Implementación de estrategia para manejo de excepciones basada en componentes: las fachadas de seguridad

Nora S. M. Blet
Departamento de Sistemas e Informática - Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario – Argentina, nblet@fceia.unr.edu.ar

Abstract

Exception handling mechanisms were conceived as a means for incorporating fault tolerance into software systems. More than two thirds of the application's code is dedicated to detecting and handling errors and exceptions. These exception handling systems are often misuse and (or) abused. The design of exception handling in an application is seen as a difficult task because: introduce significant complexity, may conflict with many of the goals of object oriented design, suffers from lack of guidelines, among other things. For these reasons the handling of exceptions and errors is one of the major subject of the software architecture and one of the most critical, but overlooked aspect of critical system design and analysis.
I will apply the concept of dynamic proxy to implement a simple but interesting exception handling design approach called safety facades, available in the software architecture literature. The article describing security façades is light on details. The security façade form a new approach that introduce and architecture and best practices to build a viable framework for systems with exception handling.

Keywords: Exception handling, Security Facades, Dynamic proxies, component software, Design patterns, Programming languages, Java.

Resumen

Los mecanismos de manejo de excepciones fueron concebidos como un marco de trabajo para implementar tolerancia a fallos en sistemas de software. Más de dos tercios del código de una aplicación, está dedicado a la detección y manejo de errores y excepciones. A menudo, los mecanismos de manejo de excepciones, son mal empleados o se abusa de ellos; el diseño de una estrategia efectiva se considera una tarea difícil puesto que: aumenta considerablemente la complejidad de los sistemas, plantea conflictos con algunos de los principios del diseño orientado a objetos, no existen pautas eficientes, entre otras causas. Por tanto, el manejo de errores y excepciones, es uno de los temas más importantes de la arquitectura del software y, uno de los aspectos más cruciales pero menos tenido en cuenta, en el análisis y diseño de sistemas críticos.
Aplicando el concepto de *proxies* dinámicos, se investigan los detalles de implementación de una simple aunque interesante, estrategia basada en componentes para el manejo de excepciones, disponible en la literatura: las fachadas de seguridad; cuya descripción original carece de información pormenorizada. Las fachadas de seguridad constituyen un nuevo enfoque, que introduce una arquitectura y directrices, estableciendo un *framework* para el manejo de excepciones.

Palabras claves: Manejo de excepciones, Fachada de seguridad, *Proxies* dinámicos, Componentes de software, Patrones de diseño, Lenguajes de programación, Java.

1 INTRODUCCION

Los mecanismos de manejo de excepciones fueron concebidos como un marco de trabajo para implementar tolerancia a fallos en sistemas de software [2]. Idealmente un mecanismo de manejo de excepciones debería permitir escribir programas donde: (i) el código normal esté léxicamente separado del que permite el manejo de excepciones y, cada uno de ellos, pueda mantenerse en forma independiente [19, 17, 13]; (ii) el impacto del código responsable del manejo de excepciones, en la complejidad del sistema completo, sea mínimo y, (iii) una primitiva versión inicial del sistema de manejo de excepciones, pueda evolucionar hacia otra, con técnicas más sofisticadas, sin cambios sustanciales en la estructura del sistema [19].

Sin embargo, a menudo en la práctica diaria, los mecanismos de manejo de excepciones son mal empleados o se abusa de ellos [21]; el diseño de una estrategia efectiva, es considerada por los desarrolladores de sistemas una tarea difícil [4], puesto que: aumenta considerablemente la complejidad de los sistemas, plantea conflictos con algunos de los principios del diseño orientado a objetos, no existen pautas eficientes [29], entre otras causas. Por tanto, el desarrollador de sistemas, tiende a concentrarse en el diseño de la actividad normal de los mismos, relacionada con los requerimientos funcionales de las aplicaciones. Finalmente se termina generando sistemas con mecanismos de manejo de las situaciones excepcionales deficientes, difíciles de mantener y con baja capacidad de reutilización.

En la práctica, la mayor parte de un sistema, [26] está dedicada a tratar con situaciones consideradas anormales, excepcionales o no deseadas; la parte más pequeña corresponde a la propia aplicación: Cristian [7] observa que más de dos tercios del código! está dedicada a la detección y manejo de errores y excepciones y que; es muy probable que esta porción, contenga errores (*bugs*) ya que, además de su complejidad, rara vez se ejecuta y tampoco se ensaya adecuadamente.

Por todo esto, el manejo de errores y excepciones, es uno de los temas más importantes de la arquitectura del software y, uno de los aspectos más cruciales, pero menos tenido en cuenta, en el análisis y diseño de sistemas críticos. El ejemplo más citado en la literatura para ilustrar este punto, es el accidente del cohete Ariane 5 debido a una excepción no manejada.

Habiéndose examinado distintas soluciones para obtener un manejo de excepciones eficiente y practicable, en este artículo se escogió, en particular, investigar una estrategia basada en componentes, propuesta por Siedersleben [26], simple, totalmente factible y basada en la experiencia de muchos proyectos reales de software a gran escala: las fachadas de seguridad. La estrategia elegida está incluida dentro de la arquitectura de referencia Quasar (por *Quality Software Architecture for Business Information Systems*), resultante del trabajo de 7 años del área de investigación y desarrollo de la empresa sd&m de Munich, Alemania, cuyo líder es el Profesor Dr. Johannes Siedersleben. Klaus Renzel, miembro de la misma empresa, presenta en [22], un lenguaje de patrones para el diseño orientado a objetos de los componentes necesarios para el manejo de errores, en sistemas de información de negocios a gran escala; perfectamente documentado en más de 100 páginas.

La descripción original de las fachadas de seguridad [25, 26] ofrece pocos detalles; en este trabajo se analiza la implementación de las mismas, basada en el concepto de *proxies* dinámicos.

El código de ejemplo está escrito en Java, aunque los resultados pueden aplicarse también a otros lenguajes orientados a objetos.

2 BACKGROUND

2.1 Terminología y conceptos básicos

2.1.1 Componente software

Se adopta aquí la definición ampliamente aceptada de Clemens Szyperski [24, 1]: un componente de software es una unidad de composición con interfaces contractualmente especificadas y explícitas sólo con dependencias dentro de un contexto. Un componente de software puede ser utilizado independientemente y está sujeto a la composición por parte de terceros.
Mientras que, un componente arquitectural es un concepto abstracto [1], un componente de software es una implementación concreta de uno o más componentes arquitecturales. Hasta ahora, sigue existiendo una brecha entre la concepción de la arquitectura de un sistema y su implementación concreta en un lenguaje de programación.

2.1.2 Excepciones

Aunque el trabajo de Goodenough [15] puso los cimientos de la terminología relacionada con las excepciones, aún hoy día, no existe un acuerdo en la definición del concepto de excepción, como así tampoco en cuanto a su utilización [5]. En realidad, la palabra excepción sugiere "algo que ocurre muy raramente", o, "desastre". Sin embargo, esto se condice sólo parcialmente, con el uso que comúnmente se hace de ellas en la práctica diaria de la programación. Goodenough [15] considera a las excepciones como un medio para informar al invocante de una operación, acerca de distintas condiciones, tales como: errores ocurridos durante la ejecución de la misma, clasificación de los resultados retornados al ejecutarla (por ejemplo: *overflow*, fin de archivo, etc.) o monitoreo de determinados eventos significativos, que no siempre representan fallas (por ejemplo: indicar la cantidad de registros que pudieron procesarse).
En Java, sólo las excepciones de tipo `RuntimeException` tales como, `NullPointerException` o `ClassCastException`, indican problemas severos mientras que, las excepciones comprobadas son usadas para propósitos muy diferentes, alguno de los cuales no son para nada excepcionales: `InterruptedException` está relacionada con la sincronización de hilos de control.
Por tanto, el término excepción se tornó ambiguo e, intentó reemplazárselo por otros más descriptivos de su función: Parnas [19] habla de evento no deseado y Siedersleben [25] propone desastre o emergencia. Sin embargo, todos implican una valoración, y por tanto, en cada proyecto particular, pueden generarse extensas discusiones acerca de qué es menos o más malo (o sea, si es un error del sistema o una emergencia o un desastre). Siedersleben [26] considera que el punto en cuestión para determinar qué considerar como excepción es muy distinto: cada operación puede retornar básicamente dos diferentes tipos de resultados: normal o anormal (utilizado aquí como su opuesto).

2.1.3 Resultados normales y anormales

Los resultados normales se corresponden con el nivel de abstracción de la interface, no contienen detalles de implementación y pertenecen a la misma categoría de software. Los errores de aplicación, aunque son resultados no deseados, son esperados y se los considera incluidos dentro de los resultados normales. Los errores deben manejarse localmente, nunca propagarse: el invocante de una operación puede reaccionar apropiadamente ante la ocurrencia de los mismos. Siedersleben [25, 26] recomienda, en lo posible, usar valores de retorno para indicar un error, en caso contrario hacerlo mediante una excepción (en Java: excepciones comprobadas).

Los resultados anormales, por el contrario, son inesperados, pertenecen a otra categoría de software y, a menudo, revelan detalles de implementación. Siedersleben [25, 26] recomienda señalarlos tan pronto como sea posible (programación defensiva); en el caso de Java, mediante excepciones no comprobadas. Aunque el invocante de una operación es libre de manejar o ignorar los resultados anormales, casi siempre el procesamiento de los mismos está a cargo de mecanismos de manejo de excepciones a un nivel superior; en este trabajo será responsabilidad de las fachadas de seguridad.
En términos de arquitectura del software, excepción y resultado anormal significan exactamente lo mismo. En términos de lenguajes de programación, ambos tipos de resultados pueden expresarse como un valor de retorno o como una excepción.
De [26], se extrae un ejemplo donde se muestra que los resultados anormales son más que errores del sistema, desastres, emergencias o eventos no deseados: Un componente `account` ofrece un método `withdrawmoney`. Los resultados normales son: `ok`, `nofunds`, `accountblocked`. Los problemas relacionados con la comunicación remota (expresados mediante una `RemoteException`) o, aquellos relacionados con las bases datos utilizadas (expresados por ejemplo con una `SQLException`) son resultados anormales: exponen detalles de implementación que deberían permanecer ocultos. Sin embargo, no es ésta realmente la cuestión que los hace anormales, sino que, el método `account` no es responsable del manejo de estas dificultades dependientes de la implementación. En esta misma aplicación bancaria, en la capa de acceso a las bases de datos, para un método `connectDatabase`, cuyos resultados normales son: `ok` y `nok`; un problema de accesibilidad a las mismas, a diferencia del método `account`, es un resultado normal; para este método sería anormal una falla en la conexión remota (indicado por `RemoteException`).
La decisión entre considerar un resultado normal o anormal, por tanto, debe hacerse localmente, bajo la responsabilidad del diseñador de la interface.
En [26] se listan los puntos más importantes que pueden guiar dicha decisión:

1. La interface de un método define todos los resultados normales que son independientes de la implementación. La decisión normal *vs.* anormal es binaria, no hay términos medios.
2. En casi todos los casos sólo existen dos posibles resultados, si el número de ellos es considerablemente mayor, debería cambiarse la interface. Todos los resultados normales deben enumerarse completamente como parte de las especificaciones.
3. Casi todos los resultados anormales son dependientes de la implementación, por tanto, básicamente no es posible especificarlos en la interface independiente de la implementación. Nadie sería capaz de enumerar todos los resultados anormales imaginables.
4. Los resultados anormales son raros, si ocurren frecuentemente es una falla de diseño y, deberían rediseñarse los métodos.
5. Los resultados anormales interrumpen el flujo normal de una aplicación y demandan un tratamiento especial; en algunos casos es posible continuar después de su ocurrencia, mientras que, en otros, la única opción es detener completamente el sistema.
6. Cualquier cosa que no pueda manejarse en tiempo de ejecución es siempre anormal. Todas las excepciones `RuntimeException` de Java son anormales desde el punto de la Máquina Virtual Java (JVM, por *Java Virtual Machine*): la JVM no tiene forma de resolver el problema.

2.1.4 Opciones para el manejo de excepciones

El diseño de un sistema de manejo de excepciones apropiado es particular de cada proyecto pero, las tareas que típicamente realizan este tipo estos sistemas son [26]:

1. Registrar las circunstancias y continuar el trabajo, sólo en caso que los resultados anormales no sean graves desde el punto de vista del invocante de un método.

2. Registrar las circunstancias para los encargados de mantenimiento del sistema, limitar los daños (cerrar conexiones, resetear transacciones, etc.) y, finalmente decidir qué parte de la aplicación debe terminarse (normalmente una sesión, en el peor caso, el sistema completo).
3. Esperar un tiempo o ejecutar un número específico y razonable de reintentos de una operación.
4. Reconfiguración, es decir, reemplazar componentes por otros redundantes.

Por tanto, luego de efectuar el manejo de excepciones apropiado, cada método tendrá sólo dos posibles resultados:

1. Resultado normal: Esto incluye también aquellos que son errores de aplicación. El invocante del método no se entera si actuó el sistema de manejo de excepciones, a lo sumo puede percibir un tiempo de respuesta mayor al normal.
2. Falla "segura" y definitiva: El método finalmente falla; todos los intentos realizados por los mecanismos de manejo de excepciones no tuvieron éxito y, se tomaron todas las medidas apropiadas para limitar daños. No tiene sentido realizar ninguna acción de reparación posterior; la única opción que tiene el invocante es abortar la operación.

2.2 Arquitectura del sistema de manejo de excepciones escogida

El problema de incorporar en forma sistemática el manejo de excepciones en el desarrollo de sistemas basados en componentes, hasta ahora no ha sido tratado adecuadamente, aún cuando se reconoce la importancia de esta necesidad [9], desde hace mucho tiempo.

2.2.1 Fachadas de seguridad

Siedersleben [25, 26] sugiere agrupar uno o más componentes en comunidades de riesgo (RC, por *risk communities*), cada una de ellas "envuelta" o "protegida" por una fachada de seguridad (SF, por *security facade*). Las SF, un caso especial del conocido patrón de diseño fachada (*facade pattern*) [12], centralizan el manejo de excepciones de los componentes de la RC a su cargo. Siedersleben [26] las considera como una versión moderna de la metáfora de "trampas" propuesta por Parnas y Würges [19]: se oculta el mecanismo de detección y se expone la interface de manejo de excepciones. Para Parnas y Würges [19] estas trampas pueden pensarse como paredes que protegen módulos de software, del daño que podrían causar "eventos no deseados". En [28] se indica que las SF son comparables a la idea de compartimentos de software propuesta, por Robillard y Murphy [23].

Todos los componentes de una RC comparten los mismos mecanismos de manejo de excepciones. Las RC deben diseñarse cuidadosamente y, tal vez, una de las tareas más difíciles cuando se decide sobre ellas, es determinar una lista de excepciones semánticamente coherentes que serán las únicas señaladas por las SF. Guerra *et.al.* [16] proponen el uso de una jerarquía de excepciones genéricas, para expresar en forma precisa la semántica de fallas anticipadas y no anticipadas de componentes y conectores; facilitando al mismo tiempo la evolución de las interfaces de excepciones. En [1] se mapea esta jerarquía de excepciones genéricas a otra, de clases de excepciones en la plataforma Java. Cabe destacar que la propuesta de las SF pueden complementarse perfectamente con la de Guerra *et.al.* [16], como así también con un conjunto de directrices generales presentadas en [25].

Existen dos formas posibles de invocar a un componente: pesimista u optimista, según haya o no precauciones contra posibles excepciones, respectivamente. Todas las llamadas internas dentro de la RC son optimistas; parafraseando a Siedersleben: los componentes de la comunidad "sobreviven juntos o mueren juntos". Por el contrario, todas las llamadas externas a la RC se reciben únicamente a través de la SF que la protege, el resultado de dicha invocación sólo puede ser normal o falla "segura" y definitiva.

Las SF pueden reemplazarse, dependiendo de las necesidades de los usuarios de las RC (por ejemplo: cambiar una para modo *batch*, por otra para modo interactivo), tarea normalmente a cargo de un administrador de composición (CM, por *Composition Manager*). Es posible utilizar SF con componentes existentes o, construirlos de tal modo que operen teniendo en cuenta dichas fachadas. Usualmente, las SF se comunican en cascada: las emergencias son interceptadas por la SF que protege al componente que falla; el resultado de una invocación (éxito o falla definitiva y "segura") se informa al componente invocante (a través de la SF de su comunidad), que es libre de considerar la falla definitiva como un resultado anormal o no. De tal forma, en cada etapa de la cascada un resultado anormal puede enmascararse o propagarse a la próxima SF; el último recurso es abortar la operación en el nivel más externo (en Java: probablemente, la clase que tiene el método `main()`).
Con el uso de SF se logra separar el manejo de resultados normales, de los anormales, lo cual trae aparejado tres ventajas principales [26]:

1. Preservar el principio de ocultamiento de la información.
2. Promover la reutilización de los componentes.
3. Simplificar el proceso de desarrollo de aplicaciones.

El lado oscuro: la imposibilidad de retomar la ejecución de un método después que ocurre una excepción. En principio, la única opción para solucionar esta desventaja, es invocar el mecanismo de manejo de la excepción, directamente en el código del componente, en la cláusula `catch` asociada al bloque `try`, que "protege" la invocación crítica. Según la experiencia del autor, la interrupción del flujo de control raramente produce problemas.

2.2.2 Expertos en diagnóstico y reparación

Las SF no tienen acceso a los datos privados de los componentes protegidos; éstos pueden proveer una interface para diagnóstico y reparación (interface de D&R), conocidas sólo por el CM y la SF. Estas interfaces son más útiles en el caso de componentes técnicos (por ejemplo en la capa de acceso a una base de datos) que, en aquellos propios de la aplicación. Las interfaces de D&R proveen servicios, tales como: información del estado del componente con fines de diagnóstico, *reset* (por ejemplo: *rollback* de un reloj y, otras tareas que normalmente no se ejecutan en una cláusula *finally* de Java), reintento de operaciones o, desactivación total de un componente.
Un ejemplo de interface D&R es [26]:

```
//ejemplo de interface de D&R para capa de acceso a una base de datos (BD)
public interface PersistenceDR{
     boolean ok(); //provee información del estado del componente, true es OK
     boolean reset();
     boolean reconnect(); //trata de reestablecer una conexión con la BD
     boolean resign(); //falla, el componente es desactivado
}
```

En caso de excepciones muy complejas, se las divide en categorías apropiadas y, a cada de una de ellas se le asigna un objeto, denominado por Siedersleben [25, 26]: "experto en D&R" (por ejemplo: experto en SQL, en RMI, etc.), el cual implementa las interfaces de D&R ofrecidas por uno o más componentes, de los que se hace responsable de monitorear.

3 IMPLEMENTACION DE FACHADAS DE SEGURIDAD EN JAVA

Las SF implementan las mismas interfaces que los componentes que protegen; es decir son componentes simétricos: exportan las mismas interfaces que importan y, se encargan de manejar todas las excepciones que puedan ocurrir en los componentes de la RC que envuelven. . Existe una

librería J2EE de terceros, *open source*: EL4J [11, 28] de la empresa ELCA (http://www.elca.ch/) que, a fines del 2005, implementa una versión de las SF basada en programación orientada a aspectos; al conocimiento del autor, no existen aplicaciones que usen esta implementación.

3.1 Implementación manual de una fachada de seguridad

Los pasos a seguir para la implementación "manual" de una SF que envuelva un componente *C*, son [26]:

1. Debe crearse una nueva clase para la *SF* que implemente un subconjunto apropiado de las interfaces exportadas por el componente *C* encapsulado
2. Dentro del código de dicha clase, para cada uno de los métodos implementados, debe invocarse a los originales del componente *C*; protegidos dentro de un bloque `try-catch`. Cada resultado anormal de *C* es enviado a la *SF*. En código Java se tendría algo similar a:

```
public class SecurityFacade implements R, S{
    private C x;
    public setC(C x){this.x=x;}
    public void foo(){//foo y bar son métodos de R y S
      try{
        x.foo();
      }
      //Manejo de excepciones con acceso opcional a x
      catch(ExceptionA a){//Manejo de excepción tipo A
      }
      catch(ExceptionB b){//Manejo de excepción tipo B
      }
      catch(RuntimeException t){//otros problemas}
    }
    public void bar(){
      try{
        x.bar();
      }
      catch(ExceptionA a){//...     }//sigue como en
foo()
    }
    //demás métodos de R y S
}
```

Este procedimiento "a mano", en el caso real de una RC formada por varios componentes, cada uno de los cuales implementan varias interfaces, no resulta muy eficiente. En [28] se provee un ejemplo muy simple de la implementación de una SF rudimentaria, en código Java, utilizando este método de implementación.

3.2 Implementación de una fachada de seguridad mediante *proxies* dinámicos

A partir de Java 1.3 el concepto de *proxies* dinámicos (DP) provee un mecanismo alternativo, potente aunque subutilizado [14], para implementar muchos de los patrones de diseño, incluyendo el facade. Este enfoque resulta mucho más conveniente y compacto que la opción de implementarlos manualmente [20, 24, 11, 3]. En otros lenguajes orientados a objetos existen conceptos similares al de los DP.

Se puede pensar en los DP como en "camaleones" que se mimetizan con el objeto "sobre" el cual están situados; en Java esto significa que exhiben las mismas interfaces que el objeto para el cual

actúan como "procuradores" (*proxies*). Más formalmente: Un DP es una clase (`java.lang.reflect.Proxy`) que implementa una lista de interfaces especificadas en tiempo de ejecución, cuando se crea dicho *proxy*. Las invocaciones de métodos sobre una instancia de la clase DP son enviadas, usando el mecanismo de reflexión de Java, a la única instancia del manejador de invocaciones que posee y que, constituye el corazón de la funcionalidad del DP. Este manejador implementa la interface `java.lang.reflect.InvocationHandler`, con un único método declarado: `public Object invoke(Object proxy, Method method, Object[] args)`, al que se le pasa como argumentos, la única instancia del *proxy*, un objeto `java.lang.reflect.Method` que identifica el método invocado y, un *array* de tipo `Object` conteniendo los argumentos del mismo. El manejador de invocaciones procesa el método invocado y, retorna a la instancia del *proxy* propietario del mismo, el valor correspondiente, el cual es enviado al objeto que en realidad realiza la operación. Este mecanismo permite interceptar invocaciones a métodos y "desviarlos" dinámicamente. En caso de implementación de SF, el método `invoke` estaría protegido dentro de un bloque `try`, en cuyas cláusulas *`catch`* asociadas, la SF podría realizar el "trabajo" de interceptar los resultados anormales de la RC que protege.

Los DP, siguiendo el principio general del diseño que indica que las interfaces se usan para definir tipos y, las clases para definir implementaciones, pueden implementar una o más interfaces pero no clases. Existen actualmente algunos trabajos que presentan estrategias que soportan DP para "clases" en Java.

Debido a que las clases *proxies* no tienen nombres accesibles, no pueden tener constructores; deben crearse mediante métodos factorías, los cuales se encargarán de invocar a: `public static Object newProxyInstance(ClassLoader loader, Class[]interfaces,InvocationHandler h)`, método que retorna una instancia de la clase *proxy* para las interfaces especificadas y que, redirige la invocación de métodos al manejador indicado en la lista de argumentos.

La reflexión en Java es una herramienta muy potente pero, su principal desventaja es el efecto en la *performance* [27]; este problema fue uno de los que más atención recibió cuando la empresa Sun desarrolló la JVM 1.4, logrando sustanciales mejoras en este aspecto, con respecto a la versión anterior de la JVM: a modo de ejemplo, en [27] se indica que construir una instancia de un `java.lang.Object`, invocando `newInstance( )` tarda 12 veces más que con `new Object()`, usando la versión 1.3 de la JVM, mientras que; usando la versión 1.4 de la JVM toma aproximadamente 2 veces más. Podría concluirse, con respecto al tema de la *performance*, que el enfoque de DP funciona bastante bien en caso de invocación de métodos que son "pesos livianos", mientras que, cuando los métodos invocados se vuelven más "pesados" (por ejemplo: invocación a métodos remotos, que usen serialización, realicen operaciones de I/O o busquen información en bases de datos), la sobrecarga que introduce el uso de la reflexión en Java se aproxima a cero. Esta discusión se vuelve irrelevante si las ventajas que pueden obtenerse del uso de DP, tienen un mayor impacto en la *performance*, que la del uso del mecanismo de reflexión, en sí mismo.

La implementación de SF, basada en DP, de una RC que contenga más de un objeto, tal como ha sido presentada, tiene un punto débil a resolver: todas las interfaces del *proxy* deben estar implementadas en una única clase, puesto que, `InvocationHandler` delega todas las invocaciones de los métodos, a la única instancia del objeto que está siendo *proxificado*. La solución: el uso de *mixins* [10, página 462]. El término *mixin* ha adquirido distintos significados a lo largo del tiempo, pero el concepto fundamental se refiere a la mezcla (*mixing*) de capacidades de múltiples clases con el fin de producir una clase resultante que represente a todos los tipos del *mixin*. Este tipo de labor suele realizarse en el último minuto, lo que hace que resulte bastante conveniente para ensamblar fácilmente unas clases con otras. Otras ventajas de los *mixins* que resultan de utilidad para implementar DP: fomentan la programación basada en interfaces y la reutilización de software. Con un DP, el tipo dinámico de la clase resultante, es igual a los tipos combinados que se hayan mezclado.

Eckel, con respecto al ejemplo de aplicación que presenta para graficar el tema de modelado de *mixins* con DP [10, páginas 458 y 462], opina que la solución obtenida no es tan elegante como la que se alcanzaría usando los *templates* de C++; en Java el mecanismo de reflexión hace que el *mixin* sea un tipo dinámico, lo cual obliga a realizar una operación de especialización (*downcasting*) sobre el tipo apropiado, antes de poder invocar cualquier método de alguno de los objetos "*proxificados*".
Finalmente, es importante remarcar que el `InvocationHandler` del *proxy* "mimetiza" las interfaces reales, también en lo que respecta a la interface de excepciones de los métodos declarados en ellas. Si el *handler* adopta [8] estrategia diferentes, con respecto al manejo de excepciones, el *proxy* se comportará en forma diferente a las clases reales que "representa"; ya no será más transparente y, se deberá añadir código de propósito especial, para manejar las diferencias.
Para ilustrar la técnica de implementación de una SF con DP, se añade a continuación un ejemplo adaptado de [8], en el cual se "protege" a modo de una SF, una instancia de una única clase (`TestImpl`) que implementa una interface (con lo cual no se usa la estrategia de *mixins*) :

```
//código propio de la aplicación
//clase de Excepción que lanza ping()
public class ApplicationException extends Exception
{
public ApplicationException(Throwable cause) {
               super(cause);

}
public ApplicationException(String msg) {
                       super(msg);
}
}
//interface que implementa TestImpl
public interface Test {

       public void ping() throws
ApplicationException;
}
//clase "protegida" por el proxy
public class TestImpl implements Test {
       public void ping() throws
ApplicationException {
               System.out.println("ping()");
       }
}
//clase principal (main())
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;
public class Principal {
  public static void main(String[] args) {
       try {
                       Test t = new TestImpl();
  // Crea una instancia de TestImpl
  // Crea InvokeHandler
  InvocationHandler handler = new
LoggingProxyHandler(t);
  // Crea Proxy (puede usarse método
  //factoría en el handler y separar
  //el código de la aplicación del código del proxy
  //hace un cast a Test
  (Test) Proxy.newProxyInstance(
       t.getClass().getClassLoader(),
               t.getClass().getInterfaces(),
                                       handler);
  // Test clase real
       t.ping();
  // Test clase Proxy
       proxy.ping();
       } catch (Exception e) {
               e.printStackTrace();
               }      }}
```

```
import java.lang.reflect.InvocationHandler;
import
java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
public class LoggingProxyHandler implements
InvocationHandler {
     protected Object delegate;
     public static final int RETRY_LIMIT = 3;
  public LoggingProxyHandler(Object delegate) {
       this.delegate = delegate;
  }
  public Object invoke(Object proxy, Method
method, Object[] args)
       throws Throwable {
       System.out.println(
                                "Calling method "
                                       + method
                                       + " at "
                                       +
System.currentTimeMillis());
       Object result =
invokeRemoteMethod(method, args);
       return result;
}
//hace 3 reintentos,
protected Object invokeRemoteMethod(Method
method, Object[] args)
       throws ApplicationException {
       int trys = 1;
       while (trys <= RETRY_LIMIT) {
       try {
            System.out.println("try(" + trys +
") to access (" + method + ")");
       return method.invoke(delegate, args);
       } catch (IllegalAccessException ie) {
       throw new ApplicationException(ie);
       } catch (InvocationTargetException e) {
              if (trys < RETRY_LIMIT) {
                   ++trys;
              } else {
              System.out.println("Reached
retry limit " + trys + " to access (" + method
+ ")");
              throw new
ApplicationException(e.getTargetException());
} //fi
       } //end catch
       } //end while
return null;}}
```

Cuando se ejecuta la aplicación se observa el siguiente resultado, donde se comprueba que efectivamente el *proxy* mimetiza a la clase que protege, hasta en la interface de excepciones:

```
ping()
Calling method public abstract void Test.ping() throws ApplicationException at
1218365590687
try(1) to access (public abstract void Test.ping() throws ApplicationException)
ping()
```

El ejemplo sirve para evidenciar claramente que, la parte del código de manejo de excepciones, queda totalmente separada del código propio de la aplicación, lo cual representa el espíritu con que fueron pensadas las SF.

4 CONCLUSIONES Y TRABAJO FUTURO

Aplicando el concepto de *proxies* dinámicos, se analizaron los detalles de la implementación de una simple, aunque muy interesante estrategia de manejo de excepciones, disponible en la literatura [25, 26]: las fachadas de seguridad, cuya descripción original carece de información pormenorizada. Las fachadas de seguridad constituyen un nuevo enfoque, que introduce una arquitectura y una serie de pautas que establecen un *framework* factible para el manejo de excepciones. Teniendo en mente este hecho y, con vista a aplicarlo a sistemas concurrentes/distribuidos complejos, el próximo paso será pensar en la ejecución del sistema completo, estructurada como un conjunto de acciones atómicas, en las cuales tomen parte los componentes integrantes del mismo, tal como se propone en [24, 6].

REFERENCIAS

[1] Aguilar Gayard, L., Rubira, C. M. F. y Guerra, P. A. de C., COSMOS*: a Component System Model for Software architectures. Reporte Técnico, Instituto de Computação, Universidade Estadual de Campinas, Febrero del 2008. Accedido por última vez: 06/08/2008 en: http://www.ic.unicamp.br/reltec-ftp/2008/Abstracts.html

[2] Anderson, T. y Lee, P. A., *Fault Tolerance: Principles and Practice*, Springer-Verlag, 2º edición, 1990.

[3] Baumann, A., "Fehler- und Ausnahmebehandlung mit Quasar.", Ph.D. diss., Technische Universität München, Institut für Informatik, 2005.

[4] Black, A. P., "Exception Handling: The Case Against." Ph.D. diss., University of Oxford, 1982.

[5] Buhr, P.A., y Mok, W. Y. R., Advanced Exception Handling Mechanisms, *IEEE trans. Software Eng.*, 26, pp. 820-836, 2000.

[6] Capozucca, A., Guelfi, N. y Pelliccione, P. "The Fault-Tolerant Insulin Pump Therapy". Capítulo del libro "Rigorous Engineering of Fault-Tolerant Systems", Co-editores: M. Butler, C. Jones, A. Romanovsky, E. Troubitsyna. LNCS 4157, Lecture Notes in Computer Sciences, Springer-Verlag, Berlin Heidelberg, pp. 59–79, 2006

[7] Cristian, F., Exception handling and software fault tolerance. *IEEE Transactions on Computers*, 31(6), pp. 531–540, 1982.

[8] D'Abreo, L., Java Dynamic Proxies: One Step from Aspect-Oriented Programming, 2004, Accedido por última vez: 06/08/2008 en: http://www.devx.com/Java/Article/21463/1954

[9] Dellarocas, C., Toward Exception Handling Infrastructures in Component-based Software. *International Workshop on Component-based Software Engineering*, Kyoto, Japon, 1998.

[10] Eckel, B., *Piensa en Java*. Pearson Prentice Hall, Madrid, 4º edición, 2007.

[11] EL4J.Documentación de referencia. Accedido por última vez: 06/08/2008 en: http://el4j.sourceforge.net/el4j/ReferenceDoc.pdf

[12] Gamma, E., Helm, R., Johnson, R. y Vlissides, J., *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 1995

[13] Garcia, A. F., Rubira, C. M. F., Romanovsky, A. y Xu, J., A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. *Journal of Systems and Software*, Elsevier, 59(2), pp. 197-222, 2001.

[14] Goetz, B., Java theory and practice: Decorating with dynamic proxies, Accedido por última vez: 06/08/2008 en: http://www.ibm.com/developerworks/java/library/j-jtp08305.html

[15] Goodenough, J. B., Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12), pp. 683–696, 1975.

[16] Guerra, P. A. de C., Filho, F. C., Pagano, V. A. y Rubira, C. M. F., Structuring exception handling for dependable component-based software systems. *EUROMICRO*, pp. 575–582, 2004.

[17] Lippert, M. y Lopes, C. V., A study on exception detection and handling using aspect-oriented programming., *22nd International Conference on Software Engineering (ICSE'2000)*, Limerick, Ireland, pp. 418–427, 2000.

[18] Miller, R. y Tripathi, A., Issues with exception handling in object-oriented systems. *ECOOP'97 — Object Oriented Programming*, número 121 en Lecture Notes in Computer Science, pp. 85–103. Springer Berlin / Heidelberg, 1997.

[19] Parnas, D. L. y Würges, H., Response to undesired events in software systems. *2nd ICSE*, San Francisco, USA, pp. 437–446, 1976.

[20] Popov, P. Strigini, L., Riddle, S. y Romanovsky, A., On Systematic Design of Protectors for Employing OTS Items, *27th Euromicro Conference, Workshop on Component-Based Software Engineering*, Warsaw, Poland, pp. 22-29, 2001.

[21] Reimer D.,y Srinivasan, H., Analyzing exception usage in large Java applications. *ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms*, pp.10-19, 2003.

[22] Renzel, K., "Error Handling for Business Information Systems, Version 1.1", ARCUS sd&m Muenchen, 2003. Accedido por última vez: 06/08/2008 en: http://www.eso.org/~almamgr/AlmaAcs/OnlineDocs/ARCUSErrorHandling.pdf

[23] Robillard, M. P. y Murphy, G. C., Designing robust java programs with exceptions. *SIGSOFT Softw. Eng. Notes*, 25(6), pp. 2–10, 2000.

[24] Romanovsky, A. B., Exception handling in component-based system development. *COMPSAC '01: 25th International Computer Software and Applications Conference on Invigorating Software Development*, Washington, DC, USA, IEEE Computer Society, pp. 580-586, 2001.

[25] Siedersleben, J., *Errors and Exceptions—Rights and Obligations*, Lecture Notes in Computer Science, vol. 4119, pp. 275–287. Springer Berlin / Heidelberg, 2006.

[26] Siedersleben, J., Errors and Exceptions - Rights and Responsibilities., *Workshop on Exception Handling in Object Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms*, Darmstadt, Germany, pp. 2-9, 2003.

[27] Sosnoski, D., Java programming dynamics, Part 2: Introducing reflection. Accedido por última vez: 06/08/2008 en: http://www-128.ibm.com/developerworks/library/j-dyn0603/

[28] Tellefsen, C., "An Examination of Issues with Exception Handling Mechanisms." Master's thesis, Norwegian University of Science and Technology, 2007.

[29] Wirfs-Brock, R., Toward Exception-Handling Best Practices and Patterns., *Software IEEE*, vol. 23(5), pp. 11-13, 2006.