It's Testing Time!

Patterns for Testing Software

Author: Manfred Lange
Gemplus GmbH
Otto-Lilienthal-Strasse 36
71034 Boeblingen
Germany
e-mail: Manfred_Lange@acm.org

Revision: 1.78

Last saved: 6/18/01 6:39 AM

Introduction

Have you ever heard of defect free software? I have not. I think it is fair to say, that virtually there is no software that does not contain defects. However, we as software engineers still try to make as few errors as possible and we are also always looking for ways of improving the quality of what we deliver.

One way to improve software quality on the functional level is to have good tests in place. This paper does not cover everything that tests can be, e.g. design tests, black box test, white box tests etc. Instead, this paper contains a collection of test techniques, that I have not found elsewhere, but have proven to be useful to improve functional tests.

Sure, testing does not guarantee defect free software. In addition, tests should never be used to "test quality in". However, some scenarios occur repeatedly when testing software. This paper tries to describe a few of these patterns in order to allow for new perspectives on how to test software.

The first pattern *Separate Test Interface* shows how to employ a dedicated interface for test purposes. In addition to black box tests, a separate test interface allows for automatically inspecting internal state or performing additional tests for classes only visible inside of a software component.

The *Error Simulator* allows for simulating errors during the runtime of a component. This is especially useful, when testing a component in an environment with multiple threads or processes. If resources are shared among threads and processes, race conditions can lead to unexpected behavior during runtime. In some cases, a particular scenario cannot be tested by just having a test driver invoking methods in the component to be tested. Here the *Error Simulator* comes into play.

Testing a component can mean that errors occur in functions belonging to one of the lowest levels of a component. Here it is essential to have a tool at hand that allows finding out, which execution path (call stack) was used until the function in question has been called or what function path was executed just before the error occurred. This issue can be addressed using the *Call Stack Tracer*.

Finally, with the advent of Extreme Programming [Beck2000], test cases became even more important than in the past. Essentially, test cases are codified requirements. However, often programmers do not specify

the timely behavior of such test cases. The *Test Case with Time Specification* deal with this issue and makes important suggestions in the solution section.

The patterns presented in this paper are focused, but not limited to object-oriented programming languages.

Extreme Programming

The patterns described in this paper are a perfect fit for Extreme Programming (XP), where test cases build a central element of the methodology.

Especially *Test Case with Time Specification* is an excellent technique for improving functional or feature specification being written as test cases. The timely behavior of a test cases is often overlooked, when writing a test case.

In other words: As automated test cases form an integral part of XP, using the methodology creates an additional force for all of the patterns presented here: You want to run fully automated tests for XP.

Separate Test Interface

Motivation

Many software programmers use components as the mean for modularizing their software systems[1].

One result of using components is a set of interfaces implemented by one or more components. Then the some other components may use the same interface. These technique leads to very strong encapsulation, one of the major advantages of object-oriented programming.

However, all non-trivial components also contain classes, which do not expose any interface to the outside of the component. They are only used internally to that component.

On the other hand, you also want to be able to test the internal classes, without changing the official interfaces of the component.

In short: How do you test internal classes of components?

Forces

1. Black box testing of components is not sufficient

2. No interface available for forcing a component into a specific state

3. No interface available for checking particular parts of the internal state of a component

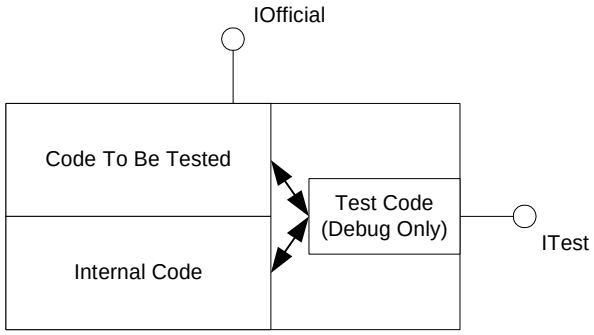
Solution

Introduce a dedicated test interface (*Separate Test Interface*). In general, this serves two purposes: First, it is now possible to gain access to internal storage and/or state in order to check the value of the storage or

[1] I use the term "component" for everything that can be used to build a software product. Typically, this is a binary – executable or runtime library – that implements more than one classes.

the internal state of the component. Second, the test interface can also be used to perform specific tests on classes that are visible only internally in the component.

As an example, consider a component that implements an interface IOfficial. This interface is the official interface of the component. However, there might be code that is only visible internally ("Internal Code"). Using the official interface the internal code is not accessible. Therefore, test cases cannot cover this code. The introduction of the separate test interface fixes this problem.



Note, that the test interface is not restricted to execute internal code only. It may well be, that the test code invokes also functions in the code that can be tested using the official interface, too.

The test interface can be left in the component, if the resulting binary size is not an issue. Otherwise you may want to set up your build environment in a way that you have anther version of the component that does not compile in all the code required for the ITest interface.

Consequences

1. (+) You have a mean to test classes that are internal to a component.

2. (+) You can put your component into states that you typically cannot achieve by using the official interface. This may be additional states, or it might be even additional states, that you need for testing only, e.g. for simulating that the component does accept additional client connections.

3. (+) You can automatically check internal states of you component. This is useful when executing test cases. Invoking a function of the official interface leads to a specific state of the component, e.g. "Activated". The official interface may not provide API's to query the internal state, however, for automated test cases, it is mandatory to find out, whether the invocation of a function leads to the expected result.

4. (-) You have one additional interface to implement. However, you would have had to implement the test cases anyhow, but would have used a different interface.

5. (-) With an interface being used for testing only, clients may be tempted to use the testing interface in production code. In order to avoid this, for production code the interface should indicate for its API's, that no implementation is available.

6. (-) The additional test interface may need additional code within the component. The binary will have a bigger size than without the test interface.

Implementation Issues

Adding a separate test interface is not complex.

However, preventing "normal" clients from using it for production purposes may be worth some additional effort.

One possible solution would be to compile code for the test interface conditionally. A build for production code would not contain any implementation, but would just return an error code indicating that there is no implementation. Here is some C++ code, how to do this:

```
HRESULT ITest::ExecuteTestCase1(DWORD param) {
#ifdef _TESTING
   // put the implementation of the test case here.
#else // !_TESTING
   return E_NOT_IMPL;
#endif // !_TESTING
}
```

Another possibility is to compile the interface conditionally. In this case, the interface would not even be visible. The client is not able to use it at all. The according C++ code would look like this:

```
#ifdef _TESTING
HRESULT ITest::ExecuteTestCase1(DWORD param) {
// put the implementation of the test case here.
}
#endif // _TESTING
```

If you decide to compile code or interface conditionally, you must make sure that you do not produce different behavior of the official interfaces.

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect and Network Documentation Tool (NDT).[2]

Related Patterns

Façade hides a set of complex interfaces behind an object, that itself exposes an interface that is easier to use. The *Separate Test Interface* makes invisible internal interface visible. Furthermore, the intention of using a *Separate Test Interface* is not to improve the design. In the first place, better testing was the goal. With *Façade*, the object was to improve the design of a system.

Error Simulator

Motivation

Systems, which consist of multiple threads and processes, and systems, which use shared resource inherently contain an additional set of errors that are caused by race conditions. Shared resources can be locks, mutexes, semaphores, critical sections, and so on.

A database application is a typical example for such a system. These applications run into problems, when deadlock situations occur. A deadlock happens for instance, when two processes want to access data for which the other process holds a lock already.

[2] At present, all these applications are not available standalone. All of them are part of service or support offerings.

This scenario – multiple threads and processes and shared resources – becomes much more complex, if time comes into play. Depending on the actual path of execution, in practice it is impossible to predict, when conflicts for accessing shared resources will occur. However, to test a component sufficiently, you also need to test the case, when a shared resource conflict occurs.

Example: Assume, that you have to use a certain interface for accessing a file. Multiple threads and/or processes are accessing the same file. All of them read from the file, lock (overlapping) portions of the file, and write to the file. Opening a file, seeking to the correct position, or opening a memory-mapped file requires a couple of operating system calls. Each of the calls can fail with different error conditions. However, in some cases it is not possible to test all thinkable combinations.

In short: How can you improve testing for complex race conditions?

Forces

1. You have many different race conditions, which you cannot simulate with test cases. E.g., a database system may have the ability to detect deadlock situations. In this case, one of the clients will be selected as victim, and the transaction of that client will fail and will need to be rolled back. It is not possible to write a test case for this, as the database system has no API to select the victim in advance.

2. You have error handling code in place, and you want to test it.

3. Because of a low density of conflicts, testing for all conflicts is very time consuming.

Solution

Provide a mechanism to simulate shared resources conflicts in the code.

To do this, implement your error handling in a way, that no matter where an error occurs, or what kind of error has occurred, the very same code is executed.

In that code, add code that is executed conditionally depending on a condition. The condition might be that a random number falls into a certain range. In this case, execute the test very frequently in order to ensure, that all branches of the error handling code have been executed at least once.

Consequences

1. (+) Errors are now simulated on a random basis. However, you will have to execute the code many times to have a close to 100 percent probability that all error conditions are tested. You should use a coverage analysis tool from time to time to ensure that the test case executes all paths.

2. (+) Error handling code can now be executed and tested as well.

3. (+) Testing is much faster, as the error density is much higher.

4. (-) All error handling must be routed through the same function, which might not be feasible for existing code, where error handling is scattered throughout the code.

Implementation Issues

The best way to implement the error simulator is, to have a list of all typical error codes available. Then, based on the random number, an error is chosen from the list and then reported to the client. The client then has to handle the error.

The following code shows the idea, in this case using a hard coded list consisting of two possible error conditions:

```
   ...
   DWORD result = OpenFile(. . .);
   HandleResult(result);
   ...

   void HandleResult(DWORD result) {
#ifdef _DEBUG
   // Error Simulator
   int rn = GetRandomNumberInRange(1,2);
   switch(rn) {
   case 1:
      // simulate wrong file name
      throw WrongFileNameException();
   case 2:
      // simulate access denied
      throw AccessDeniedException();
   }
#endif
```

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).[3]

Related Patterns

In *Controlled Exception Test* [Bind1999] a similar approach is taken. *Controlled Exception Test* runs the testee against an implementation that throws exceptions in a predefined way, thus ensuring that all possible paths of the exception handling are executed. After all paths have been tested the execution stops. Opposed to that the *Error Simulator* runs the testee against the production code.

Call Stack Tracer

Motivation

Assume you are using a runtime library that can throw exceptions[4]. The runtime library can throw exceptions in arbitrary functions. However, you have a multitude of places in your code, where you call functions of the runtime library.

In short: How do you find out, where the exception has been thrown (, while avoiding too much "noise" in the debug trace)?

Forces

1. You want to avoid "noise" in the debug trace.

[3] At present, all these applications are not available standalone. All of them are part of service or support offerings.

[4] Obviously, this technique works only for languages that provide exceptions, e.g. C++ and Java.

2. The tracing facility must be easy to use.

3. You want to exactly know, where the function was called, when the exception has been thrown.

Solution

Implement a class *CallStackTracer*, that internally traces the call stack, but writes to the debug trace only, if an exception has occurred.

The mechanism is as follows: An instance of the class is bound to a block (this works for at least C++ and SmallTalk). The constructor for that instance takes as a parameter information about where the instance has been created, e.g. file name and line number.

If everything is OK, at the end of the block, the function SetComplete() is invoked on the instance of *CallStackTracer*.

When the instance goes out of scope the destructor is called. In the implementation of the constructor, the instance checks, whether the SetComplete() function was called. If not, an exception must have been thrown, so the destructor of the instance writes the information, e.g. file name, line number, etc., to the debug trace. If the SetComplete() function was called before, then the instance traces nothing thus avoid the "noise" in the debug trace.

```
class CallStackTracer {
   CallStackTracer(string sourceFileName,
                   int sourceLineNumber) {
      m_sourceFileName = sourceFileName;
      m_sourceLineNumber = sourceLineNumber;
      m_bComplete = false;
   };
   ~CallStackTracer() {
      if( !m_bComplete ) {
         cerr << m_sourceFileName << m_sourceLineName;
      }
   }
   void SetComplete() {
      m_bComplete = true;
   }
private:
   string m_sourceFileName;
   int m_sourceLineNumber;
};

// Some code to be covered by a CallStackTracer
void foo() {
   CallStackTracer cst(__FILENAME__, __LINENUMBER__);

   // all normal code goes here.

   cst.SetComplete();
}
```

Note, that __FILENAME__ and __LINENUMBER__ are preprocessor macros in C++, which will be replaced during compilation by their value.

Finally, you can use the *CallStackTracer* not only for testing purposes, but also to gain additional information in case, your production system runs into problems. In case of exceptions, the additional information may become critical to get the customer productive again.

Consequences

1. (+) Tracing of exceptions without the noise of normal trace statements.

2. (+) Easy to use.

3. (-) An additional class must be implemented, potentially leading to a more complex code.

4. (-) Runtime overhead might be significant, depending on the remaining code.

Implementation Issues

Some languages, e.g. Java, do not guarantee, that the destructor is executed when the instance goes out of scope. So *CallStackTracer* does no work in these cases.

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).

Related Patterns

The *Diagnostic Logger* [Neil Harrison in PLOPD3, p. 277] describes a pattern for reporting diagnostic information in a consistent manner. This is a perfect match for the *CallStackTracer*, as when the *CallStackTracer* writes debug information it can use the *Diagnostic Logger*.

Also: *The Bottom Line* [Bob Hanmer in PLOPD4], and *Scoped Transaction* [POSA2].

Test Case with Time Specification

Motivation

You have opted to use practices from Extreme Programming (XP). "Any program feature without an automated test simply doesn't exist" [Beck2000, p. 57]. Therefore, you implement a test case first. Then you implement the feature.

Also, documenting the XP way means, that the code IS the documentation. Therefore, you do not write any additional documents.

This means, that you specify features using test cases. However, in some cases the functional specification is not sufficient, e.g. a function should be executed within a time limit.

In short: How do you add time specifications to your test cases?

Forces

1. You want to run fully automated test cases.

2. The test cases include the functional specification of software being tested.

3. You want to specify the time a test case may take.

4. You want to specify and verify the number of CPU cycles the execution of a sequence takes during testing.

Solution

Add a time or CPU cycle check to your test case.

A fully automated test case is simply a piece of code executed. At the end of the test case, you typically check, whether the result of the operations, which build up the test case, have lead to the correct result.

In addition to only checking the functional correctness, the current system time (or CPU cycle count) is stored when the test case starts. Once the test case has finished, the current system time (or CPU cycle count) is retrieved again. The difference is then put into an assertion for checking whether the difference is within the specification.

The following is pseudo code for how an implementation of a test case may look like, after the time specification has been added:

```
CTime start;
CTimeSpan duration;
start = CTime::GetSystemTime();
// code of test case goes here
duration = CTime::GetSystemTime() - start;
if( duration > 10.0 ) {
   throw CTestFailed();
}
```

Consequences

1. (+) Each test case not only test the functional correctness, but also checks that the test case is executed within predefined time limits.

2. (+) If the test case also contains time limits, the test case guarantees that an arbitrary code change of the implementation will have a negative side effect on performance.

3. (+) Using time specifications for each test case makes the customer, implementer think about the timely behavior of a program's feature.

4. (-) For each test case additional code might be necessary.

5. (-) This approach may not be feasible for real-time applications. I simply have no experience with real-time applications.

Implementation Issues

Each platform provides only a certain resolution for their times. If a test case is extremely short, only few lines of code need to be executed, the resolution may not be sufficient to get a useful number. In this case, you may want to implement a loop around the test case and execute the same test case several times, e.g. 100 or 1000 times.

Known Uses

Multiple Hewlett-Packard network support applications use this pattern, e.g. AutoCollect, Network Documentation Tool (NDT).[5]

Related Patterns

N/A

References

Beck2000	Kent Beck, "Extreme Programming Explained – Embracing Change", Addison-Wesley, ISBN 0201616416
Bind1999	Robert V. Binder: "Testing Object-Oriented Systems – Models, Patterns, and Tools", Addison-Wesley, ISBN 0201809389. This book contains 37 patterns for testing software. It is therefore an extremely valuable source.
PLOPD3	Robert Martin, Dirk Riehle, Frank Buschmann (Editors): "Pattern Languages of Program Design 3", Addison-Wesley, ISBN 0201310112
PLOPD4	Brian Foote, Neil Harrison, Hans Rohnert: "Pattern Languages of Program Design 4", Addison-Wesley, ISBN 0201433044
POSA2	Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann: "Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects", John Wiley & Sons, 0471606952

Acknowledgments

I would like to thank my two shepherds. Thank your to Bob Hanmer for his gentle, yet extremely helpful suggestions for improving this paper. Thank you also to Andreas Rüping for help in preparing this paper for EuroPLoP 2001. It was a pleasure working with both of them.

[5] At present, all these applications are not available standalone. All of them are part of service or support offerings.