Fail fast - Jim Shore

Traducción realizada por Santiago Langer con ayuda de Mauro Rizzi para
Ingeniería del software 1, cátedra Leveroni.
Facultad de Ingeniería de la Universidad de Buenos Aires. Abril 2024.

El aspecto más molesto del desarrollo de software es, en mi opinión, el debugging. Pero no me molestan los bugs que se arreglan en unos pocos minutos. Los bugs que odio son aquellos que aparecen después de horas de uso exitoso, bajo circunstancias inusuales o aquellos cuyos stack traces no llevan a ningún lado.
Afortunadamente, existe una técnica sencilla para reducir drásticamente el número de este tipo de bugs en el software. Esta técnica no va a reducir el número de bugs totales (no al principio, al menos) pero sí hará más fácil encontrar la mayoría de los defectos.
La técnica consiste en construir software que *falle rápido* (fail fast).

Fallo inmediato y visible

Algunas personas recomiendan construir software robusto que lidie con los problemas automáticamente, pero esto resulta en programas que fallan *lentamente*. Estos programas siguen corriendo luego del error pero fallan de formas inesperadas e inexplicables tiempo después.
Un sistema que falla rápidamente hace exactamente lo contrario: cuando sucede un error falla inmediatamente y de forma visible. Fallar rápidamente puede sonar contra intuitivo, “fallar inmediatamente y de forma visible” suena frágil, pero en realidad hace que el software sea más robusto. Los errores son más fáciles de encontrar y solucionar, evitando que lleguen a producción.
Por ejemplo, imaginemos un método que lee una propiedad de un archivo de configuración ¿qué debería pasar si la propiedad no está en el archivo? Frecuentemente se envía null o un valor por default:

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

En cambio, un programa que falla rápido lanzará una excepción:

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

Supongamos que este método es parte de un sistema web al que le estamos aplicando una actualización menor y un programador comete un error ortográfico en el archivo de configuración. Para el código que retorna un valor por defecto parecerá que todo está bien. Pero cuando los clientes empiecen a usar el software ellos encontrarán ralentizaciones inesperadas. Encontrar el error, en este caso, tomará días y días de sufrimiento.
El desenlace es completamente distinto si desarrollamos software que falla rápido. En el instante que el desarrollador comete el error de ortografía, el software dejará de funcionar y dirá *maxConnections property not found in c:\projects\SuperSoftware\config.properties.* El desarrollador se llevará las manos a la cara y en menos de 30 segundos tendrá el problema solucionado.

Fundamentos de fail fast

Las aserciones (assertions) son la clave para fallar rápido. Una aserción es una pequeña parte de código que evalúa una condición y falla si esa condición no se cumple. Cuando algo comienza a andar mal, una aserción detecta el problema y lo hace visible.
La mayoría de los lenguajes de programación ya vienen con aserciones incluidas, pero no siempre estas lanzan excepciones. También tienden a ser genéricas, limitando la expresividad y causando duplicación. Por estas razones, usualmente prefiero implementar mi propia clase Aserción, como se ve en la figura 1.

```
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);
  }
}
```

Figura 1. Una clase Assert en Java

Sin embargo, debemos tener en cuenta que es difícil saber cuándo agregar un assert.
Una buena forma es mirar donde están los comentarios en el código. Los comentarios usualmente documentan suposiciones sobre cómo una parte del código funciona o cómo debería ser llamado. Cuando te encuentres con esos comentarios (o sientas que es importante escribir uno) tal vez debas convertirlo en una aserción.
También, cuando escribas un método, evita escribir aserciones para problemas en el método de por sí. Es mucho mejor usar tests (en particular mediante Test Driven Development) para asegurar el correcto funcionamiento del método. En cambio, las aserciones brillan para encontrar problemas en las uniones del sistema. Usalos para poner a la vista errores de cómo el resto del sistema interactúa con ese método.

Cómo escribir aserciones

Un buen ejemplo de la delicadeza necesaria para usar aserciones es *Assert.notNull()*. El uso de Null es un problema frecuente en mis programas, así que me gustaría que mi software me avise cuando una referencia a null se usa inadecuadamente.
Por otro lado, si usase *Assert.notNull()* cada vez que se asigna una variable, mi código se inundaría con aserciones inútiles. Para evitar esto, me pongo en los zapatos del desafortunado desarrollador que debe debuggear el sistema. Cuando salta una excepción por referencia null me pregunto ¿cómo puedo hacer para facilitarle al desarrollador encontrar el problema?
A veces, el lenguaje informará automáticamente dónde está el problema. Java y C#, por ejemplo, lanzan una NullReferenceException cuando se llama un método sobre una referencia null. En casos sencillos, el stack trace nos llevará a la fuente del problema:

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

En este caso, seguir el camino del stack nos llevará a la línea 3, donde un null es enviado a un método (véase figura 2).

```
1   public static void Main() {
2     WriteCenteredLine(null);
3   }
4
5   public void WriteCenteredLine(string text){
6     int screenWidth = 80;
7     int paddingSize = (screenWidth - text.Length) / 2;
8     string padding = new string(' ', paddingSize);
9     Console.WriteLine(padding + text);
10  }
```

Figura 2. Un stack trace que lleva a una referencia a null

La respuesta no siempre es obvia, pero unos pocos minutos de búsqueda nos permitirá encontrar el error.
Veamos un caso más complicado, como el siguiente:

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

```
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 }
```

Figura 3. Un stack trace que no lleva a ningún lado (C#)

En este caso el stack trace lleva a las líneas 6 y 22 (figura 3), cuando en realidad el problema está en la línea 17: getProperty retorna null, causando la excepción cuando se desreferencia _titleBorder en la línea 22. El stack trace nos lleva a un callejón sin salida (típico de código diseñado para fallar lentamente), lo que requerirá debugging largo y tedioso.
Este último ejemplo es lo que busco prevenir. Necesito que el programa me de suficiente información para encontrar los bugs fácilmente.

```
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)
```

Figura 4. Uso de Assert.notNull(). A) Aserción innecesaria, B) Aserción necesaria

Por esto, para mi código tengo la siguiente regla: normalmente el programa fallará rápido por defecto, por lo que no necesito hacer nada especial cuando aparezcan referencias null (figura 4a). Sin embargo, cuando asigno un parámetro a una variable el programa no fallará sin mi ayuda, por lo que necesito introducir un assert en el código para verificar que el parámetro no sea null (figura 4b).

Esta regla particular puede ser muy útil para tus programas, pero la idea de este ejemplo es mostrarte el proceso mental en mi cabeza. Cuando agregues aserciones a tu código, sigue una línea de razonamiento como la que usé yo para excepciones de referencias null. Piensa los tipos de defectos posibles y cómo pueden ocurrir. Pon tus aserciones para que tu programa falle cuanto antes y lo más cerca posible del problema original, así facilitarás el debugging del problema.

¿Qué clase de problemas son comunes en tu código y cómo podés usar aserciones para facilitarte arreglarlos?

Eliminar el debugger

En muchos casos, el stack trace es todo lo que vas a necesitar para encontrar la causa de un error. En otros tal vez tengas que conocer el contenido de algunas variables. Pero incluso aunque tengas todas las variables, algunos errores son difíciles de reproducir ¿no sería mejor que el programa te diga exactamente qué salió mal?

Cuando escribas una aserción, piensa qué clase de información vas a necesitar para solucionar el problema que la aserción detectó. Incluí esa información en el mensaje de error de la aserción. No repitas la condición de la aserción (ya el stack trace te lo señalará). En cambio, pon el error en contexto.

Volvamos a nuestro lector del archivo de configuración para ver otro ejemplo:

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

Podríamos escribir la aserción de varias maneras distintas. Una manera posible podría ser la siguiente:

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

Esta opción simplemente repite la condición de la aserción, no es útil.
Veamos otra opción:

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

Esta opción es superior a la primera, pero no nos alcanza para evitar usar el debugger. La mejor aserción podría ser:

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

Esta última opción nos da la información perfecta. Explica qué falló, qué propiedad buscaba y dónde no la encontró.
Tampoco es necesario abrumarse redactando mensajes de aserción, las aserciones son para programadores, no tienen porqué ser amigables para el usuario. Solo deben ser informativas.

Fallo robusto

Podría parecer que fallar rápido resultará en software frágil. Perfecto, facilita encontrar los errores ¿pero qué pasa cuando presentamos el programa a los clientes? No queremos que la aplicación crashee por pequeños detalles como un error ortográfico en el archivo de configuración.
Una reacción posible frente a este miedo podría ser desactivar los asserts una vez que pasemos a producción ¡No hagas eso! Recuerda, un error que ocurre con el uso del cliente es un error que se filtró a tu proceso de testeo. Probablemente tendrás problemas reproduciendo ese mismo error, así que sigue siendo buena idea tener un assert que nos ahorre todos esos días de trabajo.

```
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);
}
```

Figura 5. Global Error Handler para C#

Por otro lado, un crash nunca es apropiado. Afortunadamente, existe un punto medio. Puedes crear un administrador global de errores (GlobalExceptionHandler) que maneje las excepciones inesperadas (como las aserciones) de forma elegante y que las presente claramente al desarrollador. Por ejemplo, una interfaz gráfica podría mostrar ”*Se ha producido un error inesperado”* al usuario y darle la opción de contactar a servicio técnico. Un sistema de procesamiento por lotes podrá notificar a un administrador del sistema y continuar con el siguiente lote (Ver figura 5)

Si usas un GlobalExceptionHandler, evita usar manejadores de errores en el resto de tu aplicación. Estos prevendrán que las excepciones lleguen al administrador global de errores. También, cuando uses recursos que deben ser cerrados (como archivos) asegurate de usar bloques *finally* para limpiarlos y cerrarlos (debes cerrar los archivos luego de usarlos). De esta manera, si aparece un error, la aplicación volverá a su estado original y funcional.

Conclusión

Los bugs agregan costo y riesgo a nuestros proyectos (y además son un dolor de cabeza de solucionar). Entendiendo que la peor parte de debuggear es reproducir y reconocer el error, fallar rápido reduce el costo y la molestia de debuggear.
Además, es una técnica que puedes aplicar hoy mismo. Te invito a implementar un administrador global de errores, luego busca todos los manejadores de excepciones y eliminalos o refactorizalos. Cuando estés listo, gradualmente introduce aserciones. Con el paso del tiempo más y más errores fallarán rápidamente y verás que el costo de debuggear baja, y la calidad de tu sistema sube.