Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

Fail Fast

Jim Shore

The most annoying aspect of software development, for me, is debugging. I don't mind the kinds of bugs that yield to a few minutes' inspection. The bugs I hate are the ones that show up only after hours of successful operation, under unusual circumstances, or whose stack traces lead to dead ends.

Fortunately, there's a simple technique that will dramatically reduce the number of these bugs in your software. It won't reduce the overall number of bugs, at least not at first, but it'll make most defects much easier to find.

The technique is to build your software to "fail fast."

Immediate and visible failure

Some people recommend making your software robust by working around problems automatically. This results in the software "failing slowly." The program continues working right after an error but fails in strange ways later on.

A system that fails fast does exactly the opposite: when a problem occurs, it fails immediately and visibly. Failing fast is a nonintuitive technique: "failing immediately and visibly" sounds like it would make your software more fragile, but it actually makes it more robust. Bugs are easier to find and fix, so fewer go into production.

For example, consider a method that reads a property from a configuration file. What should happen when the property isn't present? A common approach is to return `null` or a default value:

```
public int maxConnections() {
  string property =
    getProperty("maxConnections");
  if (property == null) {
    return 10;
  }
  else {
    return property.toInt();
  }
}
```

In contrast, a program that fails fast will throw an exception:

```
public int maxConnections() {
  string property =
    getProperty("maxConnections");
  if (property == null) {
    throw new NullReferenceException
     ("maxConnections property not
          found in " +
          this.configFilePath);
  }
  else {
    return property.toInt();
  }
}
```

Imagine this method is part of a Web-based system that's undergoing a minor upgrade. In this release, let's say the developer accidentally introduces a typo in the configuration file, triggering the error-handling code. For the code that returns

```
public class Assert {
  public static void true(bool condition, string message) {
    if (!condition) throw new AssertionException(message);
  }

  public static void notNull(object o) {
    if (o == null) throw new NullReferenceException();
  }

  public static void cantReach(string message) {
    throw new UnreachableCodeException(message);
  }

  public static void impossibleException(Throwable e, string message) {
    throw new UnreachableCodeException(e, message);
  }
}
```

Figure 1. An assert class in Java.

```
public static void Main()
{
  WriteCenteredLine(null);
}

public void WriteCenteredLine(string text)
{
  int screenWidth = 80;
  int paddingSize = (screenWidth – text.Length) / 2;
  string padding = new string(‘ ’, paddingSize);
  Console.WriteLine(padding + text);
}
```

Figure 2. A stack trace that leads to a null reference (C#).

a default value, everything will seem fine. But when customers start using the software, they'll encounter mysterious slowdowns. Figuring it out could take days of hair pulling.

The outcome is much different when we write the software to fail fast. The instant the developer introduces the typo, the software stops functioning, saying `maxConnections property not found in c:\projects\SuperSoftware\config.properties`. The developer slaps his or her forehead and spends 30 seconds fixing the problem.

Fail-fast fundamentals

Assertions are the key to failing fast. An assertion is a tiny piece of code that checks a condition and then fails if the condition isn't met. So, when something starts to go wrong, an assertion detects the problem and makes it visible.

Most languages have built-in assertions, but they don't always throw exceptions. They're also usually pretty generic, limiting expressiveness and causing duplication. For these reasons, I usually prefer to implement my own assertion class, as Figure 1 shows.

However, it's tough to know when to add assertions. One way to tell is to look for comments. Comments often document assumptions about how a piece of code works or how it should be called. When you see those comments, or feel like writing one, think about how you can turn it into an assertion instead.

When you're writing a method, avoid writing assertions for problems in the method itself. Tests, particularly test-driven development, are a better way of ensuring the correctness of individual methods. Assertions shine in their ability to flush out problems in the seams of the system. Use them to show mistakes in how the rest of the system interacts with your method.

Writing assertions

A good example of the finesse needed to use assertions well is `Assert.notNull()`. Null reference exceptions are a common symptom of defects in my programs, so I'd like my software to tell me when a null reference is created inappropriately.

On the other hand, if I used `Assert.notNull()` after every single variable assignment, my code would drown in a sea of useless assertions. So I put myself in the shoes of the luckless developer debugging the system. When a null reference exception occurs, how can I make it easy for the developer to find the problem?

Sometimes, the language will automatically tell us where the problem is. For example, Java and C# throw a null reference exception when a method is called on a null reference. In simple cases, the exception's stack trace will lead us to the source of the problem, as in this example:

```
System.NullReferenceException
    at Example.WriteCenteredLine()
        in example.cs:line 9
    at Example.Main() in
        example.cs:line 3
```

Here, tracing backwards through the stack trace leads us to line 3, where we see a null reference being passed into a method (see Figure 2). The answer's not always obvious, but a few minutes of digging will find it.

Now consider a more complicated case, such as

```
System.NullReferenceException
   at Example.Main() in
        example.cs:line 22
   at FancyConsole.Main() in
        example.cs:line 6
```

Here, the stack trace leads to lines 6 and 22, which doesn't help at all (see Figure 3). The real error is at line 17: `getProperty()` returns `null`, causing an exception when `_titleBorder` is dereferenced at line 22. The stack trace leads to a dead end in this case—typical of code designed to fail slowly—requiring tedious debugging.

It's this latter case that I want to prevent. I need the program to give me enough information to find bugs easily. So for my code, I've instituted the following rule of thumb: in most cases, the program will fail fast by default, so I don't do anything special about null references (see Figure 4a). However, when I assign a parameter to an instance variable, the program won't fail fast without my help, so I assert that the parameter is not null (see Figure 4b).

This rule of thumb could be helpful for your programs, too, but the main point here is the thought process I went through. When adding assertions to your code, follow a line of reasoning like the one I used for null reference exceptions. Think about what kinds of defects are possible and how they occur. Place your assertions so that the software fails ear-

```
1  public class Example
2  {
3    public static void Main()
4    {
5      FancyConsole out = new FancyConsole();
6      out.WriteTitle("text");
7    }
8  }
9
10 public class FancyConsole()
11 {
12   private const screenWidth = 80;
13   private string _titleBorder;
14
15   public FancyConsole()
16   {
17     _titleBorder = getProperty("titleBorder");
18   }
19
20   public void WriteTitle(string text)
21   {
22     int borderSize = (screenWidth - text.Length)
            /(_titleBorder.Length * 2);
23     string border = "";
24     for (int i = 0; i < borderSize; i++)
25     {
26       border += _titleBorder;
27     }
28     Console.WriteLine(border + text + border);
29   }
30 }
```

Figure 3. A stack trace that leads to a dead end (C#).

```
public string toString(Object parameter) {
  return parameter.toString();
}
```

(a)

```
public class Foo {
  private Object _instanceVariable;

  public Foo(Object instanceVariable) {
    Assert.notNull(instanceVariable);
    _instanceVariable = instanceVariable;
  }
}
```

(b)

Figure 4. An `Assert.notNull()` rule of thumb: (a) no assertion necessary and (b) assertion needed.

lier—close to the original problem—making the problem easy to find. What kinds of problems are common in your code and how can you use assertions to make them easy to fix?

Eliminate the debugger

In some cases, a stack trace is all you need to find an error's cause. In other cases, you must know the contents of some variables. Although you can find the variable data by reproducing the error in a debugger, some errors are hard to reproduce. Wouldn't it be better if your program told you exactly what went wrong?

When writing an assertion, think about what kind of information you'll need to fix the problem if the assertion fails. Include that information in the assertion message. Don't just repeat the assertion's condition; the stack trace will lead to that. Instead, put the error in context.

For an example, we return to our configuration file reader:

```
public string readProperty
    (PropertyFile file,
    string key) {
  string result =
    file.readProperty(key);
  // assertion goes here
  return result;
}
```

You could write the assertion message in several different ways. One possibility is

```
Assert.notNull(result, "result was
   null");
```

but that merely repeats the assertion condition.

Another possibility is

```
Assert.notNull(result, "can't find
   property");
```

which gives some context but not enough to eliminate the debugger.

A better assertion is

```
Assert.notNull(result, "can't find
   [" + key + "] property in config
   file [" + file + "]");
```

which gives just the right amount of information.

You don't need to go overboard when writing assertion messages. Assertions are for programmers, so they don't need to be user friendly, just informative.

Robust failure

Failing fast seems like it could result in pretty fragile software. Sure, it makes defects easier to find, but what about when you deploy the software to customers? We don't want the application to crash just because there's a typo in a configuration file.

One reaction to this fear is to disable assertions in the field. Don't do that! Remember, an error that occurs at the customer's site made it through your testing process. You'll probably have trouble reproducing it. These errors are the hardest to find, and a well-placed assertion explaining the problem could save you days of effort.

On the other hand, a crash is never appropriate. Fortunately, there's a middle ground. You can create a

Figure 5. Global error handler for a C# batch-processing system.

```
public static void Main() {
   try
   {
      foreach (BatchCommand command in Batch())
      {
         try
         {
            command.Process();
         }
         catch (Exception e)
         {
            ReportError("Exception in " + command, e);
            // continue with next command
         }
      }
   }
   catch (Exception e)
   {
      ReportError("Exception in batch loader", e);
      // unrecoverable; must exit
   }
}

private static void ReportError(string message, Exception e)
{
   LogError(message, e);
   PageSysAdmin(message);
}
```

global exception handler to gracefully handle unexpected exceptions, such as assertions, and bring them to the developers' attention. For example, a GUI-based program might display

```
an unexpected problem has occured
```

in an error dialog and provide an option to email tech support. A batch-processing system might page a system administrator and continue with the next transaction (see Figure 5 for an example).

If you use a global exception handler, avoid catch-all exception handlers in the rest of your application. They'll prevent exceptions from reaching your global handler. Also, when you use resources that have to be closed (such as files), be sure to use `finally` blocks or `using` statements (in C#) to clean them up. This way, if an exception occurs, the application will be returned to a fresh, working state.

Bugs add a lot of expense and risk to our projects—not to mention, they're a pain in the neck to figure out. Since the hardest part of debugging is often reproducing and pinpointing errors, failing fast can reduce debugging's cost, and pain, significantly.

Furthermore, it's a technique you can start using right away. Be sure to implement a global error handler so your overall stability doesn't suffer. Search your existing code for catch-all exception handlers and either remove or refactor them. Then you're ready to gradually introduce assertions. Over time, more and more errors will fail fast, and you'll see the cost of debugging decrease and the quality of your system improve.

Jim Shore is the founder of Titanium I.T., a Portland, Ore., consultancy specializing in helping software teams work more effectively. Contact him at jshore@titanium-it.com.