Effective Java Exceptions

Pages: 1, 2, **3**

Contingency Handling in Your Architecture

With fault processing relegated to the barrier, contingency communication between primary components becomes much simpler. A contingency represents an alternative method result that is just as important as the principal return result. Therefore, checked exception type is a good vehicle to convey the existence of a contingency condition and supply the information needed to contend with it. This practice enlists the help of the Java compiler to remind developers of all aspects of the API they are using and the need to provide for the full range of method outcomes.

It is possible to convey simple contingencies by using a method's return type alone. For example, returning a null reference instead of an actual object can signify that the object could not be created for a defined reason. Java I/O methods typically return an integer value of -1 instead of a byte value or byte count to indicate an end-of-file condition. If your method's semantics are simple enough to allow it, alternative return values may be the way to go, since they eliminate the overhead that comes with exceptions. The downside is that the method caller is responsible for testing the return value to see if it is a primary result or a contingency result. The compiler will not insist that the method caller makes that test, however.

If a method has a void return type, an exception is the only way to indicate that a contingency occurred. If a method is returns an object reference, the vocabulary that the return value can express is limited to two values (null and non-null). If a method returns an integral value, it may be possible to express several contingency conditions by choosing values that are guaranteed not to conflict with the primary return values. But now we have entered the world of error code checking, something the Java exception model was developed to avoid.

Supply something useful

It made little sense to define different fault reporting exception types, since the fault barrier treats them all the same. Contingency exceptions are quite different, because they are meant to convey diverse conditions to method callers. Your architecture would probably specify that these exceptions should all extend `java.lang.Exception` or a designated base class that does.

Do not forget your exceptions are complete Java types that can accommodate specialized fields, methods, and even constructors that can be shaped for your unique purposes. For example, the `InsufficientFundsException` type thrown by the imaginary `CheckingAccount processCheck()` method could include an `OverdraftProtection` object that is able to transfer funds needed to cover the shortfall from another account whose identity depends on how the checking account is set up.

To log or not to log

Logging fault exceptions makes sense because their purpose is to draw the attention of people to situations that need to be corrected. The same cannot be said for contingency exceptions. They may represent relatively rare events, but every one of them is expected to happen during the life of your application. If anything, they signify that the application is working the way it was designed to work. Routinely adding logging code to contingency catch blocks adds clutter to your code with no actual benefit. If a contingency represents a significant event, it is probably better for a method to generate a log entry recording the event before throwing a contingency exception to alert its caller.

Exception Aspects

In Aspect Oriented Programming (AOP) terms, fault and contingency handling are crosscutting concerns. To implement the fault barrier pattern, for example, all the participating classes must follow common conventions:

The fault barrier method must reside at the head of a graph of method calls that traverses the participating classes.
They must all use unchecked exceptions to signify fault conditions.
They must all use the specific unchecked exception types that the fault barrier is expecting to receive.
They all must catch and translate checked exceptions from lower methods that are deemed to be faults in their execution context.
They must not interfere with the propagation of fault exceptions on their way to the barrier.
These concerns cut across the boundaries of otherwise unrelated classes. The result is minor bits of scattered fault handling code and implicit coupling between the barrier class and the participants (although still a great improvement over not using a pattern at all!). AOP allows the fault handling concern to be encapsulated in a common Aspect applied to the participating classes. Java AOP frameworks such as AspectJ and Spring AOP recognize exception handling as a join point to which fault handling behavior (or advice) can be attached. In this way, the conventions that bind participants in the fault barrier pattern can be relaxed. Fault processing can now reside within an independent, out-of-line aspect, eliminating the need for a "barrier" method to be placed at the head of a method invocation sequence.

If you are exploiting AOP in your architecture, fault and contingency handling are ideal candidates for aspects that apply throughout an application. A full exploration of how fault and contingency handling could work in the AOP world would make an interesting topic for a future article.

Conclusion

Although the Java exception model has generated spirited discussion during its lifetime, it provides excellent value when it is applied correctly. As an architect, it is up to you to establish conventions that get the most from the model. Thinking of exceptions in terms of faults and contingencies can help you make the right choices. Using the Java exception model properly will keep your application simple, maintainable, and correct. Aspect Oriented Programming techniques may offer some definite advantages for your architecture by recognizing fault and contingency handling as crosscutting concerns.

References

Sun's Exception Tutorial - all of the basics on Java exceptions
Does Java Need Checked Exceptions? - Bruce Eckel argues against checked exceptions in Java
Exceptional Java - a good discussion of these topics and an architectural exceptions policy to emulate
The Exceptions Debate - more exceptions background from developerWorks
The Apache Struts Web Application Framework - the source for Struts information
The Spring Framework - the source for the information on Spring
Wikipedia: Aspect Oriented Programming - a good introduction to AOP concepts
The AspectJ Project - the source for the information on AspectJ

Barry Ruzek has been named a Master Certified IT Architect by the Open Group. He has over 30 years of experience developing operating systems and enterprise applications.