Issues with Exception Handling in Object-Oriented Systems

Robert Miller[1] and Anand Tripathi[2]
Computer Science Department
University of Minnesota
Minneapolis, Minnesota 55455 USA

Abstract

The goals of exception handling mechanisms are to make programs more reliable and robust. The integration of exception handling mechanisms with object-oriented languages raises some unique issues. The requirements of exception handling often conflict with some of the goals of object-oriented designs, such as supporting design evolution, functional specialization, and abstraction for implementation transparency. This paper demonstrates these conflicts, illustrates that the use of exception handling in object-oriented systems poses potential pitfalls, and suggests that their resolution is one of the first steps necessary to make exception handling robust in object-oriented designs and languages.

1. Introduction

The general goals of object-oriented designs includes support for abstraction, design evolution, functional specialization, and conceptual modeling. These goals are supported by basic elements of object-oriented languages, which include constructs for supporting abstraction, encapsulation, modularity, and specialization and reuse through inheritance. Towards the building of robust and reliable systems, object encapsulation mechanisms create natural domains for confining the effects of errors within an object and propagating such effects in a well-controlled fashion across the object boundaries [11].

The importance of exception handling is for program reliability and robustness, since many object-oriented languages consider an exception to be some kind of error. Recovering from errors has traditionally been error prone, and is itself a significant cause of program failures. One study [5] has shown that perhaps two-thirds of a program may be for error handling. The rising importance of exception handling is evident in object-oriented languages such as C++ [20], Java [18], Ada [4], and Smalltalk [8].

By combining object-oriented design methods with exception handling techniques, it is hoped that more reliable error handling can occur within a program. However, exception handling often does not exactly fit into the object-oriented paradigm. We show in this paper that the needs of exception handling often conflict with the goals of object-orientation in the areas of object interface definition, composition relationships,

[1] E-mail rm@vnet.ibm.com. Currently with IBM, Rochester, MN.
[2] E-mail atripath@nsf.gov. Currently with National Science Foundation, Arlington, VA on a temporary assignment.

exception context, exception conformance, and program control flow. The conflict often manifests during the specialization of object functionality and design evolution.

The objective of this paper is to show, mainly through examples, why exception handling may need to conflict with traditional object-oriented goals. The purpose is to aid programmers that must write exception handlers and designers of exception handling mechanisms in avoiding the potential pitfalls that exception handling poses. With this insight, perhaps programmers and language designers can avoid these problems, and truly robust exception handling can be realized.

Section 2 of this paper provides background with terminology, exception models, and exception handling in some of the major object-oriented languages. We make a distinction between modeling and implementation domains because their exception handling requirements are different. Section 3 details four key areas of differences between the basic design goals of object-oriented systems and exception handling models: complete exception specification, partial states, exception conformance, and exception propagation. To keep the discussion focused, simple and somewhat contrived examples are used. Section 4 shows how these four differences can affect four of the major elements of object-oriented designs: abstraction, encapsulation, modularity, and inheritance. Lastly, Section 5 summarizes the conclusions of this paper.

2. Background and Related Work

Based on the work by other researchers in the field [9,12,23], we briefly describe the seminal concepts and terms related to exception handling models. Next, we discuss the different inheritance domains because there can be significant differences in the meaning that an exception can have in these domains. Finally, a brief overview of the exception handling models of some of the most common object-oriented languages is presented. The scope of this discussion is limited to the concepts that are relevant from the viewpoint of the central theme of this paper.

2.1 Terminology and Exception Models

Goodenough's work [9] forms the foundation of exception handling terminology and models. Over the past 20 years, there have been several further elaborations and extensions of these models. Goodenough considered exceptions as a means to communicate to the invoker of an operation various kinds of conditions, such as errors during an operation execution, classification of the result returned by an operation, or the occurrence of certain significant events (not necessarily failures) during the execution of an operation. Yemini and Berry [23] define an exception condition in an operation invocation as the occurrence of a state that does not satisfy the input assertion for the operation. Knudsen [12] defines an exception as a characterization of an unusual or abnormal condition that requires extraordinary computation. In many languages (e.g., C++, Java, Ada, Eiffel, and Smalltalk) an exception is normally viewed as an error.

An *exception* is an abnormal computation state. An *exception occurrence* is an instance of an exception. An exception can occur due to an *error* (an invalid system state that is not allowed), *deviation* (an invalid system state that is allowed), *notification* (informing an invoker that an assumed or previous system state has changed), or *idiom* (other uses in which the exception occurrence is rare rather than abnormal, such as detecting EOF). This paper deals mostly with errors, but touches upon deviations and notifications.

The terms *procedure*, *function*, and *block* are used in this paper in the usual sense, as implied in block-structured languages such as Algol-68, Pascal, or Ada. An *object* is an encapsulation of data and code that has a well-defined behavior. An *object's interface* (or *interface*) is the external view of an object that describes the object's behavior as a set of operations (i.e., functions, procedures, or methods) that can be invoked on the object, by its users. An *object context* (or *context*) is the information visible inside the object's encapsulation. This can include internal object variables and global variables.

An exception is *raised* when the corresponding abnormal state is detected. *Signaling* an exception by an operation (or a syntactic entity such as a statement or block) is the communication of an exception occurrence to its invoker. The recipient of the notification is a syntactic entity, called the *exception target* (or *target*); the originator of an exception is the *signaler*. The target is determined either by static scope rules or by dynamic invocation chain.

An *exception handler* (or *handler*) is the code invoked in response to an exception occurrence. It is assumed that the handler's code is separate from the non-exception (or *normal*) control path in the program. Searching for eligible handlers begins with the target (i.e., the search starts with the handlers associated with the target). An exception is considered *handled* when the handler's execution has completed and control flow resumes in the normal path. An exception handled by the target *masks* the exception from the target's invokers.

An *exception context* is the information available to the exception handler concerning the exception occurrence. The exception context may contain some data that is explicitly passed by the signaler, as in CLU and C++. Additionally, it may contain information implicitly passed by the language runtime; for example, the name of the method or the class that signaled the exception. Guide [13] allows a handler to distinguish between exceptions by additionally using the class and method names of the signaler.

Exception propagation is the signaling of an exception to the target's invoker. There are two ways to propagate an exception: *implicit* (i.e., automatic), and *explicit.* In some languages, e.g. Ada, an unhandled exception is implicitly propagated. The second way is to require that any exception to be propagated must be explicitly signaled in a handler; CLU follows this approach. Such exceptions are also called *resignaled* or *reraised* exceptions. An implicitly propagated exception is automatically signaled and is the same as the original exception; on the other hand, a resignaled exception must be explicitly raised by the target's handler and can be different from the original exception. For explicit propagation, an unhandled exception (i.e., an exception not handled by any handler in the target) is either implicitly converted into some predefined exception by the exception handling mechanism, or the program is terminated.

An *exception model* defines the interactions between the signaler, target, and handler. The *termination model* automatically terminates the signaler, destroys any objects within the scope of the signaler, and considers the target to be the signaler's invoker. Once the exception is handled by the target, control resumes at the next statement following the target's syntactic unit. For example, if the target's syntactic unit is a statement or a block, then normal execution resumes at the next statement or at the next statement after the block, respectively. In the *resumption model*, computation continues from the point where the exception was originally raised. The *retry model* is a combination of termination and resumption; when the exception is handled, the signaler's block is terminated and then is re-invoked (or retried). The *replacement model* [23] is a variant of termination. The handler can return a value in place of the result expected from the signaler. The signaler's block is terminated and the result from the handler is returned to the signaler's invoker. Languages that allow a handler to replace a signaler's result include Guide [13]. Lastly, it is not required that a language only support one exception model, though that is usually the case. The main reasons for this restriction are ease of implementation and that one model can usually emulate the main characteristics of the other models (though it may be grossly inefficient).

2.2 Domains of Exception Handling

There are two views (or *domains*) of inheritance [22]: *modeling* and *implementation*. The modeling domain is for conceptual inheritance, while the implementation domain is for code reuse. *Subtyping* is conceptual specialization, while *subclassing* is implementation reuse of code classes. With respect to exception handling, the two domains can be quite different: exceptions in the implementation domain are generally considered to be errors, while exceptions in the modeling domain are generally considered to be deviations.

In the implementation domain, one major concern is reliable programming. Program failures are often caused by unexpected conditions that cause an error and the error is not properly handled. Unexpected conditions are often treated as exceptions, so an exception handler effectively becomes an error handler. The typical exception model in object-oriented languages is termination because the termination model is considered easier to implement than other models, and it is usually safer to destroy an object that has signaled an exception and restart the computation than trying to restore the object to a consistent state. Over the years, experience has indicated that (a variant of) the termination model is easier to use than the resumption model in the implementation domain [21]. One reason is that it is hard for a handler to know the system state for all situations in which the handler may be invoked, so it is easier to terminate (or cleanup) to a well-known system state rather than proceed and possibly cause more problems by making an incorrect assumption. In [13], arguments are made against resuming a signaler inside an encapsulation after the execution of some actions by a handler outside the encapsulation. This makes it difficult to prove the correctness of an interface by only examining the encapsulated implementation.

In the modeling domain, a major concern is flexibility [2]. An example from [2] illustrates the use of exceptions in the modeling domain. A real estate database has fields for the address and price of each listed house. An exception occurs for two kinds of contradictions within the database: the database constraints may need to be violated occasionally, and some subtypes may need to violate parts of the supertype definition. An example of violating a constraint is if the house is in a foreign country. If the price of the house is in the foreign currency, then, depending on the exchange rate, the price field may be exceeded. Another example is if a new subtype for address is defined for a foreign country, it may not be a proper subtype of its parent type because the parent may have fields that have no meaning to the child. An example is US and Canadian addresses: US states are not a subset of Canadian provinces, or vice versa. Such abnormal cases are considered as acceptable deviations in the modeling domain.

Most languages are designed for the implementation domain since everything (including modeling domain abstractions) must eventually be implemented. However, in the modeling domain an exception is not an error that may compromise program reliability, but is instead a way of increasing program flexibility. An exception cannot be considered an error and the use of the termination model in a similar fashion as in the implementation domain may not allow exceptions as deviations. It would seem that the resumption model would be more suitable for modeling domain exceptions.

There is another way that the mixing of the two domains affects exception handling: an exception context may need to be modified as an exception is signaled from one domain to the other. To illustrate this, assume an accounting spreadsheet program is executing within an environment, such as an accounting framework that provides features unique for accounting. The accounting program and framework could be in the modeling domain, while runtime libraries used by the framework are in the implementation domain. Within the program, an arithmetic operation in a library routine signals an overflow exception. The operation knows the exception occurred during an integer addition, so resignals to the framework an 'integer addition failed' exception. The framework, in turn, resignals the exception to the program as a 'tally operation failed' exception. The same exception has had its abstraction (and so its context) changed from integer overflow in some hardware register to a tally operation failing. Another program, say a database, could also encounter the same integer overflow exception, but the exception's changing abstraction may follow a completely different path than that of the accounting program.

This example raises two significant issues regarding how to adequately represent the difference in abstraction that an exception represents. The issues are how does an exception context get modified during exception propagation, and what is the form of the exception propagation (along the invocation chain or along the class hierarchy) that can correctly represent the change of domain.

2.3 Exception Handling in Object-Oriented Languages

Exceptions are represented in a variety of ways in the various object-oriented programming languages. At one end, Beta [15] does not provide any special constructs or primitives to represent and handle exceptions. Instead, Beta uses its basic language constructs (such as *virtual patterns*). At the other end, some systems [7] use a special class of objects to represent exceptions that have certain predefined operations, such as *signal* or *raise*. In between are languages such as Guide [13] that represent exceptions as symbols, which can be organized according to some inheritance hierarchy. Such an organization permits incremental specialization of exceptions and their handlers, and can provide default and generic support for exceptions defined at the higher levels of the inheritance hierarchy.

There are many variations on representing exceptions as symbols. In CLU, an object-based language, exceptions are represented as symbols with an associated set of typed parameter objects that can be used to pass information from the signaler to the handler. In C++, an exception is represented by an object which is not necessarily differentiated from other objects of the computation. An object representing the exception is created when an exception is raised.

Representing exceptions as objects allows inclusion of context information that can be explicit (passed by the signaler) as well as implicit (passed by the runtime) to communicate to the handler the nature and cause of the exception. Guide [13] implicitly provides the handler with the names of the class and the method that signaled the exception. In C++, context-related information is explicitly passed to the handler with the object that is created when the exception is raised.

Many languages have been concerned with the ability to determine, at compile time, the set of exceptions that an operation can raise. For this reason, the interface of an object should also list the set of exceptions that an operation can raise [9,14,23]. In general, an attempt to propagate to the invoker an exception not in this list can cause the program to terminate. In C++, it leads to the execution of a function named *unexpected*, which can be redefined by the programmer.

The propagation of exceptions along the dynamic invocation chain is considered to increase reusability so that the invoker of an operation can possibly handle it in a context-dependent manner. This can help in increasing the usability of an object in a wider context. However, the goal of compile time determination of a possible set of exceptions requires disallowing automatic propagation of exceptions along the dynamic invocation chain. All exceptions propagated from an operation to its invoker should be either explicitly propagated or resignaled. If an unhandled exception is not explicitly propagated, it is either considered a fatal error or is implicitly propagated after being changed to some predefined general exception, such as *FAILURE* in CLU or *UNCAUGHT_EXCEPTION* in Guide.

In Smalltalk-80, exceptions are handled by the object encountering the abnormal condition. There is no propagation along the dynamic invocation chain. There are three selectors defined for each class to deal with exceptions. Two of these have predefined meanings, and the third selector, called *Error*, is used to communicate all programmer defined errors. Thus, the meaning of *Error* may become overloaded, as there is no way to define new exceptions.

In Eiffel, exceptions are defined as the violation of certain assertions on the object state or the pre- or post-conditions of an invoked operation. A handler is attached to a method as a *rescue clause*, which can possibly direct the re-execution of the method using the *retry* statement after it has restored the object to a consistent state. Any termination of a method through the rescue clause results in the failure of the operation. Within a rescue clause, the exception can be determined by comparing a predefined variable called *exception* with an exception name. The exception names form a global name space.

In Beta [15], there is no explicit construct to represent exception conditions. When an abnormal condition is detected, an appropriate pattern, which is a Beta object, is invoked. This pattern defines the exception handler for the detected condition. Beta's approach is like building "fault tolerant encapsulations" [6] that try to handle abnormal conditions within the object at the points where they arise. The program or object's enclosing scope is terminated if the exception cannot be handled within the object. An exception condition is detected and handled as part of the normal flow of execution. For that reason, the concept of searching for a handler does not exist in Beta. The *inner* construct and virtual patterns of Beta together provide a way for an invoker of an operation to affect the handling of an exception inside the object. The code extension (called *binding*) given by the invoker at the time of an invocation is executed by substituting it in the operation's code at the place where *inner* is declared. The *inner* mechanism also allows a subclass to extend or augment the exception handling of the parent class. This use of *inner* can often require a careful understanding of the pattern's code.

3. Exception Handling vs. Object-Orientation

Many other researchers [e.g., 2,7,13,19] have recognized that exception handling may have different requirements than object-orientation. It is this paper's position that the differences are more like conflicts because exception handling can contradict the conventional object-oriented paradigm. As noted in Section 1, exception handling is often error handling, which is notoriously difficult to do. The conflicts can lead to a 'false sense of security' [3] that exception handling in object-oriented languages will automatically yield more robust programs. To the contrary, the conflicts may lead to a less robust program. To begin the discussion, it is necessary to see how exception handling may differ from object-orientation. The differences to be discussed in this paper are:

- Complete exception specification.
- Partial states.
- Exception conformance.
- Exception propagation.

3.1 Complete Exception Specification

The object interface is the outside view or specification of the object. In many languages, the interface only contains a list of the exceptions that an interface method may signal (an *exception specification*). However, this does not unambiguously define the exception behavior of the interface. We use a new term, *complete exception specification*, that extends upon an exception specification in the following ways:

- Exception masking: how is the abstract state of an object affected if the implementation masks an exception.
- Exception consistency: each exception that can be signaled has an associated meaning that is consistent regardless of the signaling location.
- Exception context: what is the exception context provided for each signaled exception.
- Object state: the state of the object immediately prior to the exception signaling is indicated within the exception context.

A complete exception specification is similar to a normal (non-exception) interface in two ways. The exceptions that can be signaled are similar to the *return codes* that the interface defines. Exception consistency is similar to a return code having the same meaning, regardless of the location in the implementation from where the code is returned.

None of the remaining extensions have equivalents in a normal interface. The exception masking equivalent would be similar to the normal interface detailing how an internal call is handled within the interface implementation. Exception context's equivalent is similar to the interface being able to return different types of results. Lastly, object state for the normal interface is usually assumed to be valid upon return. An explicit declaration of state is unnecessary for the normal interface but may be necessary for exception handling.

The key difference between exception handling and normal object-orientation is that more information is revealed in the exception specification. This is to provide enough information to the handler to allow recovery or termination. No known language implements a complete exception specification as outlined here.

3.2 Partial States

Normal object-orientation (i.e., without exception handling) generally considers an object state to be either *valid* or *invalid*. The difference with exception handling is that there is a third state, which we call a *partial* state. Such a state may arise in a collection of objects, called a *composition*, where some invariant relationships among the components (or members) are required to be maintained. A partial state occurs when all the component objects of a composition are valid, but the composition relationships are not satisfied, implying an invalid composition state.

For normal object-orientation, changing a component object of a composition from one state to another may transition through a partial state. Normal programming generally considers object state transitions to be atomic, so the partial state is not observable to the programmer. However, the assumption of atomic state changes may not hold for exception handling because an exception may occur before the state change has completed.

The major difference between exception handling and normal object-orientation is that exception handling cannot always assume that state transitions are atomic. No known language has a total solution for partial states. Languages that support finalization or destruction (e.g., Ada and C++) address, in a limited way, the problems due to partial states by keeping track of which objects have been created or destroyed. But they do not keep track of relationships between objects.

A partial state is, in general, an invalid state. The difficulty is in indicating to the users of the composition, and possibly composition members, what the partial state is. A possible indicator for an invalid state is for the object to signal a failure exception. However, several questions arise. If a failure exception is signaled, should the object be allowed to exist in an invalid state or be destroyed? Should an object in an invalid state be allowable due to unforeseen side-effects if the object is accessed again? How is a user of the composition informed that the object is in a different partial state than what is expected?

3.3 Exception Conformance

Conformance between types also includes exceptions in some languages. Assume that class *B* is derived from class *A* and is to be conformant with *A*. For all methods that *B* inherits or redefines from *A*, *B* cannot signal an exception that is not a subtype of an exception that can be signaled from the corresponding method in *A*. For example, if *A* has method *M* that can signal exceptions *E1* or *E2*, then *B* cannot extend or redefine *M* with new exceptions that are not subtypes of *E1* or *E2*. We call conformance between exception types *exception conformance*.

Exception handling may need to be non-conformant. For normal object-orientation, implementations can change without users needing to know. New functionality can be provided as a redefined virtual method, however, the redefined method must still be conformant with the original method interface. For exceptions, new functionality may need new exceptions that are not subtypes of exceptions from the parent method. The conformance rules would prevent the new functionality because the exceptions would be non-conformant. To make the exceptions conformant requires the parent methods to use sufficiently generalized exception types so that any possible new exception that may be signaled by a redefined virtual method would be a subtype. The generalized exception types may be so general that they have limited value in defining a clear and consistent exception interface.

The difference between exception handling and normal object-orientation is that methods can be overloaded to have similar meaning in a wide variety of situations, but exception information (particularly for errors) generally needs to be specific. The overloading of functionality is acceptable and often desirable, but the overloading of exception information is usually not. This difference clearly surfaces in conformance.

There is a conflict between exception conformance and complete exception specification. To have a complete specification implies exception conformance, but to allow evolutionary program development suggests exception non-conformance, which in turns suggests an incomplete exception specification.

3.4 Exception Propagation

In normal object-oriented programming, program control flow follows the call/return path. If method *A* calls *B*, which then calls *C*, then *C* will return to *B* which will return to *A*. It is not expected that code from *A* will execute before *B* and *C* return. Even if *C* is terminated, the normal flow would not expect code from *A* to execute before code from *B*. Suppose that *C* signals an exception, then, depending on how the exception is propagated, handler code from *A* may execute before code from *B*, or even before *C* returns or is terminated.

The key point is that exception handling differs from normal object-orientation because exception propagation can change the control flow. It becomes important for a programmer to understand the possible control flow change, so there are at least two paths (not one) that the programmer must consider: the normal path, and the propagation path.

4. Exception Handling and Object-Oriented Goals

The preceding section showed how exception handling is different in four ways from normal object-orientation. In this section we see how these four differences manifest themselves in four of the major elements of object-orientation: abstraction, encapsulation, modularity, and inheritance.

4.1 Abstraction

One goal of abstraction principles is to hide the implementation details of an object from its users and expose only the necessary functionality. There are two aspects of abstraction that are relevant here:

- Generalization of operations.
- Composition.

4.1.1 Generalized Operations

A goal of exception handling mechanisms is to generalize operations of an object and make them usable in a wider range of conditions [9]. Often this may require exposing more implementation details, as a part of the abstraction, to the object's users. We observe here that the motivation for introducing exception handling may sometimes impose conflicting demands in regard to hiding implementation details. The following example illustrates this point.

Consider the design of a fault-tolerant disk storage system that is required to be more reliable and available as compared to a single disk drive system. Suppose that this reliable disk system is implemented using two drives acting as a primary-backup pair. There are two opposite solutions to building such a fault-tolerant object [6]. One is to perform all recovery within the object in case of any exception and not propagate the exception to the object's users. The second approach is to mask exceptions within the object, and propagate only relevant exceptions to the object's users. With the first approach in implementing the desired reliable disk system, one can completely hide from the users the internal implementation detail that the disk system consists of a pair of disk drives. In case of a failure of one of the drives, it would be meaningless to inform the user that one of the drives has failed. Thus, the system would be working at a reduced reliability level without the user knowing of or correcting such a situation.

The second approach is to expose the primary-backup nature of the implementation as a part of the abstraction visible to the users. An exception can be signaled to the user in case of a failure of one of the drives. The reliable disk system's interface can also provide suitable functions to the users to perform appropriate reconfiguration (such as replacement of the failed drive with a new one) in the event of such an exception. This approach of communicating exceptions to the users gives a greater flexibility to recover from failures, but it involves making the abstraction closer to the implementation.

4.1.2 Composition

Composition is a form of abstraction by grouping together related objects and presenting them as a single entity. Recall from Section 3.2 that a partial state is one in which all component objects of a composition are in valid states, but the relationships between the objects are not. Each individual component object may have no indication that the composition is invalid. We show two examples in which partial states are important: exception masking by a component object, and a component object signaling an exception to the composition user.

A component object masking an exception may need to inform the composition that an exception has occurred that is of interest to the composition. Assume that a composition class *C* is defined that has an individual component object that is an array: *Array[100]*. Each element of *Array* has a field that is a pointer to another element of *Array*, so that the pointers form a doubly-linked list of *Array* elements. Suppose that *Array[50]* is to have its pointer field set to *Array[1]*, and vice versa. Assuming that *Array* is initialized in ascending index order, *Array[1]* will be pointing to *Array[50]* when an exception is signaled during the initialization of *Array[50]*. The *Array* code masks the exception, and as part of the masking the pointer for *Array[50]* is set to a predefined value, NULL. Class *C* methods cannot assume that the pointer fields represent a doubly-linked list anymore. Since the exception is masked, *C* cannot invoke its exception handler to even scan *Array* and perform consistency checks. The masking of the exception may require class *C* methods to assume the possible existence of partial states, which conflicts with the assumption of object-oriented programming that internal state transitions are atomic.

A component of a composition could signal an exception to a composition user. However, not only the user, but also composition members may need to be informed of the exception. Thus, an exception may not only need to be propagated upward to the composition user, but also across or downwards to composition members.

Suppose that, in the example above, the exception signaled from *Array[50]* during its initialization is not masked, and the exception is signaled to the composition *C*. Two problems can arise as described below: the possible inability of *C* to correct the invalid composition relationships, and the possible need to propagate the exception downward to a component object. First, *C* needs to know which element of *Array* signaled the exception so *C* can identify which composition relationships are invalid. Since an individual element may not know what its array index is, the exception context may not have any information as to which element signaled the exception. So, *C* may not be able to do any recovery since it cannot determine where the exception occurred. Second, it may be desired to undo the link that *Array[1]* has set pointing to *Array[50]*. Undoing of actions is normally considered an error recovery path and usually not a part of the normal execution path, hence *Array[1]* needs to have an exception signaled to it. It would be preferred if the exception is signaled to *Array[1]* before signaling the exception to *C* so that *C* need not concern itself with invalid composition relationships. To accomplish this, the exception from *Array[50]* would need to be signaled to *Array[1]*, then signaled to *C*.

As an aside, no known language propagates exceptions downward and upward in the fashion described here. When this situation arises in current languages, what may be done is to create a method that performs the undo. When *C*'s handler is invoked, the handler calls the undo method to simulate an exception being signaled. The difficulty arises in specifying in *C*'s handler all the composition members that may need to be undone if more than one member is affected. *C* may need to know details of composition member implementations in order to determine which members' undo methods need to be called; because the undo method is considered a part of the 'normal' code (because it is not in a handler), it could be used inappropriately within normal execution.

4.2 Encapsulation

The encapsulation defined by an object hides its internal implementation from the users. However, implementation details of a signaler can be exposed by an exception object if it contains information pertaining to the signaler's implementation that is more than what is permitted by the encapsulation. It is also possible that the parent classes of an exception object may have extra information, unknown to the interface, using which the exception handler may be able to deduce implementation details. A programmer may be tempted to use the extra information for error recovery, which may become invalid if the implementation is changed. There are two ways that extra information can be used detrimentally: trying to determine the location of the signaler, and accessing the signaler without using the established interface.

Using implementation details can be useful when determining why an exception is signaled. Often, one handler is enabled over a region of code. The handler may need to determine where in the region did the exception occur. If the handler has access to extra exception context information, then the handler may be able to use that to assume where the exception was signaled. However, if this information is easily available to the programmer, then there is a high probability that the programmer will use it.

For example, suppose that the exception context contains information about the location of the signaling operation in the code, and this information is available to a handler. Let's say that one particular signaling code location is within a library routine, and the programmer knows that this library routine is only called at one location within the implementation. The programmer can use this information directly in the handler. Of course, the potential difficulty is if the implementation code (or anything that it may invoke) is changed to use the library routine in a different way, the handler may perform the wrong action. However, if this is the only way for the programmer to determine why the exception was signaled, then there is a high probability that the programmer will use the extra information.

The second way to expose implementation is if the exception context contains a pointer to private data within the signaler. Suppose that the exception context contains a pointer to the static data area of the signaler, then the handler may view or modify the static data. Though the handler may not actually know how the signaler is implemented, it may be able to access private data within the signaler without using the signaler's interface.

4.3 Modularity

The goal for modularity is to ensure that the changes in one module have little effect on other modules. However, exception handling often increases the coupling between modules due to evolution of the modules. At the time the modules are designed, interfaces can be adhered to. After a module is designed and implemented, evolution in the form of incremental change can occur, which is one of the advantages of object-orientation. However, evolution of a design may require the redefined methods or new implementation of an object to signal some additional exception conditions to its users. The non-updated user modules of such an object may still continue to rely upon the validity of the original interface of the object. Evolution related problems can occur in three ways:

- *Exception evolution*: evolution of exception signaling by a method, leading to signaling of new exceptions while conforming to the current exception interface.
- *Function evolution*: evolution of the functionality of a method, leading to the definition and signaling of some new exceptions by the method not in the current exception interface.
- *Mechanism evolution*: evolution of the underlying implementation (or mechanism) used by a method, leading to the overloading of existing exceptions in the current exception interface.

Mechanism evolution is similar to function evolution and exception evolution. The difference with function evolution is that function evolution signals new exceptions not in the existing exception hierarchy, while mechanism evolution signals new exceptions that must be in the existing exception hierarchy. Mechanism evolution is different from exception evolution in the sense that it overloads the meaning of an existing exception.

4.3.1 Exception Evolution

Exception evolution is the incremental changing of exception signaling from a method. A typical example is exception specialization (creating a more specialized exception by subclassing from an existing one). To uphold modularity, the specialized exceptions should preserve *exception conformance* to remain compatible with existing exception handlers in any user methods and modules. The basis of exception specialization is that only those modules that want to use the specialization need to be changed, and other modules should not be affected. However, exception specialization can increase module coupling due to the side-effect the new exception creates because the new exception has to be handled by existing handlers in unchanged user modules.

The code of Figure 1 is used to illustrate, through an example, the issues that can arise due to exception evolution. This code is from a module that calls the *open* function of a file system module to open a file. Assume that *open* can signal the exceptions *FileErr* or *InputErr*. *FileErr* indicates a fatal error, while *InputErr* indicates a user error, e.g., non-existent path or filename, and is not fatal. The *FileErr* exception handler terminates the program, while the *InputErr* exception handler prompts the user for a new filename and retries the file open operation. When a file is already opened, *open* signals a *FileErr* exception. After the program in Figure 1 is designed, *open* is modified by the file system module so that an already opened file is not a fatal error. Therefore, a new specialized exception named *FileOpened* is defined indicating an already opened file. *FileOpened* cannot be derived from *FileErr* because *FileOpened* is not a fatal error. So, *FileOpened* is derived from *InputErr*. However, in Figure 1, if *FileOpened* is signaled, the *InputErr* handler gets invoked because *FileOpened* is derived from *InputErr*. The handler prompts the user for a filename, which is irrelevant for an already opened file. For *FileOpened* to be correctly handled by the modules that use this new version of the file system, all *InputErr* handlers in those modules need to be examined to determine the effect of signaling *FileOpened*. This illustrates an increase in module coupling.

```
try {
    open(file);
} catch(FileErr)  {...};
  catch(InputErr) {...};    // prompt filename and reopen
```

Fig. 1. Example of exception specialization problem

4.3.2 Function Evolution

Function evolution is the incremental changing of the functionality of a method. It affects exception handling because unchanged modules may have to cope with new exceptions that are introduced by the new functionality. An example of function evolution is the definition of polymorphic methods that incorporate new functionality. Virtual methods can be redefined as new subclasses are created, and can implement a completely new function. A redefined virtual method can signal a new exception that is not within any existing exception hierarchy of the method.

Figure 2 is a modified example from [22], and it is used here to illustrate the problems arising due to function evolution. The code shown there is from a module that uses an object of class *Window*. Class *Window* has a virtual method *drawFrame* that draws a window on the terminal. The code block shown in Figure 2 draws a window, and has an exception handler for *DrawErr* if the drawing operation fails. Suppose that the virtual method *drawFrame* is redefined by a new class, *TitleWindow*, that is subclassed from *Window*. *TitleWindow* places a text title in the window, and its version of *drawFrame* can signal *DrawErr* and a new exception, *TitleErr*, which corresponds to the increase in functionality provided by *TitleWindow*. If the code block in Figure 2 invokes *TitleWindow*'s version of *drawFrame*, a *TitleErr* exception would not be expected. All modules that could potentially use *TitleWindow* have to be examined to determine the effects of *TitleErr*. In general, exception handling in modules using *Window* objects may need to be changed appropriately, which may imply an increase in module coupling.

```
try {
      win.drawFrame()
} catch(DrawErr) {...}
```

Fig. 2. Example of function evolution

4.3.3 Mechanism Evolution

Mechanism evolution is the incremental changing of the implementation of a method, possibly using a different set of mechanisms to implement the same functionality. The problems in mechanism evolution arise when the new mechanism uses the existing exception hierarchy in the system. The new mechanism may have to overload the meaning of existing exceptions in the system, with the possible result of an unintended handler action by the user modules.

The code in Figure 3 is used here to illustrate the problems arising from mechanism evolution. The procedure *proc*, shown in this figure, reads from a file and transmits the data across a network to a target machine. It uses a file system module and a network module. There are two main classes of exceptions: *FileErr* (error signaled from the file system) and *CommErr* (error with the network communications). The code shown in Figure 3 has handlers for both types of exceptions. Suppose that the *FileErr* exception context contains the file name for which the *open* operation failed, and similarly *CommErr* contains information about the failed connection request. The handler for *FileErr* assumes the file to be read is non-existent or bad and creates it. The handler for *CommErr* assumes that the connection that is indicated by the exception object has a failure, and it then retries that connection. After the procedure *proc* is designed, suppose that the file system module is reimplemented as a network file system and the new implementation now also signals *CommErr,* which is a currently defined exception in the system, to also indicate a network error encountered by the file system. Now *FileErr* and *CommErr* may have overloaded meanings in the modules, such as *proc*, using the file system. This is illustrated in the situation when the network file system times out.

When the network file system times out, it can signal either *FileErr* or *CommErr*. Unfortunately, either exception can cause the wrong action to be performed by *proc*. If *FileErr* is used, then *proc*'s handler may attempt to recreate the file, and will fail again. When the network error that caused the timeout is corrected, the file system will signal another exception due to trying to create or overwrite an existing file. If *CommErr* is signaled to indicate the timeout, then the connection will be retried through the *CommErr* handler. If connection information is passed to the handler through the exception object, the connection that is to be retried would be the one to the network file system. In many network file systems, directly connecting to it is not allowed, so the connection may fail repeatedly inside the handler. The handler may go into a loop, trying the connection only to have it refused. This shows that changing an underlying mechanism may increase module coupling if existing exceptions are overloaded.

It is easy to see that exception conformance inhibits evolution. If exception non-conformance is allowed, would the effect be any different? Probably not, since the exception would not be recognized and eventually converted into an unhandled exception. What is needed is exception non-conformance and the ability to add exception handlers to existing code without the need to re-compile the code. Beta [15] supports this kind of augmentation of code, using its *inner* mechanism.

```
proc() {
        ...code...
        try {
                ...code...
                open(file);
                open(connection);
                ...code...
        } catch(FileErr)    {...}  // assumes file is bad or non-existent
          catch(CommErr) {...}  //assumes network connection terminated
```

Fig. 3. Example of mechanism evolution

4.4 Inheritance

The inheritance goal is to promote code reuse and conceptual specialization. The aspect to be considered here is the ***inheritance anomaly***. An inheritance anomaly occurs when a subclass method has to re-implement a parent class method to get the derived class's functionality [16]. The anomaly can occur when a subclass's exception handling replaces rather than augments the parent's handling of exceptions. Beta [15] is one of the few languages that supports augmentation of the parent class code by a subclass, rather than completely rewriting the method's code. Therefore, some of the problems mentioned here do not apply to Beta. The inheritance anomaly due to exception handling can occur in two ways:

- A subclass wants to specialize or handle an exception that a parent class method is signaling.
- A virtual method is redefined to signal a new exception that is not expected by any of the ancestor class methods calling that virtual method.

To illustrate the first case, say a subclass wishes to signal a specialized parent class exception, but still inherit the parent's handling. In Figure 4, the *put* method from class *Buffer* is shown. All exceptions from *put* are resignaled as *PutErr*. Suppose that *BetterBuffer* is derived from *Buffer*. *BetterBuffer* wishes to add a new exception to its *put* method that further specializes memory exceptions. *BetterBuffer* does not want to re-implement the *put* method, only to signal a specialized exception. The inheritance anomaly may occur if there is no way for *BetterBuffer* to signal the new exception. The parent *put* method only signals *PutErr* exceptions, and *BetterBuffer* does not receive any other exceptions from *Buffer*. So, *BetterBuffer* cannot resignal any specialized exceptions. Prior to signaling *PutErr* from *Buffer*, it is needed to augment the exception handling of *Buffer* with the extensions desired by the *BetterBuffer* implementation. Here also we notice that Beta supports this kind of augmentation of parent methods using the *inner* mechanism.

Lastly, the inheritance anomaly may occur due to a virtual method call in a parent method. With virtual methods, the subclass may be designed after the parent class. If a redefined virtual method in a subclass signals certain exceptions not handled by a parent class method calling this virtual method, then the parent class method may need to be re-implemented. Using Figure 4 again, *put* uses a virtual method for *createElement*. If *BetterBuffer* has a virtual method for *createElement*, and that version signals a new exception unknown to the *put* method of *Buffer*, then *put* may need to be re-implemented. Suppose that the *createElement* method for *BetterBuffer* can signal a new exception indicating a buffer element creation error. If the parent *put* receives this exception (due to a virtual method call to *createElement*), *put* resignals the exception as *PutErr*. Users of *BetterBuffer* may be expecting the new exception, which will not be signaled by *put*. The anomaly is that the parent *put* may need to be re-implemented to accommodate the requirements of *BetterBuffer* users.

```
class Buffer
    put() throws PutErr {
            try {
                ...
                self.createElement();
                ...
            } catch(...) { throw PutErr; }
```

Fig. 4. Example of inheritance anomaly

5. Summary

In this paper, exception handling in object-oriented languages is examined to determine which object-oriented techniques do not quite fit with exception handling. The goal is to be able to better identify the aspects of exception handling that make it hard to use or understand. This identification may be of value to programmers and language designers who wish to use or design exception handling in a robust way.

Four aspects of exception handling that are different from normal object-orientation are identified:

- Complete exception specification: extra information may be needed in the exception specification (via the exception context) to provide information for the handler that is beyond what is in the object interface.
- Partial states: exception handling cannot always assume that state transitions are atomic.
- Exception conformance: functions can be overloaded to have similar meaning in a wide variety of situations, but exception information (particularly for errors) generally needs to be specific.
- Exception propagation: propagation can change the control flow, putting an extra burden on the programmer to understand at least two control paths: the normal execution path, and the exception handling path.

The above four differences can affect the major elements of object-orientation. The examination of the effects of exception handling on some of the object-oriented design elements leads to the following conclusions:

- Abstraction: generalization of operations and composition construction may involve changing of abstraction levels and dealing with partial states.
- Encapsulation: the exception context may leak information that allows implementation details or private data of the signaler to be revealed or accessed.
- Modularity: design evolution (function and implementation) maybe inhibited by exception conformance.
- Inheritance: the inheritance anomaly can occur when a language does not support exception handling augmentation in a modular way.

Acknowledgments

The authors wish to express gratitude to the reviewers for their insightful comments and detailed suggestions in revising this paper to its current form. The first author is also grateful to IBM in supporting his efforts. The second author wants to thank NSF for providing him an opportunity to work on this paper.

References

[1] G. Booch, "Object-Oriented Analysis and Design with Applications, 2nd Ed", Benjamin/Cummings, 1994.

[2] A. Borgida, "Exceptions in Object-Oriented Languages", *SIGPLAN Notices*, vol. 21, no. 10, pp. 107-119, Oct. 1986.

[3] T. Cargill, "Exception Handling: A False Sense of Security", *C++ Report*, vol. 6, no. 9, pp. 21-24, Nov.-Dec. 1994.

[4] N.H. Cohen, "Ada as a Second Language, 2nd Ed", McGraw-Hill, 1996.

[5] F. Cristian, "Exception Handling", *IBM Research Report RJ5724*, 1987.

[6] C. Dony, "An Object-Oriented Exception Handling System for an Object-Oriented Language", in *Proceedings of ECOOP'88*, pp. 146-159, 1991.

[7] C. Dony, "Exception Handling and Object-Oriented Programming: Towards a Synthesis", in *Proceedings OOPSLA 90,* pp. 322-330, Oct. 1990.

[8] A. Goldberg and D. Robson, "Smalltalk-80: The Language", Addison-Wesley, 1989

[9] J.B. Goodenough, "Exception Handling: Issues and a Proposed Notation", *Communications of the ACM*, vol. 18, no. 12, pp. 683-696, Dec. 1975.

[10] J.D. Ichbiah, J.C. Heliard, O. Roubine, J.G.P. Barnes, B. Krieg-Brueckner, and B.A. Wichmann, "Rationale for the Design of the Ada Programming Language", *SIGPLAN Notices*, vol. 14, no. 6, Part B, Jun. 1979.

[11] A.K. Jones, "The Object Model: A Conceptual Tool for Structuring Software," in Operating Systems and Advanced Course - Lecture Notes in Computer Science, Vol. 60, pp. 7-16, Springer-Verlag, 1979.

[12] J.L. Knudsen, "Better Exception-Handling in Block-Structured Systems", *IEEE Software*, vol. 17, no. 2, pp. 40-49, May 1987.

[13] S. Lacourte, "Exceptions in Guide, an Object-Oriented Language for Distributed Applications", in *Proceedings ECOOP 91*, pp. 268-287, 1991.

[14] B. Liskov and A. Snyder, "Exception Handling in CLU", *IEEE Transactions on Software Engineering*, vol. SE-5, no. 6, pp. 546-558, Nov. 1979.

[15] O.L. Madsen, B. Moller-Pedersen, and K. Nygaard, "Object-Oriented Programming in the Beta Programming Language", Addison-Wesley, 1993.

[16] S. Matsuoka and A. Yonezawa, "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages", *Research Directions in Concurrent Object-Oriented Programming*, chapter 4, MIT Press, 1993.

[17] B. Meyer, "Eiffel: The Language", Prentice-Hall 1992.

[18] T. Ritchey, "Programming with Java!", New Riders, 1995.

[19] A.B. Romanovsky, L.V. Shturtz, and V.R. Vassilyev, "Designing Fault-Tolerant Objects in Object-Oriented Programming", in *Proceedings 7th International Conference of Technology of Object Oriented Languages and Systems (TOOLS Europe 92)*, pp. 199-205, 1992.

[20] B. Stroustrup, "The C++ Programming Language, 2nd Ed.", Addison-Wesley, 1991.

[21] B. Stroustrup, "The Design and Evolution of C++", Addison-Wesley, 1994.

[22] A. Taivalsaari, "On the Notion of Inheritance", *ACM Computing Surveys*, vol. 28, no. 3, pp.438-479, Sept. 1996.

[23] S. Yemini and D.M. Berry, "A Modular Verifiable Exception-Handling Mechanism", *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 2, pp. 214-243, Apr. 1985.