Clasificación Fundamental de las Excepciones

Juan Manuel Cabo
(jcabo <at> manas.com.ar)
Manas Technology Solutions

Creación: 2003-09-16
Actualización: 2005-12

En este artículo se discute la manera en que se debe organizar una jerarquía de clases de excepciones. Se propone una clasificación top-level en dos clases fundamentales. La novedad de nuestro enfoque se encuentra en la introducción de un criterio para usar y clasificar excepciones basado en la observación del invariante del programa.

Se asume que el lector está familiarizado con la programación usando excepciones y las ha utilizado en algún lenguaje como C++, Java, C# o Python. El contenido de este artículo vale para cualquiera de dichos lenguajes.

Para qué dividir en categorías

Dividir las excepciones en categorías o clases generales es útil solamente si a las excepciones dentro de una categoría se las puede tratar igual desde el punto de vista del catch y de lo que se debe y puede hacer cuando se producen.

Tener las excepciones distribuidas en 20 categorías top-level distintas no sirve de mucho si dentro de, por ejemplo, la categoría *VideoExceptions* tenemos algunas que hay que mostrar al usuario, otras que no, algunas que impiden continuar con la ejecución del programa y otras que son recuperables. Tampoco se puede hacer un *catch* para cada una de las excepciones concretas y actuar específicamente. Hay que abstraer.

Se debe encontrar entonces una **clasificación** basal que las abstraiga pero que nos brinde información sobre qué acción tomar sin conocer la excepción en concreto.

Luego debemos subclasificar las excepciones. Cuanto mas cerca esté mi *catch* del método que produjo la excepción, más específico quiero que sea. Cuanto mas alejado, es decir, cuantos mas *callers* haya entremedio, quiero que sea más general y abarcativo. **Nuestra clasificación, entonces, nos tiene que permitir ir de lo particular a lo general de una manera coherente y predecible.**

Qué clasificación usar

Una clasificación según la **causa** del error puede ser:

- **Program errors**. Puede estar en la sintaxis o en la lógica del programa. El error es "culpa"del programador, por lo tanto requiere un cambio en el código. Este tipo de errores no debería existir en las versiones *release.* Los errores en la lógica de la aplicación son difíciles de encontrar y pueden llegar hasta el diseño.

- **Data Errors**. Son típicamente errores en la entrada de datos del usuario. Se pueden validar los datos y permitir al usuario corregirlos.

- **System Errors.** Pueden provenir de varias causas, incluyendo problemas en bases de datos, timeouts, falta de memoria RAM, errores de disco,etc.

Estas categorías no son muy útiles como clasificiación raíz. Al escribir un bloque *catch* no interesa tanto la causa del error, sino, primeramente, si lo vamos a poder solucionar o no. Sin embargo, es más fácil conocer la causa de un error cuando escribimos un *throw*, que es cuando elegimos qué *exception* usar. De manera que una clasificación como esta tiene relevancia sólo si nos centramos en éste último tipo de código

Tambien se puede clasificar los errores según su efecto, pero éste es difícil de asegurar desde el punto en que se tira el error, que es la función más enterrada en el *call stack*, la cual puede poseer menor información sobre el estado del programa que las superiores.

Por lo tanto, hay que hallar una clasificación top-level que nos indique mejor cómo proceder desde el *catch*, si *catch*ear o no. Tampoco queremos perder la información de la causa del error. Para esto podemos usar la clasificación por causas como **subclasificación**. Cuanto más información tengamos de la causa, mejor podremos recuperarnos del error en los *callers* cercanos al método que produjo el *throw*.

De todas formas, en el mensaje incluido en el *exception* se encuentra la mayor información sobre la causa, aunque este mensaje no es sistematizable, es más bien para informar al usuario y al programador.

La Clasificación de la Standard C++ Library

La libreria standard de C++ emite *exception objects* de los tipos que se ven a continuación en la siguiente jerarquía de clases:

```
exception
    logic_error
        domain_error
        invalid_argument
        length_error
        out_of_range
    runtime_error
        range_error
        overflow_error
        underflow_error
bad_alloc
bad_cast
bad_exception
bad_typeid
```

Citando al Standard de C++ [7]:

> "The distinguishing characteristic of logic errors is that they are due to errors in the internal logic of the program. In theory, they are preventable. [...] The class logic_error defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants."
>
> "By contrast, runtime errors are due to events beyond the scope of the program. They cannot be easily predicted in advance. The class runtime_error defines the type of objects thrown as exceptions to report errors presumably detectable only when the program executes."

Chuck Allison (ver [8]) nos lo aclara:

> "A *bad_alloc* exception occurs when heap memory is exhausted. C++ will generate a *bad_cast* exception when a *dynamic_cast* to a reference type fails. If you re*throw* an exception from within an unexpected handler, it gets converted into a *bad_exception*. If you attempt to apply the *typeid* operator to a null expression, you get a *bad_typeid* exception."
>
> "A logic error indicates an inconsistency in the internal logic of a program, or a violation of pre-conditions on the part of client software. For example, the *substr* member function of the standard *string* class throws an *out_of_range* exception if you ask for a substring beginning past the end of the *string*. Run-time errors are those that you cannot easily predict in advance, and are usually due to forces external to a program. A *range_error*, for instance, violates a post-condition of a function, such as arithmetic overflow from processing legal arguments."

El que un error sea predecible o no en tiempo de ejecución, no es, en mi opinión, la manera mas fuerte de basar la jerarquía, si bien se acerca a lo que buscamos.

La Clasificación C#

Esta es la jerarquía de las *exception*s estándar del .Net Framework. En [6] encontramos la descripción de cada una y los casos en que se aconseja emitirlas.

```
Exception
    SystemException
        IndexOutOfRangeException
        NullReferenceException
        InvalidOperationException
        ArgumentException
            ArgumentNullException
            ArgumentOutOfRangeException
        ExternalException
            COMException
            SEHException
    ApplicationException
```

La dicotomía *SystemException* y *ApplicationException* es muy poco útil o interesante: las *exception*s producidas por el framework o el entorno de ejecución deben derivar de *SystemException* mientras que las producidas por otras clases deben derivar de *ApplicationException*. Más aun, las producidas por objetos COM son *COMException,* sin importar si se tratan de errores de acceso a base de datos, a memoria, de comunicación, etc.

En definitiva, este enfoque no nos ayuda desde el punto de vista del *catch*, puesto que no permite agrupar fácilmente *exception*s similares. Las clasificaciones importantes aquí terminan siendo transversales a la jerarquía. El hecho de que en C# no exista la herencia múltiple nos impide hacer un *workaround*. Para extender la jerarquía de un modo útil, seguramente nos veremos forzados a derivar de subclases de *SystemException*.

La Clasificación Java

En Java tenemos dos tipos superiores de *exception*s: *RuntimeExceptions* y *Exceptions* .

Las primeras indican situaciones de error irrecuperables, que pueden dejar al programa en un estado inconsistente, por lo cual es imposible continuar garantizando todo aquello que el programa asume sobre su estado. Conviene detener la ejecución, informando al usuario si es posible. Por ejemplo: un *new* no allocateó la memoria pedida, o se quiso desreferenciar un puntero nulo o un puntero a memoria fuera del proceso. En general son producidas por la *virtual machine* de Java.

En cambio, las *Exceptions* indican situaciones también excepcionales, pero de las que

algún *caller* puede recuperarse. La consistencia del estado del programa se puede volver a garantizar.

La Clasificación Fundamental

Nosotros adoptaremos la clasificación Java, con algunos pequeños cambios, y extenderemos los conceptos para dar una definición más clara. Además la fortaleceremos con un criterio basado en el invariante.

Definimos dos categorías superiores para las excepciones: ***FatalException*** y ***CheckedException***. También podríamos haberlas llamado *UnrecoverableException* y *RecoverableException* respectivamente.

FatalException:

- Tirar cuando el error es definitivamente irrecuperable (por perderse el invariante del programa al pasar a un estado inconsistente o indefinido).
- Nunca catchear.
- No es necesario dividirlas en subcategorías.
- No se deben relanzar como *CheckedException*'s.

CheckedException:

- Tirar cuando el error puede recuperarse.
- Catchear lo antes posible, en el primer caller que pueda corregir la situación.
- Subdividir en categorías. Cuanto más específico sea el tipo de la *exception*, más información sobre la causa del error tiene un caller para corregirlo.
- Se pueden relanzar como *FatalException*'s.

Las excepciones de tipo *logic_error* de C++ caen todas bajo *FatalException* según nuestras categorías. Tambien las de tipos *bad_alloc, bad_cast, bad_exception* y *bad_typeid*. Como quedará claro al desarrollar nuestro criterio mas adelante, algunas de las de tipo *runtime_error* corresponden a *FatalException*'s y otras a *CheckedException*'s.

¿Cuándo tirar una exception cualquiera?

Cuando el método no puede cumplir su postcondición, es decir, no puede devolver un resultado válido y definido como se indica en su documentación, y no puede hacer nada para arreglar la situación.

El mecanismo que se pone en marcha cuando se ejecuta un *throw* es costoso en performance, por lo que sólo se debe dejar el uso de excepciones para, justamente, situaciones excepcionales y no como un modo de modificar el flujo del programa o emitir eventos. Otro argumento para no utilizar las exceptions como shortcuts para saltar de un punto del programa a otro, más fuerte aun que el de la performance, es el mismo aplicado en contra de los *goto*'s (ver [4]): el progreso del programa se haría más difícil de caracterizar por su estado.

Nunca olvidar que el ideal de la programación con excepciones es el de poder separar el código útil del aquel que se dedica el manejo de errores. Debemos poder programar el código útil, fuera de los *catch*'s, asumiendo que todas las operaciones fueron exitosas. También es una separación del código que detecta el error de aquel que lo resuelve, el cual puede contar con más información de contexto.

Ejemplo de código incorrecto:

```
int dividir (unsigned a, unsigned b) {
    if (b == 0)
        return -1;
    else
        return (a/b);
}
```

Las funciones que llamen a dividir() no deberían andar verificando si la salida fue -1 o si fue correcta. La idea no es dar un error como un resultado especial. Esto impide separar el código útil del código para manejar errores. Dividir() debería producir una *exception*, puesto que su postcondición es: `dividir(a,b) == a/b`.

¿Cuándo tirar una FatalException?. Criterio.

Cuando el programa no debería continuar ejecutándose dado que su estado puede haber quedado inconsistente. Cuando la situación es irrecuperable. **El estado de un programa es inconsistente cuando no se puede garantizar el invariante. Este último es la conjunción de todas las suposiciones que el programa hace sobre sus datos en cualquier momento de la ejecución.**

Todo fragmento de código puede garantizar una salida correcta y a su vez dentro del invariante solo sí la entrada ya cumplía con el invariante. Con todo aquello que se asume de ella. Fuera del invariante nuestro código no puede asegurar que su salida sea correcta y definida. Será **indefinida** e **impredecible**.

Si en un *catch* sobre una *CheckedException* se determina que la situación de error es irrecuperable, se debe tirar desde ese mismo bloque una *FatalException*, así "convirtiendo" de *checked* a *fatal*. ¡Pero atención!: la inversa no es posible. Una vez que se tira una *FatalException* no se la puede convertir a una *CheckedException*, puesto que todo código ejecutado después de lanzada la *FatalException* está "sucio" y su respuesta es indefinida. Esto incluye al código en los *catch*. Ya nada tiene sentido (ver la ``defensa Chewacca" en [3] para más información sobre la falta de sentido y para un poco de humor).

El que un error sea recuperable o irrecuperable es relativo al programa y su invariante. Si uno construyó todo un programa asumiendo que un archivo existe, por ejemplo, un archivo de configuración, una DLL, etc. y la API de Input/Output emite una *FileNotFoundException* al querer abrirlo, se trata de una situación irrecuperable y dicha *exception* se debe "convertir" a una *FatalException*. Ahora, si el archivo buscado era algo menos importante para la integridad del programa, tal como un documento que el usuario pidió, el error es bien recuperable. Toda situación específica es relativa a la construcción general del programa.

Podemos estar seguros de tirar una *FatalException* desde un método si en su invocación no se cumple su precondición. Ejemplo: llamar a `Resize(wnd_handle, width, height)` con `width` negativa o `wnd_handle` igual a *null*. El caller que llamó al método está asumiendo que su precondición se cumple. De otro modo no lo habría invocado.

Atención: Esto no quiere decir que hay que producir *FatalExceptions* cuando se viola la precondición de un método y emitir *CheckedExceptions* cuando se quiebra su postcondición. Al contrario. Ambas se tiran si el método no podrá ejecutarse cumpliendo su postcondición. Y muchas veces se necesita tirar una *FatalException* aunque sí se cumpla la precondición. En esto diferimos con la clasificación estándar de C++ en *logic_exceptions* y *runtime_exceptions*. Nuestro criterio se basa en el invariante del programa (que sería algo así como la precondición de todo el programa), no simplemente en la de cada método en particular (ver [5] para más información sobre pre- y post-conditions).

¿Cuándo catch*ear una FatalException?*

Nunca. Permítante citar [2]:

> "[..]As a rule of thumb, you should always catch a checked exception once you reach a point where your code can make a meaningful attempt at recovery. However, **it is best not to catch [fatal] exceptions**. **Instead, you should allow [fatal] exceptions to bubble up to where you can see them.**
>
> If you do catch [fatal] exceptions, you risk inadvertently hiding an exception you would have otherwise detected and fixed. As a result, catching [fatal] exceptions complicates unit and regression testing. While testing, seeing a stack trace or allowing the test to catch and report [fatal] exceptions lets you quickly identify problems."

El hecho de haberse producido una *FatalException* indica que el programa debe ser modificado, o bien, que se trata de una situación verdaderamente impredecible luego de la cual el programa no está preparado para continuar.

Por ejemplo, una clase de una librería puede tirar una *FatalException* cuando no se la está usando debidamente. El desarrollador deberá modificar el código que la usa para remediar esto.

¿Cuándo tirar una CheckedException?

Cuando algún caller del programa **puede recuperarse** del error. Evidentemente, el método que tira la *exception* no pudo recuperarse del error y quizás el caller inmediato tampoco pueda. La diferencia con las *FatalException* es sutil, pero fundamental: en estas últimas la consistencia del programa ha quedado claramente comprometida, ningún caller podrá recuperarse del error de manera totalmente segura y confiable. El programa no prevee la situación que produjo una *FatalException* y, a lo sumo, su código debe ser modificado para estar preparado para ella. Si esto no es posible, al menos el programa debe ajustarse para suspender su ejecución.

En general, pero no es lo que las define, las checked exceptions se producen en respuesta a errores que se espera que puedan ocurrir, están dentro del uso normal del sistema o biblioteca de código.

¿Cuándo catch*ear una CheckedException?*

Apenas estemos en un nivel en que podamos recuperarnos del error y restaurar el flujo normal del programa.

Se llaman *CheckedExceptions* porque con sus equivalentes en Java, el compilador obliga a *catch*earlas desde un método si en su declaración indica que no tira ninguna *exception*, o a declararlas en su signatura. **Tarde o temprano, si el método top-level no puede tirar *exceptions*, todas las *CheckedExceptions* deben ser *catch*eadas dentro del código.**

Ejemplos

Es difícil elaborar un ejemplo sobre *FatalExceptions*, dado que ponerse a verificar cosas que están en el dominio de lo que el programa asume, de su invariante, puede parecer algo paranoico. Al escribir un programa uno asume cosas razonables. Pero el espíritu de las condiciones para tirar una *FatalException* es el mismo que el de las condiciones verificadas con los viejos ASSERTs de C.

- FatalException:

Método invocado fuera de su precondición:

```
int dividir (unsigned a, unsigned b) {
    if (b == 0)
        throw InvalidArgumentFatalException(
                  "Can't divide by zero");
    else
        return (a/b);
}
```

En máquinas de estado (ie: clases que definen objetos con muchos estados) tirar *FatalExceptions* cuando el invariante de la clase dejó de cumplirse (ver [5] para más información sobre *class invariants*):

```
void VideoCapture::GetLastFrame(char* buffer, size_t bufferLength) {
     if (this->state == VideoCaptureState::Capturing &&
         this->captureBuffer.size() == 0)
     {
          throw InvalidStateFatalException(
                "Capturing state should imply an available image buffer.");
     }
     if (buffer == NULL) {
          throw InvalidArgumentFatalException(
                "Parameter 'buffer' is a NULL pointer.");
     }
     ....
}
```

En este caso, la precondición del método va más allá de sus parámetros para abarcar el estado interno de los datos de la clase a la que pertenece.

En una máquina de estados, el estado del objeto implica una serie de cosas sobre sus datos. Debemos poder programar el código útil asumiendo que dichas cosas son ciertas. Por eso es muy recomendable disponer de un método, por ejemplo, *CheckInvariant(),* en toda máquina de estado de cierta complejidad, que revise que los datos del objeto concuerden con el garantizado por su estado. Un método tal debe ejecutarse como primera y última instrucción en todo método de la interfaz pública y su uso puede desactivarse en las versiones *release*. *CheckInvariant()* debe fallar con un *FatalException* si encontrara una inconsistencia en el estado.

Otro caso claro en donde emitir *FatalException*'s se da en aquellos métodos que si no devolverían un puntero a `NULL` cuando no pueden devolver un objeto válido.

En C++, desreferenciar un puntero a `NULL` suele generar un error por parte del sistema operativo, mientras que desreferenciar un puntero a "basura" puede corromper código o datos del programa o bien resultar en un fallo a memoria protegida por parte del sistema operativo. En C# y Java el entorno de ejecución tira una *exception* automáticamente. Esta *exception* generada automáticamente tiene el espíritu de una *FatalException.*

¿Por qué tirar una *FatalException* en lugar de devolver `NULL`? Porque el resto del programa asume que el método siempre devuelve un objeto válido y no podría seguir adelante si no fuera así: se desreferenciaría un puntero a `NULL`. El programa podría no asumir esto y consultar antes el estado del objeto al que pertenece el método, pero eso ya es parte del *fix*.

Algún lector podría quejarse y señalar: *"pero mi método no deberia tirar una* exception*, deberia devolver null, si igualmente lo puedo checkear después que lo llamo...".* ¡Mal. No y no!. A tal lector le respondo: disculpe, mi estimado, pero usted no entendió nada. Antes

de mandarlo a leer todo de vuelta, le explico un poco, por compasión nomás: (1) Hay que separar el código útil del código para errores. (2) Para ésto los métodos no deben devolver errores codificados en valores especiales (*return codes*). (3) ¿Qué sucedería si usted se olvida de verificar que el valor retornado no sea `NULL` o -1 según el caso?.

El motivo por el que muchas funciones en varias APIs devuelven `NULL` o cosas así, es que dichas APIs también aspiran a funcionar con lenguajes sin *exception handling*, como el viejo C.

- CheckedException:

```
Printer TcpConnection::Connect(string address, int port) {
     ...
     if (socketError == SOCKERR_NOTREACHEABLE) {
         throw ConnectException("The specified address is not reacheable");
     }
}
```

Algunas reglas generales. Construcción del resto de la jerarquía

Para usar, derivar y construir el resto de la jerarquía de excepciones, sugiero seguir las siguientes reglas:

1. Subclasificar las excepciones siempre pensando en el *catch*.
2. Elegir qué excepción tirar pensando en el *catch*. No conformarse con una existente, ni tener miedo a derivar nuevas *exceptions*. Cuanto más específicas sean, más información expresarán.
3. No tiene mucho sentido crear subcategorías bajo *FatalException* porque no se *catch*ean. Se pueden dejar sus clases derivadas sueltas directamente bajo la categoría *FatalException*..
4. Derivar una nueva *CheckedException* solamente si tiene sentido escribir un *catch* para ella sola.
5. Nunca, pero nunca, hacer un *catch-all* (*catch(...)* en C++ o *catch(*Exception) en C#) en otro lugar que no sea el método top-level del programa (*main,* etc.). Por lo tanto las *class libraries* **nunca** deben hacer un *catch-all*.

Dos fuerzas deben coexistir en la mente del desarrollador al intentar derivar una nueva clase *exception*: (1) el buscar que el tipo de la *exception* sea lo mas específico posible, es decir, que esté bien enterrada en la jerarquía y (2) el buscar reusar lo mas posible una categoría, y así poder agrupar coherentemente desde el punto de vista del *catch*.

Debemos notar que cuantas más *exception*s hijas distintas encierre una categoría, menos información nos dará su *throw*. Por otro lado, cuanto menos subclases tenga, menos poderoso sera su *catch*. Encontrar la proporción justa es una cuestión de gusto y práctica.

Bibliografía Online

- [1] "Understanding Errors"

  http://livedocs.macromedia.com/coldfusion/6/Developing_ColdFusion_MX_Applications_with_CFML/Errors3.htm
- [2] Anthony Sintes. ″Exceptions: Don't Get Thrown for a Loss."

  http://www.javaworld.com/javaworld/javaqa/2002-02/01-qa-0208-exceptional.html
- [3] South Park. "The Chewbacca Defense."

  http://www.connect-dots.com/Poofs/chewbacca.html

  http://en.wikipedia.org/wiki/Chewbacca_Defense
- [4] Edsger W. Dijkstra. ″Go To Statement Considered Harmful."

  http://www.acm.org/classics/oct95/

  Original: *Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147-148.
- [5] Chung, Fisher, Simonet. "Dynamic Analisis – Assertion Checking."

  http://www.cis.ksu.edu/~hankley/d841/old/assertion/
- [6] Microsoft. "Error Raising and Handling Guidelines." *DotNet Framework.*

  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconerrorraisinghandlingguidelines.asp
- [7] "December 1996 C++ ISO Standrard Draft."

  http://www.itga.com.au/~gnb/wp/
- [8] Chuck Allison. "Error Handling with C++ Exceptions"

  http://www.freshsources.com/Except1/ALLISON.HTM