****

**Paradigma**

**Orientado a Objetos**

**Módulo 10:**

**Manejo de errores**

**por Fernando Dodino, Pablo Tesone**

**Versión 2.0**

**Marzo 2016**

Distribuido bajo licencia [Creative Commons Share-a-like](https://creativecommons.org/licenses/by-sa/4.0/legalcode)

**Indice**

[1 Errores y excepciones](#_gjdgxs)

[2 Breve ejemplo: modelar un monedero](#_v6rbsmzcmpiu)

[3 Nuestro objetivo](#_499qw9lslakc)

[4 Solución en Wollok](#_cp8fb6qlnjol)

[5 Alternativas para modelar una excepción](#_lmv41ohjcxhl)

[5.1 Mirar a un costado](#_1czd0pkson7l)

[5.2 Códigos de error](#_wgrtsbrbq62z)

[5.3 Excepciones](#_fmnis9l4zdi8)

[6 Cómo generar una excepción](#_su23n4ew0yhf)

[7 Dónde se atrapa](#_gh204527ke9g)

[8 Excepciones de usuario y de programa](#_8ldbjnjzlghu)

[9 Tareas para el lector](#_a8zp8xjgvahc)

[10 Buenas y malas prácticas](#_s69a4yyuiici)

[10.1 To throw or not to throw](#_rkpsn8550meq)

[10.2 Consistencia en el valor de retorno de un método](#_4rxqxtc6hssn)

[10.3 Mejorando los mensajes de error](#_aa9v82qe28ep)

[11 Resumen](#_jkaxxuepyf7t)

# 

# 

# 1 Errores y excepciones

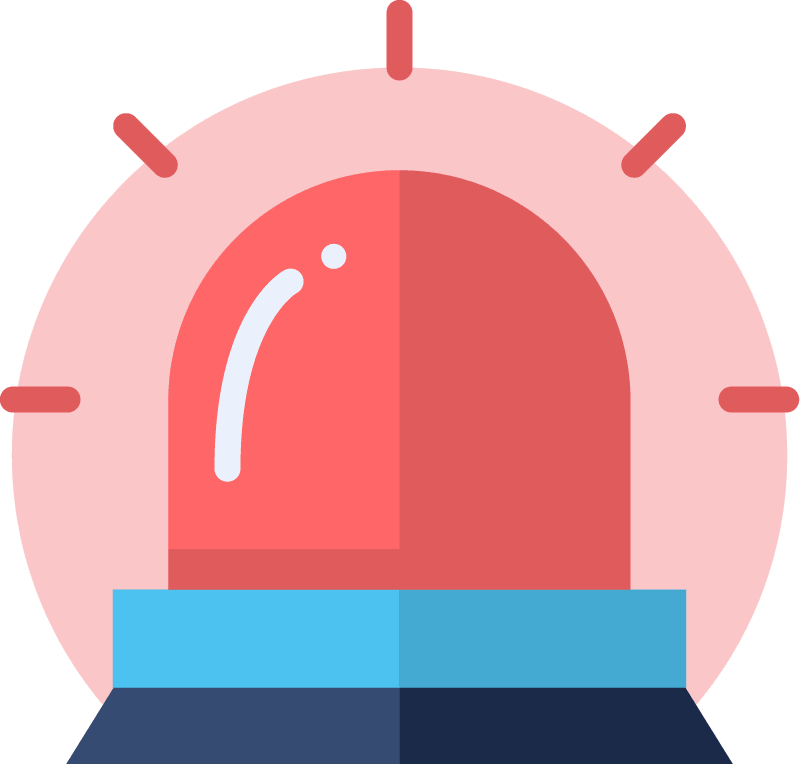
Recordemos la definición de sistema dentro del paradigma de objetos

|  |
| --- |
| ***Sistema =*** conjunto de objetos que colaboran para un objetivo común. |

Entonces precisamos que haya comunicación entre los objetos. Cuando un objeto le envía un mensaje a otro,



el emisor espera que el receptor pueda realizar esa tarea. Ahora ¿qué sucede si falla algo en el método x?



Una ***excepción*** es un evento que altera el flujo normal de mensajes entre objetos.

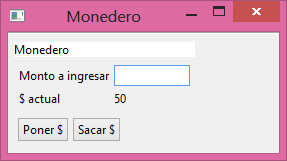
Las excepciones tienen orígenes diferentes: desde errores serios de hardware, como la ruptura del disco rígido, hasta errores simples de programación, como tratar de acceder a un elemento fuera de los límites de un vector, dividir por cero o enviar un mensaje a una referencia que no apunta a ningún objeto.

# **2 Breve ejemplo: modelar un monedero**

Se quiere modelar un monedero al cual se le puedan realizar dos operaciones:

* Poner plata
* Sacar plata

Tenemos una interfaz de usuario que tiene el siguiente formato:



En el campo “$ actual” figurará la cantidad de plata que tiene el monedero. Inicialmente el monedero comienza con 500 (quinientos) pesos. El sistema debe validar lo siguiente:

* para sacar o poner plata en el monedero, el monto debe ser positivo, mayor a cero
* no se puede sacar más plata de lo que tiene el monedero

# 3 Nuestro objetivo

* entender cómo hay que manejar errores en el dominio
* discriminar el tratamiento de errores de usuario y los de sistema

# 4 Solución en Wollok

Modelamos en Wollok al objeto de dominio monedero[[1]](#footnote-0)

**object** monedero {

**var** plata = 500

**method** plata() = plata

**method** poner(cantidad) { plata = plata + cantidad }

**method** sacar(cantidad) { plata = plata - cantidad }

}

Nada extraño, pero todavía no contemplamos las validaciones que el usuario nos pidió. Tenemos que definir qué vamos a hacer si la cantidad es un número negativo...

# 5 Alternativas para modelar una excepción

## 5.1 Mirar a un costado

La peor decisión que podemos tomar es ignorar un valor de entrada que no satisface las condiciones que pide el negocio[[2]](#footnote-1):

**method** poner(cantidad) {

**if** (cantidad > 0) {

plata += cantidad

}

}

¿Por qué decimos que se ignora? Porque el resultado es desconcertante para quien envía un mensaje

**>>>** monedero.**poner**(**-**2)

**>>>**

No da error, pero tampoco se produce efecto colateral en el monedero, que sigue teniendo la misma plata. Y lo que pasa en la interfaz de usuario es que el usuario presiona una y otra vez el botón “Poner $”, sin notar cambios en el monedero.

Esta opción por lo tanto queda descartada de plano.

## 5.2 Códigos de error

Una opción más interesante podría ser utilizar códigos de retorno numéricos:

**method** poner(cantidad) {

**if** (cantidad > 0) {

plata += cantidad

} **else** {

**return** -1

}

}

Entonces el método poner ya no solo tiene efecto colateral, sino un valor de retorno que hay que inspeccionar a su salida.

Esta estrategia tiene algunas desventajas

* ahora la definición del método no es consistente, ¿qué pasa si no hay error? ¿qué devuelve?
* obliga a quien llama a preguntar el valor de retorno antes de continuar con la siguiente línea de una secuencia de instrucciones,
* cuando hay varios mensajes de error posibles las preguntas se encadenan en ifs que dificultan el seguimiento
* los números mágicos -1, -2, no tienen descripciones representativas. Por eso a veces se reemplazan los números con mensajes de error alfanuméricos, aunque sigue siendo difícil rastrear el origen del problema.

Todos estos problemas lo vemos en el siguiente ejemplo, donde queremos verificar si un alumno puede inscribirse a una materia:

**method** inscribirse(alumno, materia) {

**var** codigoError = OK

**const** codErrorMateria = materia.validarCupo()

**if** (codErrorMateria = 0) {

**const** codErrorAlumno = alumno.validarCorrelativas(materia)

**if** (codErrorAlumno = 0) {

... generamos la inscripción ...

}

**else** {

codigoError = ERROR\_ALUMNO\_NO\_CUMPLE\_CORRELATIVAS

}

} **else** {

codigoError = ERROR\_MATERIA\_SIN\_CUPO

}

**return** codigoError

}

La lógica de la inscripción podría complicarse aún más:

* si agregamos un plazo límite para la inscripción
* si aparecen múltiples validaciones sobre la materia o el alumno, por ejemplo que el alumno haya aprobado al menos dos finales el último año

Aquí vemos que no solo se recibe el código de error como valor de retorno del mensaje materia.validarCupo(), sino que este código también debe propagarse hacia quien esté llamando al método inscribirse()[[3]](#footnote-2).

## 5.3 Excepciones

La tercera opción es lanzar una excepción por cada condición que salga del flujo normal, en particular

1. al sacar o poner una cantidad menor a cero
2. al sacar más de lo que permita el monedero

Esto lo haremos a continuación.

# 6 Cómo generar una excepción

En el método poner tenemos que contemplar que la cantidad sea positiva:

**method** poner(cantidad) {

**if** (cantidad < 0) {

**throw new** UserException("La cantidad debe ser positiva")

}

plata += cantidad

}

¿Cómo funciona? Si la cantidad es menor a cero, se lanza una excepción cortando la secuencia de envío de mensajes que tenga el método: nunca llegarán a incrementar la cantidad de plata al monedero. Entonces quien envió el mensaje poner() al objeto monedero recibirá la excepción.

La clase UserException la definimos con un constructor que recibe el mensaje como parámetro[[4]](#footnote-3):

**class** UserException **inherits** Exception { }

En el método sacar, una vez más, tenemos que validar la cantidad:

**method** sacar(cantidad) {

**if** (cantidad < 0) {

**throw new** UserException("La cantidad a retirar debe ser positiva")

}

**if** (cantidad > plata) {

**throw new** UserException("Debe retirar menos de " + plata)

}

plata -= cantidad

}

El lector podrá pensar, ¿no se repite la misma validación para poner y sacar? Sí, podemos definir una validación común para ambas operaciones:

**method** poner(cantidad) {

**self**.validarMonto(cantidad)

plata += cantidad

}

**method** sacar(cantidad) {

**self**.validarMonto(cantidad)

**if** (cantidad > plata) {

**throw** **new** UserException("Debe retirar menos de " + plata)

}

plata -= cantidad

}

**method** validarMonto(cantidad) {

**if** (cantidad < 0) {

**throw** **new** UserException("La cantidad debe ser positiva")

}

}

# 7 Dónde se atrapa

Por lo general, los objetos de dominio (el cliente, el alumno, el héroe, un ave) no suelen atrapar los errores. Más bien lo único que hacen es lanzar los errores o dejar que ocurran, porque es raro poder hacer algo en el dominio cuando recibimos una excepción.

Volvamos al ejemplo anterior de la inscripción, ahora codificado mediante excepciones:

**method** inscribirse(alumno, materia) {

materia.validarCupo()

alumno.validarCorrelativas(materia)

... inscripción propiamente dicha ...

}

El método quedó bastante más corto, porque si la validación del cupo de la materia falla, no hay mucho que podamos hacer. Tampoco podremos hacer mucho si el alumno no tiene las correlativas al día. Si bien Wollok ofrece la posibilidad de envolver el código que puede fallar en un bloque try/catch, de manera de “atrapar” la excepción y poder definir comportamiento por el flujo alternativo...

**method** inscribirse(alumno, materia) {

**try** {

materia.validarCupo()

alumno.validarCorrelativas(materia)

... inscripción propiamente dicha ...

} **catch** e : UserException{

¿¿¿¿¿ Qué hacer ?????

}

}

...no sabemos muy bien qué hacer en el bloque catch. Por eso, lo mejor que podemos hacer en este lugar, es simplemente dejar que la excepción que se dispare en materia o alumno siga su curso hacia quien nos mandó el mensaje inscribirse().

**method** inscribirse(alumno, materia) {

materia.validarCupo()

alumno.validarCorrelativas(materia)

... inscripción propiamente dicha ...

}

¿Quién puede atrapar **realmente** una excepción?

* un test unitario[[5]](#footnote-4)
* la interfaz de usuario, que está fuera del alcance del presente apunte

Ellos deben ser los encargados de atrapar la excepción antes de que la aplicación falle y se cierre.

# 8 Excepciones de usuario y de programa

En base al cliente que va a recibir la excepción, podemos clasificar a las excepciones en dos tipos diferentes:

* **Excepciones de usuario o de dominio:** ocurren en el uso de la aplicación y son entendibles para el usuario final (“no hay saldo en la cuenta corriente”, “no hay stock del producto a facturar”, “no hay precio del producto a facturar”, etc.)
* **Excepciones de programa:** se producen cuando se ejecuta código de la aplicación y las puede analizar un especialista técnico (“falló el acceso al motor de la base de datos”, “hubo división por cero”, “el objeto no entiende este mensaje - MessageNotUnderstoodException”, etc.)

La naturaleza de ambos tipos de excepción son diferentes: en general las excepciones de negocio (o de aplicación) requieren que el usuario corrija la información que quiere ingresar al sistema (y valorice el producto, o bien seleccione un producto alternativo para facturar, o trate de sacar plata de otra cuenta bancaria), en tanto que las excepciones de programa requieren una corrección por parte de un usuario técnico (que chequeará la conexión a la base de datos o bien corregirá el código que originó el error).

Por lo tanto, las acciones a tomar cuando armamos cada tipo de excepción son diferentes: en las excepciones de negocio intentamos que el usuario vea una pantalla amigable donde le mostramos el problema que hubo al tratar de completar una acción con un mensaje representativo (e incluso proponiéndole soluciones alternativas para que la tarea se realice), mientras que en las excepciones de programa también mostramos una pantalla amigable al usuario, pero reservamos todos los detalles internos al desarrollador. De esa manera, las excepciones terminan siendo una herramienta más que ayuda a que nuestra aplicación se vuelva más robusta y confiable.

Entonces, para el componente de UI (interfaz de usuario), quizás sea más conveniente no tener que atrapar distintas excepciones, sino diferenciar las que son de negocio y las que no. En el ejemplo del monedero, si tenemos una tecnología de interfaz de usuario similar a la que venimos trabajando en el dominio, podremos hacer algo parecido a ésto:

**method** sacar() {

**try** {

monedero.sacar(<<monto>>)

} **catch** e : UserException {

**self**.mostrarMensajeDeAdvertencia(e.getMessage())

} **catch** e : Exception {

e.printStackTrace()

**self**.mostrarMensajeDeError("Ocurrió un error en la aplicación. Consulte al administrador.")

}

}

# 9 Tareas para el lector

Hacer los cambios necesarios para incorporar las siguientes validaciones:

* no se pueden hacer más de 3 depósitos el mismo día
* no se puede extraer más de 1.000 pesos diarios

# 10 Buenas y malas prácticas

Ya sabemos que no hay recetas, pero sí libros de auto-ayuda... aquí van algunas palabras que pueden servir de puntapié inicial para debates a la hora de trabajar con excepciones:

* Sólo las atrapa el que las sabe tratar. Dado que en el contexto de una materia inicial de objetos estamos concentrados en modelar el negocio, es poco frecuente que necesitemos usar el bloque try/catch.
* Las excepciones deberían tratarse en pocos lugares, excederse en el control produce código poco claro.
* Utilizar mensajes claros en las excepciones, tanto para los usuarios como para los desarrolladores.
* Representamos como excepciones las condiciones que salen del flujo normal del negocio.
* No debemos nunca atrapar una excepción (“cachearla” por así decirlo) y no hacer nada. No hacer nada incluye hacer e.printStackTrace(), console.println(obj) o cosa similar, donde **no se avisa del error al componente llamador, con lo cual es probable que este error propague errores posteriores en otros componentes de la aplicación y estemos buscando el error en otro lugar durante un buen tiempo**.
* Este ejemplo muestra cómo el uso de excepciones se desaprovecha para trabajar con valores de retorno, algo que no es aconsejable:

**method** metodo() {

**try** {

**var** chiara = **new** Cliente()

chiara.comprar(25)

**return** 0

} **catch** e : NoTengoUnMangoException {

**return** -1

}

}

¿Cómo debería quedar? Mucho más simple si no tratamos de controlar las cosas y dejamos que fallen:

**method** metodo() {

**var** chiara = **new** Cliente()

chiara.comprar(25) --> sabemos que puede fallar

}

Lo mismo si retornamos true / false

## 10.1 To throw or not to throw

Al modelar la contratación de un servicio, hay una regla de negocio que pide que validemos que haya suficiente presupuesto para pagarle al contratista. Esto se implementa con una excepción de negocio, y está bien que así sea:

**method** contratarA(contratista) {

**if** (contratista.costoCasa(casa) > **self**.presupuesto())

**throw** **new** UserException("Presupuesto insuficiente")

...

No es posible contratar a un contratista sin pasar esta validación.

Ahora al querer conocer cuáles fueron los contratistas a los que llamé...

**method** contratistasContratados(){

**if** (serviciosContratados.isEmpty()){

**throw new** UserException("No contrató servicios aún")

}

**return** serviciosContratados.map({ ... })

}

Esto es conceptualmente incorrecto, si un cliente no contrató servicios no puedo saber los contratistas contratados ni cuánto gastó en servicios. Además un cliente comienza sin servicios contratados, no es una condición excepcional del negocio.

Además, el map además funciona bien cuando no hay elementos en la colección:

**>>> [].map({ element => element \* 2 })**

**[]**

## 10.2 Consistencia en el valor de retorno de un método

Consideremos el siguiente ejemplo, donde tenemos un taxista Daniel que toma viajes si el destino es Alejandro Korn:

**object** dani {

**method** puedeTomar(viaje) {

**if** (viaje.destino() == "Korn") **return** **true**

**throw** **new** Exception("No puede tomar el viaje")

}

Si la pregunta es “¿puede tomar un viaje?”, es inconsistente que el método

* devuelva true si Daniel puede tomar el viaje
* o tire error en caso contrario. Lo que debe hacer es... devolver false si no lo puede tomar.

**object** dani {

**method** puedeTomar(viaje) = viaje.destino() == "Korn"

Ahora sí estamos siendo consistentes, el método devuelve siempre un valor booleano. Y devolvemos una expresión booleana como valor de retorno del método.

Por otra parte, distinto es el comportamiento cuando buscamos agregar un viaje a dani:

**method** tomarViaje(viaje) {

**if** (!**self**.puedeTomar(viaje)) {

**throw** **new** Exception("No puede tomar el viaje")

}

viajes.add(viaje)

}

Claro, aquí el método tomarViaje no devuelve nada, entonces está bien que

* no retorne nada si funciona ok
* o tire un error en caso de que no se pueda tomar el viaje

## 10.3 Mejorando los mensajes de error

Ahora Daniel toma viajes

* si son a Alejandro Korn
* y si el pasajero tiene más de 50 años

Elaboramos una primera versión

**method** puedeTomar(viaje) = viaje.destino() == "Korn" && viaje.pasajero().edad() > 50

**method** agregarViaje(viaje) {

**if** (!**self**.puedeTomar(viaje)) {

**throw** **new** Exception("No puede tomar el viaje")

}

viajes.add(viaje)

}

¿Cuál es el motivo por el cual no puede tomar un viaje? No lo sabemos. Cuando tenemos que pasar muchas validaciones a la vez, está bueno que el mensaje de la excepción refleje el motivo por el cual la operación no puede satisfacerse.

Vemos una segunda validación, que

* por un lado trata de evitar duplicaciones en las preguntas
* y busca que los mensajes de error sean representativos

**method** destinoCorrecto(viaje) = viaje.destino() == "Korn"

**method** edadPasajeroCorrecta(viaje) = viaje.pasajero().edad() > 50

**method** puedeTomar(viaje) = **self**.destinoCorrecto(viaje) &&

**self**.edadPasajeroCorrecta(viaje)

**method** agregarViaje(viaje) {

**if** (!**self**.destinoCorrecto(viaje)) {

**throw** **new** Exception("El destino debe ser Korn")

}

**if** (!**self**.edadPasajeroCorrecta(viaje)) {

**throw** **new** Exception("El pasajero debe tener 50 años o más")

}

viajes.add(viaje)

}

De todas maneras, esta técnica solo necesitamos implementarla cuando queremos tener dos métodos: la pregunta puedeXXX (sin efecto) y la acción hacerXXX (con efecto).

# 11 Resumen

Las excepciones son un mecanismo que nos sirve para modelar condiciones que salen del flujo normal del negocio. Para los objetos de dominio las formas correctas de usar este concepto es muy simple:

* en el caso de los errores de programa, no se pueden prever, así que no debemos hacer nada
* en el caso de los errores de usuario, debemos detectarlos y alertar al usuario lo más tempranamente posible

Por último, hay mecanismos para atrapar las excepciones y volver al sistema a un estado consistente, aunque por lo general son raros los casos en el que un objeto de dominio puede hacer algo al respecto.

1. El ejemplo completo se puede descargar en <https://github.com/wollok/excepciones-monedero> [↑](#footnote-ref-0)
2. Con la palabra “negocio” o “dominio” nos referimos a los requerimientos que estableció el usuario en alguno de los relevamientos [↑](#footnote-ref-1)
3. esto produce un grado de acoplamiento mayor entre los objetos involucrados [↑](#footnote-ref-2)
4. Por el momento no nos detendremos a explicar qué implica escribir “inherits”, para el lector curioso recomendamos leer [este apunte](https://docs.google.com/document/d/1KdG7NrKPgPh4bAcyLuDG2G1iWP7Ze2GFs91qzlvDKqI/edit?usp=drive_web) [↑](#footnote-ref-3)
5. El lector interesado puede estudiar el capítulo de [Testing avanzado](https://docs.google.com/document/d/1caDE_mlP1QMfzyVpyvh-tKshjAeYLXBkXDYrTX5zFUI/edit?usp=sharing) que cubre los tests que buscan cubrir situaciones excepcionales [↑](#footnote-ref-4)