Diseño de manejos de errores

Juan Jacobs, Franco Bulgarelli, Gastón Prieto, Juan Ignacio Villarejo
versión 1.0-borrador

1 ¿Qué es un error?

Podemos decir que un error se produce cuando hay alguna condición (por ejemplo, una precondicion) que no se cumplió al realizar una operación. Por ejemplo, el compilador esperaba un paréntesis que no está, el programador escribió una condición lógica errónea porque confundió los signos o un método esperaba que una variable no fuera nula pero de hecho lo es.

2 Clasificaciones

Podemos clasificar a los errores de varias formas, clasificaciones que más adelante nos ayudarán a construir un manejo de errores consistente[1].

[1] Ninguna clasificación escapa a parcialidades, ni es útil por sí misma

2.1 Según el momento y forma en que se manifiestan

- **De tiempo de compilación**[2]: Son los que no permiten que el sistema se construya y en consecuencia, evitan que se ejecute.

  Estos errores son de los que menos nos preocupan. ¿Por qué? Porque, por un lado, la mayoría de los IDEs que usemos para escribir nuestro código nos informarán de la existencia de los mismos y hasta tal vez nos sugiera cómo arreglarlos. Y aún si no tuviéramos IDEs, estos surgirán rápidamente, al compilar o cargar los programas.

  Cabe marcar que estos errores varían según la tecnología que estamos utilizando: los lenguajes con tipado dinámico por ejemplo realizan pocas verificaciones de tipos en tiempo de compilación, difiriendo la falla tiempo de ejecución.

  Por otro lado, como tienen la característica fundamental de que no dejan que nuestro sistema actúe de ninguna forma, no tienen ninguna posibilidad de *romper* otras cosas. No pueden hacer fallar otros componentes, no pueden corromper datos ni tener ningún otro comportamiento perjudicial sobre otro componente de nuestro ambiente.

- **De tiempo de ejecución (falla):** Son los que no permiten que el sistema continúe ejecutándose. Ante la aparición de un error de este tipo, el programa aborta su ejecución normal.

  Ejemplos de este tipo de error son por ejemplo mandarle un mensaje a null/nil, intentar dividir por 0, realizar un casteo inválido o intentar utilizar una conexión que está cerrada.

  Un error de este tipo puede ser fatal ya que el programa directamente deja de funcionar. Sin embargo, si la falla es temprana, podemos evitar por ejemplo la corrupción de datos o introducir fallas de seguridad en otros componentes.

- **De lógica:** Son los que permiten que el sistema se construya y se ejecute, pero con un comportamiento que no es el esperado.
  Ejemplos de este tipo de error serían una condición de un if con los signos invertidos, confundir una multiplicación con una división o retornar una variable que no es la que quiero retornar.

  Estos errores son los más engañosos y peligrosos, porque hacen que nuestro sistema "falle" silenciosamente. Detectar y corregir estos errores suele ser difícil.

  Como estos errores provocan que el sistema se ejecute realizando operaciones inválidas a nivel negocio, tienen una alta probabilidad de *romper* otros componentes de nuestro ambiente. Pueden por ejemplo corromper datos o permitir realizar operaciones a usuarios que no tienen autorización para realizarlas o bloquear

[2] Acá usamos compilación en sentido amplio, como sinónimo de todo procesamiento que ocurre antes de la ejecución del programa

funcionalidades que deberían ser accesibles.

2.2 Según su origen

- **Error de programa/sistema:** Son los errores genéricos que son propios de la tecnología que estemos utilizando para construir nuestro sistema. Son errores básicos y genéricos que pueden ocurrir en cualquier dominio, como por ejemplo mandarle un mensaje a null/nil, dividir por 0, obtener el primer elemento de una lista vacía, etc. La mayoría de los lenguajes ya los tienen contemplados de alguna manera.

- **Error de aplicación/dominio:** Son los errores específicos de nuestro dominio, que se derivan de nuestras reglas de negocio.

2.3 Según su predictibilidad

- **Predecibles**: Decimos que un error es predecible para un cierto componente cuando fue responsabilidad de este componente que la precondición no se cumpliera.

  Un ejemplo de este error sería intentar acceder al primer elemento de un array vacío. Mi método podría haber validado que el array no esté vacío antes de intentar acceder al primer elemento. Es decir, el error era predecible y se podría haber evitado, y por tanto lo asociaremos con una mala programación.

- **No predecibles**: Decimos que un error es no predecible para un cierto componente cuando este no es responsable de haberlo generado y no tiene suficiente control o conocimiento sobre este error como para evitarlo.

  Un ejemplo de este error sería intentar abrir un archivo que no está en la ruta especificada: desde mi proceso, (normalmente) nada podría haber hecho para evitar que otro proceso borre el archivo que quería usar. Por tanto, no tenía forma de anticiparlo desde mi código.

2.4 Según su lugar de ocurrencia

- **Internos al sistema**: ocurren dentro de mi sistema por errores que se originaron en su propio código.

- **Externos al sistema**: ocurren porque un sistema externo se comunicó de una manera inválida con mi sistema, ya sea porque pasó información incorrecta a través de mi interfaz entrante, o porque devolvió resultados incorrectos a través de la interfaz saliente.

3 Estrategias, Patrones, Mecanismos

Ahora que sabemos qué es un error, nos preguntamos: ¿cómo lidiar con éste?

El manejo de errores es un problema que es dependiente de la tecnología, porque cada una tiene sus propias formas de resolverlo[3]. Sin embargo, también existen algunos principios generales sobre el manejo de errores, que nos servirán en cualquiera de ellas.

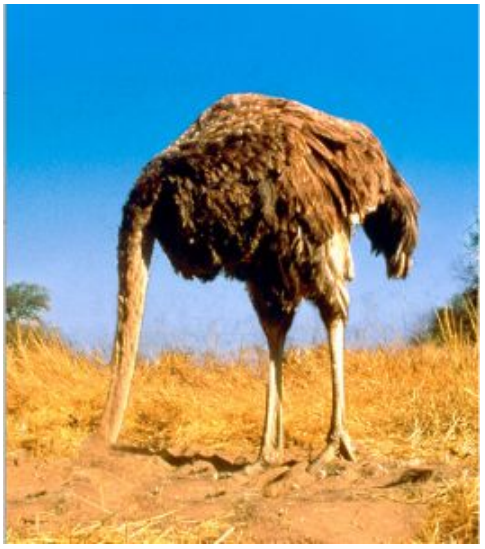
Por eso primero atacaremos el manejo de errores en general, y luego las bajaramos a detalles en algunas tecnologías concretas

4 Estrategias de manejo de errores

Supongamos que estamos ante un error, ya sea lógico o directamente una falla. ¿Qué opciones tenemos? Mencionaremos las siguientes

- Ocultarlo: hacer de de cuenta de que el error no está allí
- Manejarlo:
    - tratarlo: tomar una acción concreta
    - o propagarlo en forma de falla: permitir que falle, o incluso, hacerlo fallar.

4.1 Ocultar el error



Muchas veces cuando nos encontramos frente a un error, queremos que no se note. Intentamos ocultarlo, silenciarlo o esconderlo de alguna manera, para no tener que verlo o que los demás no lo vean.

Esta es una terrible idea[4]. Nadie quiere cometer o presenciar errores, pero si los detectamos, no debemos ignorarlos (“acá no pasó nada”), porque continuar con la ejecución del programa en un estado de error podría significar daños aún mayores, como la corrupción o pérdida de datos. Parece irónico, pero intentar ocultar el error lo único que hace es propagar errores[5] a otros componentes.

Es decir, el error ocurrió, aceptémoslo. Ahora veamos qué podemos hacer.

[3] El desconocimiento de las formas particulares de cada tecnología de manejar los errores lleva muchas veces a discusiones eternas y a la construcción de sistemas de manejo de error inconsistentes y de menor robustez.

[4] Este consejo es válido también para la vida real.

[5] Error es distinto de falla: el error es malo, la falla no, como veremos en breve.

4.2 Tratar el error

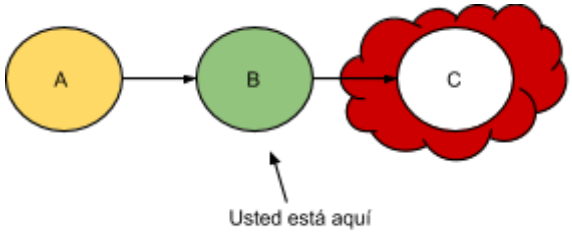
Otra opción es intentar recuperarse del error, esto es, hacer algo tendiente a reparar el mismo. Por ejemplo, si intentamos escribir en un archivo que asumimos que existía, entonces podríamos crearlo y luego proceder normalmente. Si intentamos transferir un archivo a través de la red, y perdemos la conexión, podríamos intentar reconectarnos y reiniciar la transferencia.

Esta estrategia tiene dos problemas:

1. En general el componente que que percibe la falla no tiene suficiente información como para saber cuál es la acción concreta a realizar
2. Seguir adelante intentando recuperarse del error muchas veces puede llevarnos a tener estados inconsistentes si no tenemos cuidado.

Por eso es que en general la estrategia más segura y simple para lidiar con el error no es hacer malabares para seguir sino propagar una falla al componente que nos llamó. Y **sólo tratarlo cuando realmente tenemos suficiente contexto como para determinar cómo reponernos del mismo**[6].

4.3 Propagar el error en forma de falla



Si un componente B detecta un error en un componente C, y (como normalmente ocurrirá) no es nuestra responsabildiad tratarlo, entonces lo que debemos hacer es propagar el error, en forma de falla: notificar el error al componente A que llamó a B.

Esto significa dos cosas: si estamos ante un error de lógica, debemos **convertirlo en falla tan pronto como sea posible**, para evitar continuar trabajando sobre un estado erróneo. Y si el error se manifestó ya como una falla, **debemos simplemente abrirle paso a esta falla** y permitir que escale[7].

[6] Manejar errores también es parte de la responsabilidad de un componente. Y de la misma forma que no queremos que un componente haga más de lo que debe hacer, buscaremos que un componente sólo trate los errores que sepa tratar.

[7] La forma de generar la falla y hacerla escalar es propia de cada mecanismo de manejo de errores.

5 Patrones de manejo de errores

5.1 Fallar rápido (fail fast)

El concepto de fail fast es muy sencillo. Nos plantea que ante la detección de un error de lógica (si el error ya es una falla este principio no nos ayuda), hagamos fallar al sistema **lo antes posible**.

Esto es: si hay dos momentos en los que podríamos validar una precondición inválida, deberíamos hacerlo en el primero de los momentos.

¿Para qué? Para evitar que el error se propague en el sistema a través de nuestra interacción con otros objetos.

Recuerden: si permitimos que el error se propague, y seguimos actuando como si nada hubiera pasado, es posible que sigamos generando inconsistencias tanto en nuestro componente como también en otros. Propaguemos entonces la falla, no el error.

Ejemplo: si tenemos un objeto, y una condición de error puede ser validada tanto al enviarle un mensaje como en su constructor, entonces lo debemos hacer en el constructor, porque ocurre antes.

5.2 Confiar en el adentro, desconfiar del afuera

Si aplicamos el patrón de fallar rápido, ¿cómo podemos hacer para no volvernos paranoicos y llenar nuestro código de validaciones? Anteriormente hablamos de errores que se producen dentro del sistema y errores que se producen a través de las interfaces entrantes y salientes: lo que se suele hacer es validar entonces toda la información que llega al sistema (particularmente si proviene de usuarios), pero confiar en la información que la aplicación maneja internamente.

5.3 Mantener consistencia

Aunque el sistema falle, debemos procurar que el estado en que quede sea consistente, de forma que su ejecución pueda ser continuada de forma segura.

Esto significa que si estamos ante un error, y vamos a fallar o propagar una falla, debemos hacerlo antes de que el estado de componente quede inconsistente, o intentar llevarlo nuevamente a la consistente si es posible.

6 Mecanismos de manejo de errores

Para manejar errores tenemos los mismos mecanismos que para comunicar dos componentes: call and return, memoria compartida, etc. Sin embargo, algunos de ellos calzan mejor que otros para esta tarea.

6.1 Excepciones

Las excepciones son el mecanismo más popular, robusto[8] y simple de lidiar en objetos con los errores:

- Es popular porque virtualmente todos los lenguajes de objetos las soportan, incluso con sintaxis muy similares
- Es simple porque con unas pocas primitivas de manejo de excepciones podemos implementar fácilmente los patrones antes descriptos
- Es robusto justamente porque
    - la simplicidad lleva a que sea más difícil cometer errores en el manejo de errores, que los oculten
    - genera información (stacktraces) que ayudan en el rastreo del error
    - las excepciones se propagan por defecto, lo cual no ocurre en todos los mecanismos

Ejemplo:

```
clase Comprador
  metodo comprar()
     variable producto = elegirProducto()
     variable idProducto = pagador.realizarPago(producto)
     guardarProducto(idProducto, producto)

clase Pagador
  metodo realizarPago(producto)
     si seCumplenCondicionesDePago(producto)
         variable identificadorProducto = hacerPagoPosta(producto)
         retornar identificadorProducto

     lanzar nuevo Error("no se cumplen las condiciones")
```

Es por eso es que las excepciones son nuestro mecanismo principal para manejar errores en objetos.

Lean por ello nuestro apunte principal: manejo de errores con excepciones

6.2 Continuaciones

El uso de continuaciones (~= callbacks) para manejar errores es típico de tecnologías donde las mismas también son usadas para el flujo normal (por ejemplo, porque favorecen el asincronismo) y por tanto utilizar excepciones no es posible.

Ejemplo de implementación naïf:

[8] Recordar la definición de robustez del apunte de cualidades de diseño

```
clase Comprador
  metodo comprar(anteExito, anteFracaso)
     variable producto = elegirProducto()
     pagador.realizarPago(producto, (idProducto ->
         guardarProducto(idProducto, producto)
         anteExito()
       ), anteFracaso)

clase Pagador
  metodo realizarPago(producto, anteExito, anteFracaso)
     si seCumplenCondicionesDePago(producto)
         variable identificadorProducto = hacerPagoPosta(producto)
         anteExito(identificadorProducto)
     si no
         anteFracaso(nuevo Error("no se cumplen las condiciones"))
```

En lugar de pasar las continuaciones de error y éxito, podría crear un objeto que represente la continuación:

```
clase Comprador
  metodo comprar(continuacion)
     variable producto = elegirProducto()
     pagador.realizarPago(producto,
      continuacion.cuandoTengaExito(idProducto ->
         guardarProducto(idProducto, producto)
         anteExito()
       ))

clase Pagador
  metodo realizarPago(producto, continuacion)
     si seCumplenCondicionesDePago(producto)
         variable identificadorProducto = hacerPagoPosta(producto)
         continuacion.anteExito(identificadorProducto)
     si no
         continuacion.anteFracaso(nuevo Error("no se cumplen las
condiciones"))
```

6.3 Call and return

6.3.1 Códigos de error

Es el mecanismo más endeble. Es fácil confundir un valor erróneo con un valor bueno. No es posible propagarlos en forma de una falla que aborte el flujo de ejecución. Ver el apunte de excepciones.
<TODO>

6.3.2 Cosificar el error

Es difícil confundir un valor erróneo con uno bueno. En una implementación naïf estos tampoco se pueden propagar en forma de falla pero si se aprovecha el sistema de tipos y orden superior es posible. Ejemplo, Java con Optional.

Implementación naïf:

```
clase Comprador
  metodo comprar()
     variable producto = elegirProducto()
     cuando pagador.realizarPago(producto)
        caso Exito(identificador) ->
            guardarIdentificador(identificador, producto)
            retornar nuevo Exito()
        caso fracaso ->
            retornar fracaso

clase Pagador
  metodo realizarPago(producto)
     si seCumplenCondicionesDePago(producto)
         variable identificador = hacerPagoPosta(producto)
         retornar nuevo Exito(identificador)
     si no
         retornar nuevo Fracaso(nuevo Error("no se cumplen las
condiciones"))
```

No parece mucho mejor que el manejo de códigos de error, porque la propagación de la falla sigue siendo responsabilidad del componente pero:

- No puedo acceder directamente al resultado, lo que me obliga a ser consciente de la posibilidad de error y al menos impide que lo ignore
- Los errores son objetos ahora, puedo enviarles mensajes, lo cual me da nuevas posibilidades

Implementación mejorada:

```
clase Comprador
  metodo comprar()
     variable producto = elegirProducto()
     retornar pagador
          .realizarPago(producto)
          .siTieneExito(identificador ->
             guardarIdentificador(identificador, producto))
```

Con lo que la propagación pasa a ser implícita.

7 Anti patrones de manejo de excepciones

7.1 Mezclar mecanismos de manejo de errores

Como regla general deberíamos evitar mezclar distintos mecanismos de manejo de error, o al menos, hacerlo con particular cuidado. Damos algunos ejemplos:

7.1.1 Retornar true en caso exitoso y lanzar excepciones en caso de error

El problema está en que nunca devolvería false, entonces ¿para qué retornar un valor con una única posibilidad? Directamente el método no retorna valor y lanza excepción en caso de error

```
metodo inscribir(jugador)
  si puedoInscribirNuevo() entonces
    jugadores.agregar(jugador)
    retornar true
  si no
    lanzar EspacioInsuficienteException()
```

7.1.2 Mezclar excepciones con continuaciones

<TODO>
Se pierde el potencial asincronismo y se confunden los mecanismos de propagación

7.2 Ocasionar errores adrede

Si el error es predecible (porque tenemos conocimiento y control sobre las causas del mismo) mucha veces podemos evitarlo en lugar de manejarlo.

<TODO>

8 Conclusiones

- Hay principios básicos de manejo de errores, independientes de la tecnología:
    - no ocultarlos
    - tratarlos solo si es posible
    - de lo contrario, reportarlos, lo antes posible: fail fast

- El error es malo: nadie quiere que el sistema funcione inadecuadamente. Pero la falla es buena: pone de manifiesto el error, en lugar de ocultarlo, permitiendo que un componente de más alto nivel tome una decisión respecto de cómo tratarlo (a veces este componente será el humano directamente).

- Nos interesa poder diferenciar los errores que se producen dentro de nuestro sistema de los que se ocasionan fuera de este y se manifiestan a través de las interfaces entrantes y salientes, porque deberemos ser más precavidos con estos últimos. Por

otro lado, no debemos volvernos paranoicos con los errores que se produzcan dentro de nuestro sistema.
    - Como caso particular de lo anterior, deberíamos validar todas las entradas que procedan del usuario.

- Y luego, hay distintos mecanismos para trabajar con errores:
    - Las excepciones son un mecanismo común y robusto de lidiar con errores en el paradigma de objetos. Tienen la ventaja de que son difíciles de ignorar, y de que si no son tratados, abortan el programa.
    - Otro mecanismos para lidiar con errores son call and return y continuaciones. Estos mecanismos tiene la contra de que en la mayoría de las tecnologías es más fácil equivocarse y llegar a ignorarlos. Pero dependiendo de la tecnología estos podrían ser nuestra primera opción.

- Es importante mantener un manejo de errores consistente; para ello debemos evitar mezclar distintos mecanismos de manejo de error.

9 Bibliografía y lectura sugerida:

- Effective Java, Capitulo 9
- http://www.erlang.org/doc/reference_manual/errors.html
- http://learnyousomeerlang.com/errors-and-exceptions
- http://c2.com/cgi/wiki?LetItCrash
- http://www.randomhacks.net/articles/2007/03/10/haskell-8-ways-to-report-errors