representing errors.

3.3 Problem Statement

There seems to be no single approach regarding all of our discussed arguments. Nobody seems to recognize the whole combination of problems, at least one of the arguments about accumulation, implementation dependence and abstraction and their consequences is overlooked. [2] describes a concept with declaration of exceptions within the interface and compile time checking, similar to Java's checked exceptions, disregarding accumulation and implementation dependence. Java has checked exceptions that also disregard accumulation, but usually are solved by abstraction. [4] only allows the use of exceptions for errors, all other situations must be declared within the interface. [11] recommends to abstract. We have never seen real code that consequently solves things. Until now, it has not been possible to determine all contingencies of all implementation levels at development time with conventional languages. How can they ever completely be handled reliably? They cannot at the first run! Usually, they simply crash at the user.

We feel like contingencies are a systematic source of errors and might encompass much more than teething problems, e.g., diverse inconsistencies, missing requirements and potential catastrophes. Contingencies are a general and systematic problem of sequential programs that needs to be solved. We believe, this problem can be solved better and many errors can be avoided systematically with appropriate language mechanisms. Vice versa, as long as this problem is not solved, this source of errors will continue to exist.

In our perception so far, there exists no satisfying solution to declare/document contingencies at development time, communicate them at runtime and handle them without specification violation. They are not even recognized separately from errors and normal situations. Nevertheless, they are omnipresent and are worth regarding and handling separately. Therefore, we try to establish the new term contingency within Section 2 to separate them from errors and normal situations. Surprisingly, separating contingencies opens a new perspective, providing new understandings and a plethora of innovative topics that makes this paper a bit crowded, which is also reflected in the objectives in Section 4. Once you take this perspective, some commonly made observations can be explained, for example, one need for iterations in the engineering process. The space of errors is divided with the term contingency to make the situations that are treatable, accessible for forward recovery. Conventional exception handling is not able to make this distinction and therefore appropriate handling and resumption is not possible. We intend to contribute to the discourse of errors, exceptions, handling and resumption.

4. OBJECTIVES

Contingencies must be handled in the sense of forward recovery if following errors are to be avoided. The objectives of our work and this paper are to

- distinguish contingencies from normal situations and errors.
- enable a simple communication of errors and contingencies between the call levels.
- determine contingencies automatically at development time and always get a current and complete documentation of all available contingencies of all levels.
- determine the source of some potential errors directly (not via symptoms) and before runtime, thereby reducing time and effort to analyse errors and determine the sources.
- enable forward recovery. This should generally be possible, i.e., rescue situations without database transaction mechanisms, e.g., within cross-system interfaces, and even for side effects like physical processes.
- keep information hiding as far as possible.
- avoid complications of conventional interfaces, cumbersome cleanups and partial repetitions.
- enable overriding any handling by outer context with broader knowledge and component access.
- reduce redundant code, the total amount of code and code complexity needed for communicating, handling, repairing and resuming.
- show the need of special iterations in the engineering process for gathering requirements for contingencies.

Only sequential, imperative programs should be regarded, even though we assume our approach might also be reasonable for non-sequential programs and our thoughts might be translated to event-based systems and other programming paradigms.

5. PROPERTIES OF CONTINGENCIES

The following section presents significant properties of contingencies and describes why handling of contingencies within conventional programming languages with exception handling is reasonable.

5.1 Contingencies are Omnipresent

As shown in the discussion, contingencies can be very numerous alone within one single function. They are littered over many functions within the whole system and all levels.

The following example from the database domain illustrates how numerous contingencies actually are within everyday life. Presumably, contingencies are not less numerous within operating systems or other complex environments like enterprise resource planning (ERP) systems, e.g., SAP, only less documented and apparent. This shows how important it is to handle these situations explicitly.

Example 5. Oracle maintains documentation [5] of all problem messages of the Oracle database. It encompasses over 2000 pages, each with several messages. Hence, it documents several thousands of entries and mostly contains detailed and specific (not abstracted) and therefore helpful hints for each single known situation that can occur at runtime. Not all of these entries describe situations where the database is within an unknown or undefined state (error) that must be repaired, and prevents the database from operating correctly. Instead, substantial amounts of them are numerous contingencies that are successfully recognized, intercepted and communicated at runtime and documented (quasi specified) at development time of the caller.

5.2 Contingencies are Unavoidable and Better Treatable than Errors

Function results that represent contingencies (refusals to work) will usually be avoided intuitively at system development time if possible. The contingencies that remain are unavoidable, but on the other hand their circumstances are exactly known and their conditions are defined. Conditions of errors contradict the specification and are not exactly described within the specification. Handling contingencies is therefore easier than handling errors.

5.3 Contingencies Disclose Implementation Details and Must Not Be Abstracted

When passed to the callers in the caller hierarchy, contingencies disclose implementation details, not immediately, but usually after a few call levels when a class or group of classes representing the same abstraction is left.

If different contingencies are mapped to one abstract contingency, e.g., `OutOfMemory` and `DiskFull` to `OutOfResource`, the different conditions of the different contingencies cannot be distinguished anymore and specific handling is no longer possible. A specific handling for `OutOfMemory` like swapping is neither applicable for `DiskFull` nor for the abstraction of both `OutOfResource`.

Our unconventional recommendation is to *not* abstract contingencies for information hiding reasons, because it would map the contingency of the current implementation and the potential contingencies of future implementations to one abstracted situation for which a current handler might not be appropriate. The implementation details might be indispensable [7] for repairing at runtime. Since it is mandatory to handle contingencies specifically to avoid subsequent errors, it must be assured that they remain unambiguous and are *not* abstracted.

6. DEFICIENCIES OF CONVENTIONAL EXCEPTION MECHANISMS

This section outlines deficiencies of conventional exception mechanisms to easily handle contingencies successfully and continue execution afterwards.

Ascertaining contingencies before runtime.

To handle contingencies, they must be ascertainable at development time. Conventional languages have no practicable mechanisms to determine all exceptions that can occur syntactically within a code fragment at development time, although that should be not a complex problem.

Distinguishing contingencies and errors.

Unlike errors, contingencies are not allowed to terminate and must therefore be distinguished from errors. Conventional languages have no exception mechanism to distinguish them.

Repairing lower level implementation details.

Repair measures can be taken in defined circumstances, but they require access from the higher call levels to the implementation details of the lower call levels by one means or another. Conventional languages have no language mechanisms to do so. In the following Java example 6 the lower level method call of `saveEditedData` no longer exists after throwing an exception, but even if the stack were not unwound, no language construct to easily manipulate the variable `currentPath` exists.

Example 6. The routine `main` with access to the graphical user interface executes several tasks and handles all occurrences of `DiskFull` by asking the user for an alternative path, but is not able to access and change the variable `currentPath` of the lower level:

```
void main(String[] args) { // GUI-access here
    try {
        doTasks(args);
    } catch (DiskFull) {
        String alterPath =
            AskUserForAlternative.execute().getAnswer();
        // repair lower level with alterPath, but how?
    }
}

// many call levels lower:
void saveEditedData(Data edited) { // No GUI here
    if (getFree(currentPath) < edited.size()) {
        throw new DiskFull(getFree(currentPath),
                           edited.size());
    }
    // ... writing data on disk
}
```

Resuming.

After a successful handling, the program should continue execution to fulfill the postconditions, but how? Conventional exception handling does not offer an easy mechanism to continue execution. The termination model is not helpful for continuing by any means, because the stack is always unwound and then important issues must be resolved to continue within defined conditions: abandon some work already done, undo or handle internal and external side effects, even irreversible physical effects, determine where and how to continue, redo some work and loose performance.

Language mechanisms for resumption are the only known, generally and easily applicable option to bypass the latter hurdles.

Overriding handlings.

No known language offers a mechanism to override handlers similar to polymorphy to exploit additional context knowledge, components and access available in higher levels. Therefore, the Java example 7 cannot override the handling of the exception `PaperJam` within the method `print`

by the handling in the method `printAdvanced` and access its additional hardware component `advancedPaperEmitter`.

Example 7. The routine `print` might be the implementation of a simple printer model, catches the exception `PaperJamDetected` and handles it conventionally. The routine `printAdvnc` might be the implementation of a sophisticated printer model with an additional paper emitter hardware component that reuses the routine `print` and tries to override the handling of `PaperJamDetected`, but that is not possible.

```
public void print(Object document, int fromPage) {
    try {
        // print ...
    } catch (PaperJamDetected jamDetected) {
        // Default handling: log and cancel
        logger.error("Paper jam: aborting print.");
    }
}

public void printAdvnc(Object doc, int fromPage) {
    try {
        print(document, fromPage);
    } catch (PaperJamDetected jamDetected) {
        // overriding for automatic repair impossible!
        advancedPaperEmitter.removeJam();
        printAdvanced(document, jamDetected.atPage());
    }
}
```

7. AN APPROACH FOR EXTENDED EXCEPTION MECHANISMS

A common recommendation is to use exceptions only for errors [4]. This is nearly impossible in non-trivial cases caused by contingencies, as we have shown in the discussion section. Therefore, we unconventionally recommend using exceptions also for contingencies.

Since conventional exception handling is deficient for forward recovery of contingencies and resumption, we sketch extended exception mechanisms that offer appropriate handling and resumption. Mainly concepts of the language Common Lisp are sketched for the language Java. They should also be transferable to other languages.

After giving an overview, the elements of our approach are presented in more detail. Afterwards, an example illustrating the extended mechanisms follows.

7.1 Origins and Overview

Significant ideas are coined by the language Common Lisp [6].

One important concept of Common Lisp (and our approach) is to refine the conventional twofold separation of problem detection (`throw`) and problem handling (`catch`) into a threefold separation of problem detection, problem handling and optional problem solution. The problem solution resumes the execution at any implementation level that is appropriate for the repair and is called from the handling by name. Common Lisp calls them `restart`, we call them `offer`. The problem handling is also called `decision`. The resumption model of our approach is exactly carried over from Common Lisp, particularly multiple, named offers including parameters on multiple, arbitrary levels.

The only important difference is the reversal of the search direction for decisions and offers that is explained in more detail in Section 7.2. Additional to the concepts of Common Lisp, our approach distinguishes errors from contingencies by different keywords, ascertains all contingencies at development time with the help of the development environment and lets the developer interactively choose the contingency to handle and insert the corresponding code into the program.

7.2 Elements of the Approach

All deficiencies of conventional exception handling that have been mentioned within Section 6 are solved by the following elements of the approach.

Distinguishing contingencies.

A new keyword is introduced into the language to distinguish contingencies from errors. Contingencies are marked and thrown with the new keyword `signal` and distinguished from errors that are thrown with the known keyword `throw`. For backward compatibility, the conventional behavior of `throw` should not be changed, the stack will be unwound by `throw` and resuming is no longer possible, but `signal` should behave differently. It should not unwind the stack immediately. Selecting the appropriate keyword determines whether the exception is resumable or not.

As a result, all contingencies occurring syntactically within a code fragment, including all called levels, can be distinguished from errors by keyword and before runtime.

Ascertaining contingencies automatically before runtime and choosing interactively.

It should be possible to handle arbitrary contingencies at arbitrary places, but determining all possible contingencies is very difficult with conventional mechanisms. The need to ascertain all exceptions at development time should not be satisfied by declaring all of them within the interface explicitly and manually. Therefore, we suggest an alternative tool supported approach by automatically ascertaining all of them at development time and presenting them within a dialog. Due to the expected huge amount of contingencies (see Section 5) only an interactive choice of the contingencies that should be handled seems reasonable. This requires a cooperation of all corresponding parser or compiler respectively and development environments. The dialog should offer a sorting and filtering by diverse criteria, e.g., by type, class hierarchy, location, call chain, frequency, already registered occurrence, the existence of a decision for the contingency, personal settings, name pattern matching etc. After the developer has chosen the contingency that should be handled, appropriate code is inserted into the program.

Overriding by reversal of search direction.

All conventional languages and also Common Lisp search for handlers (decisions) and restarts (offers) from the lowest levels to the highest (bottom-up). In contrast to the conventional search direction, our approach searches in reverse direction, top-down. By reversing the search direction, it becomes possible to override existing handlers by higher levels as was intended by the example 7. It is a dynamic binding similar to object oriented polymorphism, but it does not search along a class inheritance path, instead it searches along the call stack.

Distinguish handling with and without possibility of resumption.

If an exception is thrown and the stack is unwound, it can be caught with `catch`, but it cannot be resumed anymore. The behavior of `catch` should not be changed for backward compatibility, because existing handlers assume that the stack is already unwound, resources have already been cleaned etc. These existing correct assumptions should not be violated. Therefore, a resumable exception must not occur at existing `catch` handlers. Hence, a new keyword is required to distinguish and handle exceptions that are resumable. The new keyword `decide` should be used to handle signaled and resumable contingencies. If multiple call levels of the call chain exist that can handle the contingency, the decision that is the topmost in the call chain is executed due to the reversed search direction. In this way, any decision can be overridden quasi polymorphically along the call chain.

Repairing lower level implementation details by offers for resumption.

In different situations, different measures are needed to repair the implementation details of the same level. Therefore, multiple different possibilities for resumption of contingencies can be defined at every arbitrary level with separate name and parameters. These resumption possibilities are introduced with the new keyword `offer`, which is followed by a name and formal parameter declarations with usual notation. These offers are side entrances into the interrupted methods that are still on the stack. They are like procedures (without result value), because they should resume and not return. Offers can access all implementation details of that implementation level. They are additional interfaces and can keep the information hiding principle as conventional interfaces. Offers can only be called by a decision of a contingency.

Resuming.

The new keyword `resume` is used to call offers from decisions, followed by the name and the required actual parameters with usual notation. If multiple offers with the same name and parameters on different call levels exist, the topmost offer is chosen due to the reversed search direction. For this reason, the resumption is not always fixed to the level where the contingency is signaled originally. In this way, offers can be overridden quasi polymorphically along the call chain. When the offer and its level is chosen, the stack will be unwound (afterwards) till there.

Repairing by cooperation of levels.

The approach presented here enables the cooperation of the involved levels, e.g., to use the context knowledge (`advancedPaperEmitter` of the decision level of example 7 or the user interface of the example 6) and also to repair the implementation (`currentPath` of example 6) on the offer level without violating its information hiding.

7.3 Illustrating Example

In the following example that was coined by [9] and slightly extended, the concepts of Common Lisp are outlined for the language Java. The example shows all syntactic extended mechanisms in conjunction. It will be presented as source code within example 8.

Within the example, multiple log-files should be read. Within the log-files are multiple lines that should be checked for whether they are well-formed. For this purpose, two nested loops are used. The first loop iterates over the files. The second loop iterates over the entries of one file. Both loops are implemented within two separated methods `analyzeLogs` and `parseFile`, of which the first calls the second. For each line of the files a third method `parseEntry` will be called that checks whether the line is well-formed. If not, it signals the contingency `MalformedLine` by the new keyword `signal`. Both loops offer an option to resume, the abstraction of which corresponds to the according implementation level, i.e., the loop within `analyzeLogs` offers `skipFile` and the loop within `parseFile` offers `skipEntry`, each without parameters. The offers are introduced by the new keyword `offer`. Furthermore, both loops decide what should happen in the case that a line is not well-formed (`MalformedLine`). For the choice of the contingency that should be handled, the new keyword `decide` is used. Within the handler (decision) of the chosen contingency the choice of the repair and resumption offer is done with the new keyword `resume`.

Within the method `parseFile` the repair offer `returnEntry` is called and the element `defaultEntry` is passed as parameter, which only exists there. This way both involved levels can cooperate using their specific implementation details. Two possibilities exist within the example on two different call levels to decide or handle the contingency `MalformedLine`. The higher level method overrides the decision of the lower level method, therefore the decision with resumption of `skipEntry` is chosen.

Example 8.

```
void analyzeLogs(Files openFiles) {
  for (File file: openFiles) {
    try {
      use(parseFile(file));
    } offer skipFile() {
      continue; // nothing else to do
    } decide (MalformedLine x) {
      if (x.firstStackFrame().startsWith("mylib"))
        resume skipEntry();
    }
  }
}

Entries parseFile(File openFile) {
  Entry defaultEntry = new Entry();
  Entries result;
  while (!openFile.EOF()) {
    Entry entry = null;
    try {
      String logTxt = openFile.line();
      entry = parseEntry(logTxt);
    } offer skipEntry() {
      // entry = null;
    } decide (MalformedLine) {
      // entry = null;
      resume returnEntry(defaultEntry);
    }
    if (entry != null)
      result.add(entry);
  }
  return result;
}
```

```
}
Entry parseEntry(String logTxt) {
  if (entryIsWellFormed(logTxt)) {
    return new Entry(logTxt);
  } else {
    signal new MalformedLine(logTxt);
  }
}
```

8. IMPLICATIONS FOR THE ENGINEERING PROCESS

This Section explains some implications of systematically handling contingencies for the engineering process.

Need for an iterative engineering process.

As described, contingencies disclose implementation details if they are passed to the caller and (as recommended) are not abstracted. Such implementation details cannot be known before the design phase of the engineering process. Usually they are known after the implementation phase, when the requirement phase is already completed within a strict waterfall process. The requirements for contingencies cannot be clarified within the requirements phase, because they are not known at that time.

Example 9. A salesman already has a CRM and loan system for the complete process of customer purchases and credits. It already has an interface for processing the first installment and a module for the interface that can process cash payment and electronic cash (first implementation). The salesman wants to improve the first installment and also support account withdrawals (new requirement). He decides (design when requirement phase has been completed) to buy the newer version of the module (second implementation). His developers explore the new implementation with new, sophisticated tools for contingencies and find the contingencies `EletronicWalletEmpty` (already known and handled but not part of the implementation-independent interface) and `AccountOverdrawn` (new, unknown, not yet handled and also not part of the interface). Not until then, could the implementation specific contingencies have been known and reasonable requirements for recovery for `AccountOverdrawn` been gathered. In this case, the developers might ask the salesman for requirements within a second requirements phase and might be given new requirements. The contingency `EletronicWalletEmpty` is already handled by an extension of a week for the first installment. The salesman is asked what the application should do if the account is overdrawn. He answers: "The credit should be denied!"

The payment of the first installment may be done with electronic cash or by an account withdrawal. If it succeeds, the implementation does not matter, but if it does not succeed, the implementation details might disclose important additional information that is relevant for repairing or handling and must or should be handled separately even on the business process level. The `AccountOverdrawn` discloses information about a potentially bigger monetary problem of the customer's than the `EletronicWalletEmpty`. This seems to be a general principle for repairing, also for errors: if not everything works, the repairer needs to know the implementation.

Such previously unknown requirements for contingencies, like many other usual requirements, cannot easily be foreseen by the developers and must therefore be gathered in an additional requirements phase. Determining the requirements of contingencies requires a kind of an iterative engineering process approach. This is not surprising, but viewed from this new perspective: even a perfect strict waterfall process cannot deliver a completely flawless implementation, especially with respect to contingencies, and therefore such systems crash for cases happening in the real world. Even projects that follow iterative approaches must explicitly regard requirements gathering for contingencies, otherwise contingencies will bug users and later return to development as bug or crash reports.

There must be a separate phase for requirements, design, implementation and testing for contingencies in the engineering process, at least once before roll-out. They might be overlapped with a regular iteration for new features, but that iteration result still cannot be rolled-out, since the contingencies of the new features have not been processed. Contingencies should not be rolled out without having been processed. They have enough potential to fail.

Interactive forward recovery.

Sometimes the user (the caller of the top call level) must be asked to decide and support forward recovery. An interesting solution with predefined options is commonly used in Common Lisp environments: interactively present all available offers of the application to the user and ask him to take the decision only if there is no decision at all within the code for the contingency. This is the latest time to ask for the requirements, but it can take advantage of the most complete information, e.g., about the real implementation, like installed memory, hard drives, mounted directories, available amount of space, available components etc. If applications do not leave this user interview to the environment, they might offer an even more application-specific dialog.

Turning errors into contingencies.

Errors are situations, the conditions of which, contradict the specification, but if they really occur, they simply have real conditions that were not known or regarded before. Usually, developers analyse the conditions and, after understanding them, change the system to avoid them, but sometimes they realize that the conditions were unforeseen and are unavoidable. Then they have to be specified exactly, extending the previous specification. They become contingencies, have to be handled systematically and new requirements on how to handle them have to be gathered.

9. CONCLUSIONS

Contingencies are undesired results of routines and are specified as well as desired results. In this light, they are the situations that really can be handled and for which forward recovery and resumption is reasonable. Hence, contingencies may fulfill the hopes that were initially associated with error and exception handling.

[3] is often cited as argument against the resumption model and has the opinion that a more complex mechanism can only be justified if the additional expressive power it provides is frequently needed. As we have shown, resumption is needed frequently for forward recovery of contingencies.

Some may fear the additional complexity of resumption, but we do not see any other appropriate solution. Other than resumption, no alternative approaches to solve contingencies are known. Additionally, our impression is that the presented concepts are simple and less complex than other already established concepts from other domains, e.g., event oriented systems. Related and further work such as the planned evaluation approach is mentioned within [10].

A systematic inspection and handling of contingencies with these mechanisms before runtime can diagnose and avoid subsets of errors effectively. We assume that the systematic handling of contingencies has positive effects for production readiness in the first iteration, but even if the presented mechanisms are helpful, they will not solve thousands of contingencies within one single system immediately. We hope that over time increasingly more frameworks, libraries, programs and systems find reasonable algorithms for forward recovery and solve their contingencies. Especially for safety critical applications the proposed extended exception mechanisms could be very helpful immediately.

10. REFERENCES

- [1] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.
- [2] J. B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.
- [3] B. H. Liskov and A. Snyder. Exception handling in CLU. *IEEE Trans. Softw. Eng.*, 5(6):546–558, 1979.
- [4] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
- [5] Oracle. Oracle9 i database error messages, release 2 (9.2) part no. a96525-01. 2002. http://download.oracle.com/docs/cd/B10501_01/server.920/a96525.pdf.
- [6] K. M. Pitman. Exceptional situations in Lisp. In *Proceedings for the First European Conference on the Practical Application of Lisp (EUROPAL'90)*, Cambridge, UK, 1990.
- [7] M. Raento. What should exceptions look like? *Mika Raento's Blog*, July 2006. http://www.errorhandling.org/wordpress/?page_id=100.
- [8] B. Ruzek. Effective java exceptions. *dev2dev.bea.com*, January 2007. <http://www.oracle.com/technology/pub/articles/dev2arch/2006/11/effective-exceptions.html>.
- [9] P. Seibel. *Practical Common Lisp*. Apress, September 2004. PDF at <http://www.apress.com/resource/freebook/9781590592397> and HTML at <http://gigamonkeys.com/book/>.
- [10] T. van Ellen and W. Hasselbring. Extended exception mechanisms for contingencies. In *SERENE 2008: Proceedings of the Software Engineering for rEsilient systems 2008 workshop*, Newcastle upon Tyne (UK), 2008. (In press).
- [11] D. Weinreb. What conditions (exceptions) are really about. *Dan Weinreb's Weblog*, March 2008. <http://danweinreb.org/blog/what-conditions-exceptions-are-really-about>.