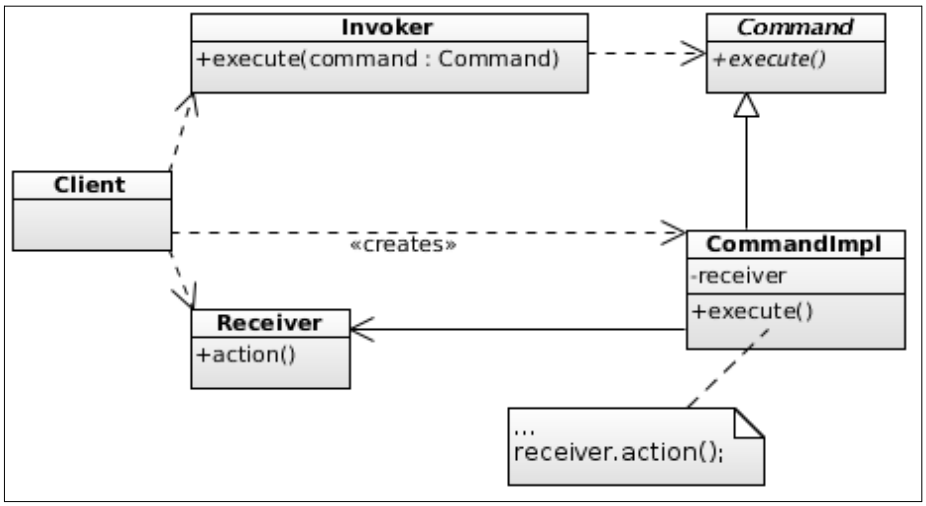
Command

Diagrama de clases e interfaces



Intención

El patrón Command, también conocido como Action o Transaction, permite que un objeto A sea requerido para ejecutar la operación b() de un objeto B cuando no es posible o no nos interesa que A y B se conozcan.

Realmente, A no solicita la operación a B, pues A ni conoce a B ni las operaciones que éste ofrece. Lo que sucede es que A invoca a un tercero, C, a través de una operación c(). C sí que es conocedor de B y de las operaciones que éste proporciona. La pregunta es ¿Por qué A puede llamar a c() y en cambio no puede llamar a b()? Respuesta: Porque A es una clase perteneciente a un framework y B es una clase creada por el usuario del framework. B también es una clase creada por el usuario, pero que implementa una interfaz que permite ejecutar operaciones.

El flujo de ejecución queda como sigue:

- Cuando A es requerido invoca la operación c() sobre C.
- La implementación de c() consiste en llamar a b() sobre B, que suele ser un atributo de C.

Por tanto, A consigue indirectamente ejecutar la operación b() de B.

Lo que aquí sucede es que el objeto C ha sido enviado como parámetro en la solicitud realizada a A. Se dice que C ha sido parametrizado, ya que se le ha dado el tratamiento de parámetro, como si se fuera un número, un String, etc. Gracias a esta parametrización de C -que es el Command-, es posible que la clase que recibe el parámetro, A, sea capaz, indirecta e inconscientemente, de ejecutar un servicio de un objeto arbitrario (B) que pueda estar encapsulado como atributo del objeto recibido por parámetro (C). Este hecho proporciona un desacoplamiento que es vital en ciertas situaciones en las que se necesita mucha flexibilidad al asociar objetos.

Adicionalmente, es posible crear listas de objetos Command, para implementar funcionalidades como hacer/deshacer/rehacer y macro comandos (lotes de comandos)

NOTA: Antes de proseguir con este documento es necesario haber leído el documento "16a-Functors".

Motivación

Los *frameworks* (aplicaciones marco) se desarrollan para servir de infraestructura en la elaboración de aplicaciones finales (reserva de vuelos, gestión comercial, contabilidad, etc). De esta manera los programadores de aplicaciones finales no tienen que comenzar de cero una nueva aplicación, sino extender el *framework* según las necesidades específicas de la aplicación a desarrollar. Con esto se consigue aumentar significativamente la productividad del desarrollo de aplicaciones finales.

Casi todos los *frameworks* utilizan el patrón Command para llamar a objetos que son específicos de la aplicación final (creada por un programador final, esto es, por un usuario del *framework*) sin saber nada sobre estos objetos o sobre las funciones que realizan.

Por ejemplo, el *framework* Swing de Java nos ofrece todo un conjunto de componentes gráficos (widgets) para crear aplicaciones de escritorio con una interfaz rica. Tomemos por caso los controles que permiten añadir a una interfaz gráfica un menú, una barra de herramientas con iconos o botones para formularios.

Para nuestro discurso, asumamos el uso de un menú. Así, cuando el usuario selecciona una opción del menú, el *framework* tiene que realizar una llamada para que se lleve a cabo alguna acción o comportamiento, y aquí viene lo importante: Swing no

implementa ni puede implementar esa acción porque no es una responsabilidad del *framework*. Únicamente la aplicación específica que esté creando el usuario -programador- del *framework* sabe lo que hay que hacer cuando se selecciona esa opción de menú. Queda claro entonces que la acción a realizar es una clase creada por el usuario.

Por tanto, ¿cómo puede el *framework* realizar peticiones a objetos para que lleven a cabo una acción cuando ni tan sólo sabe de su existencia o de las operaciones que ofrecen? La respuesta es obvia: mediante el patrón de diseño Command.

La clave está en el método execute() que define la interfaz Command. Esta interfaz permite establecer un contrato entre el programador del *framework* y el programador de la aplicación final basada en el *framework*. A saber:

- El *framework* (menú, botón, etc) se compromete a invocar al método execute() de un objeto, siempre que éste implemente la interfaz Command.
- El programador de la aplicación final se compromete a asociar un objeto Command determinado a una opción de menú, botón, etc. Este objeto incluirá en su implementación del método execute() la llamada al receptor real de la petición solicitada por el usuario al pulsar sobre el menú o el botón. Por ejemplo, el método execute() del objeto Command podría hacer lo siguiente para obtener un listado de un catálogo de productos: catalogo.mostrar()

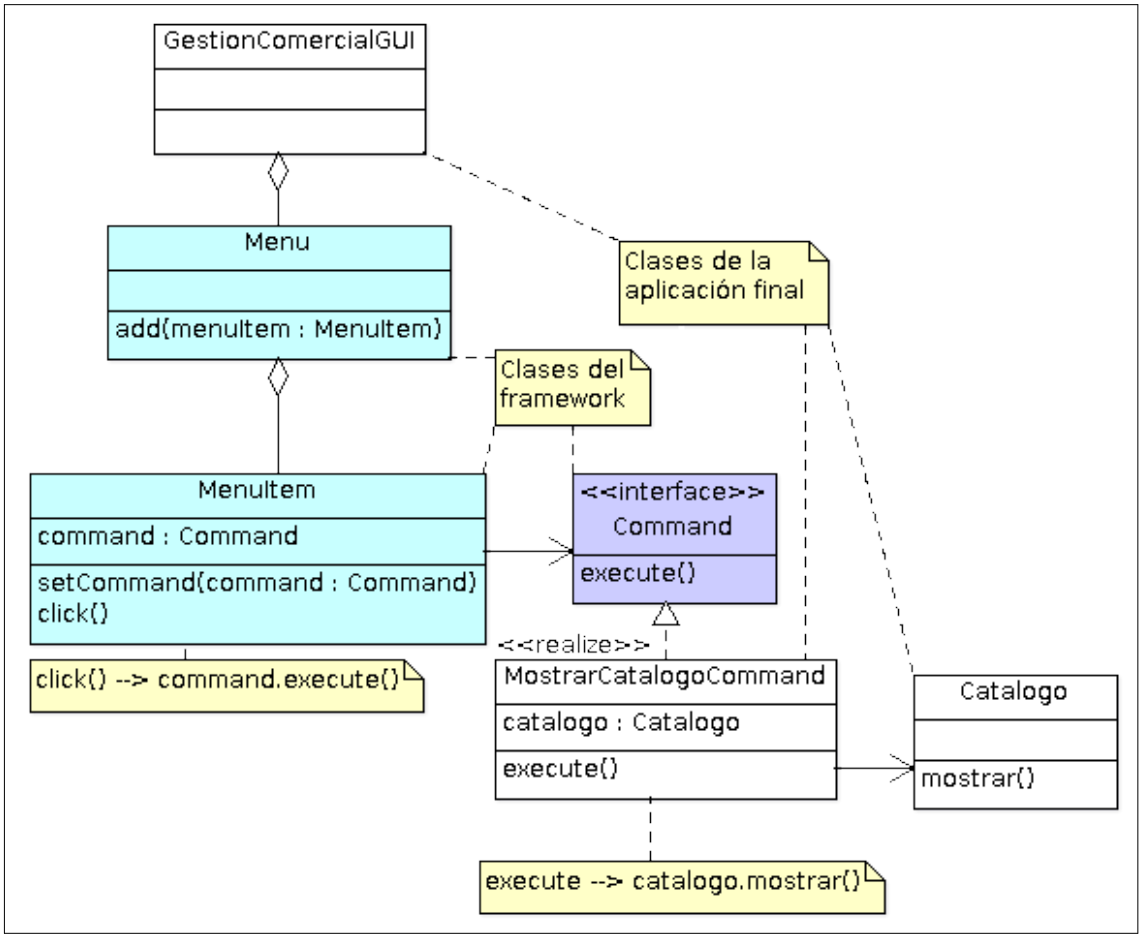
De esta manera se consigue que una opción de menú o un botón muestren el catálogo, cuando el *framework* no tiene ni idea de la existencia del mismo.

El *framework* sólo conoce a la interfaz Command, por lo que puede recibir cualquier objeto que la implemente.

Es el programador de la aplicación final quien diseña el objeto Command (una subclase de Command) con la información necesaria para ejecutar el catálogo.

Por tanto, el objeto Catalogo y el objeto concreto de Command sí que quedan altamente acoplados, lo cual es normal dado que ambos objetos pertenecen a la aplicación final del usuario y es necesario que el comando pueda llamar a las operaciones que se requieran de Catalogo. De hecho, el objeto concreto Command tendrá como atributo a Catalogo, que es el objeto encargo de llevar a cabo la petición (mostrar el catálogo de productos).

La imagen siguiente ilustra lo explicado:



De la imagen anterior, vemos que la interfaz Command es la clave para poder ejecutar operaciones. Este artefacto lo proporciona el framework, como sucede con la interfaz javax.swing.Action que veremos más adelante, o en el framework MVC Struts, la clases org.apache.struts.Action.

La cuestión de fondo es que se consigue desacoplar los componentes que realizan peticiones (clases del *framework*) de los componentes que llevan a cabo esas peticiones (clases del usuario). Gracias a este patrón los componentes del menú son totalmente reutilizables en aplicaciones finales.

¿El patrón Command sólo tiene sentido si se usan frameworks?

A lo mejor un programador nunca llega a necesitar implementar el patrón Command, eso sí, asumiendo que tal programador no se dedique a los *frameworks.*

Hasta el momento, se ha enfocado la justificación del uso del patrón Command confrontando la diferente naturaleza de una aplicación tipo *framework* con una

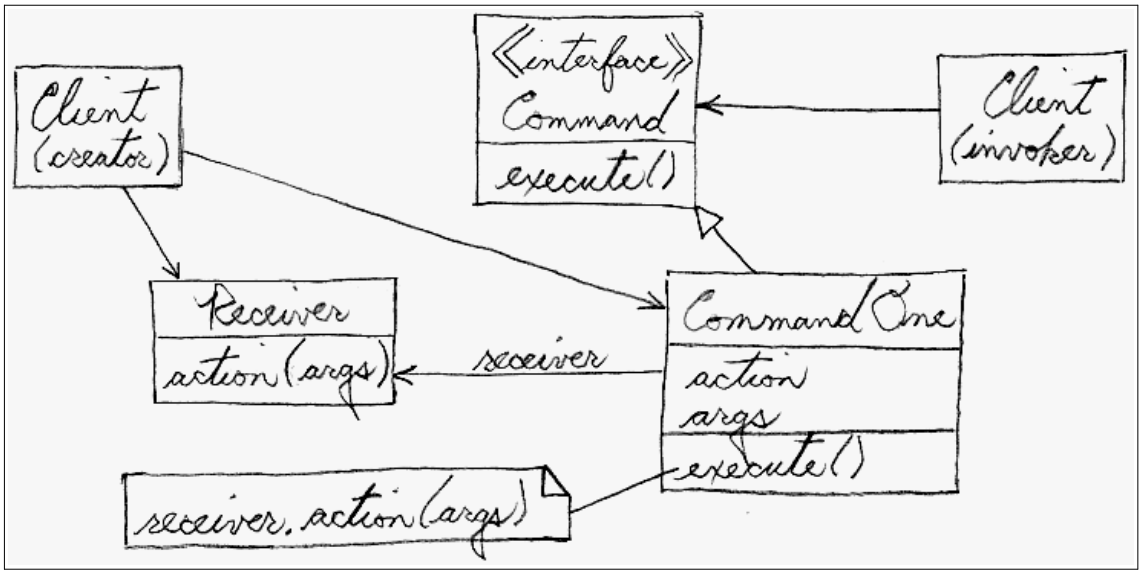
aplicación final basada en ese *framework*. No obstante, podemos emplear este patrón directamente en una aplicación que no utilice *framework* alguno.

Por lo general, cuando estemos ideando alguna clase cuya responsabilidad consista en recibir (y a veces en centralizar) peticiones y realizar algún tipo de trabajo con ellas o, lo más común, delegarlas a otras clases para que éstas las atiendan, nos encontraremos con que tal clase tendrá que distinguir una petición de otra para actuar de una u otra manera o para delegarla a una u otra clase.

Si la lógica utilizada para distinguir peticiones consiste en una serie de estructuras *if-elseif*, sucederá que cuando se añada una nueva funcionalidad a la aplicación, el código existente tendrá que modificarse para contemplar este nuevo caso, violando así el principio de diseño básico de abierto-cerrado (los módulos de software deben diseñarse de manera que pueda extenderse su funcionalidad sin tener que modificar el código existente). El patrón Command salva este inconveniente.

Implementación

El funcionamiento del patrón es el mostrado en el diagrama de clases siguiente:



Consideraciones:

- El cliente que crea un Command (creator) no es el mismo que lo ejecuta (invoker), lo que ofrece mucha flexibilidad en cuanto a la planificación temporal y secuenciación de los comandos.
- Un cliente (Invoker) de un objeto comando trata a éste como una caja negra, sencillamente llama cmd.execute() y consigue que ese ejecute receiver.action(args).
- Las subclases de Command guardan como atributos tanto el Receiver asociado como los argumentos necesarios para invocar a su método action.

Las clases participantes en el patrón son las siguientes:

Command: Interfaz perteneciente al framework que define el método execute() para la ejecución de comandos. Opcionalmente, se puede definir el método unExecute() si se quiere proporcionar la funcionalidad de deshacer (undo).

Receiver: Clase definida por el usuario que sabe cómo ejecutar una petición. Cualquier clase puede ser receptora (clase Catalogo, en el ejemplo anterior).

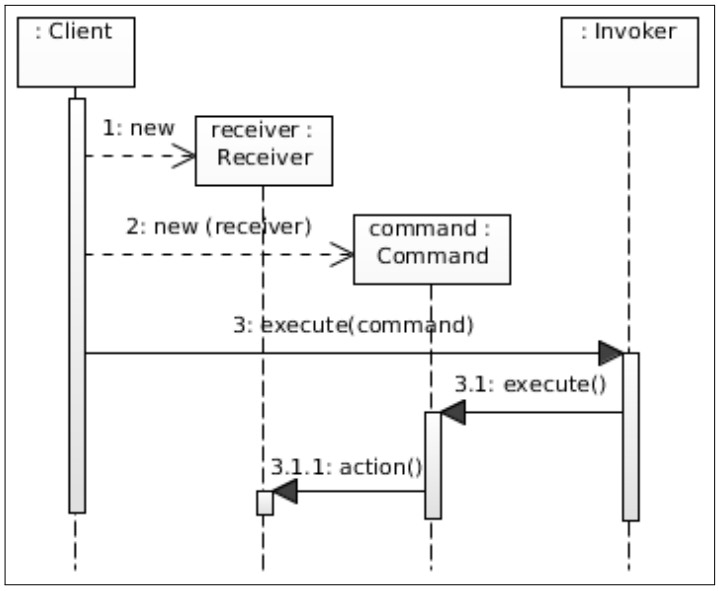
Invoker: Clase del framework que ante una petición del usuario invoca el método execute() sobre el objeto Command (realmente al objeto ConcreteCommand) que tiene como atributo (en el ejemplo anterior es la clase MenuItem).

ConcreteCommand: Clase definida por el usuario que implementa la interfaz Command y por tanto proporciona una implementación para el método execute(). Representa un comando en particular y es el enlace entre el *framework* y la aplicación final (en el ejemplo anterior es la clase MostrarCatalogoCommand). El Invoker (MenuItem) la utiliza para invocar a su método execute() y ella utiliza al Receiver (Catalogo) para que se lleve a cabo la petición. De ser definido en Command el método unExecute(), ConcreteCommand debe proporcionar una implementación para tal método.

Client: Crea un ConcreteCommand (MostrarCatalogoCommand) y un Receiver (Catalogo) y los relaciona. Crea un Invoker (MenuItem) y lo relaciona con su ConcreteCommand.

Hay que tener en cuenta que según nuestros requerimientos, puede ser que un objeto Command no delegue la responsabilidad de realizar una tarea en un objeto Receiver, por lo que en nuestro diseño no aparecería la clase Receiver, o al menos no aparecería relacionada con el objeto Command. Por tanto, podemos hablar de objetos Command “inteligentes” y objetos Command “simples”. Los primeros no delegan en un objeto Receiver, mientras que los segundos sí.

Diagrama de secuencias del patrón:



Aplicabilidad y Ejemplos

El patrón Command ofrece muchas aplicaciones útiles:

- La más importante: desacopla una clase responsable de ejecutar una solicitud (Invoker) de la clase que realmente lleva a cabo el trabajo (Receiver).
- Posibilita la ejecución de operaciones en otro entorno (en otro subproceso o nodo) o en otro momento.
- Permite deshacer operaciones que ya han sido ejecutadas (undo).
- Es posible tener un historial de los cambios realizados, lo que permite volverlos a aplicar si es necesario. Esto es muy conveniente si, por ejemplo, se debe realizar acciones repetitivas, o para volver a hacer una secuencia de pasos,

como sería el caso de la recuperación de una base de datos con inconsistencias a partir de un archivo de log.

- Estructurar un sistema mediante operaciones de alto nivel. Este sería el caso de tener una operación por cada funcionalidad presente en un caso de uso. Por ejemplo: Ver catálogo, Buscar producto, Añadir producto a la cesta, Facturar,...

Pasemos a ver algunos ejemplos.

Ejemplo 1 – Deshacer operaciones. Macro-comandos. Historial de operaciones

Supongamos que nos han pedido crear un conjunto de clases a modo de *framework* con el propósito siguiente:

- Tienen que recibir peticiones de ejecución de servicios relativos a objetos que sólo se pueden conocer en tiempo de ejecución.
- Se debe mantener un historial de las operaciones ejecutadas. Esto ha de facilitar la ejecución de todas las operaciones de nuevo, cuando así sea requerido.
- Ha de ser posible definir comandos compuestos de otros comandos, esto es, paquetes de comandos según interese.
- Es necesario que se puedan deshacer (undo) las operaciones realizadas hasta un determinado punto.

El patrón Command es quien nos puede ayudar a cumplir todos estos requisitos.

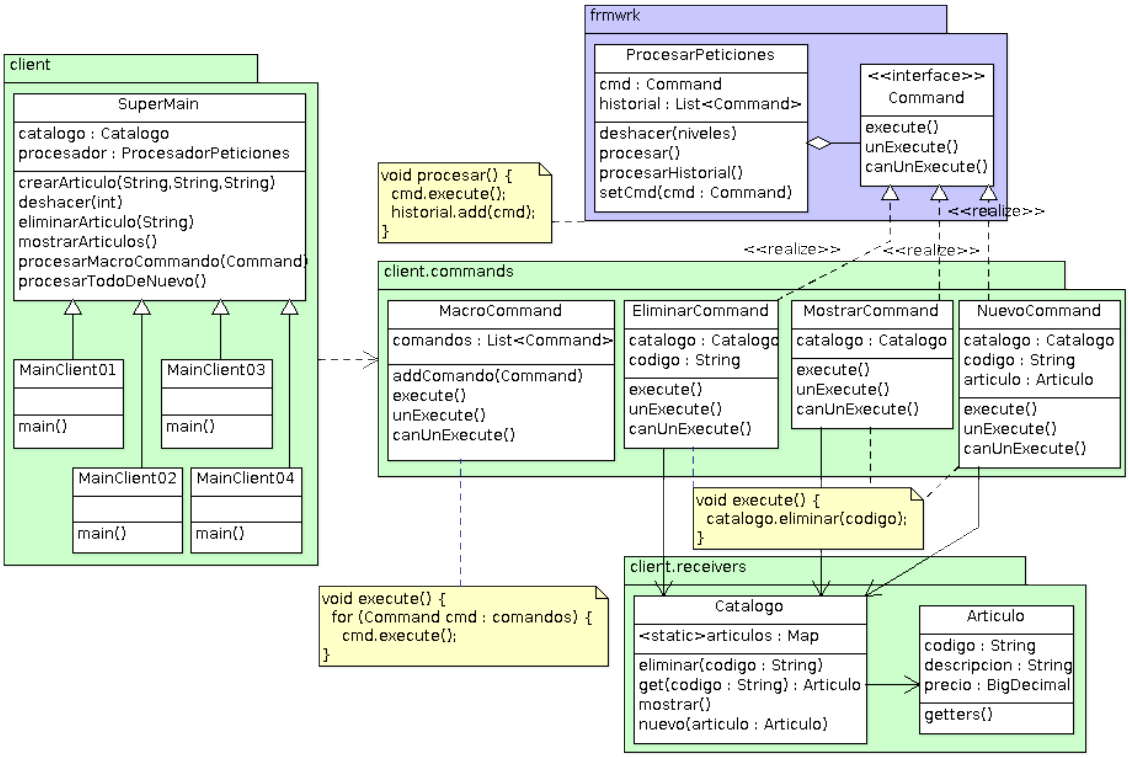
Nos dicen además, que la primera aplicación que se va a crear utilizando nuestro *framework* de procesamiento de comandos trata sobre un sencillo sistema de catalogo de artículos con los siguientes requerimientos:

- Se tienen que poder añadir, eliminar y listar artículos al catálogo (sólo estas operaciones, para no complicar más de lo necesario el ejemplo).
- Se tienen que poder utilizar las facilidades descritas anteriormente (deshacer operaciones, macro-comandos y repetir la ejecución de todas las operaciones).
- Para el ejemplo no utilizaremos base de datos alguna. El Receiver (el catálogo) se encarga de la persistencia y de la materialización de los datos mediante un

Map estático que se inicializa con algunos valores predeterminados al cargarse la aplicación.

Esta funcionalidad será de utilidad para el programa cliente que usará nuestro *framework* de procesamiento de peticiones.

A continuación se muestra el diagrama de clases para el sistema, donde podemos ver interactuando a las clases de la aplicación final con las clases que conforman el *framework*:



Consideraciones para el diseño de la aplicación:

1. Necesitamos definir una interfaz Command con los tres métodos siguientes:

- execute(): Las subclases de Command implementarán este método con la tarea que les corresponda. Normalmente, se tratará de almacenar en atributos los valores necesarios para poder invocar la operación en un Receiver. Por ejemplo, para el caso de añadir un artículo al catálogo, la subclase de Command pertinente recibiría por parámetros el catálogo (que es el Receiver) y el artículo; a continuación haría algo del estilo: *catalogo.nuevo(articulo)* -asumiendo que nuevo() es el método de la clase Catalogo que lleva a cabo la inserción de un nuevo artículo.

- unExecute(): Las subclases de Command implementarán este método para deshacer (undo) la tarea llevada a cabo en el método execute(). No todos los comandos tendrán la posibilidad de deshacer su ejecución, por ejemplo, la orden de listar los artículos del catálogo no se puede revertir; por tanto, este método debe lanza la excepción OperationNotSupportedException.
- CanUnExecute(): Indica si el comando puede o no deshacerse. A veces, un comando que en determinadas condiciones sí puede deshacerse, no lo puede hacer en otras. Por ejemplo, si ha fallado la eliminación de un artículo porque no ha encontrado en la base de datos (Map en nuestro caso), entonces la operación no se podrá revertir, pues no habrá nada que añadir si nada se eliminó.

Veamos el código para la interfaz Command:

Command.java

```
package comportamiento.command.procesar_peticiones.frmwrk;


import javax.naming.OperationNotSupportedException;
/*
 * Command
 * Clase del framework
 */
public interface Command {
	public void execute();
	public void unExecute() throws OperationNotSupportedException;
	public boolean canUnExecute();
}
```

2. Necesitamos una clase que reciba todas las peticiones de manera centralizada y que se encargue de que se lleven a cabo, utilizando para ello su objeto Command asociado. Esta clase adopta el rol de Invoker en el patrón Command.

Las peticiones podrán ser:

- Ejecución de un comando, que es el caso habitual.
- Deshacer el efecto producido por un comando ejecutado o los últimos N comandos ejecutados.
- Volver a ejecutar de nuevo todos los comandos.

Para las dos últimas peticiones, a modo de historial, necesitamos una estructura de datos tipo lista que permita almacenar cada comando ejecutado. De esta manera:

- Deshacer un comando o N comandos implicará recorrer la lista en orden inverso (desde el último comando) e invocar el método unExecute() de cada comando.
- Volver a ejecutar todo el historial será posible sencillamente recorriendo -en el orden natural de inserción- la lista de comandos ejecutados.

Por otro lado, parece que esta clase -Invoker- no debe preocuparse de los macro-comandos, ya que les tiene que dar el mismo tratamiento que a los comandos simples: ejecutarlos, deshacerlos, etc. Luego el diseño de un macro-comando, igual que para un comando simples, es una responsabilidad del usuario/programador del *framework*.

Veamos el código para esta clase, que puede tener por nombre ProcesadorPeticiones:

ProcesadorPeticiones.java

```
package comportamiento.command.procesar_peticiones.frmwrk;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import javax.naming.OperationNotSupportedException;
/*
 * Invoker
 * Clase del framework
 */
public class ProcesadorPeticiones {
```

```
	private List<Command> historial = new ArrayList<Command>();
	private Command cmd;

	public void procesar() {
		cmd.execute();
		historial.add(cmd);
	}

	public void setCmd(Command cmd) {
		this.cmd = cmd;
	}

	public void procesarHistorial()  {

System.out.println("**************************************************
********");
		System.out.println("EJECUTANDO HISTORIAL...");

System.out.println("**************************************************
********");
		for (Command cmd : historial) {
			System.out.println(">>>Ejecutando comando: " +
cmd.getClass().getSimpleName());
			cmd.execute();
		}
	}

	public void deshacer() {
		deshacer(1);
	}

	public void deshacer(int niveles) {
```

```
            if (niveles < 1 || niveles > 10) {
                  throw new RuntimeException("El numero de niveles
tiene que estar entre 1 y 10.");
            }
System.out.println("**************************************************
********");
            System.out.println("DESHACIENDO LAS ULTIMAS (" + niveles +
") OPERACIONES...");


System.out.println("**************************************************
********");
            if (!historial.isEmpty()) {
                  int des_de = historial.size() - niveles;
                  int hasta = historial.size();

                  List<Command> listaDeComandosBorrable =
                              historial.subList(des_de , hasta);

                  Collections.reverse(listaDeComandosBorrable);

                  for (Command cmd : listaDeComandosBorrable ) {
                        /*
                         * Un comando puede que no pueda deshacerse de
manera circunstancial,
                         * por ejemplo, se intento crear un nuevo
articulo pero ya existia,
                         * por lo que no hay nada que deshacer.
                         */
                        if (cmd.canUnExecute()) {
                              try {
                                    cmd.unExecute();
                              } catch (OperationNotSupportedException
e) {
```

```
                                    /*
                                     * Si se lanza esta excepcion es
porque se ha intentado ejecutar un comando
                                     * cuya naturaleza le impide
deshacerse, como sería el caso de MostrarCmd, ya
                                     * que si ya se ha mostrado algo
¿cómo los desmostramos?
                                     */
                                    e.printStackTrace();
                                }
                            } else {
                                System.out.println("El comando " +
cmd.getClass().getSimpleName() + " no se puede revertir.");

                            }
                        }
                    } else {
                        System.out.println("Nada para deshacer. El historial
esta vacio.");
                    }
                }
            }
```

Ahora es el turno de las clases que componen la aplicación final, basada en el framework. Comenzamos por ver la clase Articulo y la clase Catalogo. La clase Articulo es un *bean* al uso, que como su nombre indica encapsula la abstracción de lo que representa un artículo del catálogo.

Articulo.java

```
package comportamiento.command.procesar_peticiones.client.receivers;
import java.math.BigDecimal;


/*
 * Clase creada por el usuario del framework
```

```
 */
public class Articulo {


	private String codigo;

	private String descripcion;

	private BigDecimal precio;


	public Articulo(String codigo, String descripcion, BigDecimal
precio) {
		this.codigo = codigo;

		this.descripcion = descripcion;

		this.precio = precio;

	}


	public String getCodigo() {   return codigo; }

	public String getDescripcion() { return descripcion; }

	public BigDecimal getPrecio() {    return precio; }


	@Override

	public String toString() {

		return "Articulo [codigo=" + codigo + ", descripcion=" +
descripcion

				+ ", precio=" + precio + "]";

	}

}
```

La clase Catalogo es un poco más interesante que Articulo. Catalogo adopta el papel de Receiver dentro del patrón Command. Podemos tener el número de receptores que necesitemos, pero en nuestro caso nos basta con Catalogo. Un Receiver -receptor- puede ser cualquier clase, lo que la caracteriza como Receiver es el hecho de que contiene la funcionalidad necesaria para llevar a cabo la petición del usuario solicitada al Invoker.

Otras consideraciones:

- Para nuestro ejemplo, esta clase utilizará un Map para almacenar los artículos del catálogo. Concretamente utilizaremos un LinkedHashMap, ya que, sin ser algo crítico, nos interesa poder recuperar los artículos en el mismo orden en que fueron insertados.
- El Map se declara como estático (por lo cualquier instancia verá siempre los mismos artículos) y se inicializa con dos artículos de prueba para partir de un catálogo mínimo. La clase ofrece servicios para listar, insertar y eliminar artículos al Map. También se permite recuperar un artículo a partir de su código.
- El listado de los artículos se lleva a cabo en pantalla, ya que no se considera de interés devolverlos en una lista o similar.
- El Map se oculta totalmente al resto de clases, por lo que sería fácil sustituirlo por una base de datos.

Veamos el código:

Catalogo.java

```
package comportamiento.command.procesar_peticiones.client.receivers;


import java.math.BigDecimal;
import java.util.LinkedHashMap;
import java.util.Map;
/** Receiver **/
/*
 * Clase creada por el usuario del framework
 */
public class Catalogo {
	private static Map<String, Articulo> catalogo = 
inicializarCatalogo();


	private static Map<String, Articulo> inicializarCatalogo() {
```

```
            Map<String, Articulo> catalogo = new LinkedHashMap<String,
Articulo>();

            catalogo.put("1000", new Articulo("1000", "Televisor plano
X100", new BigDecimal("540.50")));

            catalogo.put("1001", new Articulo("1001", "Televisor plano
X600", new BigDecimal("920.60")));

            return catalogo;

      }


      public void mostrar() {

            System.out.println("========================");

            System.out.println("Mostrando el catalogo...");

            System.out.println("========================");


            if (!catalogo.isEmpty()) {

                  for (Articulo articulo : catalogo.values()) {

                        System.out.println(articulo);

                  }

            } else {

                  System.out.println("El catalogo esta vacio.");

            }

      }

      public void nuevo(Articulo articulo) {

            catalogo.put(articulo.getCodigo(), articulo);

      }

      public void eliminarArticulo(String codigo) {

            catalogo.remove(codigo);

      }

      public Articulo getArticulo(String codigo) {

            return catalogo.get(codigo);

      }

}
```

Siguiendo con las clases de la aplicación final, veamos ahora el conjunto de comandos definidos por el usuario del *framework*. Estas clases adoptan el papel ConcreteCommand en el patrón Command.

La responsabilidad de este tipo de clases es crucial, ya que actúan de enlace entre el framework que recibe (y solicita) peticiones y las clases de usuario (Receivers) que las llevan a cabo.

En particular hacen posible que el Invoker pueda ejecutar servicios de clases que desconoce (por supuesto, el Invoker también desconoce a los servicios de estas clases).

Lógicamente, para que esto sea posible:

- Un comando concreto debe implementar la interfaz Command, en otro caso el Invoker no la podría utilizar. Por tanto, debe proporcionar la implementación necesaria para cada uno de sus métodos.
- Un comando concreto debe poder inyectarse de alguna manera en el invoker. Normalmente como argumento en el constructor o en un método *setter* del Invoker. Para ilustrar esto, recuperemos un fragmento de código del Invoker donde el comando se inyecta vía método setter:

```
public class ProcesadorPeticiones { // Invoker
	...
	private Command cmd;
	public void procesar() {
		cmd.execute();
		historial.add(cmd);
	}
	public void setCmd(Command cmd) {
		this.cmd = cmd;
	}
	...
```

- Otra condición indispensable para un comando concreto es que tiene que conocer de primera mano al Receiver cuyo servicio ejecutará. Habitualmente, sostendrá una referencia hacia el Receiver, así como a cualquier argumento que éste pudiera necesitar para su correcta ejecución. El Receiver (y los argumento de éste) se inyectan en el comando, por lo general, vía constructor o método setter.

  Por ejemplo, un comando para crear un nuevo artículo dentro del catálogo tendrá un atributo de tipo Catalogo y otro de tipo Artículo, siendo el Catalogo el Receiver y el Articulo el argumento para el servicio nuevo() ofrecido por Catalogo. De esta manera, cuando se lo pidiera el Invoker, el comando haría algo del estilo: catalogo.nuevo(articulo). Esto se ve más claro en el código que sigue.

Comenzamos por el comando que permite listar los artículos. Notad que:

- El comando tiene un atributo de tipo Catalogo para poder así invocar en su método execute() al servicio mostrar() ofrecido por Catalogo.
- El método unExecute() lanza una excepción, ya que no es posible deshacer un impresión de artículos en la consola.

MostrarCommand.java

```
package comportamiento.command.procesar_peticiones.client.comandos;


import javax.naming.OperationNotSupportedException;
import
comportamiento.command.procesar_peticiones.client.receivers.Catalogo;
import comportamiento.command.procesar_peticiones.frmwrk.Command;


/*
 * Clase creada por el usuario del framework
 */
public class MostrarCommand implements Command {


	private Catalogo catalogo;
```

```
	private String msg = "El comando " +
this.getClass().getSimpleName() + " no se puede revertir.";


	public MostrarCommand(Catalogo catalogo) {
		this.catalogo = catalogo;
	}


	@Override
	public void execute() {
		catalogo.mostrar();
	}


	@Override
	public void unExecute() throws OperationNotSupportedException {
		throw new OperationNotSupportedException(msg);
	}


	@Override
	public boolean canUnExecute() {
		return false;
	}
}
```

Pasemos a ver el código que añade nuevos artículos al catálogo. En este caso la implementación en unExecute() se encarga de eliminar el artículo insertado previamente por el método execute(). Notad que para esto sea posible, el objeto que se añadió debe seguir referenciado por el comando, lo cual se consigue mediante el atributo 'codigo' de tipo String, que fue inicializado en este caso en el método execute():

NuevoCommand.java

```
package comportamiento.command.procesar_peticiones.client.comandos;


import javax.naming.OperationNotSupportedException;

import
comportamiento.command.procesar_peticiones.client.receivers.Articulo;

import
comportamiento.command.procesar_peticiones.client.receivers.Catalogo;

import comportamiento.command.procesar_peticiones.frmwrk.Command;

/*

 * Clase creada por el usuario del framework

 */

public class NuevoCommand implements Command {

	private Catalogo catalogo;

	private Articulo articulo;

	private String codigo;

	private boolean canUnExecute = true;


	public NuevoCommand(Catalogo catalogo, Articulo articulo) {

		this.catalogo = catalogo;

		this.articulo = articulo;

	}


	@Override

	public void execute() {

		codigo = articulo.getCodigo();

		Articulo recoverArticulo = catalogo.getArticulo(codigo);

		if (recoverArticulo == null) { // No exisite, se puede
crear

			System.out.print("Creando el articulo con codigo: " +
codigo + "...");

			catalogo.nuevo(articulo);

			System.out.println("hecho.");
```

```
            } else {
                  System.out.println("El articulo con codigo: " +
codigo + " YA EXISTE.");
                  canUnExecute = false;
            }
      }

      @Override
      public void unExecute() throws OperationNotSupportedException {
            System.out.print("Eliminado el articulo con codigo: " +
codigo + "...");
            catalogo.eliminarArticulo(codigo);
            System.out.println("hecho.");
      }

      @Override
      public boolean canUnExecute() {
            return canUnExecute;
      }
}
```

El código para el comando que elimina artículos es muy parecido al comando que los crea, ya que son justo el opuesto el uno del otro:

EliminarCommand.java

```
package comportamiento.command.procesar_peticiones.client.comandos;

import javax.naming.OperationNotSupportedException;
import
comportamiento.command.procesar_peticiones.client.receivers.Articulo;
import
comportamiento.command.procesar_peticiones.client.receivers.Catalogo;
import comportamiento.command.procesar_peticiones.frmwrk.Command;
```

```java
/*
 * Clase creada por el usuario del framework
 */
public class EliminarCommand implements Command {
	private Catalogo catalogo;
	private String codigo;
	private Articulo recoverArticulo;
	private boolean canUnExecute = true;

	public EliminarCommand(Catalogo catalogo, String codigo) {
		this.catalogo = catalogo;
		this.codigo = codigo;
	}

	@Override
	public void execute() {
		recoverArticulo = catalogo.getArticulo(codigo);
		if (recoverArticulo != null) {
			System.out.print("Eliminado el articulo con codigo: "
+ codigo + "...");
			catalogo.eliminarArticulo(codigo);
			System.out.println("hecho.");
		} else {
			System.out.println("El articulo con codigo: " +
codigo + " NO EXISTE.");
			canUnExecute = false;
		}
	}

	@Override
	public void unExecute() throws OperationNotSupportedException {
```

```
            System.out.print("Creando el articulo con codigo: " +
codigo + "...");
            catalogo.nuevo(recoverArticulo);
            System.out.println("hecho.");
      }
      @Override
      public boolean canUnExecute() {
            return canUnExecute;
      }
}
```

Ahora veremos un clase macro-comando. Un macro-comando no aportada nada en sí mismo, pero es muy útil desde el punto de vista del usuario del *framework*, ya que le permite componer secuencias arbitrarias de comandos. Su diseño puede variar, presentado normalmente las siguientes características de implementación:

- Una lista de comandos para sostener una referencia a cada uno de los comandos miembro. Un método setter para poder añadir comandos a la lista.
- El método execute() recorre secuencialmente la lista anterior e invoca al método execute() de cada comando.
- El método unExecute() invierte el orden de la lista anterior e invoca al método unExecute() de cada comando.

Para el resto de clases, un macro-comando es un objeto comando como otro cualquiera, por lo que cuando el Invoker revierte un macro-comando sólo ve que se ha deshecho un comando (independientemente de los comandos que lo compongan).

MacroComand.java

```
package comportamiento.command.procesar_peticiones.client.comandos;


import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
```

```java
import javax.naming.OperationNotSupportedException;
import comportamiento.command.procesar_peticiones.frmwrk.Command;

/*
 * Clase creada por el usuario del framework
 */
public class MacroCommand implements Command {
	private List<Command> comandos = new ArrayList<Command>();
	private boolean canUnExecute = true;

	public List<Command> getCommandos() {
		return comandos;
	}

	public void addComando(Command comando) {
		comandos.add(comando);
	}

	@Override
	public void execute() {
		System.out.println("---Ejecutando macro-comando...");
		for (Command cmd : comandos) {
			cmd.execute();
		}
		System.out.println("---Fin del macro-comando.");
	}

	@Override
	public void unExecute() throws OperationNotSupportedException {
		Collections.reverse(comandos);
		for (Command cmd : comandos) {
```

```
                if (cmd.canUnExecute()) {
                        cmd.unExecute();
                } else {
                        canUnExecute = false;
                }
            }
        }
        @Override
        public boolean canUnExecute() {
            return canUnExecute;
        }
}
```

Por último, necesitamos una clase con un método main() para ejecutar la aplicación, crear algunos objetos y demostrar que todo funciona según los esperado.

No obstante, con una sola clase esto sería bastante caótico, por lo que vamos a crear cuatro clases de “demo”, cada una mostrando un aspecto relevante de la aplicación.

También, para evitar repetir código en cada clase, extraeremos la funcionalidad común a estas cuatro clases en una clase base de la cual todas heredarán.

Veamos la superclase para las cuatro clases de “demo”. Notad que esta clase:

- ✔ Se encarga de las peticiones del sistema a alto nivel que son comunes para las cuatro clases 'demo': crear un artículo, eliminar un artículo, etc. Para ello, necesita en su constructor una referencia a un Invoker y a un Receiver.
- ✔ En cada una de estas operaciones de alto nivel, se encarga de:
  - Crear el comando y asignarle su Receiver (Catalogo).
  - Inyectar el comando en el Invoker (ProcesadorPeticiones) y solicitar a éste que procese la petición.

SuperMain.java

```java
package comportamiento.command.procesar_peticiones.client;


import java.math.BigDecimal;
import comportamiento.command.procesar_peticiones.client.comandos.*;
import comportamiento.command.procesar_peticiones.client.receivers.*;
import comportamiento.command.procesar_peticiones.frmwrk.Command;
import
comportamiento.command.procesar_peticiones.frmwrk.ProcesadorPeticiones
;
/*
 * Clase creada por el usuario del framework.
 * Contiene la funcionalidad común a todas las clases de 'demo'.
 */
public class SuperMain {
	private ProcesadorPeticiones procesador;
	private Catalogo catalogo;
	public SuperMain(ProcesadorPeticiones procesador,
		Catalogo catalogo) {
		this.procesador = procesador;
		this.catalogo = catalogo;
	}
	public void crearArticulo(String codigo,
			String descripcion, String strPrecio)
	{
		Articulo articulo = new Articulo(codigo, descripcion,
				new BigDecimal(strPrecio));

		NuevoCommand nuevoCmd =
				new NuevoCommand(catalogo, articulo);

		procesador.setCmd(nuevoCmd);
		procesador.procesar();
```

```
	}


	/*
	 * Crear un objeto MostrarCommand proporcionándole su Receiver (catalogo)
	 * Al invoker (procesador) se le pasa ahora el objeto MostrarCommand y
	 * se le solicita que procese.
	 */
	public void mostrarArticulos() {
		MostrarCommand mostrarCmd = new MostrarCommand(catalogo);
		procesador.setCmd(mostrarCmd);
		procesador.procesar();
	}

	public void eliminarArticulo(String codigo) {
		EliminarCommand eliminarCmd =
				new EliminarCommand(catalogo, codigo);
		procesador.setCmd(eliminarCmd);
		procesador.procesar();
	}

	public void procesarTodoDeNuevo() {
		procesador.procesarHistorial();
	}

	public void deshacer(int i) {
		procesador.deshacer(i);
	}

	public void procesarMacroComando(Command macroComando) {
		procesador.setCmd(macroComando);
```

```
            procesador.procesar();
      }
}
```

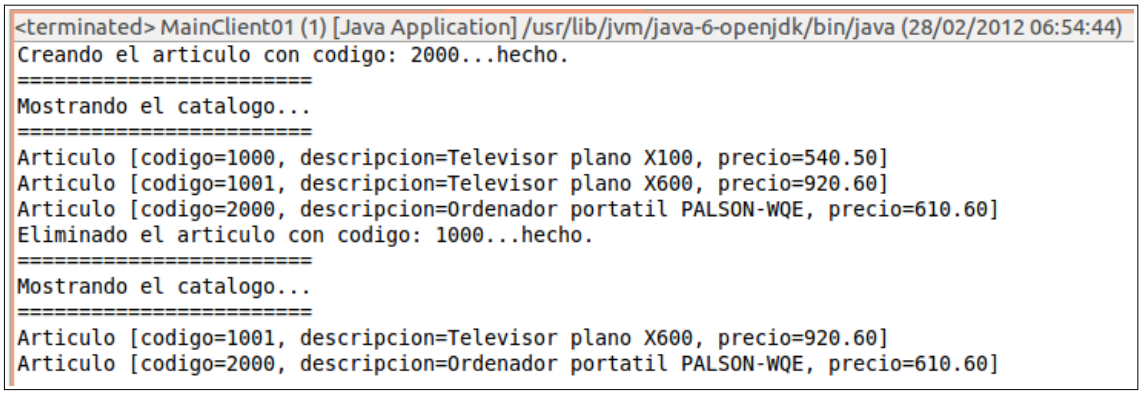
Gracias a la clase base, ahora el código para las clases 'demo' queda muy simplificado. Comenzamos viendo el caso más simple, en el que utilizamos algunos de los métodos definidos en la clase base para crear un artículo, listar el catálogo, eliminar un artículos y volver a listar. Todas estas operaciones implican la creación de comandos simples en la superclase, su asignación al Invoker (ProcesadorPeticiones) y la petición de ejecución (procesador.procesar()):

MainClient01.java

```
package comportamiento.command.procesar_peticiones.client;

import
comportamiento.command.procesar_peticiones.client.receivers.Catalogo;

import
comportamiento.command.procesar_peticiones.frmwrk.ProcesadorPeticiones
;

/*

 * Clase creada por el usuario del framework.

 * Hace lo siguiente: crea un articulo, lista el catalogo, elimina un
articulo del catalogo y vuelve a listar el catalogo

 *

 * Notad que está en un nivel de abstracción muy alto, es decir,

 * muy familiar al lenguaje natural.

 */

public class MainClient01 extends SuperMain {

      public MainClient01() {

            super(new ProcesadorPeticiones(), new Catalogo());


            crearArticulo("2000", "Ordenador portatil PALSON-WQE",
"610.60");

            mostrarArticulos();

            eliminarArticulo("1000");
```

```
            mostrarArticulos();
      }

      public static void main(String[] args) {
            new MainClient01();
      }
}
```

Salida:



La segunda “demo” muestra cómo volver a ejecutar todos los comandos. Es similar al anterior programa, sólo que al final llama al método de la superclase encargado de volver a ejecutar todo el historial:

MainClient02.java

```
package comportamiento.command.procesar_peticiones.client;

import
comportamiento.command.procesar_peticiones.client.receivers.Catalogo;

import
comportamiento.command.procesar_peticiones.frmwrk.ProcesadorPeticiones
;


/* Clase creada por el usuario del framework que demuestra el uso del
historial de comandos ejecutados */

public class MainClient02 extends SuperMain {

      public MainClient02() {

            super(new ProcesadorPeticiones(), new Catalogo());
```

```
            crearArticulo("2000", "Ordenador portatil PALSON-WQE",
"610.60");
            mostrarArticulos();
            eliminarArticulo("1000");
            mostrarArticulos();
            procesarTodoDeNuevo();
      }

      public static void main(String[] args) {
            new MainClient02();
      }
}
```

Salida:

```
Creando el articulo con codigo: 2000...hecho.
========================
Mostrando el catalogo...
========================
Articulo [codigo=1000, descripcion=Televisor plano X100, precio=540.50]
Articulo [codigo=1001, descripcion=Televisor plano X600, precio=920.60]
Articulo [codigo=2000, descripcion=Ordenador portatil PALSON-WQE, precio=610.60]
Eliminado el articulo con codigo: 1000...hecho.
========================
Mostrando el catalogo...
========================
Articulo [codigo=1001, descripcion=Televisor plano X600, precio=920.60]
Articulo [codigo=2000, descripcion=Ordenador portatil PALSON-WQE, precio=610.60]
**********************************************************
EJECUTANDO HISTORIAL...
**********************************************************
>>>Ejecutando comando: NuevoCommand
El articulo con codigo: 2000 YA EXISTE.
>>>Ejecutando comando: MostrarCommand
========================
Mostrando el catalogo...
========================
Articulo [codigo=1001, descripcion=Televisor plano X600, precio=920.60]
Articulo [codigo=2000, descripcion=Ordenador portatil PALSON-WQE, precio=610.60]
>>>Ejecutando comando: EliminarCommand
El articulo con codigo: 1000 NO EXISTE.
>>>Ejecutando comando: MostrarCommand
========================
Mostrando el catalogo...
========================
Articulo [codigo=1001, descripcion=Televisor plano X600, precio=920.60]
Articulo [codigo=2000, descripcion=Ordenador portatil PALSON-WQE, precio=610.60]
```

Notad que al volver a ejecutar los comandos no es posible insertar el que tiene el código 2000 porque ya existe en el Map (se insertó en la ejecución anterior). Igualmente, tampoco se puede eliminar el que tiene el código 1000 porque no existe (se eliminó en la ejecución anterior).

Veamos ahora el tercer programa de “demo”, en el que utilizamos un macro-comando. Aquí se utilizan también algunos de los métodos definidos en la superclase, aunque se crea un método específico en la propia subclase para definir y configurar el macro-comando, el cual está compuesto por cuatro comandos: dos para eliminar artículos, uno par añadir un artículo y otro para listar el catálogo.

MainClient03.java

```
package comportamiento.command.procesar_peticiones.client;


import java.math.BigDecimal;


import comportamiento.command.procesar_peticiones.client.comandos.*;
import comportamiento.command.procesar_peticiones.client.receivers.*;
import
comportamiento.command.procesar_peticiones.frmwrk.ProcesadorPeticiones
;


/*
 * Clase creada por el usuario del framework.
 * Demuestra el uso de un macro-comando.
 * Utiliza los métodos de la superclase para las operaciones comunes.
Sin
 * embargo crea un método especifico en esta misma clase para
concretar
 * en qué consiste el macro-comando.
 *
 */
public class MainClient03 extends SuperMain {


	// Creamos un Invoker
	private static ProcesadorPeticiones procesador = new
ProcesadorPeticiones();


	// Creamos un Receiver
```

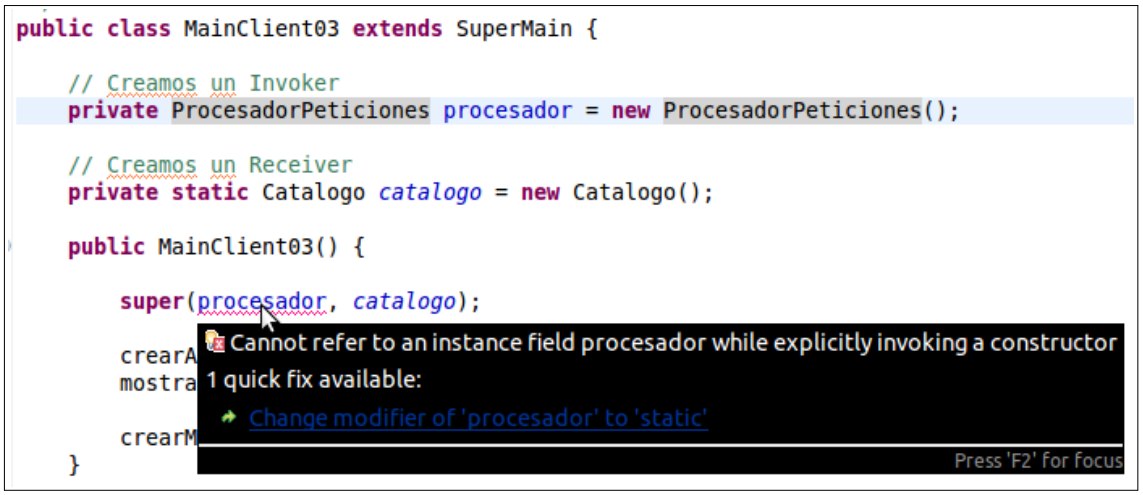
```
	private static Catalogo catalogo = new Catalogo();
	public MainClient03() {
		super(procesador, catalogo);
		crearArticulo("2000", "Ordenador portatil PALSON-WQE",
"610.60");
		mostrarArticulos();
		crearMacroComando();
	}

	private void crearMacroComando() {
		MacroCommand macroCmd = new MacroCommand();

		macroCmd.addComando(new EliminarCommand(catalogo,
"1000"));
		macroCmd.addComando(new EliminarCommand(catalogo,
"1001"));

		Articulo articulo = new Articulo("4000",
				"Disco duro externo 5TB Mantox", new
BigDecimal("190.50"));

		macroCmd.addComando(new NuevoCommand(catalogo, articulo));
		macroCmd.addComando(new MostrarCommand(catalogo));

		procesador.setCmd(macroCmd);
		procesador.procesar();
	}

	public static void main(String[] args) {
		new MainClient03();
	}
}
```

Salida:

```
<terminated> MainClient03 [Java Application] /usr/lib/jvm/java-6-openjdk/bin/java (28/02/2012 2
Creando el articulo con codigo: 2000...hecho.
========================
Mostrando el catalogo...
========================
Articulo [codigo=1000, descripcion=Televisor plano X100, precio=540.50]
Articulo [codigo=1001, descripcion=Televisor plano X600, precio=920.60]
Articulo [codigo=2000, descripcion=Ordenador portatil PALSON-WQE, precio=610.60]
---Ejecutando macro-comando...
Eliminado el articulo con codigo: 1000...hecho.
Eliminado el articulo con codigo: 1001...hecho.
Creando el articulo con codigo: 4000...hecho.
========================
Mostrando el catalogo...
========================
Articulo [codigo=2000, descripcion=Ordenador portatil PALSON-WQE, precio=610.60]
Articulo [codigo=4000, descripcion=Disco duro externo 5TB Mantox, precio=190.50]
---Fin del macro-comando.
```

¿Os habéis fijado que en el código anterior las referencias “procesador” y “catalogo” se han declarado estáticas? La razón es que tanto la superclase como la subclase necesitan la misma instancia de ProcesadorPeticiones, ya que en otro caso el macro-comando se almacenaría en un historial diferente al resto de comandos. Esto nos obliga a declararla como estática, ya que al haber una jerarquía de clases, el compilador no deja que el constructor de la subclase pase ningún atributo de instancia al constructor de la superclase, ya que la instancia de la subclase aún no se ha construido (está en ello). En cambio, cualquier miembro estático siempre queda inicializado en el primer momento que se hace referencia a una clase, independientemente de las instancias de una clase, es decir, aunque no existan o se estén construyendo justo en ese momento.

Para que se vea más claro, en la siguiente figura se ha suprimido la palabra clave 'static' de la declaración de “procesador”; mirad lo que nos dice el compilador:



No obstante, hay que observar que para el caso de la referencia “catalogo” podríamos haber omitido la declaración a nivel de atributo de clase y haber creado la instancia directamente en el constructor. Entonces, se podría haber definido una variable local en el método crearMacroComando(). Algo así:

```
public class MainClient03 extends SuperMain {

	// Creamos un Invoker

	private static ProcesadorPeticiones procesador = new
ProcesadorPeticiones();


	// Creamos un Receiver

	//private static Catalogo catalogo = new Catalogo();


	public MainClient03() {

		//super(procesador, catalogo);

		super(procesador, new Catalogo());


		crearArticulo("2000", "Ordenador portatil PALSON-WQE",
"610.60");

		mostrarArticulos();


		crearMacroComando();

	}
```

```
	private void crearMacroComando() {
		Catalogo catalogo = new Catalogo();
		MacroCommand macroCmd = new MacroCommand();

		macroCmd.addComando(new EliminarCommand(catalogo,
"1000"));
...
```

De esta manera, hubiéramos tenido dos instancias de catálogo, una para la superclase y otra para la subclase, aunque esto no supondría ningún problema, ya que todas las instancias de Catalogo trabajan contra el mismo Map (es estático), por lo que todas ven en cualquier momento los mismos artículos.

Y finalizamos con el cuarto programa 'demo", en el que se demuestra cómo deshacer un cierto números de operaciones ya ejecutadas.

Primero se utilizan los métodos de la superclase para crear dos artículos y listar el estado del catalogo; a continuación se crea un macro-comando compuesto por dos comandos para eliminar artículos, un tercero para crear un nuevo artículo y un cuarto para listar el catálogo. Respecto al macro-comando, a modo de variante, en este ejemplo se utiliza el método procesarMacroComando() de la superclase, algo que no habíamos hecho en el ejemplo anterior.

Finalmente, se llama al método deshacer() definido en la superclase para indicar que se reviertan las dos últimas operaciones ejecutadas. Dado que un macro-comando se considera una única operación, el Invoker deshace el macro-comando e intenta deshacer el comando anterior, el listado del catálogo, operación que no es posible revertir.

MainClient04.java

```
package comportamiento.command.procesar_peticiones.client;

import java.math.BigDecimal;

import comportamiento.command.procesar_peticiones.client.comandos.*;
```

```java
import comportamiento.command.procesar_peticiones.client.receivers.*;

import comportamiento.command.procesar_peticiones.frmwrk.Command;

import
comportamiento.command.procesar_peticiones.frmwrk.ProcesadorPeticiones
;


/*

 * Clase creada por el usuario del framework.

 * Clase cliente que demuestra el uso de deshacer operaciones ya
ejecutadas

 *

 */

public class MainClient04 extends SuperMain {


	public MainClient04() {


		super(new ProcesadorPeticiones(), new Catalogo());


		crearArticulo("2000", "Ordenador portatil PALSON-WQE",
"610.60");

		crearArticulo("3000", "Ordenador sobr emesa FFFR",
"920.00");

		mostrarArticulos();

		procesarMacroComando(crearMacroComando());


		/* Deshacemos las 2 ultimas operaciones.

		 * Un macro-comando se considera una única operación,
independientemente del numero de comandos que lo compongan.*/

		deshacer(2);

	}


	private Command crearMacroComando() {

		Catalogo catalogo = new Catalogo();

		MacroCommand macroCmd = new MacroCommand();
```

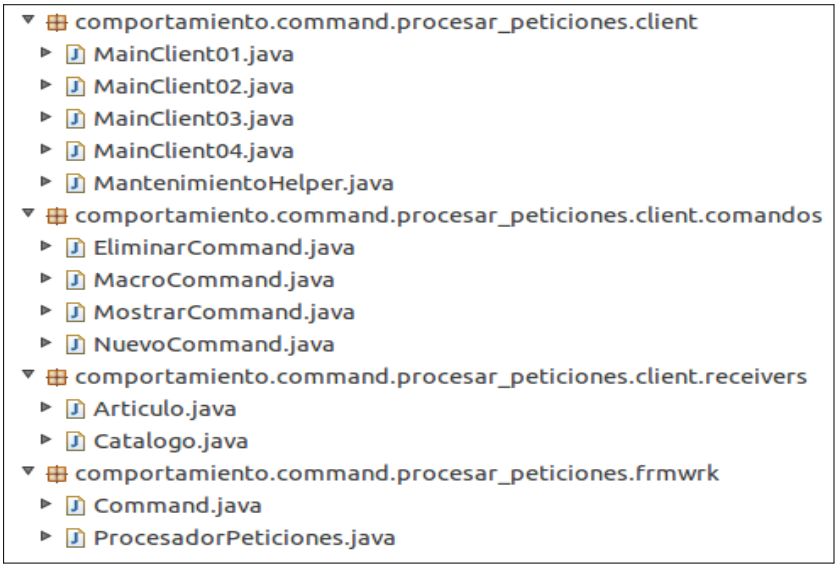
```
            macroCmd.addComando(new EliminarCommand(catalogo,
"1000"));
            macroCmd.addComando(new EliminarCommand(catalogo,
"1001"));

            Articulo articulo = new Articulo("4000",
                        "Disco duro externo 5TB Mantox", new
BigDecimal("190.50"));

            macroCmd.addComando(new NuevoCommand(catalogo, articulo));
            macroCmd.addComando(new MostrarCommand(catalogo));
            return macroCmd;
      }

      public static void main(String[] args) {
            new MainClient04();
      }
}
```

Salida:

```
<terminated> MainClient04 [Java Application] /usr/lib/jvm/java-6-openjdk/bin/java (28/02/2012 2
Creando el articulo con codigo: 2000...hecho.
Creando el articulo con codigo: 3000...hecho.
========================
Mostrando el catalogo...
========================
Articulo [codigo=1000, descripcion=Televisor plano X100, precio=540.50]
Articulo [codigo=1001, descripcion=Televisor plano X600, precio=920.60]
Articulo [codigo=2000, descripcion=Ordenador portatil PALSON-WQE, precio=610.60]
Articulo [codigo=3000, descripcion=Ordenador sobr emesa FFFR, precio=920.00]
---Ejecutando macro-comando...
Eliminado el articulo con codigo: 1000...hecho.
Eliminado el articulo con codigo: 1001...hecho.
Creando el articulo con codigo: 4000...hecho.
========================
Mostrando el catalogo...
========================
Articulo [codigo=2000, descripcion=Ordenador portatil PALSON-WQE, precio=610.60]
Articulo [codigo=3000, descripcion=Ordenador sobr emesa FFFR, precio=920.00]
Articulo [codigo=4000, descripcion=Disco duro externo 5TB Mantox, precio=190.50]
---Fin del macro-comando.
*********************************************************
DESHACIENDO LAS ULTIMAS (2) OPERACIONES...
*********************************************************
Eliminado el articulo con codigo: 4000...hecho.
Creando el articulo con codigo: 1001...hecho.
Creando el articulo con codigo: 1000...hecho.
El comando MostrarCommand no se puede revertir.
```

La gran ventaja del patrón Command es que al tener desacopladas las clases que reciben solicitudes de tareas (Invokers) y las clases que las llevan a cabo (Receivers), es posible disponer de un sistema fácilmente extensible sin infringir el principio abierto/cerrado. Por ejemplo, supongamos que ahora necesitáramos incorporar la funcionalidad de modificar un artículo del catálogo: tan sólo tendríamos que añadir este nuevo comportamiento en la clase Catalogo y crear un botón ModificarCommand.

La imagen siguiente ilustra la disposición de paquetes y clases del ejemplo:



Ejemplo 2 – Planificador de tareas

En este nuevo ejemplo tenemos una reunión con un cliente de la industria química que nos comenta lo siguiente:

El departamento de Producción utiliza una sustancia química denominado BQ (Base Química) como parte esencial de su proceso de fabricación. Este producto se activa al aplicarle corriente eléctrica y se desactiva al retirarla. Una vez que se activa genera mucha energía a una bajo coste, de ahí que sea tan interesante para el proceso de fabricación de la empresa.

Cuando la BQ se activa genera oscilaciones de temperatura, con el peligro de quedar inservible si se expone por encima de un límite de temperatura determinado, por lo que es esencial la actuación a intervalos regulares de una agente químico (agente externo) capaz de disminuir rápidamente esa temperatura. El agente químico NX200 se ha mostrado efectivo sobre la BQ, aunque llega a perder su capacidad de enfriamiento después de varias actuaciones.

Un sensor de temperatura permite monitorizar el estado de la BQ, para que, en caso de exceso de temperatura, se le pueda inyectar el agente NX200. Otro sensor permite monitorizar la efectividad del agente NX200 para que en caso de pérdida de efectividad sea posible detener el sistema ordenadamente, evitando daños mayores.

El cliente nos solicita un programa que permita ejecutar de manera planificada las tareas de monitorización sobre la BQ y el agente externo. Estas tareas se han de poder configurar para ejecutarse periódicamente.

Según lo expuesto, determinamos que se trata de un sistema donde se puede aplicar el patrón Command para permitir que un *framework* de planificación de tareas invoque a unos comandos (sensores) capaces de monitorizar sustancias químicas.

El *framework*, que no tendrá ni idea de la existencia de las sustancias químicas, conseguirá que se supervise el estado de éstas y que reaccionen según el protocolo establecido.

Notad que esto significa que el *framework* de planificación es un componente reutilizable en otros contextos.

Identificamos los siguientes componentes en el sistema:

Receivers*:*

Clases de la aplicación final que saben cómo realizar una determinada tarea:

- BaseQuimica: Clase cuyo estado y comportamiento refleja fielmente el estado y el comportamiento de la sustancia BQ. Como ya se ha comentado, la BQ se trata de una sustancia muy valiosa por su capacidad para generar energía. No obstante es inestable y frágil. Su inestabilidad se manifiesta en oscilaciones continuas de temperatura interna, con tendencia a incrementarse. Su fragilidad se debe a que queda inservible al quedar expuesta a una determinada temperatura, aunque sea durante un breve espacio de tiempo. BaseQuimica es una clase tipo Thread, la cual una vez activada sigue su propio ciclo de ejecución, manifestando los oscilamientos de temperatura ya comentados.
- AgenteExterno: Clase cuyo estado y comportamiento refleja fielmente el estado y el comportamiento del agente externo responsable de refrigerar la sustancia

BQ. Este agente permite disminuir la temperatura interna de la BQ cuando ésta se acerca al límite de su tolerancia. El agente actúa rápido pero en algún momento pierde efectividad, pasando a quedar inservible.

Supongamos que estas dos clases puede instanciarse con valores interpretados a través de dispositivos conectados a la BQ y al agente externo, respectivamente, por ejemplo mediante algún tipo de electrodos.

Hay que resaltar que el agente externo es realmente quien adopta el papel de Receiver dentro del patrón Command, pues es quien recibirá la orden de enfriar la sustancia BQ.

ConcreteCommands:

Clases de la aplicación final que implementan la interfaz Tarea, la cual pertenece al framework. Estas subclases disponen del método ejecutar() para solicitar operaciones sobre clases específicas de la aplicación (en nuestro caso, BaseQuimica y AgenteExterno). En el ejemplo, estas clases no se dedican meramente en delegar solicitudes a los Receivers, sino que implementan algún tipo de lógica que es necesaria para el correcto funcionamiento del sistema

Detectamos dos comandos:

- TareaSensorTemperatura: Solicita la temperatura a BaseQuimica y si el valor obtenido excede el límite de tolerancia entonces invoca al agente para inyectarlo en la BaseQuimica, quedando así refrigerada.
- TareaSensorAgente: Solicita al AgenteExterno su estado de efectividad. Si el AgenteExerno ya no es efectivo detiene ordenadamente el sistema para evitar daños mayores.

Invokers:

Clases del *framework* de planificación de tareas. Se detectan dos clases:

- Planificador: Se trata de una clase tipo Thread que se encarga de revisar cada cierto tiempo una lista de tareas (comandos). Durante la revisión de estas tareas se examina individualmente cada una para determinar si según la planificación temporal de la tarea ésta debe o no ejecutarse. El planificador se configura y se inicia (lanza) desde una clase cliente.

- TareaPlanificada: Las subclases de Command (los sensores en nuestro caso) no pueden ser utilizados directamente por el Planificador, ya que carecen de la información necesaria para su planificación. Por este motivo, surge la necesidad de la clase TareaPlanificada, que actuando de envoltorio sobre un comando permite al Planificador conocer detalles del tipo: frecuencia de repetición del comando, cuándo fue su última ejecución, etc.

Clase cliente

Punto de entrada al sistema. Contiene el método main() y se encarga de crear prácticamente todos las instancias de la aplicación, conectarlas e inicializarlas. Concretamente:

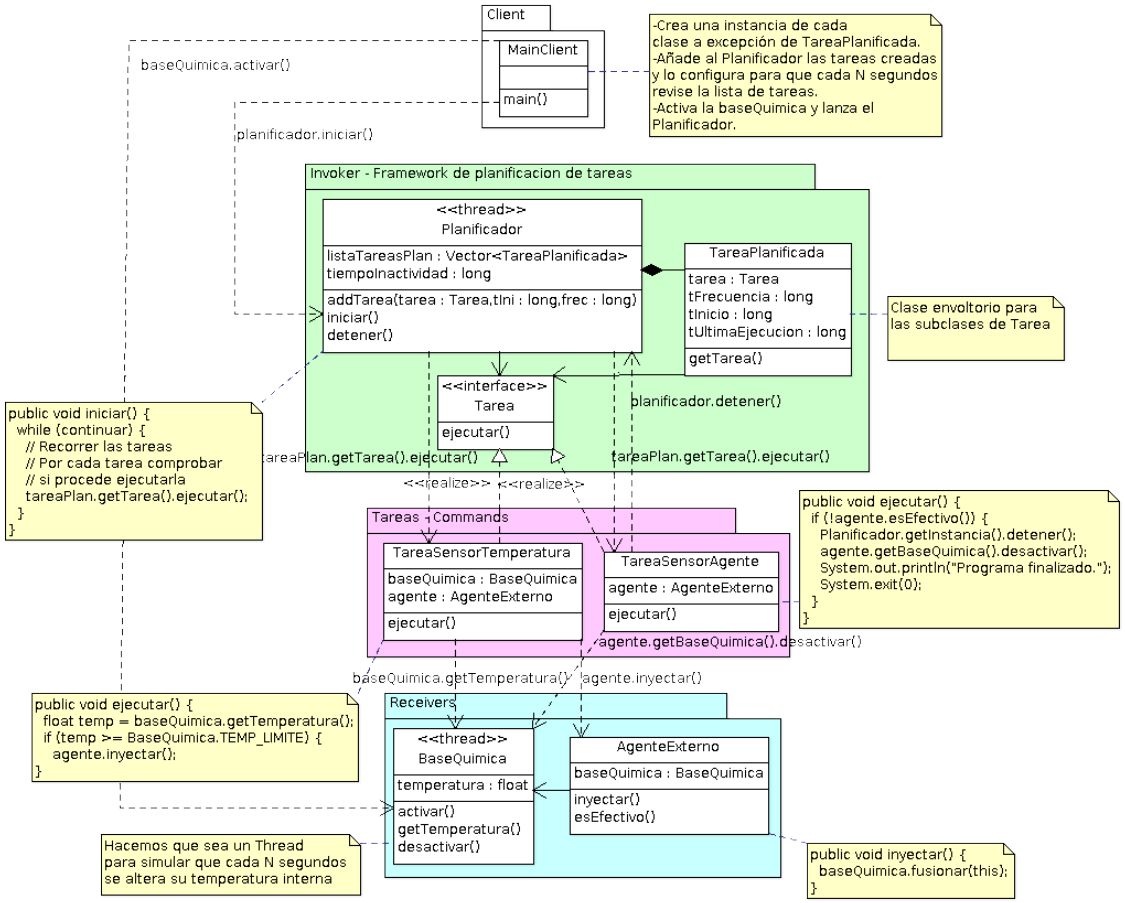
- Crea los Receivers, esto es, crea un objeto BaseQuimica y un objeto AgenteExterno conectado a la BaseQuimica.
- Crea un comando sensor de temperatura. Para crearlo es necesario pasarle una referencia a la instancia BaseQuimica cuya temperatura debe monitorizar y de la instancia de AgenteExterno que debe inyectar en la BaseQuimica para refrigerarla cuando sea necesario.
- Crea un comando sensor de efectividad de agente. Para crearlo es necesario pasarle una referencia a la instancia AgenteExterno cuya pérdida de efectividad debe monitorizar con tal de finalizar el sistema ordenadamente, llegado el caso.
- Crea un planificador de tareas, configurándolo para revisar la lista de tareas cada medio segundo y para admitir como máximo 100 tareas en la lista.
- Se añaden dos tareas al planificador, al mismo tiempo que se informa sobre la configuración de la planificación de cada una:
    - La primera es el sensor de temperatura. Esta tarea se configura para ser lanzada inmediatamente -llegado su turno dentro de la lista de tareas- y para ser ejecutada nuevamente pasados 3 segundos de su última ejecución. Debemos pensar que una tarea puede configurarse para que no se ejecute hasta pasado un tiempo determinado.
    - La segunda tarea es el sensor de efectividad del agente externo. En este caso, se configura para no ejecutarse inicialmente hasta pasados 10 segundos y para repetir su ejecución cada 2 segundos

- Se activa la instancia de BaseQuimica. Pensemos en esto como una aplicación de corriente sobre la sustancia química que provoca que ésta comience a generar energía (e implícitamente a elevar su temperatura interna) que es aprovechada para la fabricación de otras sustancias químicas.

- Se activa el planificador, por lo que según la planificación temporal configurada se lanzarán y relanzarán los comandos (sensores) que actúan sobre la instancia de BaseQuimica y AgenteExterno.

Notad que la clase cliente sobre la única clase que no interviene es TareaPlanificada, ya que en este caso, la decisión de diseño ha sido que sea la propia clase Planificador la responsable de crear instancias de TareaPlanficada a partir de instancias de Tarea.

La siguiente figura muestra el diagrama de clases necesario para nuestro ejemplo:



Veamos ahora el código del ejemplo.

Nota: Dado que en el enunciado se ha dado una buena explicación del papel que realiza cada componente dentro de la aplicación, el código que sigue que se comenta desde un punto de vista complementario, más técnico.

Comenzamos por los Receivers.

Aspectos a destacar de estas clases.

Para simular los cambios de temperatura en BaseQuimica, inicialmente hacemos que cada 4 segundos se calcule aleatoriamente su valor. Para ello utilizamos las clases de Java Timer y TimerTask, que son clases de conveniencia de tipo Thread para planificar y definir tareas, respectivamente.

El método fusionar() es llamado por la tarea TareaSensorTemperatura cuando detecta que la BaseQuimica ha rebasado el límite de temperatura. Esta llamada envía como argumento al AgenteExterno, el cual BaseQuimica utiliza para refrigerarse y para reprogramar sus cambios de temperatura internos, haciéndolos más pausados.

Notad que el método interno calcTemperatura() se ha implementado para obtener una temperatura entre 93-100, siendo 99 la tolerancia máxima para la BaseQuimica. Además se procura que el efecto del agente se vaya mitigando en cada ejecución del método.

BaseQuimica.java

```
package comportamiento.command.planificacion.client.receivers;


import java.util.Random;
import java.util.Timer;
import java.util.TimerTask;


/** Receiver **/
public class BaseQuimica  {
	private float temperatura;
	private Timer timer;
	private TimerTask timerTask;
	private float valorEnfriamientoAgente;
```

```
	public static final float TEMPERATURA_NOMINAL = 95.0F;

	public static final float TEMPERATURA_LIMITE = 99.0F;

	public static final long SEGUNDOS_CAMBIO_TEMPERATURA = 4000L;


	// MainClient activará la BaseQuimica, por lo que cada 4 segundos cambiará su temperatura

	public void activar() {

		activar(SEGUNDOS_CAMBIO_TEMPERATURA);

		System.out.println("------Base quimica activada. Variara su temperatura cada " +

				SEGUNDOS_CAMBIO_TEMPERATURA + " ms.");

	}


	// Metodo interno en el que se planifica la obtención de la siguiente temperatura

	private void activar(long tiempoCambioTemp) {

		timer = new Timer();

		timerTask = new TimerTask() {

			@Override

			public void run() {

				temperatura = calcTemperatura();

			}

		};

		timer.scheduleAtFixedRate(timerTask, 0L, tiempoCambioTemp);

	}


	// Permitir que sea posible desactivar la BaseQuimica

	public void desactivar() {

		timerTask.cancel();

		timer.cancel();

		timer.purge();
```

```
            timerTask = null;
            timer = null;
            System.out.println("------Base quimica desactivada");
      }

      // Devolver la temperatura
      public float getTemperatura() {
            return temperatura;
      }

      /*
       * Metodo que se ejecuta cuando la BaseQuimica alcanza el límite
de tolerancia.
       * El agente externo afecta a la base quimica de dos formas:
       *    -Provoca que la BQ no varie de temperatura con tanta
frecuencia.
       *    -Cuando la BQ varíe, lo hará con menores subidas de
temperatura
       */
      public void fusionar(AgenteExterno agente) {
            /*
             * Se vuelve a planificar el intervalo del calculo de
temperatura,
             * asegurando que el intervalo será mayor a 4 segundos
             */
            reactivar(agente.getSegundosCambiosTemperatura());

            /*
             * Se obtiene un valor aleatorio (entre 1.0-3.0) que
interviene
             * en el cálculo de la temperatura para refrigerarla
             */
            valorEnfriamientoAgente = agente.getValorEnfriamiento();
```

```
            System.out.println("******Fusion realizada");
      }

      private void reactivar(long tiempo) {
            timerTask.cancel();
            activar(tiempo);
            System.out.println("------Base quimica reactivada. Variara
ahora su temperatura cada " + tiempo + " ms.");
      }

      private float calcTemperatura() {
            // nextFloat() devuelve un valor entre 0.0 and 1.0
            // ([0...1] * 99-95) + 95+1-[1...3];
            // ([0...1] * 4) + 96-[1...3];
            // valor más bajo: 0 + 96 - 3 = 93 aprox.
            // valor más alto: 4 + 96 - 1 = 99 aprox., aunque
inicialmente, cuando valorEnfriamientoAgente=0, podemos obtener 100

            float nuevaTemperatura = (new Random().nextFloat() *
(TEMPERATURA_LIMITE - TEMPERATURA_NOMINAL)) +
                        TEMPERATURA_NOMINAL + 1.0F -
valorEnfriamientoAgente; // valorEnfriamientoAgente inicialmente es 0

            valorEnfriamientoAgente -= 0.33F; // el efecto
refrigerante del agente tiende a desaparecer

            System.out.println(this.getClass().getSimpleName() + " -
temperatura:" + nuevaTemperatura);

            return nuevaTemperatura;
      }

}
```

Veamos ahora la clase AgenteExterno. Esta clase tiene un método llamado inyectar() que es invocado por la tarea TareaSensorTemperatura cuando ésta detecta que la BaseQuimica ha alcanzo su temperatura máxima tolerable. El método inyectar() invoca al método fusionar() de BaseQuimica, pasando como argumento la propia referencia de AgenteExterno. De esta manera BaseQuimica puede utilizar los servicios que AgenteExterno proporciona.

Para simular el hecho de que AgenteExterno deja de ser efectivo en algún momento se obtiene un número aleatorio entre 1 y 50, siendo el 13 el valor que provoca la inefectividad.

También podemos ver un par de métodos que se utilizan para mitigar la temperatura de BaseQuimica:

- getSegundosCambioTemperatura(): colabora disminuyendo la frecuencia de oscilamientos de temperatura en BaseQuimica.
- GetValorEnfriamiento(): colabora impidiendo que BaseQuimica alcance valores de temperatura altos.

AgenteExterno.java

```
package comportamiento.command.planificacion.client.receivers;


import java.util.Random;


/** Receiver **/
public class AgenteExterno {


	// Valor arbitrario que establecemos como inefectivo
	private static final int VALOR_INEFECTIVIDAD = 13;
	BaseQuimica baseQuimica;


	public AgenteExterno(BaseQuimica baseQuimica) {
```

```
            this.baseQuimica = baseQuimica;
      }

      public void inyectar() {
            System.out.println("******Inyectando agente externo a la
base quimica...");
            baseQuimica.fusionar(this);
      }

      /*
       * Retorna un valor entre 5000ms y 11000ms
       */
      public long getSegundosCambiosTemperatura() {
            // nextInt(n) devuelve un valor entre 0 y n-1
            return 1000 * (new Random().nextInt(7) + 5);
      }

      // Devuelve un float entre 1.0 y 3.0
      public float getValorEnfriamiento() {
            // nextFloat() devuelve un valor entre 0.0 and 1.0
            return new Random().nextFloat() * 2.0F + 1.0F;
      }

      public boolean esEfectivo() {
            int numero = new Random().nextInt(50) + 1;

            if (numero == VALOR_INEFECTIVIDAD) {
                  System.out.println("#### Estado agente externo: NO
EFECTIVO!!!");
                  return false;
```

```
            }
            System.out.println("#### Estado agente externo:
EFECTIVO");
            return true;
      }
      public BaseQuimica getBaseQuimica() {
            return this.baseQuimica;
      }
}
```

Veamos a continuación el código para las tareas.

Nota: La interfaz está en un paquete diferente a las clases que la implementan, ya que las clases pertenecen a la aplicación final y la interfaz al *framework*.

Tenemos la interfaz Tarea:

Tarea.java

```
package comportamiento.command.planificacion.frmwrk;

public interface Tarea {
      public void ejecutar();
}
```

Y ahora las subclases. Estas clases almacenan una referencia de los Receivers, que utilizan cuando se dan las circunstancias apropiadas para invocar ciertos servicios que éstos proporcionan.

En particular, TareaSensorTemperatura llama al método inyectar() de AgenteExterno cuando BaseQuimica excede el tope establecido para su temperatura interna.

TareaSensorTemperatura.java

```
package comportamiento.command.planificacion.client.tareas;


import
comportamiento.command.planificacion.client.receivers.AgenteExterno;
```

```java
import comportamiento.command.planificacion.client.receivers.BaseQuimica;

import comportamiento.command.planificacion.frmwrk.Tarea;


public class TareaSensorTemperatura implements Tarea {


	BaseQuimica baseQuimica;

	AgenteExterno agente;


	public TareaSensorTemperatura(BaseQuimica baseQuimica,
AgenteExterno agente) {

		this.baseQuimica = baseQuimica;

		this.agente = agente;

	}


	@Override

	public void ejecutar() {

		System.out.print("Sensor temperatura invocando a
tarea...");

		float temperatura = baseQuimica.getTemperatura();


		if (temperatura >= BaseQuimica.TEMPERATURA_LIMITE) {

			System.out.println("\n******Base preparada para la
fusion...");

			agente.inyectar();

		} else {

			System.out.println("temperatura estable...Ok.");

		}

	}

}
```

Por su parte, TareaSensorAgente llama al método esEfectivo() de AgenteExterno según la planificación establecida por MainClient. En el momento que AgenteExterno deje de ser efectivo, TareaSensorAgente detendrá el programa.

TareaSensorAgente.java

```
package comportamiento.command.planificacion.client.tareas;


import
comportamiento.command.planificacion.client.receivers.AgenteExterno;
import comportamiento.command.planificacion.frmwrk.Planificador;
import comportamiento.command.planificacion.frmwrk.Tarea;


public class TareaSensorAgente implements Tarea {


	AgenteExterno agente;


	public TareaSensorAgente(AgenteExterno agente) {
		this.agente = agente;
	}


	@Override
	public void ejecutar() {
		System.out.print("Sensor efectividad invocando a
tarea...");
		if (!agente.esEfectivo()) {
			System.out.println("El agente externo ya no es
efectivo! Finalizando el programa según el protocolo de
seguridad...");


			Planificador.getInstancia().detener();


			agente.getBaseQuimica().desactivar();
```

```
                System.out.println("Programa finalizado.");
                System.exit(0);
            }
        }

}
```

Llega el turno de ver el código del framework de planificación de tareas.

Comenzamos por ver la clase que sirve de envoltorio para las subclases de Tarea, TareaPlanificada, que proporciona la información necesaria que permite al Planificador programar la temporalidad de ejecución de los comandos:

TareaPlanificada.java

```
package comportamiento.command.planificacion.frmwrk;


/*
 * Wrapper (envoltorio) para las tareas.
 * Una tarea planificada es una tarea con los siguientes atributos:
 * -La frecuencia de repeticion
 * -El instante de su última ejecucion
 */
public class TareaPlanificada {


      private Tarea tarea; // La tarea a realizar (el objeto Command
      private long tFrecuencia; // Frecuencia de repeticion de la
tarea
      private long tInicio; // tiempo que debe transcurrir para la
primera ejecucion de la tarea
      private long tUltimaEjecucion; // Instante de la ultima
ejecucion


      public TareaPlanificada(Tarea tarea, long tIni, long frecuencia)
{
```

```
            this.tarea = tarea;
            this.tFrecuencia = frecuencia;
            this.tInicio = tIni + System.currentTimeMillis();
            this.tUltimaEjecucion = System.currentTimeMillis();
      }

      public Tarea getTarea() { return tarea; }
      public long getIntervaloRepeticion() { return tFrecuencia; }
      public long getTiempoUltimaEjecucion() { return
tUltimaEjecucion; }
      public long getStart() { return tInicio; }

      public void setTiempoUltimaEjecucion(long tUltimaEjecucion) {
            this.tUltimaEjecucion = tUltimaEjecucion;
      }
}
```

Veamos ahora la clase Planificador. Esta clase es un Thread que sigue un diseño Singleton y que según la configuración temporal establecida en MainClient, recorrerá con mayor o menor frecuencia la lista de tareas y examinará si procede o no ejecutarlas.

Hay que destacar lo siguiente:

- Por si MainClient omite establecer la frecuencia con la que el planificador debe revisar la lista de tareas, se fija el valor de 5 segundos como valor por defecto.
- Una vez iniciada se ejecuta en un bucle “infinito” hasta que la tarea TareaSensorAgente llame al método detener() de Planificador.
- El método addTarea() recibe como argumentos:
    - La subclase de Tarea a planificar.
    - Los milisegundos a esperar antes de ejecutarla por primera vez

    - Los milisegundos que deben pasar para volver a ejecutarla (0 significa que no se debe repetir).

- addTarea() utiliza los argumentos recibidos para crear una instancia de TareaPlanificada, cuyas instancias son las que realmente almacena en su vector y que utiliza para examinar si procede o no su ejecución. Sin embargo, una vez determinado esto, necesita recuperar la tarea subyacente para poderla ejecutar.

Planificador.java

```
package comportamiento.command.planificacion.frmwrk;

import java.util.Vector;

public class Planificador {

	private static Planificador INSTANCIA = new Planificador();

	// Cada cuanto se debe comprobar si hay tareas que deben ejecutarse
	private long tiempoInactividad = 5000; // valor por defecto

	private Vector<TareaPlanificada> listaTareasPlanificadas;

	private boolean continuar = true;

	private Planificador() { // clase no instanciable

	}

	public static Planificador getInstancia() {
		return INSTANCIA;
	}
```

```
	public void configurar(long tInactividad, int maxNumTareas) {
		tiempoInactividad = tInactividad;
		listaTareasPlanificadas = new
Vector<TareaPlanificada>(maxNumTareas);
	}


	public void addTarea(Tarea tarea, long tInicial, long
frecuencia) {
		long intervaloRep = (frecuencia > 0) ? frecuencia : 0;
		TareaPlanificada tareaPlan = new TareaPlanificada(tarea,
tInicial, intervaloRep);
		listaTareasPlanificadas.add(tareaPlan);
		System.out.println("Tarea " +
tarea.getClass().getSimpleName() +
				" añadida al planificador. Programada para
ejecutarse cada " +
				frecuencia + " ms.");
	}


	// Iniciar el thread
	public void iniciar() {
		new Thread() {
			@Override
			public void run() {
				System.out.println("====================");
				System.out.println("Planificador iniciado");
				System.out.println("====================");
				while (continuar) {
					try {
						sleep(tiempoInactividad);
						long presente =
System.currentTimeMillis();
```

```
                                    for (TareaPlanificada tareaPlan :
listaTareasPlanificadas) {
                                          if (presente >
tareaPlan.getStart()) {
                                                long tiempoTranscurrido
=
      tareaPlan.getIntervaloRepeticion() +
tareaPlan.getTiempoUltimaEjecucion();
                                                if (presente >
tiempoTranscurrido) {
      tareaPlan.getTarea().ejecutar();
      tareaPlan.setTiempoUltimaEjecucion(presente);
                                                }
                                          }
                                    }
                              } catch (Exception e) {
                                    System.out.println("El tiempo de
inactividad del planificador ha sido interrumpido de manera
inesperada: " + e);
                              }
                        }
                        System.out.println("====================");
                        System.out.println("Planificador detenido");
                        System.out.println("====================");
                  }
            }.start();
      }

      public Vector<TareaPlanificada> getTareas() {
            return listaTareasPlanificadas;
      }
```

```
	public long getTiempoInactividad() {
		return tiempoInactividad;
	}

	public void setTiempoInactividad(long tInactividad) {
		tiempoInactividad = tInactividad;
	}

	public void detener() {
		continuar = false;
		System.out.println("------Planificador detenido");
	}
}
```

Finalmente veamos el código de MainClient:

Como ya se ha mencionado a lo largo del ejemplo, esta clase se encarga de crear, configurar y activar casi la totalidad de los componentes del sistema.

MainClient.java

```
package comportamiento.command.planificacion.client;


import
comportamiento.command.planificacion.client.receivers.AgenteExterno;
import
comportamiento.command.planificacion.client.receivers.BaseQuimica;
import
comportamiento.command.planificacion.client.tareas.TareaSensorAgente;
import
comportamiento.command.planificacion.client.tareas.TareaSensorTemperat
ura;
import comportamiento.command.planificacion.frmwrk.Planificador;
import comportamiento.command.planificacion.frmwrk.Tarea;
```

```
public class MainClient {

	public static void main(String args[]) {

		/*
		 * Receivers. Son quienes implementan la funcionalidad que
		 * realmente queremos que se ejecute.
		 */
		BaseQuimica baseQuimica = new BaseQuimica();
		AgenteExterno HX1000D = new AgenteExterno(baseQuimica);

		/*
		 * Creacion de tareas
		 */
		Tarea sensorTempBQ =
				new TareaSensorTemperatura(baseQuimica,
HX1000D);

		Tarea sensorEfectividadAgente = new
TareaSensorAgente(HX1000D);

		/*
		 * Creamos un planificador que:
		 * -Cada medio segundo comprobará si hay tareas que deban
ejecutarse.
		 * -Como maximo admitirá 100 tareas.
		 */
		Planificador planificador = Planificador.getInstancia();
		planificador.configurar(500, 100);

		/*
		 * Añadimos el command (la tarea) al planificador.
```

```
             * Indicamos que debe arrancarse lo antes posible y
repetirse cada 3 segundos.
             */
            planificador.addTarea(sensorTempBQ, 0, 3000);

            /*
             * Añadimos el command (la tarea) al planificador.
             * Indicamos que debe arrancarse a los 10 segundos y
repetirse cada 8 segundos.
             */
            planificador.addTarea(sensorEfectividadAgente, 10000,
8000);

            /*
             * Activamos la base quimica
             */
            baseQuimica.activar();

            /*
             * Iniciamos el planificador
             */
            planificador.iniciar();
      }
}
```

A continuación, un ejemplo de la salida del programa. Hay que tener en cuenta que cada salida será diferente, dada la naturaleza multihilo de la aplicación y por el uso de cálculos aleatorios para las simulaciones de comportamiento:

Fragmento de salida al iniciar el programa:

```
Tarea TareaSensorTemperatura añadida al planificador. Programada para ejecutarse cada 3000 ms.
Tarea TareaSensorAgente añadida al planificador. Programada para ejecutarse cada 8000 ms.
------Base quimica activada. Variara su temperatura cada 4000 ms.
BaseQuimica - temperatura:98.70645
=====================
Planificador iniciado
=====================
Sensor temperatura invocando a tarea...temperatura estable...Ok.
BaseQuimica - temperatura:98.10772
Sensor temperatura invocando a tarea...temperatura estable...Ok.
BaseQuimica - temperatura:98.03257
Sensor temperatura invocando a tarea...temperatura estable...Ok.
Sensor efectividad invocando a tarea...#### Estado agente externo: EFECTIVO
BaseQuimica - temperatura:99.97115
Sensor temperatura invocando a tarea...
******Base preparada para la fusion...
******Inyectando agente externo a la base quimica...
------Base quimica reactivada. Variara ahora su temperatura cada 9000 ms.
******Fusion realizada
BaseQuimica - temperatura:93.6114
Sensor temperatura invocando a tarea...temperatura estable...Ok.
Sensor efectividad invocando a tarea...#### Estado agente externo: EFECTIVO
Sensor temperatura invocando a tarea...temperatura estable...Ok.
```

Fragmento de salida intermedio:

```
BaseQuimica - temperatura:96.56466
Sensor efectividad invocando a tarea...#### Estado agente externo: EFECTIVO
Sensor temperatura invocando a tarea...temperatura estable...Ok.
Sensor temperatura invocando a tarea...temperatura estable...Ok.
Sensor efectividad invocando a tarea...#### Estado agente externo: EFECTIVO
BaseQuimica - temperatura:100.51452
Sensor temperatura invocando a tarea...
******Base preparada para la fusion...
******Inyectando agente externo a la base quimica...
------Base quimica reactivada. Variara ahora su temperatura cada 11000 ms.
******Fusion realizada
```

Fragmento de salida al finalizar el programa:

```
Sensor temperatura invocando a tarea...temperatura estable...Ok.
BaseQuimica - temperatura:97.20837
Sensor efectividad invocando a tarea...#### Estado agente externo: EFECTIVO
Sensor temperatura invocando a tarea...temperatura estable...Ok.
Sensor temperatura invocando a tarea...temperatura estable...Ok.
Sensor temperatura invocando a tarea...temperatura estable...Ok.
Sensor efectividad invocando a tarea...#### Estado agente externo: EFECTIVO
Sensor temperatura invocando a tarea...temperatura estable...Ok.
BaseQuimica - temperatura:98.43648
Sensor temperatura invocando a tarea...temperatura estable...Ok.
Sensor efectividad invocando a tarea...#### Estado agente externo: NO EFECTIVO!!!
El agente externo ya no es efectivo! Finalizando el programa según el protocolo de seguridad...
------Planificador detenido
------Base quimica desactivada
Programa finalizado.
```

La imagen siguiente ilustra la disposición de paquetes y clases del ejemplo:

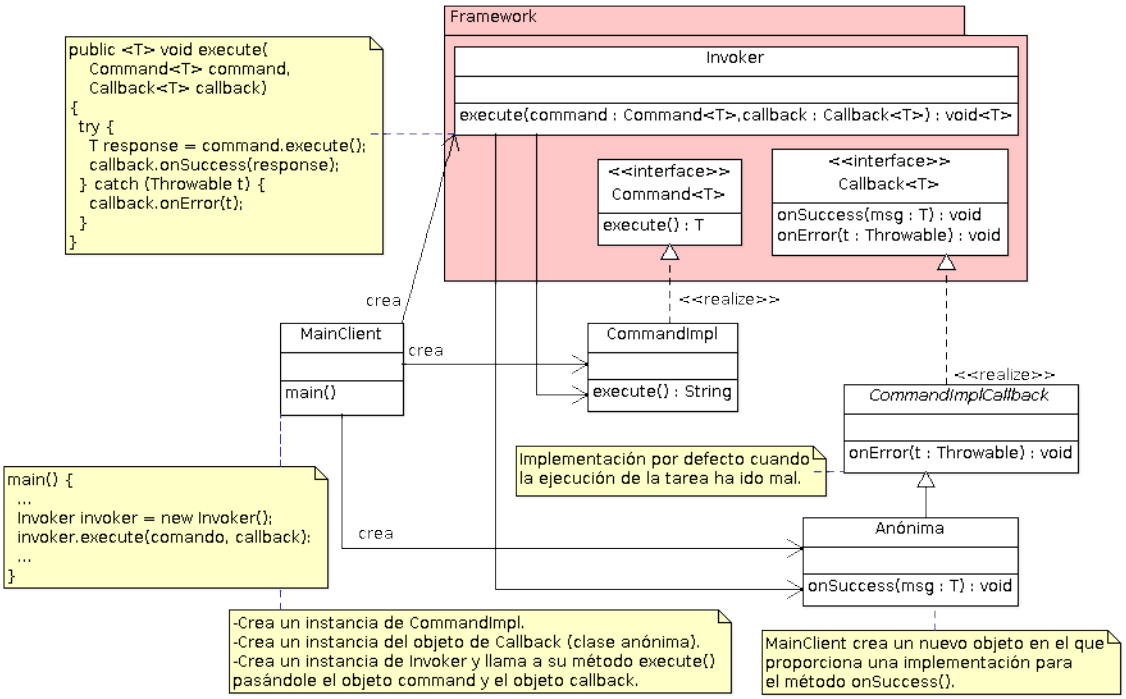
Ejemplo 3 – Clases Callback (retrollamada)

Hasta ahora hemos visto cómo el patrón Command nos ayuda a que las clases de un *framework* encargadas de recibir una petición (Invoker's) puedan ejecutar código de clases de la aplicación del usuario del *framework* que realizan una labor asociada a esa petición (Receiver). Esto es muy importante, ya que nos permite extender fácilmente las aplicaciones siguiendo buenos principios de diseño.

Ahora bien, ¿y si queremos además que el *framework*, en función del resultado de la ejecución de la petición, pueda invocar clases de la aplicación del usuario que se encarguen de proceder cómo sea conveniente según ese resultado? Por supuesto, el *framework* desconoce esas clases. ¿Cómo lo hacemos? Tan sólo tenemos que hacer que esas clases (CallBacks) implementen una interfaz con los métodos apropiados, y ya será posible que un Invoker pueda llamarlas. Por tanto, es en una situación similar a como un Invoker consigue la ejecución de los servicios proporcionados por los Receiver's.

El ejemplo que se presenta a continuación no utiliza Receiver's, ya que el trabajo a realizar lo implementa el propio comando (esta es una variante bastante frecuente del patrón en la que tenemos objetos comando “inteligentes”). El hecho de no utilizar Receiver's en este ejemplo, simplifica un poco el diseño y permite poner el foco en los objetos Callback.

La imagen siguiente muestra el diagrama de clases del ejemplo:



Habréis advertido en el gráfico que en este ejemplo se utilizan los Generics, incorporados al lenguaje Java en la versión 5. Los Generics dan flexibilidad, permitiendo parametrizar los tipos base de nuestras clases; sin embargo, el ejemplo funciona perfectamente sin utilizar Generics.

Veamos el código. Primero los componentes del framework:

Command.java

```
package comportamiento.command.callback.framework;


public interface Command<T> {
    T execute();
}
```

Callback.java

```
package comportamiento.command.callback.framework;


public interface Callback<T> {
```

```
    void onSuccess(T message);
    void onError(Throwable t);
}
```

Veamos el Invoker.

Invoker.java

```
package comportamiento.command.callback.framework;
public class Invoker {
      public <T> void execute(Command<T> command, Callback<T>
callback) {
            try {
                  T response = command.execute();
                  callback.onSuccess(response);
            } catch (Throwable t) {
                  callback.onError(t);
            }
      }
}
```

Pasemos a ver el código de la aplicación final. La implementación para el comando consiste en generar un número aleatorio entre el 0 y el 9. En caso de ser menor que 5 se lanza una excepción (lo que provocará que el Invoker llame al método onError() del objeto Callback). En otro caso, retorna el número obtenido en formato String (lo que provocará que el Invoker llame al método onSuccess() del objeto Callback).

CommandImpl.java

```
package comportamiento.command.callback.client;

import comportamiento.command.callback.framework.Command;

public class CommandImpl implements Command<String> {
```

```
    @Override
    public String execute() {
        int i = (int) (Math.random() * 10);
        if (i < 5)
            throw new RuntimeException("Se ha producido un error
simulado.");

        return String.valueOf(i);
    }
}
```

La clase Callback -en nuestro ejemplo- consiste en una clase abstracta con una implementación por defecto para el método onError(). Por tanto, en algún sitio se creará una subclase de esta clase abstracta que proporcione una implementación para el método onSuccess() -esto lo vemos a continuación en la clase MainClient:

ComandImplCallback.java

```
package comportamiento.command.callback.client;

import comportamiento.command.callback.framework.Callback;

public abstract class CommandImplCallback implements Callback<String>
{

    @Override
    public void onError(Throwable t) {
        t.printStackTrace(System.err);
    }
}
```

Y finalizamos con la clase cliente que contiene el método main():

MainClient.java

```
package comportamiento.command.callback.client;


import comportamiento.command.callback.framework.Command;
import comportamiento.command.callback.framework.Invoker;


public class MainClient {


      public static void main(String[] args) {


            Invoker invoker = new Invoker();
            Command<String> comando = new CommandImpl();


            invoker.execute(comando, new CommandImplCallback() {
                @Override
                public void onSuccess(String msg) {
                    System.out.println("El comando se ha ejecutado
correctamente, produciendo un resultado de \"" + msg + "\"");
                }
            });
      }
}
```

Notad cómo se crea la clase anónima -subclase de CommandImplCallback- a la misma vez que se llama al método execute() del Invoker.

Salida:

Tenemos el 50% de posibilidades de obtener una excepción (provocada):

```
java.lang.RuntimeException: Se ha producido un error simulado.
        at comportamiento.command.callback.client.CommandImpl.execute(CommandImpl.java:11)
        at comportamiento.command.callback.client.CommandImpl.execute(CommandImpl.java:1)
        at comportamiento.command.callback.framework.Invoker.execute(Invoker.java:7)
        at comportamiento.command.callback.client.MainClient.main(MainClient.java:14)
```

Y el 50% de posibilidades de obtener un número entre 5 y 9 ambos inclusive:

```
El comando se ha ejecutado correctamente, produciendo un resultado de "9"
```

Problemas específicos e implementación

Autonomía de los comandos

Hay que tener presente que para determinado tipo de peticiones o en función de la naturaleza de una aplicación, puede que no se necesiten clases Receiver y que los propios comandos concretos (subclases de Command) sean lo suficientemente "inteligentes" o autónomos para llevar a cabo las peticiones. Por ejemplo, una petición para salir de la aplicación o para abrir una nueva ventana puede perfectamente realizarla un comando en concreto, sin necesidad de delegar en una clase Receiver.

Operaciones de deshacer/repetir

Cuando en una determinada implementación del patrón Command se necesita soporte para operaciones deshacer/repetir se tiene que establecer algún mecanismo para obtener información histórica de un Receiver, ya que en cada ejecución el estado del Receiver es susceptible de ser modificado. Para lograr esto existen dos aproximaciones:

- Antes de ejecutar cada comando se guarda en memoria el estado del Receiver. Esto no requiere una programación especialmente compleja, aunque es un método que no puede aplicarse en todos los casos. Por ejemplo, intentar algo así en una aplicación de procesamiento de imágenes requeriría almacenar imágenes en memoria después de cada operación ejecutada sobre la imagen, lo cual es prácticamente imposible.

- Guardar en memoria el conjunto de operaciones ejecutadas sobre un Receiver. En este caso, tanto el comando como el Receiver deben implementar algoritmos inversos para deshacer cada operación (add()/remove(), insert()/delete(), ...). Esto requiere un esfuerzo de programación adicional, pero menos memoria. El patrón Memento suele ser de ayuda cuando se requiere almacenar más información sobre el Receiver.

Patrones relacionados

- Se puede utilizar Composite para implementar órdenes formadas por un conjunto de otras órdenes, es decir, macros.
- Para crear órdenes reversibles se puede utilizar Memento.
- Prototype puede ser útil para crear de manera reiterativa órdenes que contengan la misma operación u otras comunes.