

▶▶ QUICK LOOK

- Detecting program state corruption
- Designing tests to check exceptions

Testing for Exceptions

by Keith Stobie

Everyone knows that testing error or exceptional conditions can be quite fruitful in discovering defects, since the error-handling

code is usually exercised far less often—if at all. The *IEEE Transactions on Software Engineering* stressed exception handling's importance in one of its recent calls for papers:

For a variety of reasons, not least among which is the fact that more than half of the code is often devoted to exception detection and handling, many failures are caused by the incomplete or incorrect handling of these abnormal situations. The requirements for the correct system behavior during exception handling are in some sense even higher than for the system operating in a normal mode.

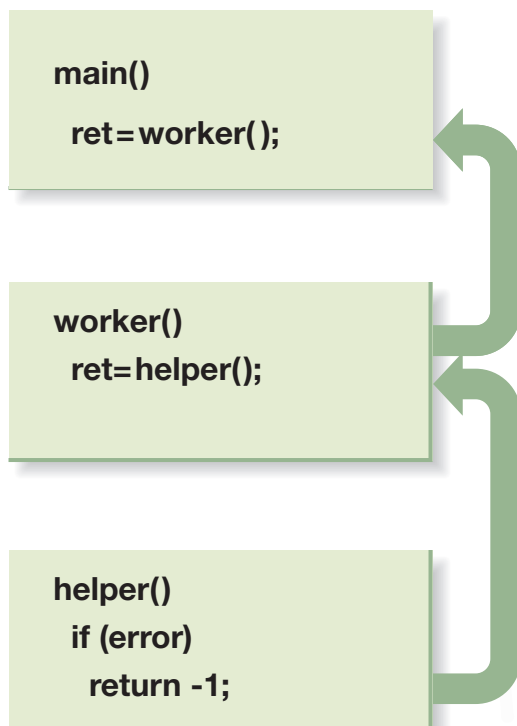


FIGURE 1 Returning error codes

In older languages like *C*, programmers would have each routine that detects an error return an error code to its caller. That routine could either handle the error or pass it on, in turn, to its caller, as shown in Figure 1.

Many modern languages like *C++* and Java have introduced a formal notion of “exceptions.” The new exception mechanisms dynamically transfer control abruptly from one part of a program to another. The programmer of a routine doesn’t need to be aware that an exception is propagating through the routine.

Figure 2 (on page 14) shows the oversight a programmer might make. He has written the routine named **worker** without considering that the called routine **helper** raises an exception. When the exception is raised, compiler-provided code that’s part of **helper** first does some invisible cleanup. The exception then propagates to **worker**, where some more compiler-generated code does some more invisible cleanup. The problem is that it may not be all of the cleanup that’s necessary in **worker**. The programmer should have done some manual cleanup in response to the exception. However, if the programmer overlooks that, the exception propagates to **main**, and the remainder of **worker**’s required cleanup is never done. The program is left in a state the programmer did not intend.

The corruption of state by an exception was dramatically brought home to our testing group while we were testing a newly implemented feature for our Java CORBA client based on the CORBA Event Notification service. The Test Design Specification described many tests for this new feature. Some of the tests had the test code call the **worker** routine with illegal data. The **worker** routine then called the **helper** routine with the same illegal data. The **helper** was expected to raise an exception, which it did.

One of our testers, Hoa Nguyen, noticed during testing that passing in illegal data twice in a row caused the **worker** to get an internal error (effectively crash) the second time through.

This discovery, like many in testing, was serendipitous (unplanned). Hoa was unfamiliar with the constraint on the data and didn’t realize the data he had picked was not legal. The Test Design already called for verifying the exception, and that test had already been created, executed, and passed before Hoa began his task!

Was the test for the exception designed or implemented incorrectly? No. It correctly set up the condition and verified the exception. What was missing was the second call. The problem was that the first call provoked an exception that **worker** should have done extra cleanup for. Since it didn’t, the second call encountered bad state.

ANNIE BISSETT

Bug Report

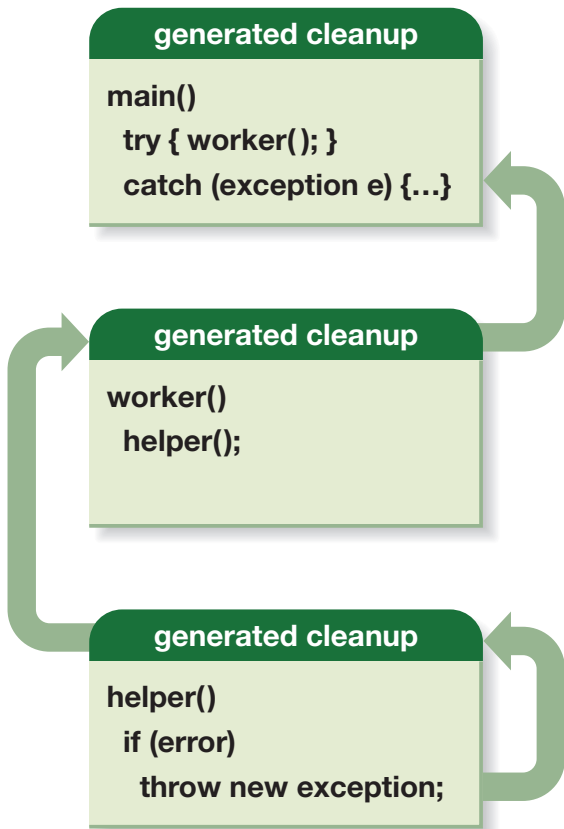


FIGURE 2 Exception propagation

Our team discussed this at our monthly improvement meeting, and we decided to create a pattern that could help us avoid missing this type of problem in the future. We had also seen cases in the past in which badly implemented tests would generate and verify an exception—but not notice that the product would then stop functioning. The problem was related to the state of the system after the missing cleanup for the exception.

Thinking in terms of test requirements (a term described in Brian Marick’s *The Craft of Software Testing* as “useful sets of input that should be tested...for example, ‘X is an even number’ or ‘the list must have a single element’”), we came up with the test requirements shown in Figure 3. They include the cases we’ve seen and described so far:

- exception → exception Does the exception leave the program state such that normal calls are okay, but **worker** can no longer handle the exception?
- exception → normal Does the exception screw up the program state such that normal calls will no longer work?

Two of the six test requirements were generally covered in our functional and system testing:

first call to **worker** →1→ normal return
 previous call returns normally →6→ current call returns normally

Note: In these examples the numbered arrows indicate a reference to the six requirements listed in Figure 3: A →3→ B →4→ C, for example, should read “A is followed by B (which satisfies requirement 3), then B is followed by C (which satisfies requirement 4).”

Our previous testing only guaranteed one of **2**, **3**, or **5**, because all of those lead to an exception being raised. Our new test pattern for testing every exception became:

first call to **worker** →2→ call raises exception →4→ call returns normal →5→ call raises exception →3→ call raises exception →4→ call returns normal

The above pattern covers all of the requirements where an exception can occur and where normal calls work after one or more exceptions.

We assume there are other normal return → normal return tests for covering **1** and **6**. The “normal return” case can also be expanded to try different ways of making the call.

We began incorporating this test pattern into our new test designs. At a recent test case inspection, we noticed that a test was not following the new pattern and we suggested it be changed to use the pattern. In addition, in this particular case the exception could be generated in one of two ways and only one way was being tried. When we used the pattern, there was plenty of opportunity to try both ways. The test verifies that **worker** returns the exception **java.rmi.NoSuchObjectException** generated by an Enterprise Java Beans (EJB) **remove** method as specified by Sun Microsystems:

“If a client makes a call to a session object that has been removed, the Container should throw the java.rmi.NoSuchObjectException to the client.”

A session object can be removed in two ways:

“A client may remove a session object using the remove () method on the javax.ejb.EJBObject interface, or the remove(Handle handle) method of the javax.ejb.EJBHome interface.”

Our test pattern asks for requirement **2** to be satisfied first. We can’t, because the **remove** method must be called with a normal return before the exception can be generat-

The arrows should be read as “followed by”:

1. first call to **worker** → normal return
2. first call to **worker** → exception raised
3. previous call raised exception → current call raises exception
4. previous call raised exception → current call returns normally
5. previous call returns normally → current call raises exception
6. previous call returns normally → current call returns normally

FIGURE 3 Test requirements for exceptions

Bug Report

ed. You can't try removing a removed object *until you've removed an object*. (In other words, in order to call a method on the handle of an object to remove the object, you must first get a handle to the object. The handle is only returned while the object exists. So first the object must be created and its handle obtained. Then the request to remove the object can be issued using the handle. Even after the remove, you still have access to the handle, so you can—mistakenly—try the removal again.)

Applying the previous pattern as a guide, we created a test case with:

```
// -1► normal return
// must first create EJB from its Home.
CommonStatelessRemote StatelessBean = ComStatelessH.create();
BeanCheck.tryValidStatelessBean(StatelessBean);
StatelessBean.remove(); // remove on EJBObject interface.

// -5► exception
BeanCheck.tryInvalidStatelessBean(StatelessBean); //contains remove

// -4► normal return
StatelessBean = ComStatelessH.create();
BeanCheck.tryValidStatelessBean(StatelessBean);
//remove via home
Handle statelessHandle = StatelessBean.getHandle();
ComStatelessH.remove(statelessHandle);

// -5► exception
BeanCheck.tryInvalidStatelessBean(StatelessBean); // contains remove
// -3► exception
BeanCheck.tryInvalidStatelessBean(StatelessBean); // contains remove

// -4► normal return
StatelessBean = ComStatelessH.create();
BeanCheck.tryValidStatelessBean(StatelessBean); // contains remove

//cleanup
StatelessBean.remove();
The routines tryValidStatelessBean and tryInvalidStatelessBean encapsulate the concept of normal return and excep-
```

tion return. They remove the bean and check that the correct behavior is observed.

```
// Invokes a method on a stateless bean and verifies correct data
// returned.
public static void
tryValidStatelessBean(CommonStatelessRemote bean)
    throws RemoteException, Exception;

// Invokes a method on a stateless bean that does not exist and verifies
// java.rmi.NoSuchObjectException exception is returned.
public static void
tryInvalidStatelessBean(CommonStatelessRemote bean)
    throws Throwable;
```

An earlier risk analysis judged **home.remove()** only medium risk since even if there were a defect, an easy workaround existed (use the **session.remove()** method). By using the pattern, we could test both routines, even though we hadn't originally planned to test **home.remove()**. We serendipitously found that **home.remove()** didn't work.

Although we have insufficient statistical information to justify our new test pattern, we have enough war stories of exception handling causing problems to justify it.

The basic problem with exception handling is that it is difficult! Exception handling in modern languages makes it easy to drastically change the contents of memory. The next instruction executed may be very distant from the site of the exception, and required cleanup might not be done. In C++ the problem can be particularly acute, with lost memory not reclaimed correctly. For these reasons, it's critical for good testing of exception handling that we test all representative sequences of normal and exceptional calls. **STQE**

Keith Stobie (kstobie@acm.org) is the QA Process and Test Architect at BEA Systems, where he directs QA and Test process and strategy. An ASQ Certified Software Quality Engineer, Keith has instructed on Testing and Inspections, and he has presented at a range of quality and testing conferences.