Disentangling Exceptions

By

Bradley A. Berg

M.S.E., Wang Institute of Boston University, 1983

Thesis

Submitted in partial fulfillment of the requirements for the
Degree of Master of Science in the Department of Computer Science
at Brown University

Providence, Rhode Island

December 2008

This thesis by Bradley Alan Berg is accepted in its present form
by the Department of Computer Science as satisfying the
thesis requirements for the degree of Master of Science.

Date_____ _____
Prof. Steven P. Reiss, Advisor

Approved by the Graduate Council

Date_____ _____
Sheila Bonde, Dean of the Graduate School

Vita

Bradley Alan Berg was born in Evanston, Illinois USA on March 14, 1953. He received a Bachelors degree in Pure Computer Science from Nothern Illinois University in DeKalb, Illinois in 1977 and a Masters degree of Software Engineering from the Wang Institute of Boston University in Tyngsboro, Massachusettes in 1983.

Publications

B. A. Berg, "Project Management with CASE," First International Workshop on Computer Aided Software Engineering, May 27-29, 1987.

N. R. Papadakis, B. A. Berg, C. M. Cook and K. A. DeJong, "A Layman's Guide to Knowledge Engineering," Smart Systems, April 1985.

D. A. Hejhal and B. A. Berg, "Some New Results Concerning Eigenvalues of the Non-Euclidian Laplacian for PSL(2, Z), "University of Minnesota Mathematics Report 82-172, June 1983.

Acknowledgements

With great appreciation to my advisor Prof. Steven Reiss, Prof. Maurice Herlihy for advice on Transactional Memory, and Prof. Stan Zdonik for bringing me on to the Borealis project. Thanks to friends Tom Maxon for support and encouragement along with Alan Martin and Dan Ligett for their thoughtful reviews.

Contents

Appendix

List of Examples

List of Figures

Abstract

Exception handlers intermeshed in the body of a method are highly coupled with primary code. Lexically separating out handlers simplifies primary code and can make exceptions easier to write and maintain. The programming model presented here uses assertions to detect exceptions. Several different mechanisms for dispatching exceptions to handlers are considered. A distinct control path is used for program faults to further decompose the overall exception architecture. Exception context is propagated in objects accessible only from handlers to enforce a separation between primary and exception objects. Default exception mechanisms implement the most simple and prevalent forms of exception management. Use of the default mechanisms greatly reduce the number of exception handlers required even in substantial programs.

1. Introduction

One of the more challenging aspects of programming is writing code to manage exceptions. From a program design perspective an exception condition occurs when a method is unable to complete it's stated objective. Reframed using client/server terminology, an exception occurs if a server can not honor a client's request. Exception handling mechanisms incorporate programming and modeling language constructs to provide support for managing exceptions.

When using conventional exception handling mechanisms programmers often encounter problems due to excessive control, data, and name space coupling. The control flow for programs can be obscure and difficult to follow, global data is shared between unrelated components, and identifiers used to reference exceptions need to be managed. This leads to difficulties as a system scales up in size and exception capabilities evolve. This paper devises novel exception handling strategies and mechanisms that addresses these concerns.

Practitioners can apply these strategies to ongoing projects and employ them to design tools and establish policy for managing exceptions. Programming constructs and a corresponding exception handling mechanisms are proposed that directly support these strategies. Language designers can use these concrete examples in conjunction with the analysis to design exception handling features in new languages and to extend existing languages. The implementation demonstrates by example that the mechanisms work well in substantial programs and that the constructs can be implemented without degrading performance.

The techniques presented separate concerns between exception handling code and primary code based on an analysis of control and data flows that occur throughout the lifetime of an exception event. Handlers are lexically separated from primary code so that new exceptions can be added without disrupting primary control paths. Program control paths are further disentangled by providing a distinct path for program faults. Method signatures are extended using preconditions; making them more precise and eliminates ambiguous or overloaded named exceptions. Consequently, overhead and conflicts associated with managing exception names is avoided. Exceptions are detected using assertions; providing simple and consistent support for integrity checks resulting in more reliable programs. Commonly occurring cases can be expressed using default mechanisms to simplify code that detects exceptions and requires fewer explicitly coded handlers.

1.1 Related Work

When an exception is raised a branch is taken from the point where an exception condition is detected into a distinct exception control path. At this point the method containing the exception point is partially executed and objects can be left in a partially transformed state which might be invalid.

Branching from an exception point deviates from the control flow of a program in a way that is inherently different from traditional block structured programming constructs. That is an exception path can leave a method‘s primary control flow at potentially any point and connect to a path determined by the exception handling mechanism.

Sinha et al. [24] point out that the exception control flow can be implicit in that it may not be obvious from the source code. An exception can occur as a side effect of an operation or a method call. The branch target from the exception point can be indirect or not explicitly declared. As a result exceptions become difficult for a programmer to reason about and are very easy to overlook. An exception handling mechanism provides the programmer with the means to better manage partial methods and partial objects.

Exception handling is now considered common practice, primarily using the *Try-Throw-Catch* mechanism as typified by C++ and Java. Widespread use has resulted in exposing software engineering issues encountered managing exceptions in general and using the *Try-Throw-Catch* mechanism in particular.

Miller and Tripathi [17] detail instances where traditional Object-Oriented programming methods are insufficient for managing partial methods and objects. As a program evolves though development and maintenance, methods are modified to include new implementation mechanisms or new functionality. Even though these changes might not change the parameters used to call the method, new exceptions could be added.

Managing new exceptions can lead to changes that ripple up the call chain resulting in violations of encapsulation. After an exception, objects may be left in partial states and new exceptions can introduce additional partial states. These changes can propagate up both the call chain and class hierarchy.

1.1.1 Handlers Embedded in Primary Code

When code to handle exceptions is nested within primary code then the two can become more tightly coupled than is neccessary. Primary code becomes cluttered with exception handling blocks making the control flow difficult to follow. Tracing an event up the call stack in and out of the method's primary code can be daunting.

Robillard and Murphy [21] observe that problems result from complex interdependent control flows in programs using *Try-Throw-Catch*. The root causes they cite are that exception handling is a global design problem and add that it is difficult to anticipate exception sources in advance.

> In our experience, this lack of information about how to design and implement with exceptions leads to complex and spaghetti-like exception structures. Even when we set out to carefully design and implement the exception handling code in one of our programs, we still ended up in the same predicament. Speaking with other Java developers and analyzing other Java source code, we have found that the situation is far from uncommon [21].

In practice programmers frequently defer exception coverage until after the primary code path is functioning. This is only natural as exception management is difficult to anticipate in advance. In the rush to deliver, exception coverage can get deferred indefinitely. It can also be difficult to revisit code later and cover the exception cases. Details of the code are freshest in the mind of the programmer when they originally write it.

The approach taken here is to separate out exception handlers from primary code so that it does not need extensive modification as exception cases are added. Should exception handling be deferred in early development, assertions serve as exception markers when revising a method. Beforehand they detect exception cases so they will be managed as an unhandled exception. Similar observations are made by Filho et al. [6] regarding Aspect Programming:

> (i) Error handling code and normal code are lexically separate and can be maintained independently; (ii) the impact of the code responsible for error handling in the overall system complexity is minimized; and (iii) an initial version that does little recovery can be evolved to one which uses sophisticated recovery techniques without a change in the structure of the system [6].

1.1.2 Named Exceptions

We are used to representing exceptions with control states; either as status codes or by overloading the type system with named exception types. This is classical control coupling; where control state is represented as data.

Names are used to identify exception events, but do not specify exception semantics. Instead they are attributed with informal descriptions at best. Different names assigned by different component developers may have the same or similar semantics which require interpretation. Programmers will reuse existing exception names and overload them with semantics for local use; which results in inconsistent usage or imprecise specification.

> ... intra-component exceptions could be re-mapped to the smaller set of exceptions that might be raised by a component. In practice, this led to problems interpreting the meaning of a component-level exception [21].

Exceptions are likely candidates for change. When they do change methods that enumerate exceptions require corresponding changes. This forms a global interdependency between methods and exception names.

> Therefore, for unchecked exceptions, the throws declarations can miss exceptions that can actually be propagated (i.e., they can be unsafe) and they can list exceptions that cannot be propagated (i.e., they can be imprecise). For checked exceptions, the compiler ensures than the throws declarations are safe; however, even for checked exceptions, the throws declarations can be imprecise [24].

Managing the names themselves is unecessary overhead. To program with named exceptions, you first have to make up a name, attribute informal semantics to it, and declare it. The names apply between components which means a global taxonomy of names need to be established and consistently used by multiple programmers.

Malayeri and Aldrich [15] claim that the *Try-Throw-Catch* mechanism is a low level control flow mechanism and is insufficient for describing broader exception policies.

It is notoriously bothersome to write and maintain throws declarations. Simple code modifications a method throwing a new exception type; moving an handler from one method to another can result in programmers having to update the declarations of an entire call chain.

There is anecdotal evidence that this overhead leads to bad programming behaviors. Programmers may avoid annotations by using the declaration throws Exception or by using unchecked exceptions inappropriately. Worse, programmers may write code to ‘‘swallow’’ exceptions (i.e., catch and do nothing) to be spared the nuisance of the declarations.

But even if programmers use checked exceptions as the language designers intended, exception declarations quickly become imprecise as code evolves; statements that throw or handle exceptions will invariably be modified [15].

1.2 Component Layers

When discussing the design and implementation of exceptions it is useful to divide a program into layers. Robillard and Murphy [21] borrow a concept from fault-tolent design called a compartment which is a more coarse grained programming unit than a class. Exception design is less complex as handlers can be provided on a per-compartment basis rather than per method. Often exceptions can be handled within a compartment and don't need to be exported. New exceptions might not propagate out of a compartment and partial objects might be contained within a compartment.

Typically a programmer has source code control over all methods within a layer. They can make interdependent lateral changes to code within a layer without it becoming burdensome. Malayeri and Aldrich [15] formally extend Java to partition programs into modules. Only methods marked ‘‘*module-public*’’ are callable from outside the module and can export exceptions.

Within a single layer the vocabulary used to describe the program's semantics is the same. Exception handlers that compose descriptive messages use vocabulary corresponding to that layer. Handlers in different layers will often need to use a different vocabulary. A message formed at a lower layer may have little meaning to an end user.

The top application layer of a program is special in that it does not have any higher layer and the vocabulary is that of the user. Conversely, a component layer can have an opaque interface through which exceptions may pass. A complete specification of component interfaces that pass exceptions can be difficult to write as exceptions are likely to change.

1.3 Handlers

After an exception is raised control is passed to an Exception Manager. In a compiled program the manager interacts closely with complementary generated code. The manager unwinds the call stack, calling destructors upon leaving scopes and dispatching control to appropriate exception handlers. Eventually the exception path will rejoin the primary code path or the manager will unwind out of a program or thread and terminate. In the lifetime of an exception event multiple exception handlers are used spanning the call stack [Appendix A].

Although programmers are free to code handlers as required by the application there are several basic operations typical of exception handlers. Context information can be passed up from lower layers, transformed, and passed up to the next higher layer. A *Reform handler* performs this transformation. For example, variables received from a lower context might be used to form a descriptive message and passed up. A *Reporting handler* receives messages and may merge them with context information to produce messages for end users or program log files.

Another prime activity is recovery. Locally, each partial method might use a *Recovery handler* to clean up and revert to a stable state. Often the destructors run when a method leaves scope is sufficient. Other times a handler is included to free or reset resources.

Recovery handlers also run various recovery strategies such as retrying operations. A recovery process involving user interaction will reside in the application layer and interact using the vocabulary of the user. In cases like this an exception in a low level process could require cooperation between handlers spanning multiple component layers [Appendix J].

1.4 Overview

This paper considers several alternatives for managing the control and data dependencies inherent in exception handling.

- In [Section 2] assertions are used to detect and raise anonymous exceptions. A descriptive message and context expression are encoded as part of an assertion as these elements are prevelent when processing exceptions.
- [Section 3] discusses program faults (bugs); which are treated separately from managed exceptions. Source code is lexically partitioned between primary code, fault handlers, and managed exceptions.
- Several techniques to dispatch anonymous managed exceptions are considered in [Section 4]. A new kind class member called a *Signal* is introduced to pass context data between handlers.

2. Assertions

Meyer [16] uses assertions in contract based programming to specify pre- and postconditions used to check a program's correctness. Preconditions check that methods were invoked correctly and postconditions check that a method completed correctly.

The notion of an assertion is extended here as a convenient way to detect and raise an exception. It does so without introducing nested structured programming constructs. This allows new exceptions to be detected in the primary code body without changing its local control structure. Consequently assertions are easy to add and can be used liberally to detect exceptions. Low effort reduces the temptation for a developer to defer exception detection. The odds of overlooking an exception decrease as more assertions are added.

2.1 Conditions in Modeling Languages

It is desirable for conditions in a modeling language to be reasonably close to that of the implementation language. Chalin [4] contends that practitioners will not want to learn two sets of logic. Leavens et al. [12] adds that slight and subtle differences between conditions in programming and modeling languages can be difficult for programmers to use easily and correctly.

A critical distinction is that conditions in programming languages can have side effects. As meta-constructs, assertion semantics imply that they can be evaluated or skipped without effect.

To extend this concept to assertion conditions in programming languages they need to be free of side effects. Enforcing these constraints using static analysis helps detect and avoid programmer errors at compile time.

In JML the concept of purity is used to avoid side effects:

- Pure methods have no side effects.
- Pure object constructors restrict assignments to their own fields.
- Pure methods and constructors can only invoke pure methods and constructors.
- Purity is inherited by overriding methods.

Leavens et al. [12] point out that the order of evaluation for compound conditions matters in Java. Conditions in a modeling language hold concurrently over all cases while programming language semantics are sequential. For example, a modeling language might use unbounded arithmetic whereas numerical types in programs are bounded.

Chalin [3] expands this to say that conditions in modeling languages are inherently more powerful than programming languages. Modeling notations can contain first order logic with universal quantifiers and operators such as implication and equivalence. Conversely, programming languages can have features not represented in modeling languages. For instance, Chalin [4] notes that a Java assertion can raise an exception or fail to terminate. In JML however, expressions always evaluate to a well defined value.

2.2 Conditions in Programming Languages

A compiler for an experimental programming language implements the notations and methods described throughout this paper. To support assertions, conditions in the language are restricted so as to have no side effects. Should an exception occur it is treated as a programming error; not as an allowable exception. Note that even if the floating point operating mode is quiet (non-signaling), some floating point operations will still fail. For example, an inequality comparing a valid value to an invalid value will raise an exception.

Evaluation of compound conditions is well defined. They are evaluated left to right and may not be nested. Only ‘‘Or’’ and ‘‘And’’ operators are supported with conjunction having precedence over disjunction. After a clause in a conjunction fails, the remaining clauses in the conjunction

are skipped. Likewise, evaluation of a condition stops after the first disjunction clause is satisfied. Ordered compound conditions are useful for avoiding exceptions in conditions. A common usage is to check for a non-null pointer in the leftmost clause before referencing it in subsequent clauses.

Conditions may only invoke pure (side effect free) functions. For the most part user defined methods are invoked using subroutine calls; not function calls as is the case for C++ and Java. A pure function is a restricted style of method that can be invoked using functional notation in expressions and conditions. Amongst other things, pure functions are useful for inspecting containers and implementing universal quantification.

- Functions can not change the global state of a program. They may only contain local variables and can not contain global state in the form of static global variables. Functions can read but not set any globals. They can not modify their parameters or any data accessed through their parameters.
- Functions can not contain input and output commands. The language has an intrinsic debug command that is used to print the value of variables in debug builds. They are permitted in functions for debugging purposes. When enabled their only effect is to print.
- Functions can not raise exceptions. If a function raises an exception it is treated as a program fault.
- Functions only invoke other pure functions or pure methods. Pure methods are allowed to modify their parameters; whereas functions can not. Pure functions and methods have the property that they are deterministic.

Constructors are also restricted to avoid side effects. They can only assign constant default values to fields in an object. A constant can also be computed as a pure function over constant arguments. Pure functions can construct any type of constant object.

At first blush these restrictions might seem to significantly reduce the expressive power of the language. Functionality that programmers habitually use is proscribed. To date, a code base of about 200,000 lines of code has been developed adhering to these constraints. While a substantial platform for experimentation, the code base is not sufficient enough to draw broad conclusions about other types of software. Still, in only a handful of cases these restrictions make the code just slightly

awkward. On the other hand, side effect free expressions and conditions makes it much easier to reason about a programs behavior when making revisions.

2.3 Assertions in Methods

The experimental language supports several different types of assertions with the following syntax.

```
ASSERT  [<Condition>]  [ PASS   <Context>
                       | FAULT [<Context>]
                       | CATCH <Handler> ] ';' <Message>

 Condition - The condition is evaluated to true or false
             and is false when omitted.  A false
             condition raises an exception.

   Context - The Pass clause evaluates to a string with
             context information used for reporting.
             The Fault clause designates a Program Fault
             and optionally sets the context string.

   Handler - The Catch clause is used to explicitly
             transfer control to an exception handler
             local to the method.

   Message - The comment contains an assertion message
             giving the reason the exception occurred.
```

Preconditions in Eiffel [16] use assertions that can be inherited. This ensures that virtual methods all contain inherited preconditions and may add their own as well. Based on their utility Rustan et al. [22] extend C# with inherited preconditions. The syntax for a precondition block in the experimental language is given below.

```
ASSERT PRECONDITION [SAFE | PURE]
    <Condition>  [ PASS   <Context>
                 | FAULT [<Context>]
                 | CATCH  <Handler> ]  ';' <Message>
      :
END ASSERT
```

A precondition can only be coded in the opening lines of a method body. It is considered part of a method's signature and is inherited. An overriding virtual method can provide additional precondition checks; which are run after first checking any inherited preconditions. Intermediate computations in a precondition can not be explicitly saved using assignments. A compiler optimization to save redundant intermediate results can avoid recomputation.

Precondition checks have no side effects so that when they fail it is guaranteed there will be no objects left in a partial state. The *Safe* or *Pure* keyword on a precondition declares the method will not propagate an exception provided the preconditions hold. After the precondition passes any further exception raised should be handled and the method returns successfully. If an exception attempts to propagate out, a guard will catch it and treats it as a program error. Note that any initial value on local variables are set before preconditions are checked.

```
METHOD Pop:  Pop the top element from a Stack.

CHANGE Stack  Stack..Word    :The stack to be modified.
  EXIT Value  WORD           :Return the top integer.

ASSERT PRECONDITION PURE
   Stack‘Size <> 0;           ERROR: Stack underflow.
END ASSERT
```

Example 1: The signature for a Precondition Pure method (to pop a value from a stack of integers) states it will not raise an exception when the precondition is met. Note that text after a colon or semicolon is a comment. The language is case insensitive. In code examples keywords are in upper case.

3. Program Faults

Checks can be placed within a program to detect when a state is reached that is unintended, unanticipated, or unsupported. Responses to these conditions are often quite different from those taken for managed exceptions which are anticipated. Note that a program fault is one of several kinds of fault that can occur in fault tolerant systems [18]. Program faults are not reached intentionally and are typically integrity checks that detect programming errors. A predefined taxonomy for program faults is used to identify these events and they are handled separately from managed exceptions. The predefined faults are detailed in Appendix B.

After a fault the program may be in an unstable state. Returning to a stable state or gracefully terminating requires case-specific considerations that depends on the nature of the exception. Unlike managed exceptions, program faults can potentially occur anywhere in the call graph. Information about where a fault occurred is largely useless in responding handlers. This is in contrast with managed exceptions; which can behave differently depending on the location where the exception was detected.

3.1 Detecting Faults

A faulty program will often wind up attempting to execute an invalid statement or an invalid machine instruction. An invalid instruction state results in a machine check detected by computer hardware. A Machine Check Handler fields these exceptions and determines if they should be interpreted as a program fault or handled by some other means. Violations of programming language semantics can be detected by code generated by a compiler or by the run-time system.

Programmers can also explicitly detect fault conditions. The experimental language uses a Fault clause on Assertions to raise a fault. Meyer [16] and Rustan et al. [22] also use assertions in class definitions to specify invariants for objects. They detect corrupt objects primarilly while developing and debugging programs.

First and Last Chance Handlers can be used to perform system wide exception related operations. Programmers can use the arguments passed to the First Chance Handler to safely respond to program faults before any other handler. The First Chance Handler is invoked by the Exception Manager; which is written to work even in most unstable conditions. The Last Chance Handler catches any unhandled exceptions before unwinding out of a program or thread.

3.2 Handling Faults

The parameters described in [Figure 1] are used for the First and Last Chance Handlers to experiment with exceptions. On a program fault the Fault parameter is set and can be used to make program control decisions. Context and Message parameters can be used to report problems to the user. They are set by Assertions or contain default values corresponding to intrinsic exceptions.

Developers debugging programs can use the Method and Line parameters to locate the origins of exceptions in the source code. Production code should not make decisions based on the location of the exception point as it is a violation of encapsulation [17]. These values will change as underlying components are modified.

The enumeration, Fault, provides a partial taxonomy and elides any platform specific details for machine checks. Programs that recover from machine checks will often need to take into account more specific details. Instead, the intent of the partial taxonomy is to provide a portable interface. It guarantees that all machine checks are fielded and a basic cause is identified. The partial taxonomy

```
Fault   - An enumeration assigned to each type of
          fault.  Zero for managed exceptions.

Message - A descriptive message used solely for
          reporting purposes.

Context - A string value used solely for reporting.

Method  - The name of the method or run-time
          component where the exception occurred.

Line    - Zero or a line number of a failed command
          in a method, or an offset a failed machine
          code instruction.
```

Figure 1: First and Last Chance Handler Parameters

enables programmers to manage common machine checks using portable code and select others for further platform specific forensic analysis.

The run-time system provides a default First Chance Handler that does nothing unless an intrinsic program fault occurs. In this case it directly calls the Last Chance Handler, attempts to close open system resources such as files, and finally terminates the program. A standard library routine overrides the default First Chance Handler to save its parameters for use by subsequent handlers.

The Last Chance Handler receives the same parameters as the First Chance Handler. The default version prints a message composed from the Message and Context parameters to the standard error stream; should it be available. When all exceptions are handled the Last Chance Handler is never invoked and there is no need to override the default. Even in this situation it might be useful to override it to catch unhandled exceptions when debugging.

Application developers can write a First Chance Handler to perform more sophisticated program fault handling. It could respond to fault events and decide whether or not to propagate an exception up the call stack. Often it is appropriate to gracefully terminate as recovery may be unreliable, impossible or take more effort to code than it is worth. Since there is only one First Chance Handler, it is reserved for application layer developers. Components should not override it or depend on it.

After a program fault the call stack is unwound up to the Last Chance Handler unless a fault handler rejoins the primary code first. Since faults can occur anywhere the set of methods that get unwound is essentially random. Calling handlers within this set of methods intended for managed

exceptions could have unintended consequences. To avoid this after a fault, any handlers for managed exceptions are skipped.

```
:  Read a calculation file and perform arithmetic.
METHOD Calculate.File  SAFE
 ENTRY Path    STRING,      &Path of the input file.
       Trace   Bit          :Trace the calculations.

 LOCAL Input   File.Path,   &Structure to access a file.
       Line    STRING       :Content of an imput line.

   USE Calculate‘Rpn,       &Stack used to calculate.
       Assertion‘Assertion  :A standard exception class.

ASSERT PRECONDITION
   Path <> ""  FAULT;  CHECK:  Missing input file path.
END ASSERT

Open.In  Input, Path;          Begin reading input.

DO UNTIL  Input‘File‘End:      Read to the end of input.
   Read.Line  Input, Line;     Read a line of input.

   IF Line <> "":              Does it have a command?
      Calculate.Line  Line, Trace;  Interpret each line.
   END IF
END DO

ASSERT Rpn‘Empty    &NOTE: Stack has remaining elements.
       PASS Rpn‘Size

DRAIN RETURN:            Print here and in all handlers.
   PRINT  "### Finished file: ", Path
RETURN;                  Primary Calculate.File return.

::::::::::::::::::::::::::::::::::::::::::::::::::::::::
METHOD CATCH FAULT:  Catch and report program faults.

PRINT‘ERR  "A fatal error has occurred:  ",
           Assertion‘Fault, Assertion‘Context

IF Assertion‘Message <> ""
   PRINT‘ERR  "   ", Assertion‘Message
END IF

RETURN‘RAISE;     Propagate fault out of Calculate.File.
```

Example 2: Example of an exception Safe Method with a Drain clause and a Fault handler. The handler will catch the fault in the precondition and any fault sent by any called method. Note that a Fault can propagate out of an exception Safe method.

Specialized fault handlers are used to intercept faults. Upon exit they can revert the fault to a managed exception, rejoin the primary code, or continue to propagate the fault.

```
METHOD CATCH  FAULT:  Introduce a fault handler.
```

[Example 2] shows how a method file with a handler is layed out in the experimental language. Each file contains a single primary method and any associated handlers. Primary methods are invoked down the call chain and the handlers are invoked up the call chain after an exception is raised. Handlers are sibling methods [18] in that they have access to all variables and parameters that the primary method can access. They may also have their own local variables. An expanded version of this example is in Appendix E.

In the absence of a fault handler in a method, any recovery code is limited to actions contained in destructors. If a method requires compensating actions to be performed regardless of the kind of exception event a destructor can be used to perform the action [2]. A Drain block can also be coded with the Return statement in the primary method body to specify commands to run when leaving the method or unwinding from any of its sibling handlers.

```
DRAIN RETURN
    <Common statements>
RETURN
```

A handler that recovers from program faults should be located above the point in the call stack where any variables it accesses are discarded while unwinding. Local variables will be unwound and discarded, however some global variables might unintentionally remain in a partial state. In order for the fault handler to recover reliably, all global variables in use need to be analyzed to determine if they need to be reset. It is easy to omit a global from this analysis; especially one added after the analysis is performed. To provide coverage for program faults, testing will be required [7].

Alternatively, select program faults can be recast as managed exceptions by the First Chance Handler and propagated upward. Another approach to recovering from a program fault within a subsystem is to contain the subsystem within a thread. In this case the thread is used as a protective barrier around the subsystem rather than to support concurrent processing. When a program fault is encountered the subsystem's thread can be terminated after safely releasing system resources it holds. Control then safely resumes in the parent process.

3.3 Faults In Concurrent Threads

In a muti-threaded system code run in an exception event needs to be thread safe to ensure that events in concurrent threads are independent. This has consequences for each component involved in an exception event. Any restrictions such as performance requirements imposed on threads apply to exception handling components as well. This is at odds with the performance convention for exceptions in serial programs; namely that exceptions are rare and exception performance is not critical to program performance [5].

Faults detected by a machine check are fielded by the operating system and control is passed to a Machine Check Handler registered with the operating system. In Unix the *signal* facility is used for fielding both system interrupts and machine checks. The *man* page for *signal* carries the ominous warning, ''*The effects of this call in a multi-threaded process are unspecified.*'' In practice machine checks in multithreaded code are common enough that particular Unix implementations can be expected to support concurrent machine checks. However, the precise interface and performance characteristics vary.

Microsoft Windows uses an undocumented interface for registering a Machine Check Handler. Fortunately it was reverse engineered and described by Pietrek [20]. An implementer should anticipate performing additional reverse engineering to sort out the details. A newer mechanism was introduced in Windows XP called Vectored Exception Handling [19]. It dispatches control to handlers according to priority and hence can speed up critical exception events. However, only the original mechanism can provide upward compatibility for older versions of Windows.

In Windows handlers are registered on a per-thread basis early on in each thread. When a machine check occurs control is dispatched to the Machine Check Handler where the cause of the check can be determined. If an exception is to be raised control is next passed to the Exception Manager in the run-time library; which in turn calls the First Chance Handler. Here the application programmer can add code to determine if the check should be treated as a fault or a managed exception.

Aside from machine checks, a fault can also be detected by checks written by the programmer, generated by the compiler or included in the run-time system. In these cases an exception is raised and control is passed to the exception manager; which again calls the First Chance Handler to see if it is a fault or a managed exception. This determination is implicitly a program fault for checks generated by the compiler or detected in the run-time system.

After a fault occurs within a thread, dispatching it through a distinct fault handling path avoids potential thread conflicts. Shared globals are protected from unintentional access. If software locks are in use then destructors can be used to automatically release locks after a fault. Minimizing the amount of time spent detecting faults ensures destructors for locks can be run quickly. Faster fault resolution helps contending threads to continue to make progress.

The response to a fault in a thread might be to simply terminate the thread rather than recover. In this case the First Chance Handler could immediately terminate the thread. The operating system can then automatically free any system locks held by the thread. If an early exit could result in thread interference due to stale software locks then it is best to unwind to the Last Chance Handler and then terminate the thread.

4. Handling Managed Exceptions

Within a program an exception event is identified not just by the point at which it is detected, but also by the place where it is fielded. That is, an exception raised by a partial method will have different meanings depending on where the method was called from. Furthermore, different call paths can be taken from an originating call down to the partial method. The application semantics of events can differ depending on which path was taken. Unique identification of an exception event is characterized by the call stack and the point in the partial method where the exception occurred.

The combination of exception points and possible call paths to each point results in a large number of potential exception events. Only a small fraction of these are reachable in a program and require handling. The number of exception events is further contracted when handlers combine and prune exceptions as the call stack is unwound.

Component designers can combine several related or semantically equivalent exception events into a single exception. For example, there might be several different kinds of I/O errors that can occur in a method. The method interface might combine them all into a single '*IOError*' exception. Also, depending on how a subroutine is called, the programmer can reason that certain exceptions will never be reached and can be safely ignored.

4.1 Exceptions Local To A Method

Exceptions can be trivially handled within the method where they are raised. A general handler catches all exceptions for a method that are not fielded by specialized handlers (described subsequently). An optional *Retry* keyword allows exceptions within a handler to be fielded. If absent a guard prevents exceptions that that can occur while in the handler. *Retry* is explicitly coded to prevent unintentional looping from circular exceptions. It is useful when retrying a recovery action, perhaps a limited number of times [Appendix D].

```
METHOD CATCH  [RETRY]  :Introduce a general handler.
```

The *Catch clause* on an Assertion statement passes control to a corresponding handler; which coexist in the same method. It can not be passed up the call stack as this would create an interprocedural dependency (i.e. a long jump). Assertions without a *Catch clause* are anonymous. The exception is unnamed and does not throw an exception object. Instead other means are used to manage the control and data state for an exception event. Subsequent sections describes techniques for discriminating between exception events and passing exception context between handlers.

```
METHOD CATCH  <Exception>, ...   :Local assertions.
```

Although these simple mechanisms for localized exception handling are inadequate for complex recovery processing, the idioms are common enough to be surprisingly effective. In the experimental code base, most exceptions were detected in the application layer. Well formed messages could be composed in the method where they were detected. For the most part annotated messages on assertions were suitable for reporting as is. Calls into component layers were by in large clean in that exceptions within the components were not reachable. Even messages issued by components on reachable assertions were often legible to end users. Other times a set of messages received from a component could be discarded and replaced by a single message.

To evaluate the utility of restricted exception signatures an experiment was conducted with a modest container library. The library was revised using method declarations with restricted exception handling semantics described in [Figure 2].

The resulting library was cleanly implemented using a combination of these method declaration styles. The resulting code was not unduly constrained, but instead using explicit exception restrictions in the signatures made the code easier to work with than the previous unrestricted im-

```
METHOD <Method Name>
   On an exception returns a descriptive message
   and a context string.  The signature can be
   constrained using a precondition block; which
   can also have Safe and Pure flags.

METHOD <Method Name> SAFE
   A method that does not throw exceptions.

METHOD <Method Name> PURE
   A method that has no side effects other than
   changing arguments and does not throw exceptions.

FUNCTION <Function Name>
   A function without side effects or exceptions.
```

Figure 2: Simple Primary Method Declarations

plementation. One awkward attribute is that *Precondition Pure* methods can not be called from a pure method since an exception can be raised. This can be avoided by design considerations or by using virtual methods to work around this situation [Appendix C].

Still there are essential events that can not be managed so simply. Primarily a formal interface is needed for opaque components that pass context information between components. Distinct sets of exception events need to be discriminated.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::
METHOD CATCH Leftover:  There are remaining values.

PRINT‘ERR‘#  "NOTE: ", Rpn‘Size

IF Rpn‘Size = 1
   PRINT‘ERR  " value was left on the stack."
ELSE
   PRINT‘ERR  " values were left on the stack."
END IF

RETURN;         Return normally from Calculate.File.

::::::::::::::::::::::::::::::::::::::::::::::::::::::::
METHOD CATCH:  Catch any other exceptions.

PRINT‘ERR  "Could not process: ", Input‘Path

IF Input‘Line <> ""
   PRINT‘ERR  "In line", Input‘Line, ": "
              Assertion‘Message
ELSE
   PRINT‘ERR  Assertion‘Message
END IF

RETURN;         Return normally from Calculate.File.
```

```
:  Original assertion in the Calculate.File method.
ASSERT Rpn‘Empty    &NOTE: Stack has remaining elements.
       PASS Rpn‘Size;

:  Modified to use a local handler.
ASSERT Rpn‘Empty    &NOTE: Stack has remaining elements.
       CATCH Leftover;
```

Example 3: A general handler is added to the Calculate.File method in Example 2 to report managed exceptions. An assertion is modified to use an additional local handler to improve reporting. Note that ”PRINT‘ERR” writes to the standard error stream.

4.2 Immediate Caller

Another simple way to dispatch a handler is with the name of a method call that propagates an exception [Appendix G]. The call usually occurs in primary code, but may occur in a handler as well. In the latter case a general handler with a *Retry* flag must also be encoded to permit multiple exceptions in the same event.

```
METHOD CATCH CALL <Method>, ...  :Immediate calls.
```

If the called method has a precondition it can be used to fashion a response by the caller. Immediate callers are in the unique position that they share some variables. Before the invocation the caller can replicate a precondition assertion to override the precondition message, it's fault status, or raise a local exception for specialized handling. Similarly the caller will have access to arguments and public globals referenced in the precondition so it can reference them within its handler.

A compiler can perform interprocedural optimization over preconditions in immediate callers. The caller will have access to parameters referenced in the precondition and the precondition will have no side effects. Unreachable preconditions can be eliminated using static analysis. Preconditions that always fail can be detected at compile time and a warning issued. When debugging keep in mind that static analysis does not apply to a system experiencing corruption.

Note that all calls to the same method are dispatched to the same *Call handler*. When several methods are declared on a *Call handler* they are all dispatched there; reducing the number of handlers. In other cases combining handlers is undesirable and specialized handling might be needed for each of multiple calls to the same method. These can be discriminated based on available

```
: Catch I/O errors in raised in Open.In or Read.Line.
:
METHOD CATCH  CALL Open.In, Read.Line

PRINT‘ERR  "An I/O error occured in: ",
           Assertion‘Context

PRINT‘ERR  Assertion‘Message

RETURN;           Return normally from Calculate.File.
```

Example 4: Extend the Calculate.File method in Example 2 with a specialized immediate call handler to catch and report I/O errors.

context information; which leads into the next section that discusses options for context-based specialization.

4.3 Partial Objects

After an exception one or more methods has been partially executed and objects can be left in an intermediate partial state; which might be invalid. Even though individual objects can be in valid states, based on the semantics of the relationship between objects collectively they can be in an invalid state. Exception handlers local to the method need to either delete invalid objects or perform compensating actions so that objects revert to allowable states.

4.3.1 Partial Objects and Exceptions

An exception state can be specified and encoded in an object so that it may be exported from a partially executed method. A partial object is an object that is in an exception state. Subsequent operations can then manage the exception using allowable partial states.

IEEE Floating Point NaN (*Not a Number*) values are an example of a partial object. After an arithmetic error certain floating point bit patterns are reserved to designate invalid results. Additional storage is not needed as a NaN is encoded by overloading the semantics of existing fields. When a NaN is is used as an input to an arithmetic operation, the result is well defined.

Similarly partial objects can be exported by methods. The consequence is that any method that accepts partial objects as input needs additional logic to process the special cases. Additional code implies additional complexity; which can in turn result in programming errors and maintenance overhead. Programmers have to take partial states into account as they write and maintain code. Checking for special cases incurs a performance penalty on the primary code path as well.

Representing an exception state in a partial object might require additional fields. Incurring a space penalty for extra fields might be undesirable. When an object is large or a program has only a few instances then the space penalty is not problematic. In fact it can be useful for primary code to augment and carry exception reporting information forward. [Example 5] declares a structure File.Path including members Path and Line to carry data forward for use by handlers. Partial objects can also be used to propagate exception state between programs or to save and restore it in persistent objects.

```
: The File.Path class signature declares a structure used to access a file,
: save it's path, and maintain a line count.  It also contains a Signal
: member containing the I/O error status.
:
CLASS File.Path                  &Access a file.
    . File,                      &File pointer.
      Path      STRING,          &Path of the file.
      Line = 0  WORD             :Current line number.

SIGNAL Status = 0  BYTE          :I/O error status.
```

```
: On an I/O error the Signal Status member in the class
: File.Path will be set for the subject variable, Input.
:
METHOD Read.Line SIGNAL:  Read a line from a file.
CHANGE Input  File.Path              :The file to read from.
  EXIT Line   STRING                 :Read and return the next line.
   USE Assertion‘Assertion           :Standard exception class.

READ  Input‘File => Line;             Read a line.
Trim.Outer.Blank  Line;               Remove outer spaces and tabs.
Input‘Line += 1;                      Bump up the line count.
RETURN;                               Primary Read.Line return.

METHOD CATCH:  Only I/O errors reach here.

Input‘Status = STATUS( Input‘File );  Set status code.
Assertion‘Context = Input‘Path;        Report the path.
RETURN‘RAISE;                         Propagate the signal.
```

```
: The original handler introduction in Calculate.File
: catches I/O errors in raised in Open.In or Read.Line.
METHOD CATCH  CALL Open.In, Read.Line

: This variation will catch I/O errors when the variable Input is signaled.
METHOD CATCH  SIGNAL Input

: This version will catch I/O errors when a Signal is
: received for any variable with the type File.Path.
METHOD CATCH  CLASS File.Path
```

Example 5: Signals can be used to propagate and catch exceptions either by variable or class. The signal is declared in a class, raised by a Signal method, and caught by a Signal or Class handler.

Since partial objects arise as the result of an exception, the two are intrinsically linked. For example, the File.Path object encapsulates an I/O file pointer (e.g. Ansi C FILE pointer). Stroustrup [25] uses a similar example to detail the ramifications for exception handling. When an I/O error occurs the File member will be left in a partial state.

Rather than returning File.Path in a partial state to the primary code; when an I/O error occurs the File.Path methods can raise an exception. The benefit is that a File.Path object can only be in a partial state within a handler. Primary code no longer needs special cases to manage partial state. Handlers are responsible for either deleting any partial File.Path object or recovering; typically by invoking a method to close the file.

By containing partial object processing within handlers additional simplifications can be made. First, the state of an exception event can be stored in the subject object (the ‘‘*this*’’ object) and propagated up the call chain. Immediately after an exception there may be one of several partial objects in play. The local handler can then coalesce the composite exception state into the subject. The exception state may incorporate context from other variables as well. The originating partial objects are deleted or reverted back to an impartial state.

As the exception state now contained in the subject object propagates up the call chain through handlers in different classes the subject will change. At each level the exception state needs to be passed from one subject to another by the corresponding handlers. Additional context can be accumulated along the way as well. The partial state goes out of existence after the exception event rejoins the primary code. This is essential so that stale partial objects are not referenced the next time an exception event is processed.

Following this protocol, in the course of an exception event only one object in a partial state is propagated at a time. Leveraging this characteristic the context information for the class can be collected in static class members. Allocated space in the object is no longer needed to represent the context; eliminating the per object space penalty.

In order for a handler to determine which object is associated with the static context, the object reference also needs to be saved in a static class member. An alternative hybrid approach would be to encode a flag in a dynamic member indicating the object is is in a partial state and put the remaining context in static members.

4.3.2 Language Support For Partial Objects

Handlers that coalesce and prune exceptions are not just dependent on the control path, but also access context data. The context available to a handler includes variables accessible by the method containing the handler along with data specific to the exception passed up from lower level methods. This exception specific context is short lived as it is only needed for the lifetime of an exception event. As it propagates up the call stack, the exception context will often be transformed into new context variables and the old data discarded. The different context variables pop in and out of existence for only brief periods in the course of an exception event.

A new category of class member called *Signals* is introduced to hold exception context. *Signal members* in a class can only be accessed from handlers and have a lifetime that covers a portion of an exception event. They are instantiated when a method uses that class as it's subject and propagates the *Signals*. Handlers in the sending method can set *signal members* and receiving handlers can read them. Once a method does not propagate a *Signal* then it is destroyed [Appendix F]. The File.Path class in Example 5 contains a *Signal member* named Status.

[Example 5] shows how *Signals* are used in the experimental language. Handlers with the *Catch Signal* clause field variables that are signaled. The *Catch Class* clause fields any signaled variable of a given type. Handlers that catch multiple variables can use the *Signal function* to ascertain which variable is set. This is is essential if the signaled object is an element in an array.

4.4 Exception Testing

Regression tests for exceptions can be used to associate an exception event with a handler. Each test establishes circumstances that lead to an exception. The handler that is to respond designates the test that caused the exception. The handler can be in the same method as the exception point or several levels up the call stack. In a test run the exception event and call stack up to the handler is recorded. In the production system, when the recorded exception event is detected, the corresponding handler is invoked.

This technique can be combined with Test-Driven Development. Leitner et al. [13] observes that developers create ad-hoc tests as they write code. These test cases can be captured and used as a regression test suite. The developer is in the best position to perform white box analysis to identify which exception cases are relevant and how to create a test to reach the exception point. This is

especially poignant when trying to fully cover exception cases as they can be obscure and easy to overlook.

Rather than recording the call path of an entire exception event, only the exception point and the associated handler are used to dispatch test handlers. Each test handler can then be assured to cover all alternate paths to the exception point. For example a component may use different code paths to process an even or odd input. A test that only covers the even case will miss the odd case. Since the component can be opaque it may not be apparent that distinct code paths are used.

The limitation of this approach is that a test may not detect all exception points that have been combined. For example a component may use different code paths to process an even or odd input. A test that only covers the even case will miss the odd case. Since the component can be opaque it may not be apparent that distinct code paths are used. Unless the component designer merges the distinct exceptions by passing them through a single handler then exception coverage can be incomplete.

Test paths that are not unique will be caught in a test run. Should a collision occur the programmer will need to either relocate one of the handlers up or down the call chain or combine the handlers [Appendix I]. Since tests can establish state to uniquely trigger colliding cases, that state can instead be forwarded to composite handlers to discriminate between them.

4.4.1 Language Support For Exception Tests

For testing a program is built that runs each of the unit tests for exceptions. As they are run the exception point reached in each test is collected in an array. The index into the array corresponds to the name of a single test case. The exception point is available to the First Chance Handler via the parameters Method (name of the failed method) and Line (assertion line or instruction offset). A test version of the First Chance Handler records the exception points.

After the exception point is recorded, the call stack is unwound until the method is reached containing a handler that declares the test name. In the experimental language the syntax to introduce a test handler is:

```
METHOD CATCH TEST <Test Name>
```

If in the course of the exception event, no such handler is reached then there is no handler associated with the test. A warning is issued so that the programmer can remedy the situation.

Collisions between tests are also detected here. After all tests are run, a completeness check can also be performed to ensure that all handlers that catch tests are reached.

The output of the test run is a module containing an array of exception points and handling methods ordered by an enumeration over the test names. A production program build will then use this table to generate code to invoke handlers. As the call stack is unwound during an exception event a check is made at each level for any test handlers. Each test handler is checked against the table for a corresponding exception point. On a match control is passed to the test handler.

A test version of the program needs to be built in addition to the production version. The test version does not need to be fully rebuilt, but just relinked with aditional test modules. When performing an incremental build the table only needs to be reconstructed when tests are changed or handlers referenced in the table are relocated. In this case the test version is relinked and rerun to produce a new dispatch table.

```
METHOD Underflow:   Test to cause Stack underflow.

Calculate.Line  "x =";        ERROR: Stack underflow.

RETURN;                       Should not reach here.
```

```
:  This handler is added to the Calculate.Line method.
METHOD CATCH TEST Underflow:  Catch an underflow.

Assertion‘Message =         &Assign a new message.
   "ERROR:  Underflow on the arithmetic stack."

RETURN‘RAISE;                Propagate the exception.
```

Example 6: Use a Test to associate underflow with a specialized handler. The Underflow test causes a stack underflow detected in the Pop method declared in Example 1. Using Example 2 the Underflow handler is placed in the Calculate.Line method; which is called from Calculate.File. A test run associates the handler with the underflow condition. When undeflow occurs the Underflow handler will catch it, reformat the message, and propagate the exception up to Calculate.File.

4.4.2 Programming With Exception Tests

Using tests to designate handlers for exceptions maintains both lexical and semantic independence. The handler is reached directly instead of explicitly chaining an exception from one handler to the next up the call stack. Code to propagate the exception is not needed. Since the exception event is specified by a test, changing a low level component will not require changes to the test or handler.

Exception tests must be well behaved since they are integrated into the build process. They should run reasonably fast and work in a test sandbox environment. External events such as I/O failures need to be either emulated or avoided. As developers write code they can also write simple exception tests and benefit immediately by using them right away.

Since this technique requires the additional test builds and runs, it might be best to restrict it to relatively benign handlers. A *Reforming handler* is one that replaces a lower level message with a more meaningful and semantically rich message. Rich messages are for the benefit of users and are not required during development. By restricting exception tests to *Reforming handlers*, the control paths of a system under development are assured to be complete. Also note that the test suite only needs to be rerun when adding new tests and handlers.

Mapping one of many possible exception points to a handler is central to programming with exceptions. If a component developer combines multiple exceptions into a single event then they can not be discriminated. Conversely, propagating too many exception states is difficult to manage. Programmers can easily overlook or misidentify exceptions. The testing technique can discriminate exceptions without explicitly identifying or propagating an exception.

Using tests to associate exception events establish an ephemeral control path. This might be considered a strength or weakness. It's strength is that the test case nor the handler is affected by changes to intermediate methods. The weakness is that there is no explicit encoding associating the two that can be traced by simply by looking at the code. Until more experience is gained using this technique, it is unclear whether or not this would be problematic.

To help compose exception tests the all exceptions propagated by public methods needs to be detailed. Preconditions and *Signal members* are explicitly specified. Any unspecified exceptions can be determined by collecting assertion messages that might be propagated. Component developers will still need to identify propagated exceptions so that a complet set of tests can be written.

4.5 Exception Mechanism Performance

Ideally an exception mechanism should impose no performance overhead on primary code in a compiled program. By convention compiler developers presume that exceptions are rare occurences and can sustain mechanism overhead. Pushing overhead for exception management into exception events not only ensures the primary path executes fast, but encourages the use of exceptions when they can be used to avoid performance penalties. For example, using an exception can be faster than explicitly checking return codes after invoking a method.

Even if the exception mechanism itself imposes no overhead, detecting exceptions occurs on the primary path and has unavoidable performance costs regardless of the implementation. The mechanism proposed here uses assertions to detect exceptions. Assertions offer several opportunities for optimization that can reduce the cost of detecting exceptions.

Conditions are read-only so machine instructions can be freely reordered with respect to surrounding statements. Preconditions and assertions at the end of a method can be relocated using interprocedural optimization as well. A store that occurs after an assertion condition can be reordered before or within the assertion condition. In this case should the exception be raised then the original value needs to be restore for semantic consistency.

Since assertions have no side effects other than raising an exception they can be removed if static analysis shows they are unreachable. This may not be desirable for assertions that detect program faults in concurrent threads. An assertion may be reachable in a race condition, but unreachable in a serial execution.

When an assertion condition fails instructions can be ordered such that branches are taken out of the primary path. This prevents the instruction pipeline from stalling along the primary code path. Furthermore, a context expression on an assertion only needs to be evaluated after the branch is taken. Syntactically distinguishing assertions from other conditional statements provides the compiler with an optimization hint so to apply these optimizations.

For certain language constructs detecting faults resulting from a violation of language semenatics entails overhead. For example, assertion conditions are not permitted to raise an exception. Static analysis can not always ensure this, so run-time checks must be added. Where performance is a concern, checks for violations that incur overhead can be enabled only in code under development and disabled in production systems.

```
In primary code for a method:
   Detect an exception.
   IF detected in hardware,
      Filter it through the Machine Check Handler.
   ELSE
      Evaluate the context string.
   END IF
   Invoke the exception Manager.

In the exception Manager:
   Invoke the First Chance Handler.
   Go to unwinding code in the method.

In the method's unwind code:
   IF a METHOD CATCH <Local> then run it.
   ELSE IF a program fault,
      IF a METHOD CATCH FAULT then run it.
   ELSE a managed exception,
      IF the retry guard fails,
         Invoke the exception Manager with a fault.
      ELSE IF a METHOD CATCH TEST then run it.
      ELSE IF a METHOD CATCH SIGNAL then run it.
      ELSE IF a METHOD CATCH CALL then run it.
      ELSE IF a METHOD CATCH CLASS then run it.
      ELSE IF a METHOD CATCH then run it.
      END IF
   END IF
   Run destructors for the method.
   IF rejoining the primary code,
      Return from the method.
   ELSE propagating the exception,
      Resume unwinding in the exception Manager.
   END IF

Resuming in the exception Manager:
   Unwind a stack frame.
   IF at the top of the stack,
      Invoke the Last Chance Handler.
      Terminate the thread or program.
   ELSE propagating the exception,
      Go to unwinding code in the next method up.
   END IF
```

Figure 3: Control Flow Of An Exception Event

Machine checks efficiently detect certain numeric exceptions such as division by zero. Floating point units can choose between raising an exception and providing a masked response by generating a NaN. The only overhead is establishing processor control settings to enable or disable detection. Ideally this is done outside performance critical code sections.

The proposed mechanisms use the exception Manager to maintain state for an exception event. The state is stored in Thread Local Storage so that events in different threads can be managed separately. In some cases the mechanism creates internal local variables allocated on the stack upon entry to a method. However these internal variables do not require initialization so they can be created without performance overhead.

Programmers can also explicitly declare local variables in exception handlers. In cases where they need to be initialized, the initial value does not need to be stored before entering primary code. Instead they can be initialized upon entry to the handler. When a handler is allowed to be retried they need to be set only when the handler is run the first time. First use can be determined by checking the retry guard stored in and maintained by the exception Manager. In practice concern about initialization overhead is unlikely, but is mentioned for completeness.

Exception events are allowed to be nested. That is a handler can invoke a method that raises a separate nested event. The nested event can then rejoin the primary code and safely return to the handler. When an exception occurs within an exception event in progress the Manager keeps a stack of nested event frames. Stacked frames can be heap allocated and do not need to be located in a methods local context; which could incur overhead.

4.6 Exceptions in Threads

Threads are subject to performance constraints intended to minimize the time that other threads are blocked. Any critical path; including exception events need to be scrutinized for worst case performance. Exceptions originating from all paths go through the Exception Manager and the First Chance Handler; whether or not they are subject to any performance constraints. Other exceptions detected by hardware will go through the Machine Check Handler as well.

The Exception Manager establishes initial state and passes control to the First Chance Handler. For use in performance critical threads, this initial code segment needs to be optimized and thread safe. The First Chance Handler is provided by the programmer, so it's performance can be tailored to the system. It can be written such that a specialized path is taken to perform restricted operations; enabling it to respond quickly. Non-critical events can be directed to a more general handler that is allowed to incur additional overhead.

All run-time exception components also need to be thread safe. The Exception Manager in the experimental implementation uses *Thread Local Storage* to store state for each active exception. Upon entry to a thread, space for *Thread Local Storage* is pre-allocated on the thread's stack and is used to store exception state independently for each thread.

If a handler propagates a *Signal*, space is allocated for it on the heap. A pointer in the Exception Manager is set to reference the currently active *Signal*. The pointer is allocated using *Thread Local Storage* along with the other state variables in the Exception Manager. As each handler unwinds the pointer is reset to reference the newly propagated *Signal*. This technique not only ensures thread safety, but allows exceptions to be nested as well.

The Machine Check Handler passes control through the operating system and is a blocking call. Any performance analysis is subject to the implementation of the operating system. For this reason it might be easier to avoid using machine checks to detect exceptions rather than including operating system paths in the performance analysis. This is an unusual case in that hardware support can actually be detrimental to performance.

Critical locks are only allowed to be held for a limited amount of time to avoid blocking other threads. Reasoning about locks is hard enough, but reasoning about them through all potential exception paths is not just difficult, but an error prone process.

When locks are released by destructors, the stack will need to be unwound to a frame above any critical locks before performing operations that could cause excessive blocking. For instance, it's common to have a handler prompt users to determine a course of action. Performing I/O (particularly user input) before critical locks are released can block the rest of the system. If the exception event is rare enough this might go unnoticed until after the system is fielded.

Output operations might avoid long lock delays if records are merely queued for output later. For example, a threaded implementation of the Debug command (see section 2.2) might multiplex output strings in a queue shared by other threads. A separate thread could then perform the actual output operation. This would enable debug traces to be used in a multi-threaded system with modest overhead.

This strategy is generalized by [11] to perform asynchronous exception propagation between threads to avoid blocking urgent exception. Their proposed mechanism gives priority scheduling to

exceptions originating from an asynchronous raise statement. They present formal semantics that safely unblock a propagating task to ensure a timely response.

5. Exceptions in Atomic Sections

Atomic sections [14] were introduced as a means to automatically manage locks between threads. Shared variables can be updated by a sequence of operations in a thread while other threads are prevented from making modifications that would interfere.

Transactional Memory is a mechanism that implements atomic sections by implicitly performing fine grained locking over shared variables. There are many proposed variations of Transactional Memory involving both software and hardware implementations along with supporting language constructs. A consensus on a semantic model and implementation for Transactional Memory has yet to be been reached. However, this is an active area of research and much progress is being made. This section speculates about possible semantics for exceptions in transactions.

5.1 Restricted Exceptions in Restricted Transactions

A method can be invoked from within atomic sections or in serial code. It is desirable for program semantics to be the same within or without atomic sections [8] to make use of available software components and increase opportunities for reuse. While it might simplify matters to restrict exception semantics so that they are not permitted or reachable within atomic sections; this is likely to conflict with programming conventions for sequentially executed components.

The advantage to adopting restricted signatures for atomic sections is that restrictions can be leveraged by a transactional mechanism. Evaluation of preconditions is a read-only operation. An atomic method with a failed assertion in a precondition can terminate the section immediately before transferring control to the Exception Manager. After termination exception processing can proceed serially to propagate the exception out of the atomic section. The transaction can be optimized taking the read-only mode into account.

Program faults are unavoidable and any transactional model must at least survive faults. There is no reason then to restrict programmers from detecting and raising program faults. Although faults may be rare they still need to be considered in performance analysis. A plausible way to restrict faults is to limit them to precondition checks and then demonstrate that faults can not occur in the remainder of the transaction.

Pure methods and functions and *Precondition Pure* methods are correlated with the side effect free semantics of restricted transactions. Disallowing exceptions in atomic sections altogether and using only pure methods would be possible, but onerous for programmers. The programmer would need to take care to never violate any preconditions. This is contrary to the intent of using atomic sections to make programming easier. Eliminating exceptions would make the resulting code undesirable for reuse in serially executed programs.

5.2 Unrestricted Exceptions In Restricted Transactions

Unrestricted exceptions permit an exception anywhere within a transaction. At the start of an exception event the method where it occurs will be partially executed. Should the exception propagate out of the method, objects it has modified can be left in a partial state. In transactions shared objects left in a partial state are of particular concern.

One approach is to roll the transaction back [10]; in which case objects modified in the transaction automatically revert to their original state. While this may provide an assurance of safety [23], rollback semantics are inconsistent with exception semantics for serial executions. When a method that propagates an exception completes, any external state that has been changed is committed. Run serially, no transactional facilities are engaged and it is unable to undo the operation.

On the other hand, unrolling a transaction after a program fault is reasonable since faults leave the program in an indeterminate state. A program fault results in a partial execution, but the transaction is uncontrollably disrupted and never completes. Unrolling provides the assurance of safety for shared objects in a transaction. A serial execution can not roll back, but after a program fault the state is undefined so there is no semantic conflict. Processing faults separately from other exceptions makes it possible for a translator to manage faults differently. Unless rollback semantics are employed the transaction could leave shared objects in partial states that would interfere with other threads.

A transaction is complete after a managed exception and needs to be committed. In a serial execution one strategy is for the caller to detect and clean up any invalid partial state. This practice violates the principle of encapsulation since the cleanup code occurs over a method boundary. Ideally a handler for the partial method should clean up any partial state itself. In serial executions

this is just good programming practice, but transactional methods require handlers to clean up before terminating.

Internal exception mechanisms must be able to operate within atomic sections. The Exception Manager does not share state between threads so this is not a problem. When fielding exceptions the First Chance Handler is run by both transactional and nontransactional code. The mode can be determined early on and exceptions in transactions can take a restricted code path. If the First Chance Handler accesses any shared objects, it must be instrumented for transaction processing.

The Machine Check Handler passes control through the operating system; which is a blocking operation. In some situations this might cause the transaction to be suspended. The transactional mechanism itself has to survive machine checks and suspended transactions. In the situation where a machine check is not a program fault, the transaction can not be aborted. For example, if a zero divide is treated as an exception and not a fault; this needs to be supported by the transaction mechanism.

5.3 Unrestricted Exceptions In Unrestricted Transactions

Unrestricted transactions allow atomic sections to contain any programmed action [1]. The consequence of generalized functionality is that the benefit of fine-grained concurrency control is sacrificed. Programmers will have to perform a thorough analysis to take the performance implications into account. Mechanisms supporting unrestricted transactions will invariably resort to nontransactional (blocking) locks.

Unrestricted transactions can perform irrevocable system calls and I/O. Once a transaction has performed an irrevocable action, termination is not an option. Should an exception occur the transaction will terminate and be partially completed. Serial programs face the problem of managing interactions between exceptions and irrevocable actions as well.

Managed exceptions that occur after an irrevocable action complete the transaction and can be managed as they would in a sequential execution. Program faults, however, result in an incomplete execution of a transaction. This is a highly specialized situation, but one that occurs in fault tolerant systems. That is; a software error has occurred and was detected in a concurrent thread after performing an irrevocable operation.

One solution is to let variables remain in a partial state with respect to irrevocable actions that have already been performed. For example a count of occurrences of an irrevocable action may not be properly updated. Shared variables that can not be returned to a valid state can instead encode a partial state. Operations on these variables can have cases to cover these situations.

Alternatively, notifications can also be issued corresponding to irrevocable actions. While the actions can not be undone a notification can log the event or inform an operator. Notifications can either be performed by destructors or by fault handlers. In the case of a fault handler it needs to be run before clearing the transaction state so the log can be accessed. Destructors are run in primary code and could introduce overhead if they are used to detect partial states.

Some implementations of Transactional Memory permit data to become inconsistent [9]. Stale data and data updated by another thread can comingle, but only in a transaction that is doomed to abort. Inconsistent data can unintentionally lead to program faults and machine checks. If inconsistent sates are allowed, before performing any exception action the transaction is checked to see if it is doomed. Likewise, inconsistent data in unrestricted transactions needs to be detected before performing any irrevocable operation.

6. Future Work and Conclusion

Evaluating language constructs as fundamental and complex as exception handlers requires extensive system building experience. Test-driven exception handling was retrofitted to an existing program, but should be applied to a test-driven development project from the start. *Signal members* holding exception context has had limited use, but has been run successfully in concurrent threads. *Signals* should be tried in more programs to see if the constructs need to be extended.

Exceptions in transactions are highly dependent on the transaction model. The proposed constructs need to be applied to a transactional programming system in order to determine their effectiveness.

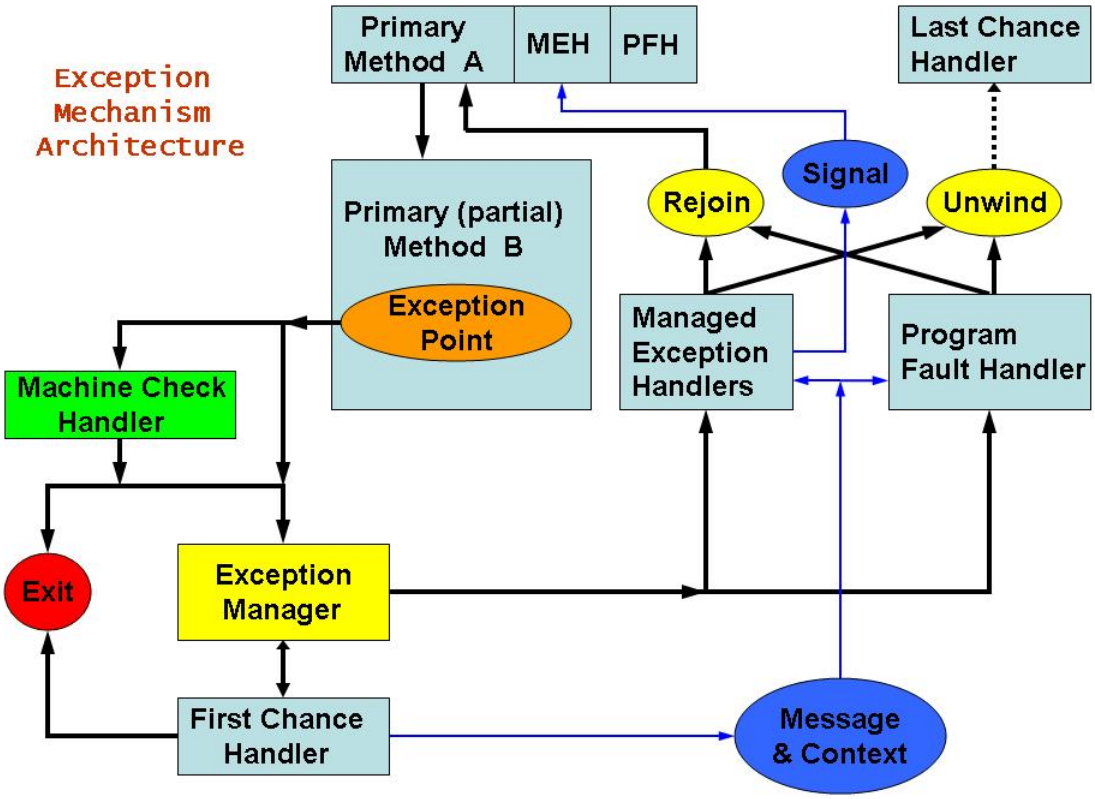
The language design challenge for exceptions entails devising constructs that closely match intended usage. Using anonymous assertions obviates the overhead of managing a global naming taxonomy for exceptions. The combination of global scoping and static organization leads to naming conflicts and programming errors. Names can be overloaded or have conflicting semantics; often

because programmers are tempted to do what is expedient. Adding new names becomes intractable in large evolving systems.

In a program under development an assertion can act as a reminder for a case that remains to be handled. Assertions can be added and removed without changing control structures and nesting levels in a method. Using assertions liberally provides integrity checks that help programmers track down errors well before they become symptomatic. A programmer can get the primary code working and then add handlers as the system matures without making significant changes to the primary code.

Separating programs into primary code, fault handlers and handlers for managed exceptions follows a natural partitioning. Experiments using the proposed language constructs have resulted in effective exception handling capabilities. Only a few simple handlers were required in substantial programs; which demonstrates that exceptions can be handled well without undue complexity. Reliable software can not be written unless exception cases are covered and properly managed.

A. Event Architecture



An exception is detected and raised at an Exception Point. If it was detected by hardware control is passed to the Machine Check Handler. A default Machine Check Handler can be used or one can be providded. It decides if the exception is a Program Fault or a Managed Exception. When completed it can either terminate or invoke the Exception Manager.

All other exceptions pass control directly to the Exception Manager. If it was detected by an assertion then it will be flagged as either a Program Fault or a Managed Exception. If a violation of language semantics was detected then it is a Program Fault.

The Exception Manager recives an exception message and context string, location information, and a Program Fault flag. Early on it invokes the First Chance Handler. The default version does nothing, but a version is provided to save the parameters to the Exception Manager for subsequent handlers to access. The First Chance Handler can also be overridden. The replacement can examine the exception and decide if it should terminate and can also set or clear the exception's Program Fault status.

Next the Exception Manager unwinds the program call stack. At each frame the exception is dispatched to a handler if there is one present certified to receive it. After a handler is run it can either rejoin the primary code or continue unwinding the next frame. A handler that rejoins behaves as if the method performed a return from its primary method. Handlers for Managed Exceptions can optionally propagate context in a Signal object.

If the stack is unwound out of the program or thread then the Last Chance Handler is invoked. The default version prints a warning message. It too can be overridden by the programmer.

B. Program Faults

B.1 Out of Memory

Normally there is plenty of memory available on a computer to run common applications. With the advent of virtual memory and inexpensive disk drives used for paging memory is bounded by the size of the address space. Address spaces over 32 bits have pushed this limit beyond what can reasonably be consumed even by memory intensive programs.

Still low memory conditions can occur. The amount of memory available depends on the deployment platform and is usually not under the control of the program. For example, memory limits may be exceeded depending on the utilization of swap space by other programs running on the same computer. Certain program errors can result in unintentionally consuming memory until a limit is reached. It requires memory to handle an out of memory condition, so care must be taken to ensure the handler consumes very little additional memory.

B.1.1 Heap Overflow

Dynamically allocated data is allocated in a region of memory called the heap. Programmers can explicitly allocate heap memory or an object can implicitly be allocated in code generated by a translator or by the run-time system. Implicit allocations can potentially occur anywhere in a program. Explicit allocations that can not be honored can either be managed by the requestor or by raising a program fault. Arguments for the general heap overflow conditions are:

```
Fault   - Memory.Heap
Context - "Memory Heap"
Message - "CHECK:  Memory heap overflow."
Method  - Name of the method with the allocation.
Line    - Offset to call to the allocation method.
```

Strings or other dynamic objects that are intrinsic to a programming language or supporting library can be given distinct arguments for more specific responses. This case describes arguments for strings that can not be allocated. Note that this exception might also be raised should a string exceed the maximum allowed size.

```
Fault   - Memory.String
Context - "Memory String"
Message - "CHECK:  String allocation overflow."
Method  - Name of the method allocating the string.
Line    - Offset to call to the allocation method.
```

B.1.2 Stack Overflow

The stack is used to store local variables and information used to maintain the program‘s call stack. It is extended each time a method is called or a nested scope is entered and shrinks upon exit. The stack is managed in code generated by a compiler or within the language interpreter and not explicitly by the programmer. Some times mechanisms are provided for a programmer to explicitly extend the stack, but this is typically confined to low level programming.

```
Fault   - Memory.Stack
Context - "Memory Stack"
Message - "CHECK:  Memory stack overflow."
Method  - Name of the method with the failure.
Line    - 0
```

A specialization of the stack overflow exception can occur at the entry to a method when a new stack frame is established. For example, a method containing a local variable that is a large array might require more memory than is available. In this event the new stack frame can be partially constructed. The current program location is uncertain as it is in a method that does not have an established frame on the call stack.

After failing to establish a stack frame any partial frame needs to be destroyed. If desired it is possible to determine the name of the method from the context available as the entry point to the method can be extracted by a post mortem analysis. It is possible that if the stack is out of space a stack frame for the First Chance Handler can not be constructed either. The run-time system needs to be able to detect and avoid this situation as well.

Should the programmer decide they want to propagate this exception up the call stack, the lowest catch reachable is in the method that called the method that failed. A catch in the method with the failed frame will not be reached.

```
Fault   - Memory.Frame
Context - "Memory Frame"
Message - "CHECK:  Method stack frame overflow."
Method  - The name of the method with a bad frame.
Line    - 0
```

B.2 Programming Checks

Programs can contain checks to see that they are functioning according to the intentions of the programmer. When one of these checks fails a programming error is detected. In this case the program is considered unstable and special consideration is needed should an attempt to safely recover be taken.

Ideally these self-checks should only occur while debugging a program. However, these checks may be optionally be left in a production program and are are still reachable when an undiscovered mistake remains in the program.

Since performance is not an issue when running exception handlers checks can be left in handlers even in production system. An advantage of lexically separating handlers from primary code is that this distinction can be made by the translator.

As a program reaches an unstable state due to a programming error it is likely to wind up in a handler. If checks are enabled in the handler the programming error is likely to be caught earlier and managed appropriately. More meaningful feedback might be given to the users and developers. This might help prevent a corrupt system from destructively propagating errors.

B.2.1 Guarding Against Exceptions

Exceptions can be disallowed at certain specified points in a program. When an exception is raised and propagated to these guarded points a program fault is raised. In such cases there are two exceptions processed; the original exception and the fault raised by the guard. The arguments to the First Chance Handler contains some of the state information for the original exception, but not all of it. To retain all arguments of the original exception the First Chance Handler can save the arguments on the previous call.

When a program fault is raised while handling a program fault, this can indicate the program is in an unstable state. Furthermore, if it is not processed explicitly it can result in runaway recursion within exception handlers. Note that policy for a non-fault raised within a program fault is at the discretion of the programmer.

To detect this condition, the run-time system can flags the start of a program fault. Should another program fault occur this condition can be detected. After a fault handler recovers and rejoins the primary code path the flag can be cleared. Another subsequent program fault is allowed after that.

A method can be annotated to specify that it should not raise any exceptions or only a set of specific exceptions. Should an exception be raised anyway that violates this constraint then a programming error has occurred. Note that in this case the developer can choose to let the method raise the Exception.Safe exception. However, if it is caught and handled by the First Chance Handler this will not be done. Presumably the original exception is not a program fault.

```
Fault   - Exception.Safe
Context - Original exception
Message - Original exception
Method  - Name of the method containing the guard.
Line    - 0
```

An exception handler can detect when an exception occurs while handling exceptions. An annotation can be provided to defeat this guard in cases where retries are desired. Since failing to manage this condition explicitly can result in a loop, the default notation (no annotation) should guard against it. Note that within a handler an exception can be handled within a called method without violating the guard.

```
Fault   - Exception.Retry
Context - Original exception
Message - Original exception
Method  - Name of the method with the CATCH.
Line    - Line number of the CATCH with no retry.
```

Should a fault occur within a fault handler this indicates that the program is failing to recover. These are detected by the exception Manager and a Fault.Retry exception is raised. The First Chance Handler is called; which has the opportunity to clear the double fault condition. If it is unchanged the Last Chance Handler is called. If that call does not terminate and instead returns then the program or thread is terminated.

```
Fault   - Fault.Retry
Context - Original exception
Message - Original exception
Method  - Name of the method with the CATCH.
Line    - Line number of the CATCH with no retry.
```

B.2.2 Assertion Fault

The programmer can check failures with assertions that should not fail when the program is running properly. In the experimental language these are denoted by using a Fault clause in an assertion. A translator could use static analysis to skip provably true Assertion checks in production code to improve performance.

```
Fault   - Assert.Fault
Context - <user defined>
Message - "<user defined>"
Method  - Name of the method with the assertion.
Line    - Assertion line number.
```

Invarients can also be implicitly inserted by a translator to help developers locate errors based on language semantics. Common examples are arithmetic overflow and array bounds checking. These checks are normally not enabled in production systems as they can significantly reduce performance without the programmer's consent. In fact, at times they may slow down performance to unacceptable levels even when debugging code. If so, a language annotation could be included to selectively disable implicit debugging checks for performance sensitive code.

```
Fault   - <language defined>
Context - <language defined>
Message - "DEBUG:  <language defined>"
Method  - Name of the method containing the check.
Line    - Line number of the statement that failed.
```

Objects can contain invarients to ensure that they are in a valid state before and after they are modified. Object invarients are applied to public methods. Private methods may perform only part of a transition and can leave the object in an invalid state. At certain points they might want to explicitly check the validity of an object.

B.2.3 Invalid Pointer Access

Failure to dereference a pointer is also a programming error. An attempt to a access location zero is always a failure and the Fault parameter is Address.Null. If an attempt to access the low 64KB of memory fails then Fault is set to Address.Low. This distinction is made because a convenient way of encoding additional enumerated states in a pointer besides zero is to use the low 16 address bits.

Unmapped memory is memory that has not been mapped into the virtual address space for a process. It can not be accessed because it does not yet exist. If a pointer to unmapped memory is dereferenced it fails and Fault is set to Address.Unmapped. Finally some pages of memory can be reserved and access results in a memory protection (Address.Protect) exception.

```
 Fault  - Address.Null
          Address.Low
          Address.Protect
          Address.Unmapped

Context - The address as a hexadecimal number.

Message - "CHECK:  Tried to address location 0."
          "CHECK:  Tried to address below 2^16."
          "CHECK:  Tried to access protected memory."
          "CHECK:  Tried to access unmapped memory."

 Method - Name of the method with the invalid access.
 Line   - Offset to the failed operation.
```

B.2.4 Arithmetic Error

Arithmetic exceptions can be allowed and handlers provided for recovery. They can also be disallowed and result in a programming error should they occur anyhow. Floating point operations can alternatively return NaNs instead of raising an exception.

```
 Fault   - Integer.Divide
           Float.Divide
           Float.Overflow
           Float.Underflow
           Float.Inexact
           Float.Invalid

Context  - "Integer Divide"
           "Float Divide"
           "Overflow"
           "Underflow"
           "Inexact"
           "Invalid"

Message  - "CHECK:  Integer divide by zero."
           "CHECK:  Floating point divide by zero."
           "CHECK:  Floating point underflow."
           "CHECK:  Floating point overflow."
           "CHECK:  Floating point inexact result."
           "CHECK:  Floating point invalid operand."

Method   - Name of the method with the failure.

Line     - Offset to the failed operation.
```

C. Calling Impure Methods From Pure Methods

Conditions require pure functions to invoke a method. However, impure methods can not be called from a pure functions or methods since an exception can be raised. This can lead to usage conflicts making it difficult to invoke some methods in conditions or pure methods. There are several ways impure methods can be called in conditions:

- Design component classes to provide pure public methods.

  Take exception behavior into consideration when designing methods in reusible components. If they are likely to be used in conditions then the method will need to be pure. Items such as iterators over containers fall into this catergory.

- An exception can either be managed or a fault.

  Determine carefully whether an exception should be written as a managed exception or a program fault. Faults are allowed in pure methods, but are treated as programming errors. A managed exception is more versitile. It can be raised for either a fault condition or managed exception depending on how it is used by client code. Within a condition any exception is interpreted as a fault.

- Provide an alternative pure version of a method.

  If it makes sense for a method to raise exceptions, alternative versions can be provided. One version can raise exceptions and another can be pure. Exceptions in the pure version would raise a fault or return status information. For example, a pure method can return either a pointer to an object or a pointer set to a status code (typically null).

- A client can use a pure wrapper method to invoke an impure method.

  When source code for a component method is unavailable, changes can only be made outside the component class. A wrapper can catch any exceptions raised and demote them to faults. How this is done is highly dependent on language features. A friend or virtual method would be suitable. A new language feature to explicitly declare a pure wrapper could also be devised.

D. Case Study Using Retry

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
: This is sample code extracted from a text editor.
: It illustrates a Recovery Handler with Retry capability.
: When an I/O occurs writing an edit session to a file, the editor
: attempts to write the file in a safe temporary directory.
: A subsequent error causes the method to give up writing and resumes editing.
:
METHOD Write.File:  Write the current text session to the named file.

CHANGE Session,                    &Current edit session.
       File       STRING           : In - Tame Slack path of the file.
                                   :Out - Unchanged or saved recovery file.

 LOCAL Id         File,            &File Id.
       Os.Path    STRING,          &Raw file path.
       Info       Path.Info,       &File path permissions
       Position   Edit.Line,       &Current position in the edit session.
       Retry = "" STRING,          &Set when a failure durring a second attempt.
       Unwind     Bit              :Set if throwing an excption up level.

   USE Edit.Command‘Last.Write,
       Tube.Display‘Height,
       Tube.Display‘Clear.Screen
:
:.....................................................................
```

```
Get.Position  $, Position;             Save the current edit line position.
Slack.To.Os.Path  File, Os.Path;       Logical path to OS specific path.

IF $Encrypt:                           IF encrypting the target file,
   Write.File.Encrypt  Os.Path, Id;       Setup an encrypted write.

ELSE:                                  ELSE writing plain text,
   OPEN‘OUT  Id = Os.Path;                Open the output file.
END IF;                                END IF

IF @Id‘Id:                             IF the user did not cancel,
   Write.File.Body  $, File, Id;          Write to the target file.
   Set.Position  $, Position;             Return to the current position

   Path.Info  Os.Path, Info;              Get the file creation time.
   $Time    = Info‘Date;                  Save it for this session.
   $Changed = 0;                          Reset the session change indicator.
END IF;                                END IF

RETURN
```

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
METHOD CATCH RETRY:  Catch an exception while writing an edit session.
:
:.............................................................................


Unwind = 1;                              Just unwind unless a write error.

IF Id'Open:                              IF the target file was opened,
   CLOSE  Id;                               Close the file that failed to write.

   IF Retry = "":                            IF the primary write failed,
      Temporary.Directory  Retry;               Find a safe temporary directory.
      Retry != "save.me";                       Name of the temporary file.

      Out.Tube     Clear.Screen;             Draw a blank screen.
      Cursor.Tube  5

      Out.Tube  "*****************************************************************^J"
      Out.Tube  "***  A unrecoverable write error has occured while saving:^J"
      Out.Tube  "***      " ! File ! "^J"
      Out.Tube  "***^J"
      Out.Tube  "***^J"
      Out.Tube  "***  The file has been truncated and data has been lost.^J"
      Out.Tube  "***      Attempting to save it in ...^J"
      Out.Tube  "***          " ! Retry ! "^J"
      Out.Tube  "***^J"

      Slack.To.Os.Path  Retry, Os.Path;      Logical path to OS specific path.

      IF $Encrypt:                           IF encrypting the target file,
         Write.File.Encrypt  Os.Path, Id;       Setup an encrypted write.

      ELSE:                                  ELSE writing plain text,
         OPEN'OUT  Id = Os.Path;                 Open the output file.
      END IF;                                END IF

      IF @Id'Id:                             IF the user did not cancel,
         Write.File.Body  $, Retry, Id;          Write to the temporary file.

         Out.Tube  "***  ... It was saved okay.^J"
         Out.Tube  "***  You need to repair the corrupted file.^J"

         File = Retry
      END IF;                                END IF

   ELSE:                                  ELSE an error writing a temprary file,
      Out.Tube  "***  ... It could not be saved.  You should try and save it.^J"
      File != " was corrupted."
   END IF;                                END IF
```

```
   Out.Tube  "***^J"
   Out.Tube  "***       Hit any key to resume editing.^J"
   Out.Tube  "***^J"
   Out.Tube  "*************************************************************^J"

   Get.Key;                                Wait until the user hits a key.
   Refresh.Screen;                         Redraw the edit session screen.
   Cursor.Tube   Height;                   Cursor to the bottom of the screen.

   Set.Position  $, Position;              Reset to the current edit position.
   Unwind = 0;                             Stop unwinding and return.
END IF;                                  END IF

RETURN  Unwind;                       Exception if Unwind = 1; else normal return.
```

E. Expanded Version of the Running Example

```
\small
<<< begin file:  calculate.file.b >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
:  A safe method does not propagate exceptions, but can propagate faults.
:
METHOD Calculate.File  SAFE:  Read a calculation file and perform arithmetic.

 ENTRY Path       STRING,    &A relative path name for the input file.
       Trace = 0  Bit        :Write a trace of operations.

 LOCAL Input   File.Path,    &Wrapper for a File pointer, Status, and Line number.
       Line    STRING        :Content of the current line.

   USE Assertion‘Assertion   :Access exception status in a stardard library.
:
:.....................................................................
```

```
ASSERT PRECONDITION;                    ASSERT
   Path  FAULT;                            CHECK:  No input file path was given.
END ASSERT;                             END ASSERT

Open.In  Input, Path;                   Begin reading the input file.

DO UNTIL  Input‘File‘End:               DO UNTIL at the end of the file,
   Read.Line  Input, Line;                 Read a line of text.

   IF Line:                                IF not an empty line,
      Calculate.Line  Line, Trace;            Compute based on the line.
   END IF;                                 END IF
END DO;                                 END DO

ASSERT  Rpn‘Empty  CATCH Leftover;      ERROR: Stack has remaining elements.

DRAIN RETURN
   PRINT  "### Finished file: ", Path;     Print here and after all handlers.
RETURN;                                 Primary return to the caller.
```

```
<<< continue file:  calculate.file.b >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
:   A local handler is used to format a rich message.
:
METHOD CATCH Leftover:  Elements were left on the stack.
:
:.............................................................................


PRINT‘ERR‘#  "NOTE: ", Rpn‘Size;                Number of remaining values.

IF Rpn‘Size = 1:                                IF only 1 value was left,
   PRINT‘ERR  " value was left on the stack.";      Singular message.

ELSE:                                           ELSE several were left,
   PRINT‘ERR  " values were left on the stack."; Plural message.
END IF;                                         END IF

RETURN;                                         Normal return.



::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
: Note that faults can still propagate from an exception Safe method.
:
METHOD CATCH FAULT:  Catch all program faults and report them.
:
:.............................................................................


PRINT‘ERR  "A fatal error has occurred:  ",     &Kind of fault and context.
           Assertion‘Fault, Assertion‘Context

IF Assertion‘Message:                           IF a message to report.
   PRINT‘ERR  "   ", Assertion‘Message;             Indent the message.
END IF;                                         END IF

RETURN‘RAISE;                                   Propagate the program fault.
```

```
<<< continue file:  calculate.file.b >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
: Catch an I/O error received (via Open or Read.Line).
: Note that the file path has been returned in:  Assertion‘Context
: It is also available in File.Path‘Path; which is used in the final handler.
:
METHOD CATCH  CALL Open.In, Read.Line
:
: The following could also be used to catch I/O error in this example:
:
:     METHOD CATCH  CLASS  File.Path
:     METHOD CATCH  SIGNAL Input:    Catch any exceptions and report them.
:.....................................................................


PRINT‘ERR  "An I/O error occured in: ",   &Report the file path.
           Assertion‘Context

PRINT‘ERR  Assertion‘Message;              Report an I/O error message.

RETURN;                                    Return from the primary method.



::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
: Catch all other exceptions and report them.
:
METHOD CATCH:  Catch any exceptions and report them.
:
:.....................................................................


PRINT‘ERR  "Could not process: ",       &Report the file path.
           Input‘Path

IF Input‘Line:                             IF any lines have been read,
   PRINT‘ERR  "Line", Input‘Line, ": ",    &Report the line number
              Assertion‘Message            :   and the message.

ELSE:                                      ELSE not reading the file,
   PRINT‘ERR  Assertion‘Message;              Just report the message.
END IF;                                    END IF

RETURN;                                    Return from the primary method.
<<< end file:  calculate.file.b >>>
```

```
<<< begin  file:  calculate.line.b >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:  The precondition will raise a fault if it fails.
:  If a managed exception is raise the line could not be processed.
:  A descriptive message is returned in:  Assertion‘Context
:
METHOD Calculate.Line:  Process a line for a calculator instruction.

 ENTRY Line       STRING,     &Line to calculate (must not be empty).
       Trace = 0  Bit         :Write a trace of operations.

 LOCAL I       WORD,          &Index to a character in the line.
       J       WORD,          &Index to a character in the line.
       C       PARCEL,        &A character from the line.
       Name    STRING,        &A variable name.
       Result  WORD,          &An arithmetic result value.
      @P       Key..Word      :A pointer to a variable name and value.

   USE Assertion‘Assertion    :Access exception status in a stardard library.
:.....................................................................


ASSERT PRECONDITION;                        ASSERT
   Line <> ""  FAULT;                          CHECK:  Can not evaluate an empty line.
END ASSERT;                                 END ASSERT

I = Scan.Right( Line );                     Index to space after the first token.

IF I:                                       IF a second token
   Non.Space  Line, I, J;                      Index to the front of the next token.

   ASSERT PARCEL( Line, J ) = '='             &ERROR:  Expected an equal sign.
          PASS  "Column " ! FORM( I )         &
              ! " in:  "  ! Line

   ASSERT  J = LENGTH( Line )  PASS Line;      ERROR: Junk after an store command.
   Pop  Rpn, Result;                           Might raise:  ERROR: Stack underflow.
   Name = UPPER( LEFT( Line, I - 1 ));         Extract the uppercased name.
   Put  Variable, Name, Result

   IF Trace:                                   IF tracing operations,
      PRINT  Proper( Name ), '=', Result;         Trace the assignment.
   END IF;                                     END IF

ELSE IF UPPER( Line ) = "PRINT":            ELSE IF a print command,
   Pop    Rpn, Result;                         Might raise:  ERROR: Stack underflow.
   PRINT  Result
```

```
<<< continue  file:  calculate.line.b >>>
ELSE:                                     ELSE producing a result,
   C = PARCEL( Line );                       First character in the line.

   IF Is.Alpha( C ):                         IF the token is a variable,
      P = @Get.At( Variable, UPPER( Line ));    Reference the stored variable.
      ASSERT  @P  PASS  Proper( Line );         ERROR:  Variable is not set.
      Result = P‘Tag

   ELSE IF ’0’ <= C  AND  C <= ’9’          &ELSE IF an integer,
        OR  C = ’-’  AND  LENGTH( Line ) > 1   &Example where a condition has
        OR  C = ’+’  AND  LENGTH( Line ) > 1   &a redundant clause.
        OR  C = ’#’  OR   C = ’_’

      From.Integer  Result, Line;               Convert the integer to binary.

   ELSE:                                     ELSE the token is an operator,
      ASSERT  LENGTH( Line ) = 1  PASS  Line;   ERROR:  Unrecognized calculation.
      Arithmetic.Operator  C, Result, Trace;    Perform a binary operation.
   END IF;                                   END IF

   Push  Rpn, Result;                        Add the result to the stack.
END IF;                                   END IF

RETURN



::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
: Catch a stack underflow associated with the Underflow test.
:
METHOD CATCH TEST Underflow:  Catch any exceptions and report them.
:
:.......................................................................


Assertion‘Message = "ERROR:  Underflow on the arithmetic stack."

RETURN‘RAISE



::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
METHOD CATCH CALL From.Integer:  Collapse conversion errors.
:
:.......................................................................


Assertion‘Message = "ERROR:  Invalid integer value."

RETURN‘RAISE
<<< end file:  calculate.line.b >>>
```

F. Case Study Using Signals

```
<<< begin class file:  file.path.p >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
: This is sample code extracted from interfaces for a component that assembles
: machine code.  This class defines a wrapper for a file path and includes a
: Signal set after an I/O error occurs.
:
CLASS File.Path          &Combine a path name and a status code signal.
    . Path   STRING      :Name of the path.

SIGNAL Status  BYTE      :I/O error status issued by the run-time system.

   : The remainder of the class goes here.

<<< end class file:  file.path.p >>>



<<< begin method file:  close.ca.b >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
: The client invokes a series of methods (not shown) to create a
: private structure containing the machine code to be assembled.
: It then passes the path of the object file to be written to this
: public method; which writes the object file.
:
: If the object file is written successfully this method simply returns.
: Otherwise it propagates a Signal set in the Write.As method.
:
: On an I/O error it raises a Signal with the Status member set to the code
: returned from the intrinsic I/O Status function.  The Context member
: declared in a standard library Assertion class is also set to the file path.
: For all other errors the Status member will be zero.  After any exception
: the Assertion'Message variable will be set to a descriptive error message.
:
:     Typical reporting for an I/O error is:
:
:         ca( hello.s ) ERROR:  Tried to open a file that does not exist.
:
:
METHOD Close.Ca  SIGNAL:  Write out the assembled file.

 ENTRY File.Path         :Relative path of the file to write.
:
:.....................................................................


   :
Write.As  $;                    Write the object file.
   :

RETURN
<<< end method file:  close.ca.b >>>
```

```
<<< begin method file:  write.as.b >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
: This is a private method to write the object file.
: An I/O error is detected here and the handler sets a Signal Status member.
:
METHOD Write.As  SIGNAL:  Write out the assembler file.

 ENTRY File.Path             :Relative path of the file to write.

 LOCAL File                  :File handle.
        :
   USE Assertion‘Assertion   :Access standard library assertion context.
:
:.....................................................................


OPEN‘OUT  File = $Path;           Open the object file for writing.

   : Code to write the object file goes here.

RETURN;                           File destructor closes the file.



::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
METHOD CATCH:  Check for I/O exceptions and report the status and context.
:
:.....................................................................


IF File‘Error:                    IF an output error,
   $Status = STATUS( File );         Signal the I/O status code.

   Assertion‘Context = $Path;        Set the context to the failed file path.
END IF;                           END IF

RETURN‘UNWIND;                    Destructor closes the file; Raise exception.
<<< end method file:  write.as.b >>>
```

G. Case Study Catching a Method Call

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
: This example was extracted from a compiler back end.
: It illustrates catching a method call that raises exceptions.
: Forensic analysis is used to construct a rich exception message with context.
:
METHOD Clean.Local.Numeric:  Generate code to initialize a numeric scalar.

 ENTRY Type,                    &Local variable numeric type.
       Value   STRING           :Value as a text string.

 LOCAL Data    CELL             :Value encoded in binary form.

   USE General‘Ra,              &Low order register.
       General‘Rb,              &High order register.
       Set‘S,                   &S points to the variable symbol.
       Assertion‘Assertion      :Access standard library assertion context.
:
:.............................................................................


IF  : Code to process other cases goes here.

   Ra = Value;                                  Assume value in the low register.

   : This call will raise an exception if the value in Ra exceeds 64 bits.
   :
   From.Integer  Data, Ra;                      Convert a register to binary.

   IF Type‘Is( Cell ):                          IF the value is in 2 registers,
      Ra = FORM( Data /\ #ffff_ffff, "=#H#" );    Low order register in hex.
      Rb = FORM( Data // 32, "=#H#" );            High order register in hex.

   ELSE IF Type‘Is( Word ):                     ELSE IF a Word,
      ASSERT  Data // 32 = 0  OR                  &ERROR:  The default value is
              Data \/ #7FFF_FFFF = -1                    &  too big for a Word.
              PASS  S‘Key ! " = " ! Value

   ELSE IF Type‘Is( Parcel ):                   ELSE IF a Parcel,
      ASSERT  Data // 16 = 0  OR                  &ERROR:  The default value is
              Data \/ #7FFF = -1                         &  too big for a Parcel.
              PASS  S‘Key ! " = " ! Value

   ELSE:                                        ELSE a Byte,
      ASSERT  Data // 8 = 0                       &ERROR:  The default value is
          OR  Data \/ #7F = -1                           &  too big for a Byte.
              PASS  S‘Key ! " = " ! Value
   END IF;                                      END IF
END IF;                                       END IF

Gen.Store.Flat;                               Generate a store to set a variable.

RETURN;                                       Successful return.
```

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
METHOD CATCH CALL From.Integer:  Catch default values over 64 bits.
:
:.............................................................................


Assertion‘Message = "ERROR:  The default value is too big for a "

IF Type‘Is( Cell ):                          IF the variable is a Cell (64 bits),
   Assertion‘Message != "Cell.";                Report a Cell target.

ELSE IF Type‘Is( Word ):                     ELSE IF it is a Word (32 bits),
   Assertion‘Message != "Word.";                Report a Word target.

ELSE IF Type‘Is( Parcel ):                   ELSE IF it is a Parcel (16 bits),
   Assertion‘Message != "Parcel.";              Report a Parcel target.

ELSE IF Type‘Is( Byte ):                     ELSE IF it is a Byte (8 bits),
   Assertion‘Message != "Byte.";                Report a Byte target.

ELSE:                                        ELSE it can’t happen here,
   ASSERT  FAULT  S‘Key;                        CHECK:  An integer type was expected.
END IF;                                      END IF

Assertion‘Context  = S‘Key ! " = " ! Value; Report the variable and bad value.

RETURN‘UNWIND;                               Raise an exception.
```

H. Example Using Local Exceptions

```
<<< begin file:  dispatch.command.b >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
METHOD Dispatch.Command:  Primary (non-exception) code for a high level method.

 ENTRY Command   STRING      :An input parameter containing.

: Access the Assertion variable in the Assertion class available in a standard
: class library.  After an exception it's fields are set by the First Chance
: Handler to the arguments.
:
   USE Assertion‘Assertion   :Access standard library assertion context.
:
:.....................................................................


IF Command
   Parse.Command  Command, Signature;     Parse a command into a structure.

   Run.Command  Signature;                Execute the parsed command.
END IF

RETURN



::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
METHOD CATCH:  A standalone catch handles any exceptions.
:
: This is an example of a reporting handler.  It optionally prints out the
: assertion context followed by a diagnostic message to the standard error
: output stream.
:.....................................................................


IF Assertion‘Context
   WRITE‘ERR  "<", Assertion‘Context, "> "
END IF

PRINT‘ERR  Assertion‘Message

RETURN;        Return from Dispatch.Command; rejoining the primary code path.
<<< end file:  dispatch.command.b >>>
```

```
<<< begin file:  update.index.b >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
METHOD Update.Index:  Primary (non-exception) code for a low level method.

CHANGE Index   BYTE     :An 8 bit integer parameter passed in and back out.

   USE Assertion‘Assertion      :Access standard library assertion context.
:
:.....................................................................


: When this precondition fails the Index variable is formatted as a signed
: decmal integer into a context string.  The
:
ASSERT PRECONDITION
   Index => 0  PASS  Index;                 ERROR:  The index is negative.
END ASSERT


: Code to do some work happens here.


ASSERT  Index <= Max  CATCH Big.Index;      ERROR:  The index is too big.


: Code to do some more work happens here.

RETURN
```

```
<<< continue file:  update.index.b >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
METHOD CATCH Big.Index:  Handler the corresponding assertion when it fails.
:
: After the assertion fails, control is passed to the Exception Manager;
: which calls the First Chance Handler.  It detects any severe exceptions
: and terminates or otherwise sets the Assertion variable.
:.............................................................................


Index = 0;          Pass back a zero on an overflow.

RETURN;             Return from Update.Index; rejoining the primary code path.



::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
METHOD CATCH:  A standalone catch handles any remaining exceptions.
:
:.............................................................................


: In debug builds this will print the original message.
:
debug  Assertion‘Message


: Overwrite the current message.  Any non-specialized exception is coalesced
: into this single case.
:
Assertion‘Message = "ERROR:  Could not update the index."


: The Unwind switch passes control to the next handler up the call chain.
:
RETURN‘UNWIND
<<< end file:  update.index.b >>>
```

I. Example of Colliding Tests

Collision are detected durring the test run and not in production runs. The programmer can change either the test or program to avoid the collision. It is also possible to change the precision of the dispatching mechanism for tests. The test driver generates code used to dispatch tests independent of the compiler. The generated code overrides a default method in the run-time system that does not dispatch any tests.

The test dispatch mechanism uses only the endpoints of the exception event to ensure complete coverage over alternate call graphs. Each test handler can be assured to cover all paths to the exception point. Altering the precision of the test dispatch mechanism may lead to incomplete coverage of test cases.

```
METHOD Test_Even TEST            METHOD Test_Odd TEST
MERGE  2                         MERGE  7
RETURN                           RETURN

            METHOD Merge
             ENTRY N   WORD
            IF N /\ 1
               Split_Odd

            ELSE
               Split_Even
            END IF
            RETURN

            METHOD CATCH Test_Even:  Run on Assert
            <Handle Even>
            RETURN

            METHOD CATCH Test_Odd:   Also Run on Assert
            <Handle Odd>
            RETURN


METHOD Split_Odd                  METHOD Split_Even
Common                            Common
RETURN                            RETURN

              METHOD Common
              ASSERT;   An assertion common to different callers.
              RETURN
```

Figure 4: A collision occurs when multiple tests are in the same method and lead to the same exception point.

The programmer can recode a collision:

* Combined the handlers and check:

```
METHOD CATCH Test_Combine
IF N /\ 1
   <Handle Odd>
ELSE
   <Handle Even>
END IF
RETURN
```

* Don't use tests for this case:

```
METHOD CATCH CALL Common
IF N /\ 1
   <Handle Odd>
ELSE
   <Handle Even>
END IF
RETURN
```

* Relocate a handler: Move test even to even case.

```
METHOD Split_Odd
COMMON
RETURN

METHOD CATCH Test_Odd:  Relocate this handler.
<Handle odd>
RETURN
```

J. Example of Interacting Layers

```
<<< begin members of the Upper class >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
:  This is an example of a handler interacting between component layers.
:  An exception raised in the lower layer needs to know if it should retry.
:  It can not unwind to the higher layer without losing its current state.
:  Instead it provides a hook (the User.Retry method).
:  The upper layer overrides the hook with a method to interact with the user.
:  Note that in the case where multiple upper layer classes want to override
:  the hook, virtual overring methods can be used.  The program output is:
:
:     ERROR: An exception in an operation.
:     Try again [No]? yes
:     ERROR: An exception in an operation.
:     (Lower @ 16)  Terminating after an exception in a Catch with no Retry.
:
:  The first error message is issued by the User.Retry method.  After the
:  operation is retried and fails again the handler in the Lower method is
:  invoked a second time.  Since it does not permit retries, an Exception.Retry
:  fault is raised.  The exception Manager unwinds to the Last Chance Handler.
:  It issues the second error message plus a fault message and terminates.
:
METHOD Upper:  The example begins here in the Upper class.
:
:.....................................................................


Lower;                Invoke a low level operation in the Lower class.

RETURN
```

```
<<< continue members of the Upper class >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
METHOD User.Retry:  Override the hook provided in the class Lower.

  EXIT Retry = 0   Bit         :Return 0 for no retry; 1 to retry.

 LOCAL Reply   STRING,         &Reply to the prompt.
       Key     BYTE            :Single key in a reply.

   USE Assertion‘Assertion     :Access standard library assertion context.
:
:.....................................................................


PRINT  Assertion‘Message;            Message on a failed operation.
WRITE  "Try again [No]? ";           Prompt for action.

DO ALWAYS:                           DO over the reply,
   FLUSH
   READ   Key;                          Read a character from the keyboard.

UNDO IF Key = ’^M’;                  UNDO IF a return key.

   WRITE  Key;                          Echo a character.
   Reply != UPPER( Key );               Append the uppercase key.
END DO;                              END DO

PRINT;                               Wrap the response line.

IF Reply = "YES"  OR  Reply = "Y":   IF the user wants to try again,
   Retry = 1;                           Set the return bit.
END IF;                              END IF

RETURN
<<< end members of the Upper class >>>
```

```
<<< begin members of the Lower class:  file lower.b >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
METHOD Lower:  Perform an operation in the Lower class.
:
:.............................................................................


Operation

RETURN



::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
METHOD CATCH:  Unwind here when the operation fails.

 LOCAL Retry  Bit         :0 for no retry; 1 to retry.
:
:.............................................................................


User.Retry  Retry;         Hook for a user level interaction.

IF Retry:                  IF Retrying,
   Operation;                 Run it again.
END IF;                    END IF

RETURN
<<< end file lower.b >>>



<<< begin file operation.b >>>
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:
METHOD Operation:  An operation that for this example always fails.
:
:.............................................................................


ASSERT;      ERROR: An exception in an operation.

RETURN
<<< end file operation.b and members of the lower class >>>
```

References

[1] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions. Technical Report TR-CIS-06-09, Department of Computer and Information Science; University of Pennsylvania; Philadelphia, PA 19104-6389 USA, May 2006.

[2] Hans-J Boehm. Destructors, Finalizers, and Synchronization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–272, January 2003.

[3] Patrice Chalin. Improving JML: For a Safer and More Effective Language. Technical Report 2003-001.3, Faculty of Engineering and Computer Science (ENCS), Concordia University, June 2003.

[4] Patrice Chalin. JML Assertion Semantics Revisited. Technical Report 2004-001, Faculty of Engineering and Computer Science (ENCS), Concordia University, April 2004.

[5] S. Drew, K. J. Gouph, and J. Ledermann. Implementing Zero Overhead Exception Handling. Technical Report 95-12, Faculty of Information Technology, Queensland University of Technology, 1995.

[6] Fernando Castor Filho, Alessandro Garcia, and Cecilia Mary F. Rubira. Error Handling as an Aspect. In *Workshop on Best Practices in Applying Aspect-Oriented Software Development*, March 2007.

[7] Chen Fu, Richard P. Martin, Kiran Nagaraja, David Wonnacott, Thu D. Nguyen, and Barbara G. Ryder. Compiler-Directed Program-Fault Coverage for Highly Available Java Internet Services. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, pages 595–604, June 2003.

[8] Tim Harris. Exceptions and Sideeffects in Atomic Blocks. Technical Report 2004-01, University of Cambridge Computer Laboratory; Cambridge, UK, July 2004.

[9] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. *ACM SIGPLAN Notices*, 38(11):388–402, October 2003.

[10] Tim Harris and Peyton Jones. Transactional Memory with Data Invariants. In *TRANSACT06 First ACM SIGPLAN workshop on languages, compilers, and hardware support for transactional computing*, June 2006.

[11] Roy Krischer and Peter A. Buhr. Asynchronous Exception Propagation in Blocked Tasks. In *Workshop on Exception Handeling (WEH 08), November 14, Atlanta, Georgia, USA*, November 2008.

[12] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification. *Science of Computer Programming*, 55:185–208, March 2005.

[13] Andreas Leitner, Ilinca Ciupa, Oriol Manuel, Bertrand Meyer, and Arno Fiva. Contract Driven Development = Test Driven Development - Writing Test-Cases. In *Proceedings of the the 6th Joint Meeting of*

the European Software engineering conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pages 425–434, September 2007.

[14] David Lomet. Process Structuring, Synchronization, and Recovery Using Atomic Actions. *ACM SIGPLAN Notices*, 12(3), March 1977.

[15] Donna Malayeri and Jonathan Aldrich. *Practical Exception Specifications*, volume 4119/2006, pages 200–220. Springer Berlin / Heidelberg, October 2006. ISBN 978-3-540-37443-5.

[16] Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, pages 40–51, October 1992.

[17] Robert Miller and Anand Tripathi. Issues with Exception Handling in Object–Oriented Systems. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 85–103, September 1997.

[18] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open Nesting in Software Transactional Memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 68–78, March 2007.

[19] Matt Pietrek. New Vectored Exception Handling in Windows XP. *Microsoft Developers Network Magazine*, September 2001.

[20] Matt Pietrek. A Crash Course on the Depths of Win32 Structured Exception Handling. *Microsoft Systems Journal*, January 1997.

[21] Martin P. Robillard and Gail C. Murphy. Designing Robust Java Programs with Exceptions. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of Software Engineering: Twenty-first Century Applications*, pages 2–10, November 2000.

[22] K. Rustan, M. Leino, and Wolfram Schulte. Exception Safety for C#. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods*, pages 218–227, September 2004.

[23] Avraham Shinnar, David Tarditi, Mark Plesko, and Bjarne Steensgaard. Integrating Support for Undo with Exception Handling. Technical Report MSR-TR-2004-140, Microsoft Research, December 2004.

[24] Saurabh Sinha, Alessandro Orso, and Mary Jean Harrold. Automated Support for Development, Maintenance, and Testing in the Presence of Implicit Control Flow. In *Proceedings of the 26th International Conference on Software Engineering*, pages 336–345, May 2004.

[25] Bjarne Stroustrup. Programming with Exceptions. *Informit.com*, April 2001. URL `http://www.research.att.com/~bs/eh_brief.pdf`.