Defensive Programming

Chapter 8 Code Complete

La idea se basa en una conducción defensiva. Al conducir a la defensiva, adoptas la mentalidad de que nunca estás seguro de lo que van a hacer los demás conductores. De esa manera, te aseguras de que si hacen algo peligroso no te lastimarán. Usted asume la responsabilidad de protegerse incluso cuando puede ser culpa del otro conductor.

Este capítulo describe cómo protegerse del frío y cruel mundo de los datos no válidos, los eventos que "nunca" pueden ocurrir y los errores de otros programadores.

# 8.1. Protecting Your Program from Invalid Inputs

Hay tres formas generales de manejar la basura en:

1. Verifique los valores de todos los datos de fuentes externas.

Cuando obtenga datos de un archivo, un usuario, la red o alguna otra interfaz externa, verifique que los datos estén dentro del rango permitido.

2. Verifique los valores de todos los parámetros de entrada de rutina.

Verificar los valores de los parámetros de entrada de la rutina es esencialmente lo mismo que verificar los datos que provienen de una fuente externa, excepto que los datos provienen de otra rutina en lugar de una interfaz externa.

La discusión en Barricada su programa para contener el daño causado por errores, proporciona una manera práctica de determinar qué rutinas necesitan verificar sus entradas.

3. Decida cómo manejar las malas entradas.

Una vez que haya detectado un parámetro no válido, ¿qué puede hacer con él? Dependiendo de la situación, puede elegir cualquiera de una docena de enfoques diferentes, que se describen en detalle en Técnicas de manejo de errores, más adelante en este capítulo.

La mejor forma de codificación defensiva es no insertar errores en primer lugar. El uso del diseño iterativo, la escritura de pseudocódigo antes del código, la escritura de casos de prueba antes de escribir el código y la realización de inspecciones de diseño de bajo nivel son actividades que ayudan a evitar la inserción de defectos. Por lo tanto, se les debe dar mayor prioridad que la programación defensiva.

# 8.2. Assertions

● Una aserción es un código que se usa durante el desarrollo, generalmente una rutina o macro, que permite que un programa se revise a sí mismo mientras se ejecuta.

● Cuando una afirmación es verdadera, significa que todo está funcionando como se esperaba. Cuando es falso, significa que ha detectado un error inesperado en el código.

● Las afirmaciones son especialmente útiles en programas grandes y complicados y en programas de alta confiabilidad.

● Una aserción generalmente toma dos argumentos: una expresión booleana que describe la suposición que se supone que es verdadera y un mensaje para mostrar si no lo es.

● Normalmente, no desea que los usuarios vean mensajes de afirmación en el código de producción; Las afirmaciones se utilizan principalmente durante el desarrollo y el mantenimiento.

○ Durante la producción, se pueden compilar fuera del código para que las afirmaciones no degraden el rendimiento del sistema.

## Guidelines for Using Assertions

**Use código de manejo de errores para las condiciones que espera que ocurran; use aserciones para condiciones que nunca deberían ocurrir.**

● Las afirmaciones comprueban condiciones que nunca deberían ocurrir.

● El manejo de errores generalmente busca datos de entrada incorrectos; Las afirmaciones comprueban si hay errores en el código.

● Si se activa una afirmación por una condición anómala, la acción correctiva no es simplemente manejar un error con elegancia; la acción correctiva es cambiar el código fuente del programa, recompilar y lanzar una nueva versión del software.

● Una buena forma de pensar en las afirmaciones es como documentación ejecutable; no puede confiar en ellas para que el código funcione, pero pueden documentar las suposiciones de manera más activa que los comentarios en el lenguaje del programa.

### Avoid putting executable code into assertions.

● Poner código en una aserción aumenta la posibilidad de que el compilador elimine el código cuando desactive las aserciones.

○ Coloque declaraciones ejecutables en sus propias líneas, asigne los resultados a las variables de estado y pruebe las variables de estado en su lugar.

### Use assertions to document and verify preconditions and postconditions.

Las condiciones previas y posteriores son parte de un enfoque para el diseño y desarrollo de programas conocido como "diseño por contrato" (Meyer 1997). Cuando se utilizan precondiciones y poscondiciones, cada rutina o clase forma un contrato con el resto del programa.

● Las condiciones previas son las propiedades que el código de cliente de una rutina o clase promete que será verdadera antes de llamar a la rutina o instanciar el objeto.

● Las poscondiciones son las propiedades que la rutina o clase promete serán verdaderas cuando concluya su ejecución.

Los comentarios se pueden usar para documentar las condiciones previas y posteriores, pero, a diferencia de los comentarios, las afirmaciones pueden verificar dinámicamente si las condiciones previas y posteriores son verdaderas.

Example:

● Si las variables latitud, longitud y elevación provienen de una fuente externa, los valores no válidos deben verificarse y manejarse mediante código de manejo de errores en lugar de afirmaciones.

● Sin embargo, si las variables provienen de una fuente interna confiable y el diseño de la rutina se basa en la suposición de que estos valores estarán dentro de sus rangos válidos, entonces las afirmaciones son apropiadas.

### For highly robust code, assert, and then handle the error anyway

Para cualquier condición de error dada, una rutina generalmente usará un código de aserción o de manejo de errores, pero no ambos. Algunos expertos sostienen que solo se necesita un tipo (Meyer 1997).

Pero los programas y proyectos del mundo real tienden a ser demasiado complicados para depender únicamente de afirmaciones.

En un sistema grande y duradero, diferentes diseñadores pueden diseñar diferentes partes durante un período de 5 a 10 años o más. Los diseñadores estarán separados en el tiempo, en numerosas versiones. Sus diseños se centrarán en diferentes tecnologías en diferentes puntos del desarrollo del sistema. Los diseñadores estarán separados geográficamente, especialmente si partes del sistema se adquieren de fuentes externas. Los programadores habrán trabajado con diferentes estándares de codificación:

# 8.3. Error-Handling Techniques

Las afirmaciones se utilizan para manejar errores que nunca deberían ocurrir en el código. ¿Cómo maneja los errores que espera que ocurran? Según las circunstancias específicas, es posible que:

● desea devolver un valor neutral,

● sustituir el siguiente dato válido,

● devolver la misma respuesta que la vez anterior,

● sustituir el valor legal más cercano,

● registrar un mensaje de advertencia en un archivo,

● devolver un código de error,

● llamar a una rutina u objeto de procesamiento de errores,

● mostrar un mensaje de error,

● o apagar

● o quizás desee utilizar una combinación de estas respuestas.

### Return a neutral value.

A veces, la mejor respuesta a los datos incorrectos es continuar operando y simplemente devolver un valor que se sabe que es inofensivo.

● Un cálculo numérico puede devolver 0. Una operación de cadena puede devolver una cadena vacía o una operación de puntero puede devolver un puntero vacío.

● Una rutina de dibujo que obtiene un valor de entrada incorrecto para el color en un videojuego puede usar el color de fondo o primer plano predeterminado.

### Substitute the next piece of valid data.

### Al procesar un flujo de datos, algunas circunstancias requieren simplemente devolver los siguientes datos válidos.

### ● Si está leyendo registros de una base de datos y encuentra un registro dañado, puede simplemente continuar leyendo hasta que encuentre un registro válido.

### ● Si está tomando lecturas de un termómetro 100 veces por segundo y no obtiene una lectura válida una vez, simplemente puede esperar otra centésima de segundo y tomar la siguiente lectura.

### Return the same answer as the previous time.

### Ejemplo: si detecta una solicitud para pintar parte de la pantalla con un color no válido, simplemente puede devolver el mismo color utilizado anteriormente.

### Substitute the closest legal value.

En algunos casos, puede optar por devolver el valor legal más cercano,

● Dado que mi velocímetro no muestra velocidades negativas, cuando retrocedo, simplemente muestra una velocidad de 0, el valor legal más cercano.

### Log a warning message to a file.

### Cuando se detectan datos incorrectos, puede optar por registrar un mensaje de advertencia en un archivo y luego continuar.

### ● Este enfoque se puede utilizar junto con otras técnicas, como sustituir el valor legal más cercano o sustituir el siguiente dato válido.

### Return an error code.

### ● Puede decidir que solo determinadas partes de un sistema manejarán los errores.

### ● Otras partes no manejarán errores localmente; simplemente informarán que se ha detectado un error y confiarán en que alguna otra rutina más arriba en la jerarquía de llamadas manejará el error.

### Establecer el valor de una variable de estado

### Devuelve el estado como el valor de retorno de la función. Lanza una excepción mediante el mecanismo de excepción integrado del lenguaje.

### Call an error-processing routine/object.

### Otro enfoque consiste en centralizar el manejo de errores en una rutina de manejo de errores global o en un objeto de manejo de errores.

### ● La ventaja de este enfoque es que la responsabilidad del procesamiento de errores se puede centralizar, lo que puede facilitar la depuración.

### ● La compensación es que todo el programa conocerá esta capacidad central y estará acoplada a ella.

### ● Si alguna vez desea reutilizar el código del sistema en otro sistema, tendrá que arrastrar la maquinaria de manejo de errores junto con el código que reutiliza.

### Display an error message wherever the error is encountered.

### Este enfoque minimiza la sobrecarga de manejo de errores; sin embargo, tiene el potencial de difundir los mensajes de la interfaz de usuario a través de toda la aplicación, lo que puede crear desafíos cuando se necesita crear una interfaz de usuario coherente.

### Los atacantes a veces usan mensajes de error para descubrir cómo atacar un sistema.

### Shut down.

Algunos sistemas se apagan cuando detectan un error. Este enfoque es útil en aplicaciones críticas para la seguridad.

● si el software que controla el equipo de radiación para el tratamiento de pacientes con cáncer recibe datos de entrada incorrectos para la dosis de radiación, ¿cuál es su mejor respuesta de manejo de errores? ¿Debería usar el mismo valor que la última vez? ¿Debería utilizar el valor legal más cercano? ¿Debería usar un valor neutral? En este caso, apagar es la mejor opción.

## Robustness vs. Correctness

Como nos muestran los ejemplos de videojuegos y rayos X, el estilo de procesamiento de errores más apropiado depende del tipo de software en el que se produce el error.

● Corrección significa nunca devolver un resultado inexacto; no devolver ningún resultado es mejor que devolver un resultado inexacto.

● Robustez significa intentar siempre hacer algo que permita que el software siga funcionando, incluso si eso conduce a resultados que a veces son inexactos.

● Las aplicaciones críticas para la seguridad tienden a favorecer la corrección sobre la robustez.

● Las aplicaciones de consumo tienden a favorecer la robustez a la corrección. Cualquier resultado es generalmente mejor que el cierre del software.

# 8.4. Exceptions

Las excepciones son un medio específico por el cual el código puede pasar errores o eventos excepcionales al código que lo llamó.

Si el código de una rutina encuentra una condición inesperada que no sabe cómo manejar, lanza una excepción, esencialmente levantando las manos y gritando: "No sé qué hacer con esto, espero que alguien más lo sepa. cómo manejarlo! "

Las excepciones tienen un atributo en común con la herencia: si se usan con criterio, pueden reducir la complejidad. Usados ​​de manera imprudente, pueden hacer que el código sea casi imposible de seguir:

Los programas que utilizan excepciones como parte de su procesamiento normal sufren todos los problemas de legibilidad y mantenibilidad del código spaghetti clásico. - Andy Hunt Dave Thomas

● El principal beneficio de las excepciones es su capacidad para señalar las condiciones de error de tal manera que no se pueden ignorar (Meyers 1996).

● Lanza una excepción solo para condiciones que son verdaderamente excepcionales.

Las excepciones deben reservarse para condiciones que son verdaderamente excepcionales, en otras palabras, para condiciones que no pueden ser abordadas por otras prácticas de codificación: para eventos que nunca deberían ocurrir.

● Las excepciones representan una compensación entre una forma poderosa de manejar condiciones inesperadas por un lado y una mayor complejidad por el otro. Las excepciones debilitan la encapsulación al requerir que el código que llama a una rutina sepa qué excepciones pueden aparecer dentro del código que se llama.

○ Si una condición de error se puede manejar localmente, hágalo localmente. No lance una excepción no detectada en una sección de código si puede manejar el error localmente.

○ Evite lanzar excepciones en constructores y destructores a menos que los atrape en el mismo lugar.

○ Las excepciones lanzadas son parte de la interfaz de rutina, al igual que los tipos de datos específicos.

Example 8-7.

class Employee { ... public TaxId GetTaxId() throws EOFException { <-- 1 ... } ... } (1)Here is the declaration of the exception that's at an

Example 8-7. class Employee { ... public TaxId GetTaxId() throws EOFException { <-- 1 ... } ... } (1)Here is the declaration of the exception that's at an inconsistent level of abstraction.

Example 8-8. Good Java Example of a Class that Throws an Exception at a Consistent Level of Abstraction class Employee { ... public TaxId GetTaxId() throws EmployeeDataNotAvailable { <-- 1 ... } ... }

● Incluya en el mensaje de excepción toda la información que llevó a la excepción.

Cada excepción ocurre en circunstancias específicas que se detectan en el momento en que el código lanza la excepción.

● Evite los bloques de captura vacíos

A veces es tentador pasar una excepción con la que no sabes qué hacer,

● Conozca las excepciones que arroja el código de su biblioteca.

Si está trabajando en un lenguaje que no requiere una rutina o clase para definir las excepciones que genera, asegúrese de saber qué excepciones genera cualquier código de biblioteca que utilice.

Si no detecta una excepción generada por el código de la biblioteca, su programa se bloqueará tan rápido como si no detecta una excepción que generó usted mismo. Si el código de la biblioteca no documenta las excepciones que genera, cree un código de prototipos para ejercitar las bibliotecas y eliminar las excepciones.

● Considere la posibilidad de crear un informador de excepciones centralizado.

Un enfoque para garantizar la coherencia en el manejo de excepciones es utilizar un informador de excepciones centralizado.

● Manejar errores con excepciones solo porque su lenguaje proporciona manejo de excepciones es un ejemplo clásico de programación en un lenguaje en lugar de programar en un lenguaje.

● Por último, considere si su programa realmente necesita manejar excepciones, punto. Como señala Bjarne Stroustrup, a veces la mejor respuesta a un error grave en tiempo de ejecución es liberar todos los recursos adquiridos y abortar. Permita que el usuario vuelva a ejecutar el programa con la entrada adecuada (Stroustrup 1997).

# 8.5. Barricade Your Program to Contain the Damage Caused by Errors

Las barricadas son una estrategia de contención de daños.

La razón es similar a la de tener compartimentos aislados en el casco de un barco. Si el barco choca contra un iceberg y abre el casco, ese compartimento se cierra y el resto del barco no se ve afectado.

● Figura 8-2. Definir algunas partes del software que funcionan con datos sucios y algunas que funcionan con datos limpios puede ser una forma eficaz de aliviar la mayor parte del código de la responsabilidad de comprobar si hay datos incorrectos.

● Este mismo enfoque se puede utilizar a nivel de clase.

○ Los métodos públicos de la clase asumen que los datos no son seguros y son responsables de verificar los datos y desinfectarlos.

○ Una vez que los métodos públicos de la clase han aceptado los datos, los métodos privados de la clase pueden asumir que los datos están seguros.

## Relationship Between Barricades and Assertions

El uso de barricadas hace que la distinción entre afirmaciones y manejo de errores sea clara.

Las rutinas que están fuera de la barricada deben utilizar el manejo de errores porque no es seguro hacer suposiciones sobre los datos.

● Las rutinas dentro de la barricada deben usar aserciones, porque se supone que los datos que se les pasan deben ser depurados antes de pasar a través de la barricada.

○ Si una de las rutinas dentro de la barricada detecta datos incorrectos, eso es un error en el programa en lugar de un error en los datos.

# 8.6. Debugging Aids

Otro aspecto clave de la programación defensiva es el uso de ayudas de depuración, que pueden ser un poderoso aliado para detectar errores rápidamente.

## Don't Automatically Apply Production Constraints to the Development Version

## Un punto ciego común del programador es la suposición de que las limitaciones del software de producción se aplican a la versión de desarrollo.

## ● La versión de producción debe ejecutarse rápidamente. Es posible que la versión de desarrollo funcione lentamente.

## ● La versión de producción tiene que ser tacaña con los recursos. A la versión de desarrollo se le podría permitir usar recursos de manera extravagante.

## Esté dispuesto a intercambiar la velocidad y el uso de recursos durante el desarrollo a cambio de herramientas integradas que pueden hacer que el desarrollo sea más fluido.

## Introduce Debugging Aids Early

Cuanto antes introduzca las ayudas de depuración, más serán de ayuda.

Por lo general, no hará el esfuerzo de escribir una ayuda de depuración hasta después de haber sido mordido por un problema varias veces…

## Use Offensive Programming

Los casos excepcionales deben manejarse de una manera que los haga obvios durante el desarrollo y recuperables cuando se ejecuta el código de producción. Michael Howard y David LeBlanc se refieren a este enfoque como "programación ofensiva"

Aquí hay algunas formas en las que puede programar de manera ofensiva:

Un programa muerto normalmente hace mucho menos daño que uno paralizado. - Andy Hunt Dave Thoma

Asegúrate de abortar el programa.

No permita que los programadores adquieran el hábito de simplemente presionar la tecla Intro para evitar un problema conocido. Haga que el problema sea lo suficientemente doloroso como para que se solucione.

Planear eliminar las ayudas de depuración

Si está escribiendo código para su propio uso, podría estar bien dejar todo el código de depuración en el programa. Si está escribiendo código para uso comercial, la penalización del rendimiento en tamaño y velocidad puede ser prohibitiva. Planifique evitar mezclar código de depuración dentro y fuera de un programa. Aquí hay varias formas de hacerlo:

● Utilice herramientas de control de versiones y cree herramientas como ant y make. Las herramientas de control de versiones pueden crear diferentes versiones de un programa a partir de los mismos archivos fuente.

Determinar cuánta programación defensiva dejar en el código de producción

Una de las paradojas de la programación defensiva es que durante el desarrollo, le gustaría que se notara un error; preferiría que fuera desagradable antes que correr el riesgo de pasarlo por alto. Pero durante la producción, prefiere que el error sea lo más discreto posible, para que el programa se recupere o falle correctamente.