

# Session Summary: Exception Propagation

Chair: William Bail  
The MITRE Corporation  
1820 Dolley Madison Blvd  
McLean, VA 20190-5214  
USA  
wbail@mitre.org

Rapporteur: Bo I. Sandén  
Colorado Technical University  
4435 N. Chestnut Street  
Colorado Springs, CO 80907-3896  
USA  
bsanden@acm.org

## 1. Background

As an introduction to this session, a brief background of the topic was discussed.

Propagation deals with the communication of information regarding unexpected happenings from one context to another within a software system.

For exception handling, there are three dimensions to the process of design:

- Information content** – what to tell
- Receptors** – whom to tell
- Strategy** – when to tell

Within these three dimensions, there are four primary design considerations:

**Responsibility** – Which units of the structure detect the unexpected event, which units communicate the events, and which units perform recovery?

**Granularity of information** – How much detail needs to be in the information that is communicated regarding the unexpected event? Fine-grained information may be necessary for recovery purposes, while more coarse-grained information may be sufficient for simple indication of errors.

**Types of information** – The nature of the information propagated needs to be decided. Possibilities include describing the origin of the exception, the cause of the exception, or the recovery strategy to be followed.

**Exception mapping** – What transformations or manipulations of the exception information are performed as it passes over context boundaries within the system? This aspect is at the core of designing propagation into a system, and is driven

by decisions made for the first three considerations.

Mapping of exceptions across program segment boundaries, that is, the propagation of the exceptions, can take place in four general ways.

- **Refinement** - where the exceptions are filtered, with some being stopped and others being propagated.
- **Generalization** - where the exceptions are aggregated to create exception abstractions.
- **Translation** - where the exceptions are mapped one-to-one but with a change of name or information content.
- **Preservation** - where the exceptions are mapped unchanged.

## 2. Issues/problems/considerations

### 2.1 Language features for exception propagation

Considering the current state of propagation design and the features provided by modern programming languages (especially Ada 95) to control propagation, one primary question is whether we currently have enough control, and the right flavors of control, to adequately support our design styles. In other words, what do we lack in terms of language features? The presence of anonymous exceptions often causes problems, but a key question is whether anonymous exceptions are necessary or desirable to support the development of safe systems.

### 2.2 Non-sequential models of computation

Another set of issues relating to the control of propagation deals with the emergence of non-sequential models of computation, including concurrent programming and the application of object oriented designs and language features. The issue is whether our current models of propagation are adequate for such

designs. These current models were formulated in the context of sequential executions, but the presence of concurrency and the structural characteristics of OO designs challenge our understanding of these models.

### 2.3 Semantic contents of exceptions

A third set of problems concerns our ability to control the semantic content of exceptions as they propagate. The goal is to be able to adjust the levels of information as the exceptions propagate across the system, changing the level of granularity appropriately. The question is whether we have enough control of the semantics under the current exception handling schemes provided by Ada 95 and other modern languages. What more control, and of what type, do we really need? There is a danger of course if too much control is provided to the software designer, so an associated issue is how to decide when we have too much control. Ada 95 is particularly strong in providing just enough control for general programming structures, and keeping control back when such control is unsafe. But is this balance maintained for exception handling? Specifically, what types of language features should be supported?

### 2.4 Propagation to multiple clients

There are other issues to consider when evaluating propagation features. Specifically, what capabilities are needed for propagation to multiple clients? Current features are limited in allowing propagation to only one client at a time, but programs with concurrent designs or with OO structures make propagating to more than one client a possibility.

### 2.5 Efficiency of propagation

Another issue relates to the efficiency of propagation. What tradeoffs need to be made to provide flexible propagation options to developers, while ensuring that use of these features maintains execution efficiency?

## 3. Session themes

The discussion covered the following general themes:

- Anonymous exceptions
- "**raises**" or "**propagates**" clauses
- Exceptions in multi-threaded transactions
- Using metaobjects for exception handling

Each is discussed in the following subsections.

### 3.1 Anonymous exceptions

Anonymous exceptions are the coarsest grained. That is, they provide the smallest amount of information to the client (caller). The information consists simply of the fact that an exception occurred.

*Currie:* Anonymous exceptions are not that different from "program error". Anonymous exceptions are easy to catch with "others". When handling one, you have to "assume the worst".

*Bill:* It boils down to the contract made between the client and the server. Notification takes place, but there is no information regarding recovery strategies.

It was discussed whether anonymous exceptions are useful or should be replaced by another concept. It is not known however whether there is a more general model that would provide more control to the developer and still be able to capture those unknown and unpredicted events that accompany anonymous exceptions.

### 3.2 "raises" or "propagates" clauses

A **raises** or **propagates** clause (similar to Java's **throws**) would make exceptional outcomes of a subprogram (S) more visible. It cannot always be checked by a compiler because of situations where S calls T, which calls U, etc., so it is dubious whether it is better than a comment. A **raises** or **propagates** clause as opposed to a comment would have value if each exception required a handler other than an "others" clause ("others" clauses being at the coarsest level, representing a "last resort" solution).

In designing such a clause the exact semantics would be unclear. Can the code also raise exceptions that are not in the list? Presumably, predefined exceptions do not need to be specified. Exceptions raised in the subprogram could be mapped onto those specified. But the exact protocol between procedure and caller needs to be carefully specified.

*Currie:* Such clauses put the caller on notice that you may have to deal with these exceptions. That is, they are potentially raisable.

But it is not always the caller's problem. The reason for the exception may be internal and should be handled internally.

### 3.3 Exceptions in multithreaded transactions

A multithreaded transaction commits if all threads finish successfully and aborts otherwise. Exception

termination in any thread aborts the transaction. The abort propagates the exception to the enclosing scope. Resolution is used if there are simultaneous aborts in different threads.

If the transaction is implemented by means of a controlled transaction object (T), then it requires a block, which is also an exception context. The situation where the “ready to commit” is never received from one task raises an exception in the finalization of T.

*Jörg:* In object-based interfaces, the code cannot raise another exception at the object or type level. Would it be an advantage to be able to encapsulate the transaction in the type itself, and handle the exceptions directly at the type level rather than propagating it onwards to the calling routine? Currently the type cannot capture the exception and raise another one prior to propagation outside of the type.

### **3.4 Use of metaobjects for exception handling**

Mitchell, Burns, and Wellings submitted a position paper (titled "MOPping up exceptions") that addresses the issue of responsibility. In their position, they questioned whether exception handling should be a distinct part of program. The paper proposes a reflective model of computation where exceptions are raised at the normal computation level, but handled at a metalevel. It is sometimes necessary to then call the metaobject of the caller.

After an exception has been handled there is a need to return to the real object level. Because this is not an Ada paper, whether to return to the place where the exception was first raised is an open question. One could conceivably return to and execute a corrected method. If applied in an Ada context, this would have to tie in with the Ada philosophy of no automatic return.

Anonymous exceptions are not included in this model. There is no reason to put them in if the model is implemented from scratch.

## **4. Conclusion**

The discussion during this session focused heavily on anonymous exceptions. Clearly there is uncertainty and disagreement about the value of this feature, but a resolution is not imminent. Overall, the semantics of anonymous exceptions are at the coarsest level - such exceptions provide very little information about the anomalous event. Consensus is that this level is not

suitable for all models of programming, and that work needs to be done to find finer-grained solutions.