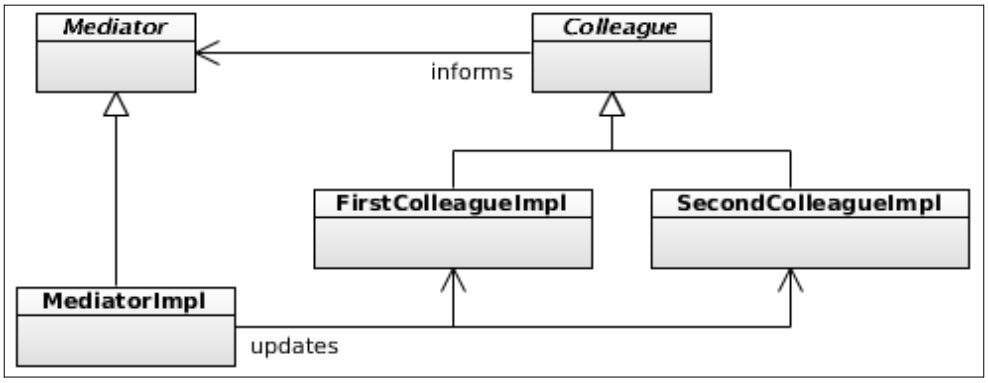
Mediator

Diagrama de clases e interfaces

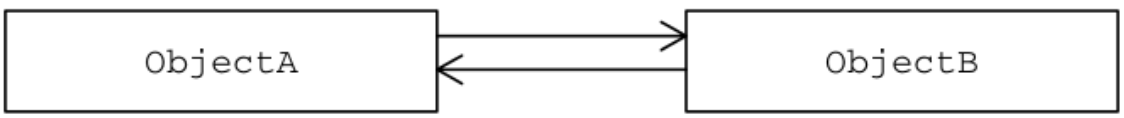


Intención

Mediator simplifica la comunicación entre los objetos de un sistema al delegar en un nuevo objeto el control de la distribución de los mensajes. Este nuevo objeto encapsula las reglas de interacción de un conjunto de objetos determinado. Mediator fomenta el bajo acoplamiento evitando que los objetos tengan que referirse explícitamente los unos a los otros, lo cual permite variar fácilmente sus interacciones.

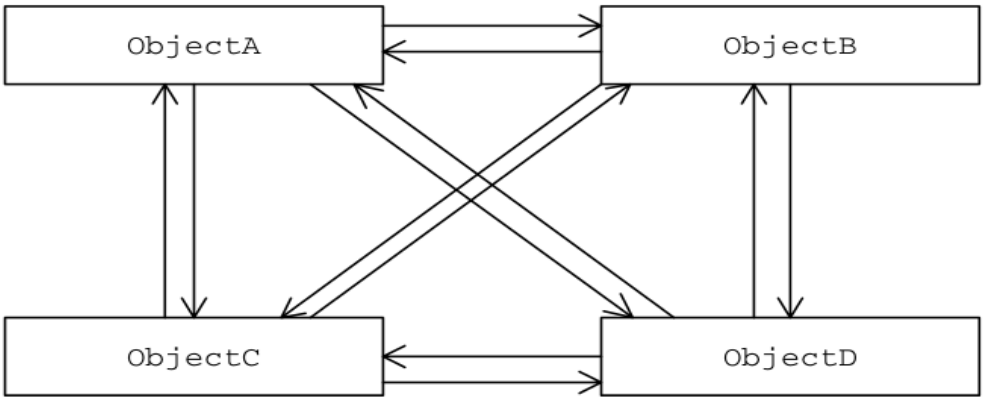
Motivación

El diseño orientado a objetos se basa en la distribución de comportamiento mediante objetos especializados con tal de proporcionar un servicio. La interacción entre objetos puede ser directa (punto a punto), tal y como se muestra en la siguiente figura:



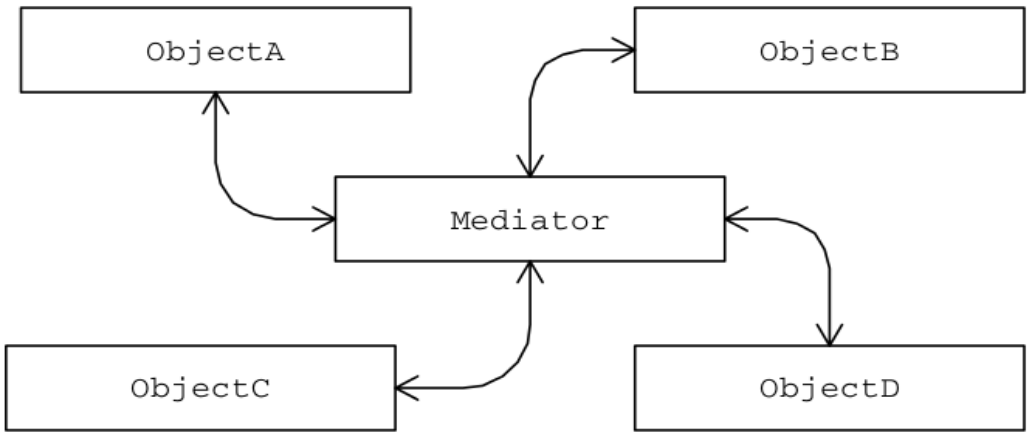
La interacción directa entre objetos es una buena opción siempre que el número de objetos implicados se mantenga bajo. Sin embargo, en cuanto el número de objetos afectados incrementa, la interacción directa supone un complejo laberinto de referencias entre objetos, lo cual nos lleva a tener una aplicación difícil de mantener.

Por ejemplo, en un sistema con 4 objetos donde cada objeto tenga que conocer a todos los demás objetos, ya supone una distribución con una estructura con excesivas conexiones. La figura siguiente ilustra este hecho:



Por lo general, desglosar un sistema en varios objetos facilita la reusabilidad. No obstante, la proliferación de interconexiones tiende a reducirla. Cuando tenemos una estructura de objetos con muchas interconexiones bajan las probabilidades de que un objeto pueda reutilizarse individualmente (sin tener que reutilizar también sus dependencias). Por otro lado, ante un (habitual) escenario como el descrito, seguramente será complicado cambiar el comportamiento del sistema de una manera significativa, ya que tal comportamiento estará disperso entre varios objetos. Ante esto, el diseñador se verá forzado a crear diferentes subclases que permitan redefinir el nuevo comportamiento deseado.

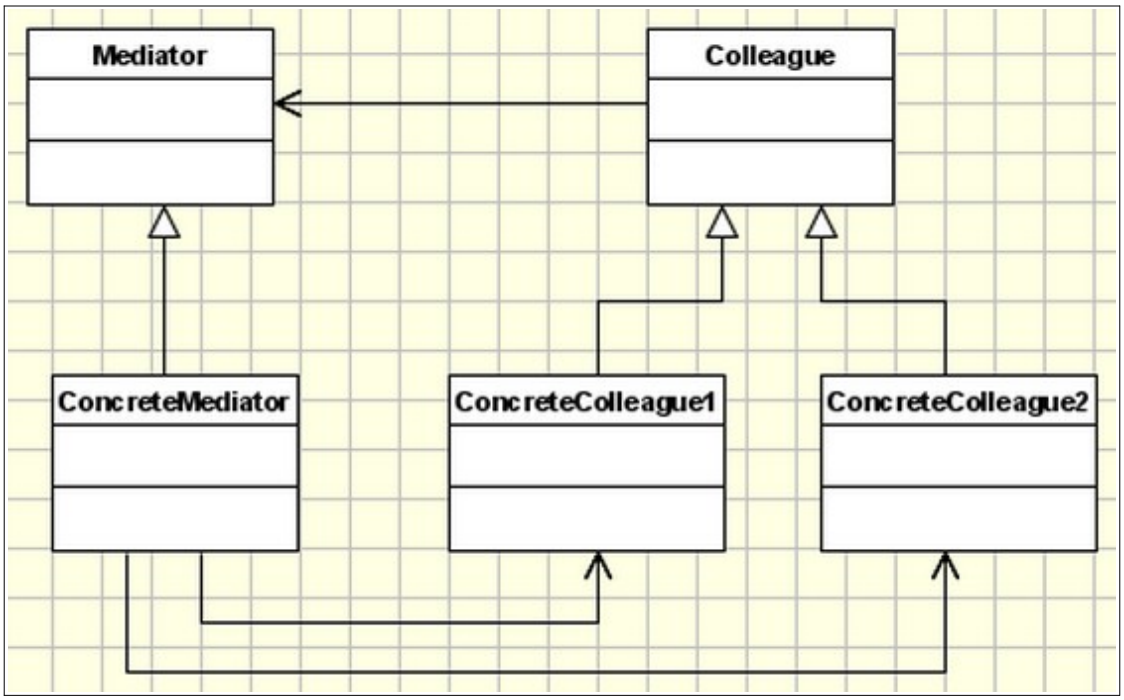
El patrón Mediator nos permite diseñar una modelo bien coordinado y controlado para gestionar la comunicación entre un grupo de objetos, eliminando la necesidad de que los objetos tengan que referenciarse los unos a los otros. La figura siguiente muestra este concepto:



Implementación

El patrón Mediator sugiere abstraer todos los detalles de las interacciones entre un grupo de objetos en una clase separada, conocida como Mediator. Cada objeto del grupo continúa siendo responsable de proporcionar el servicio para el cual fue diseñada, pero ya no debe preocuparse de interactuar directamente con otros objetos para ofrecer su servicio, ya que de tal interacción es ahora responsable el Mediator. Todos los objetos envían sus mensajes al Mediator y éste envía cada mensaje al objeto apropiado, según la lógica de interacción definida por los requerimientos de la aplicación.

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



Las clases participantes en el patrón son las siguientes:

- **Mediator:** Define una interfaz para poder comunicarse con objetos Colegas.
- **ConcreteMediator**: Clase que implementa la interfaz Mediator, estableciendo el comportamiento necesario para coordinar los objetos colegas. Por tanto, conoce y mantiene a tales objetos colegas.
- **Colleague**: Clase abstracta o interfaz común para todos los objetos colegas.

- **ConcreteColleague**: Cada colega conoce su objeto Mediator y se comunica con él siempre que necesita establecer comunicación con otro colega.

Aplicabilidad y Ejemplos

El patrón Mediator es adecuado cuando:

- Una serie de objetos se comunica de una manera bien conocida pero que es compleja. Mediator desacopla las clases colegas, a la vez que abstrae la manera en que estos interactúan.
- Se necesita un punto común de control o de comunicación (se necesita el código en una única clase en lugar de encontrarse disperso por varias clases).
- Se necesita especializar un comportamiento distribuido por varios objetos pero no podemos o no queremos crear subclases para implementar ese comportamiento especializado. Si queremos cambiar el comportamiento creamos subclases solamente del Mediator.

Veamos ahora algunos ejemplos.

Ejemplo 1 – Aplicación de mensajería

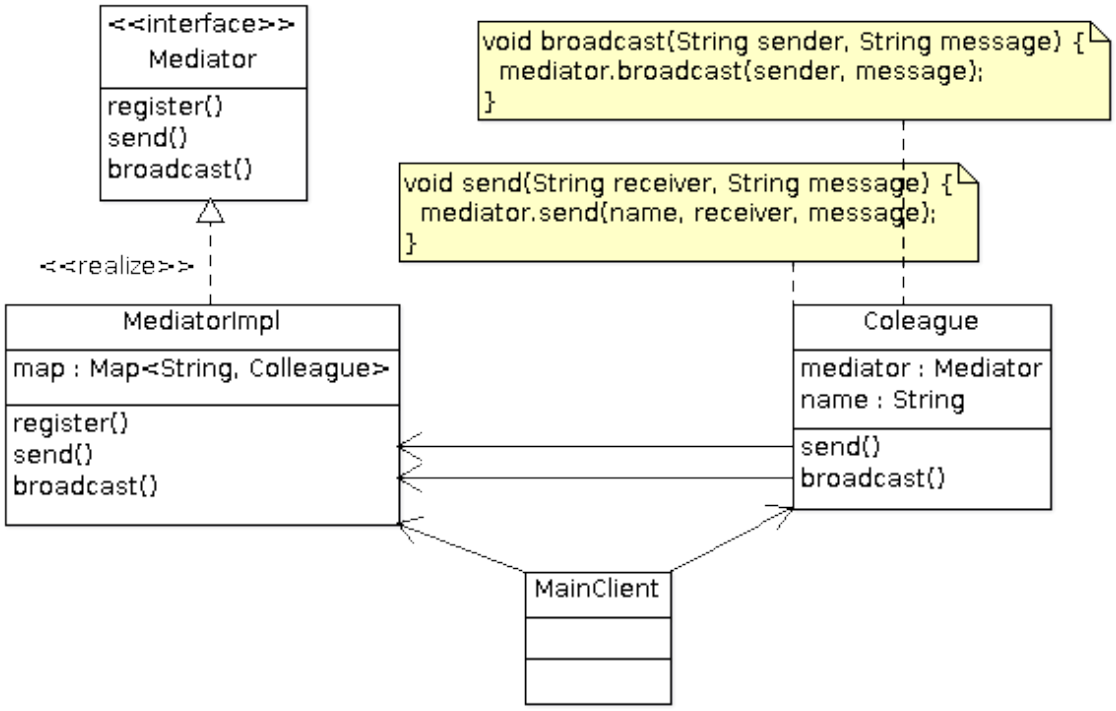
Supongamos que nos solicitan una pequeña aplicación para enviar mensajes entre los miembros de un equipo local de trabajo. Algo muy básico:

- Un usuario tiene que poder enviar un mensaje a otro. Para ello simplemente tiene que indicar el nombre del destinatario, escribir el mensaje y enviarlo.
- Cuando un usuario recibe un mensaje tiene que poder ver el nombre del emisor del mensaje.
- Cualquier usuario tiene que poder enviar un mensaje a todos los usuarios (multidifusión)

Este ejemplo pone de manifiesto una de las características más importante del patrón Mediator: cómo el Mediator al recibir la notificación de un sucedo desde un objeto

colega (quien envía un mensaje), se encarga de propagar el mensaje a otro o a todos los demás colegas. Este tipo de comunicación guarda mucha relación con otro patrón de diseño que veremos más adelante: el patrón Observer.

La siguiente figura muestra las clases necesarias para nuestro ejemplo:



Comenzamos con el código. En este caso tenemos una interfaz Mediator y una clase que la implementa:

La interfaz define tres métodos cuya signatura deja muy clara su intención:

- register: Cualquier usuario de la aplicación tiene que registrarse en el Mediator para poder enviar y recibir mensajes.
- send(): Método que envía al destinatario el mensaje.
- broadcas(): Método que envía el mensaje a todos los usuarios.

Mediator.java

```
package comportamiento.mediator.mensajeria;


public interface Mediator {
    void register(Colleague colleague);
```

```
    void send(String sender, String receiver, String message);
    void broadcast(String sender, String message);
}
```

MediatorImpl.java

```
package comportamiento.mediator.mensajeria;

import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;

public class MediatorImpl implements Mediator {

    private Map<String, Colleague> map = new HashMap<String,
Colleague>();

    @Override
    public void register(Colleague colleague) {
        map.put(colleague.getName(), colleague);
    }

    @Override
    public void send(String sender, String receiver, String message) {
        Colleague colleague = map.get(receiver);
        if (colleague != null)
            notify(colleague, sender, message);
    }

    @Override
    public void broadcast(String sender, String message) {
        for (Entry<String, Colleague> colleague : map.entrySet()) {
```

```
            notify(colleague.getValue(), sender, message);
        }
    }

    private void notify(Colleague colleague, String sender, String
message) {
            System.out.println(colleague.getName() + " recibio de " +
sender
                    + " el mensaje \"" + message + "\"");
      }

}
```

Notad que el método notify() anterior es privado, esto es, no forma parte de la inteefaz pública de la clase.

Veamos ahora la clase que representa los usuarios. La clase es muy sencilla, ya que toda implementación está delegada en el Mediator:

- En su constructor recibe una referencia al Mediator y un String con el nombre del usuario. La referencia al Mediator sirve para que la clase colega se registre en él.
- Define dos métodos públicos: send() para enviar un mensaje a un usuario particular y broadcast() para enviar un mensaje a todos los usuarios.

Colleague.java

```
package comportamiento.mediator.mensajeria;


public class Colleague {


    protected Mediator mediator;
    private String name;


    protected Colleague(Mediator mediator, String name) {
```

```
        this.mediator = mediator;
        this.name = name;
        mediator.register(this);
    }

    public String getName() {
        return name;
    }

    public void send(String receiver, String message) {
        mediator.send(name, receiver, message);
    }

    public void broadcast(String sender, String message) {
        mediator.broadcast(sender, message);
    }
}
```

Por último, veamos una clase cliente que demuestra el uso del patrón:

MainClient.java

```
package comportamiento.mediator.mensajeria;
public class MainClient {

      public static void main(String[] args) {
            Mediator mediator = new MediatorImpl();
            Colleague userOne = new Colleague(mediator, "Alicia");
            Colleague userTwo = new Colleague(mediator, "Robert");
            userOne.send("Robert", "Esta tarde tenemos reunion a las
17:00");
```

```
            userTwo.send("Alicia", "Pues no me va muy bien.");
            userOne.broadcast("Alicia", "Ya tengo vacaciones!!!");
      }
}
```

Salida:

```
Robert recibio de Alicia el mensaje "Esta tarde tenemos reunion a las 17:00"
Alicia recibio de Robert el mensaje "Pues no me va muy bien."
Alicia recibio de Alicia el mensaje "Ya tengo vacaciones!!!"
Robert recibio de Alicia el mensaje "Ya tengo vacaciones!!!"
```

Ejemplo 2 – Control del flujo de ejecución de una aplicación

El patrón Mediator se suele utilizar en el diseño de GUI's. En el siguiente ejemplo veremos una simulación de aplicación web en la que cada una de la páginas web informa al Mediator de su estado. El Mediator por su parte, dado que contiene la lógica que define el flujo de ejecución, siempre sabe qué página debe mostrarse en cada momento del flujo web según el estado enviado por cada página. El Mediator evalúa el valor del estado, determina la siguiente página a mostrar y a continuación llama al método go() de tal página.

Comenzamos por ver el código de la clase Mediator. El Mediator conecta el flujo entre páginas. Para ello crea las cuatro páginas, pasando en el constructor de cada una la referencia de si mismo. De esta manera cada página podrá acceder a él.

Mediator.java

```
package comportamiento.mediator.tienda;

import static comportamiento.mediator.tienda.Accion.*;

public class Mediator {
      BienvenidaPag bienvenidaPag;
      CatalogoPag catalogoPag;
      ComprarPag comprarPag;
      SalirPag salirPag;
```

```
	public Mediator() {
		bienvenidaPag = new BienvenidaPag(this);
		catalogoPag = new CatalogoPag(this);
		comprarPag = new ComprarPag(this);
		salirPag = new SalirPag(this);
	}

	public void handle(Accion estado) {
		if (estado.equals(bienvenida_catalogo)) {
			catalogoPag.go();
		} else if (estado.equals(catalogo_comprar)) {
			comprarPag.go();
		} else if (estado.equals(comprar_salir)) {
			salirPag.go();
		} else if (estado.equals(bienvenida_salir)) {
			salirPag.go();
		} else if (estado.equals(catalogo_salir)) {
			salirPag.go();
		} else if (estado.equals(comprar_salir)) {
			salirPag.go();
		}
	}

	public BienvenidaPag getBienvenida() {
		return bienvenidaPag;
	}
}
```

Del código anterior hay que destacar el método handle(), en el que se implementa la lógica que controla el flujo entre páginas, es decir, el código que si no se aplicara este

patrón de diseño, estaría distribuido por cada una de las páginas web, redundando en una aplicación de difícil comprensión y mantenimiento.

Continuamos con el código de cada una de las clases colegas, que en el caso del ejemplo se trata de cada una de las páginas web.

BienvenidaPag.java

Esta clase simula una página de bienvenida en la que se le pregunta al usuario si desea ver el catálogo de productos o salir de la aplicación.

Lo siguiente es común (extrapolable) para esta y para el resto de clases colegas que se presentarán a continuación:

- La selección efectuada por el usuario se pasa como parámetro al objeto Mediator para que este pueda determina la siguiente página a mostrar.
- Es importante observar que para que la clase BienvenidaPag acceda al Mediator hemos tenido que pasar la referencia del objeto Mediator en el constructor de BienvenidaPag.

```
package comportamiento.mediator.tienda;

import static comportamiento.mediator.tienda.Accion.*;
import java.util.Scanner;

public class BienvenidaPag {
	Mediator mediator;
	String respuesta = "n";
	public BienvenidaPag(Mediator m) {
		mediator = m;
	}

	public void go() {
		System.out.print("¿Desea ver el catalogo de productos
[s/n]? ");
```

```
            Scanner scan = new Scanner(System.in);
            respuesta = scan.next();
            if (respuesta.equals("s")) {
                  mediator.handle(bienvenida_catalogo);
            } else {
                  mediator.handle(bienvenida_salir);
            }
      }
}
```

CatalogoPag.java

Esta clase simula una página con un catálogo de artículos que el usuario puede seleccionar para comprarlos.

```
package comportamiento.mediator.tienda;

import static comportamiento.mediator.tienda.Accion.*;

import java.util.Scanner;

public class CatalogoPag {
      Mediator mediator;
      String respuesta = "n";

      public CatalogoPag(Mediator m) {
            mediator = m;
      }

      public void go() {
            System.out.print("¿Listo para comprar [s/n]? ");
            Scanner scan = new Scanner(System.in);
```

```
            respuesta = scan.next();
            if (respuesta.equals("s")) {
                  mediator.handle(catalogo_comprar);
            } else {
                  mediator.handle(catalogo_salir);
            }
      }
}
```

ComprarPag.java

Esta clase representa una página que permite que el usuario pueda hacer efectiva la compra del artículo seleccionado.

```
package comportamiento.mediator.tienda;

import static comportamiento.mediator.tienda.Accion.*;
import java.util.Scanner;

public class ComprarPag {
      Mediator mediator;
      String respuesta = "n";

      public ComprarPag(Mediator m) {
            mediator = m;
      }

      public void go() {
            System.out.print("¿Comprar ahora el articulo [s/n]? ");
            Scanner scan = new Scanner(System.in);
            respuesta = scan.next();
            if (respuesta.equals("s")) {
```

```
                System.out.println("Gracias por la compra.");
            }
            mediator.handle(comprar_salir);
        }
}
```

SalirPag.java

Clase en la que termina el flujo de la aplicación.

```
package comportamiento.mediator.tienda;

public class SalirPag {
        Mediator mediator;

        public SalirPag(Mediator m) {
            mediator = m;
        }

        public void go() {
            System.out.println("Vuelva pronto a visitarnos!");
        }
}
```

Veamos ahora la enum que define los diferentes estado de la aplicación:

Accion.java

```
package comportamiento.mediator.tienda;

public enum Accion {
        bienvenida_catalogo,
        bienvenida_salir,
```

```
	catalogo_comprar,
	catalogo_salir,
	comprar_salir
}
```

Por último, veamos una clase cliente que ponga a prueba el conjunto de clases anterior:

MainClient.java

```
package comportamiento.mediator.tienda;

public class MainClient {
	public static void main(String args[]) {
		new MainClient();
	}

	public MainClient() {
		Mediator mediator = new Mediator();
		mediator.getBienvenida().go();
	}
}
```

Notad que en el código anterior lo primero que se hace es crear una instancia de Mediator. Recordad que el constructor de Mediator crea una instancia por cada página web.

Una vez que se tiene la instancia de mediator se ejecutan el método getBienvenida(), el cual devuelve una referencia de la clase BienvenidaPag y sobre la que se invoca el método go(), que ejecuta lógica específica de esa clase. Esto es una manera conveniente de disponer de un punto de entrada al flujo del programa.

Veamos ahora algunos caminos de ejecución:

Por ejemplo, lo siguiente sucede cuando el usuario inicia el programa y decide no acceder al catálogo:

```
¿Desea ver el catalogo de productos [s/n]? n
Vuelva pronto a visitarnos!
```

En el siguiente caso el usuario accede al catálogo pero no quiere comprar ningún producto:

```
¿Desea ver el catalogo de productos [s/n]? s
¿Listo para comprar [s/n]? n
Vuelva pronto a visitarnos!
```

En este otro caso, el usuario decide en el último momento no comprar el artículo seleccionado:

```
¿Desea ver el catalogo de productos [s/n]? s
¿Listo para comprar [s/n]? s
¿Comprar ahora el articulo [s/n]? n
Vuelva pronto a visitarnos!
```

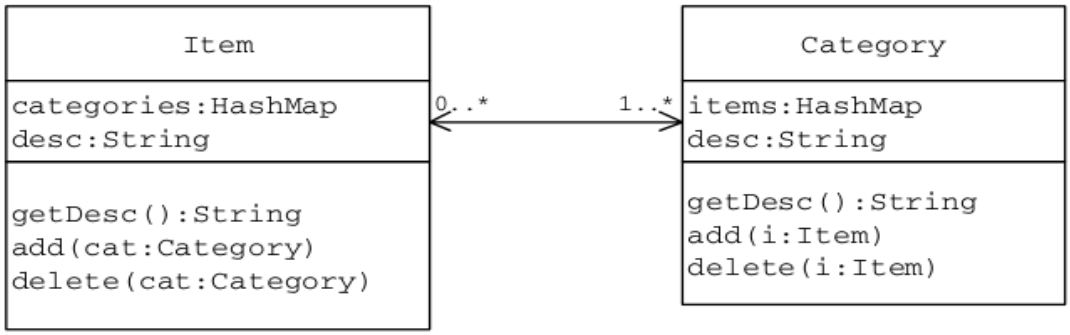
En el último ejemplo de ejecución, vemos que el usuario sí compra un artículo:

```
¿Desea ver el catalogo de productos [s/n]? s
¿Listo para comprar [s/n]? s
¿Comprar ahora el articulo [s/n]? s
Gracias por la compra.
Vuelva pronto a visitarnos!
```

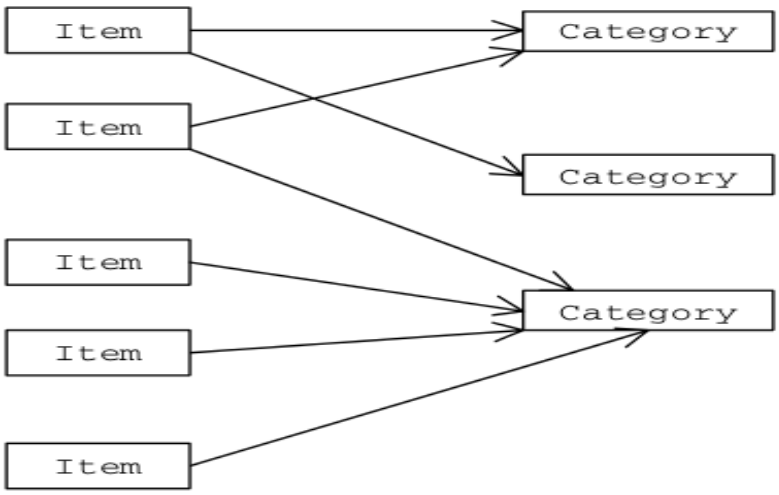
Lo importante de este ejemplo, además de identificar a cada uno de los componentes del patrón, es observar que cuando cualquiera de las páginas recibe una respuesta del usuario, tal respuesta se comunica inmediatamente al Mediator, el cuál conoce la siguiente página a mostrar.

Ejemplo 3 – Manejo de asociaciones item-categorías

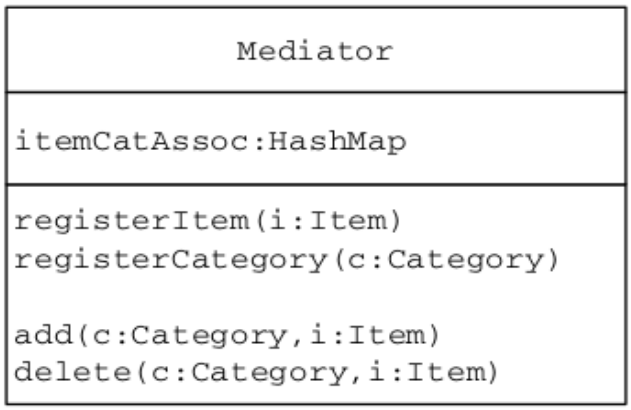
Supongamos que tenemos que crear una aplicación que permita añadir y borrar elementos a una base de datos (utilizaremos estructuras de datos en memoria para no alargar el ejemplo). Un elemento cualquiera puede pertenecer a una o varias categorías. Una manera de resolver el problema de la relación entre elementos y categorías, podría consistir en que cada elemento mantuviera una lista con todas las categorías a las que pertenece. Esta aproximación queda reflejada en la siguiente figura:



El problema de un diseño como el descrito es que las interacciones entre los objetos del programa pueden volverse complicadas de controlar. La siguiente imagen muestra un escenario tal, en el que tanto elementos como categorías se relacionan los unos con los otros:



Una buena manera de eliminar la interacción directa entre elementos y categorías consiste en extraer los detalles de la interacción de los propios objetos (Item y Category) y llevarlos a una nueva clase Mediator, como la de la siguiente figura:



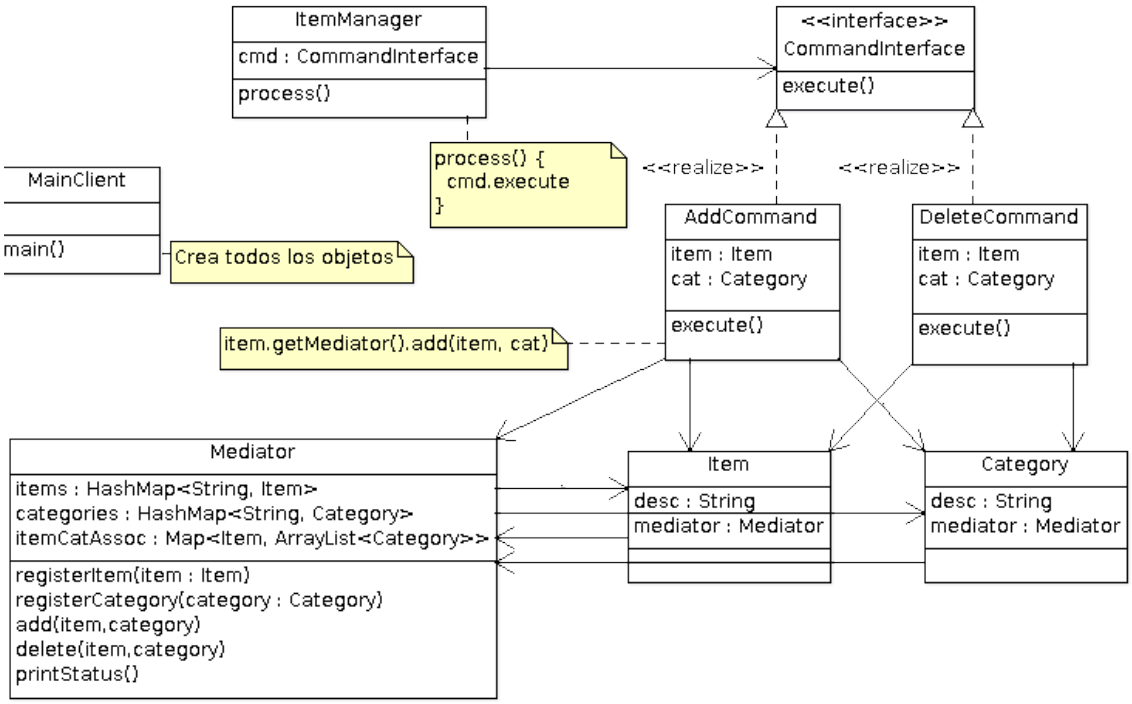
Como se aprecia en la imagen anterior, la clase Mediator puede diseñarse con los dos conjuntos de métodos siguientes:

- Una serie de métodos que permitan tanto a los elementos como a las categorías poder registrarse en el Mediator.
- Un conjunto de métodos para añadir y eliminar elementos. El Mediator es responsable de la implementación de las interacciones entre diferentes objetos como parte de estos métodos.

La clase Mediator puede mantener la asociación entre elementos y categorías en una variable de instancia. De esta manera, los objetos Item no tienen que referirse directamente a los objetos Category, es decir, un Item no necesita mantener la lista de categorías a las que pertenece, ni viceversa.

De manera similar, las operaciones add() y delete() no tiene que implementarse ni en la clase Item ni en la clase Category, sino que serán operaciones implementadas en el Mediator. Por otro lado, para ejecutar tales operaciones, podemos usar el patrón Command (por qué no), es decir, podemos tener la clase AddCommand para crear una nueva asociación elemento-categoría y una clase DeleteCommand para eliminarla. La implementación de estas clases será muy trivial, pues se limitarán a invocar a los respectivos métodos del Mediator.

El siguiente diagrama muestra las clases empleadas y sus interacciones:



Notad que en el diagrama anterior no hay comunicación directa entre clases colegas, lo cual es característico del patrón Mediator.

Pasemos a ver el código fuente. Comenzamos por las clases que conforman el patrón Command:

Primero la interfaz:

Command.java

```
package comportamiento.mediator.items_categorias;

public interface Command {

  public void execute();

}
```

Ahora las clases que la implementan:

AddCommand.java

```
package comportamiento.mediator.items_categorias;


public class AddCommand implements Command {
```

```
	Item item;
	Category cat;

	public AddCommand(Item i, Category c) {
		item = i;
		cat = c;
	}

	public void execute() {
		item.getMediator().add(item, cat);
	}
}
```

Notad que el constructor recibe como parámetros el elemento y la categoría a añadir. Esto implica que un objeto comando queda vinculado a una par Item-Category específicos. Por otro lado, fijaos que el método execute() la operación add() se invoca sobre la referencia del Mediator, la cual es obtenida a través de Item.

La clase comando que se encarga de eliminar asociaciones es muy parecida:

DeleteCommand.java

```
package comportamiento.mediator.items_categorias;

public class DeleteCommand implements Command {
	Item item;
	Category cat;

	public DeleteCommand(Item i, Category c) {
		item = i;
		cat = c;
	}
```

```
	public void execute() {
		item.getMediator().delete(item, cat);
	}
}
```

Por último en lo que respecta al patrón Command, tenemos una clase invoker para ejecutar los objetos comando:

ItemManager.java

```
package comportamiento.mediator.items_categorias;

public class ItemManager {
	Command command;

	public ItemManager setCommand(Command c) {
		command = c;
		return this;
	}

	public void process() {
		command.execute();
	}
}
```

Veamos ahora las clases Item y Category. Notad como en el constructor de la clase Item (también pasa lo mismo con Category), además de recibir la descripción, se recibe también el Mediator, el cual se utiliza para registrar el item (o la categoría) en el Mediator:

Item.java

```
package comportamiento.mediator.items_categorias;

public class Item {
	private Mediator mediator;
	private String desc;

	public Item(String s, Mediator mediator) {
		desc = s;
		this.mediator = mediator;
		this.mediator.registerItem(this);
	}

	public String getDesc() {
		return desc;
	}

	public Mediator getMediator() {
		return mediator;
	}

	@Override
	public String toString() {
		return desc;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
```

```
            result = prime * result + ((desc == null) ? 0 :
desc.hashCode());
            return result;
      }

      @Override
      public boolean equals(Object obj) {
            if (this == obj)
                  return true;
            if (obj == null)
                  return false;
            if (getClass() != obj.getClass())
                  return false;
            Item other = (Item) obj;
            if (desc == null) {
                  if (other.desc != null)
                        return false;
            } else if (!desc.equals(other.desc))
                  return false;
            return true;
      }

}
```

Category.java

```
package comportamiento.mediator.items_categorias;

public class Category {

      private String desc;
```

```
	public Category(String s, Mediator mediator) {
		desc = s;
		mediator.registerCategory(this);
	}

	public String getDesc() {
		return desc;
	}

	@Override
	public String toString() {
		return desc;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((desc == null) ? 0 :
desc.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
```

```
            Category other = (Category) obj;
            if (desc == null) {
                  if (other.desc != null)
                        return false;
            } else if (!desc.equals(other.desc))
                  return false;
            return true;
      }

}
```

Veamos ahora la clase reina: el Mediator. Esta clase declara tres mapas: uno para almacenar los elementos, otro para hacer lo propio con las categorías y un tercero para guardar por cada elemento una lista con sus categorías.

Respecto a los métodos, tenemos:

- Un par que se utilizan para permitir registrar elementos y categorías, respectivamente.
- Un par que permiten, respectivamente, crear y eliminar una relación entre un elemento y sus categorías.
- Un par que se encargan de eliminar elementos y categorías, respectivamente. Estos métodos vendrían a ser los opuestos a los métodos que registran elementos y categorías.
- Finalmente, existe un método que recorre las estructuras tipo mapa para mostrar su contenido.

Mediator.java

```
package comportamiento.mediator.items_categorias;

import java.util.ArrayList;
import java.util.HashMap;
```

```
import java.util.Map;
import java.util.Map.Entry;

public class Mediator {

	private HashMap<String, Item> items;
	private HashMap<String, Category> categories;
	private Map<Item, ArrayList<Category>> itemCatAssoc;

	public Mediator() {
		items = new HashMap<String, Item>();
		categories = new HashMap<String, Category>();
		itemCatAssoc = new HashMap<Item, ArrayList<Category>>();
	}

	public void registerItem(Item item) {
		if (items.containsKey(item.getDesc())) {
			throw new RuntimeException("El item " +
item.getDesc() + " ya existe");
		}

		items.put(item.getDesc(), item);
		System.out.println("Registrado el item: " +
item.getDesc());
	}

	public void registerCategory(Category category) {
		if (categories.containsKey(category.getDesc())) {
			throw new RuntimeException("La categoria " +
category.getDesc() + " ya existe");
		}
```

```
            categories.put(category.getDesc(), category);
            System.out.println("Registrada la categoria: " +
category.getDesc());
      }

      public void add(Item item, Category category) {
            ArrayList<Category> listCat = itemCatAssoc.get(item);
            if (listCat == null) { // No existe el item
                  System.out.println("No existe en la relacion el item
" + item.getDesc() +
                              ". Creando la categoria " +
category.getDesc() +
                              " y asociandolos.");
                  listCat = new ArrayList<Category>();
                  listCat.add(category);
                  itemCatAssoc.put(item, listCat);
            } else {
                  System.out.println("Añadiendo nueva asociacion: " +
                              item.getDesc() + " - " +
category.getDesc());
                  itemCatAssoc.get(item).add(category);
            }
      }

      public void delete(Item item, Category category) {

            for (Entry<Item, ArrayList<Category>> entry :
itemCatAssoc.entrySet()) {
                  if (entry.getValue().contains(category)) {
                        entry.getValue().remove(category);
                        System.out.println("Eliminada la relacion: " +
                                    item.getDesc() + " - " +
category.getDesc());
                        return;
```

```
                }
            }

            System.out.println("Error. No exite la relacion: " +
                        item.getDesc() + " - " + category.getDesc());

      }

      public void delete(Item item) {
            ArrayList<Category> listCat = itemCatAssoc.get(item);

            if (listCat == null) { // No existe el item en la relacion
                  items.remove(item.getDesc());
                  System.out.println("Eliminado el item: " +
item.getDesc());
            } else {
                  System.out.println("No se puede eliminar el item " +
item.getDesc() +
                              ", pues existe una relacion con
categorias.");
            }
      }

      public void delete(Category category) {
            for (Entry<Item, ArrayList<Category>> entry :
itemCatAssoc.entrySet()) {
                  if (entry.getValue().contains(category)) {
                        System.out.println("No se puede eliminar la
categoria " +
                                    category.getDesc() + ", pues
existen item que la relacionan.");
                        return;
                  }
            }
```

```
            categories.remove(category.getDesc());
      }

      public void printStatus() {

            System.out.println("Items existentes:");
            for (Entry<String, Item> entry : items.entrySet()) {
                  System.out.println(entry.getKey());
            }

            System.out.println();
            System.out.println("Categorias existentes:");
            for (Entry<String, Category> entry :
categories.entrySet()) {
                  System.out.println(entry.getKey());
            }
            System.out.println();
            System.out.println("Relaciones existentes:");
            for (Entry<Item, ArrayList<Category>> entry :
itemCatAssoc.entrySet()) {
                  System.out.print(entry.getKey() + " -> ");

                  if (entry.getValue().isEmpty()) {
                        System.out.println("Sin categorias!");
                  } else {
                        for (Category cat : entry.getValue()) {
                              System.out.print(cat + " ");
                        }
                  }
                  System.out.println();
            }
      }
```

}

Para finalizar, vemos una clase cliente que muestre el uso del patrón Mediator. Notad cómo esta clase se encarga de instanciar a todas las clases vistas anteriormente.

MainClient.java

```
package comportamiento.mediator.items_categorias;


public class MainClient {


	private Mediator mediator = new Mediator();
	private ItemManager manager = new ItemManager(); // invoker


	public MainClient() {
		Item itemBeautiful = createItem("A Beautiful Mind");
		Category catCD = createCategory("CD");
		Command cmd = new AddCommand(itemBeautiful, catCD);
		manager.setCommand(cmd).process();


		Category catDVD = createCategory("DVD");
		cmd = new AddCommand(itemBeautiful, catDVD);
		manager.setCommand(cmd).process();


		Item itemDuet = createItem("Duet");
		cmd = new AddCommand(itemDuet, catCD);
		manager.setCommand(cmd).process();
		cmd = new AddCommand(itemDuet, catDVD);
		manager.setCommand(cmd).process();


		Item itemTutuuu = createItem("Tutuuu");
		Category catNewRealese = createCategory("New Releases");
		cmd = new AddCommand(itemTutuuu, catNewRealese);
```

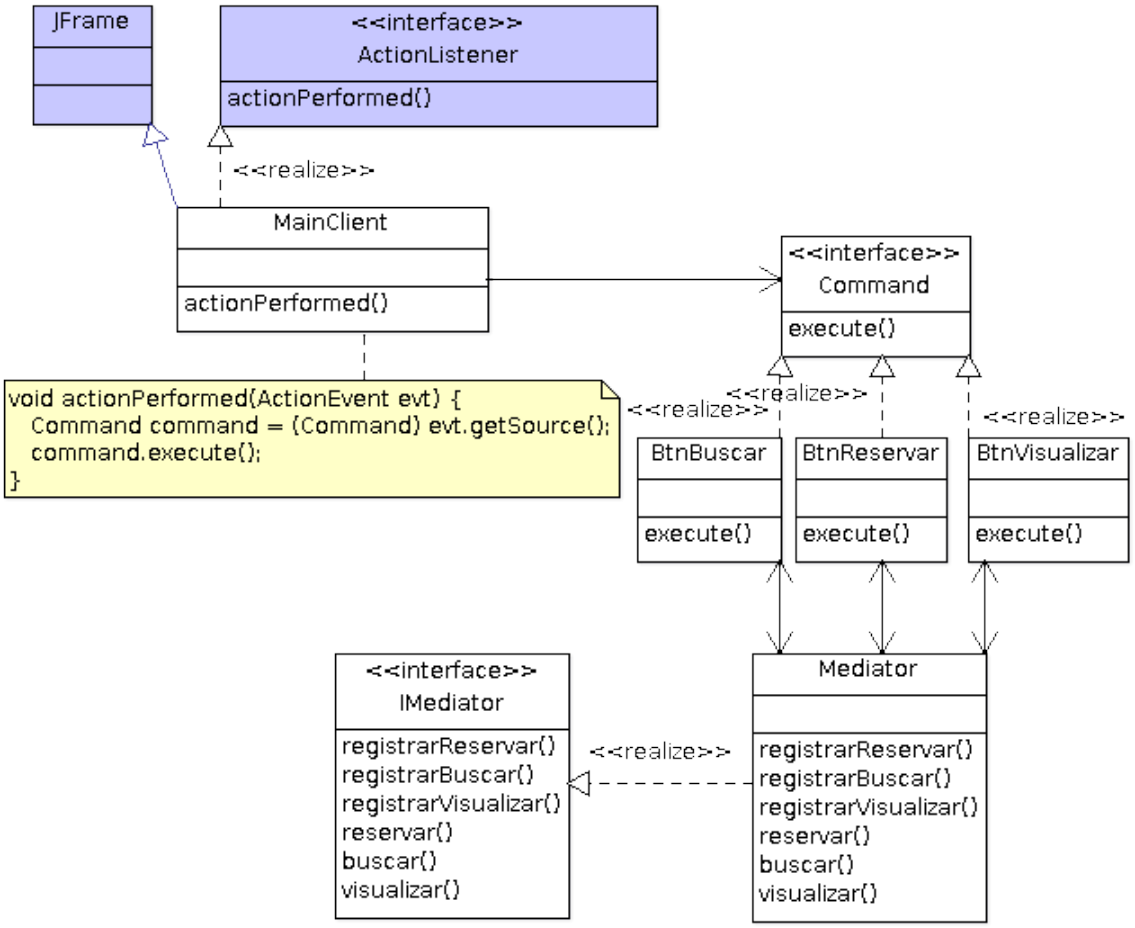
```
            manager.setCommand(cmd).process();

            // Delete the itemBeautiful from the DVD category
            cmd = new DeleteCommand(itemBeautiful, catDVD);
            manager.setCommand(cmd).process();

            mediator.printStatus();
      }

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

      private Category createCategory(String categoryName) {
            return new Category(categoryName, mediator);
      }

      private Item createItem(String itemName) {
            return new Item(itemName, mediator);
      }

}
```

Ejemplo 4 – Control de una interfaz gráfica

En este caso veremos cómo un objeto Mediator puede ser clave para controlar el correcto estado de una interfaz gráfica de usuario, a la vez que el código de los diversos componentes de GUI se mantiene simple y fácil de entender.

La interfaz gráfica que diseñaremos es muy sencilla, pero suficiente para ilustrar el uso del patrón. A partir de este ejemplo no resultaría difícil extrapolar lo visto a otro proyecto más complejo.

El diagrama siguiente muestra la relación de clases de la aplicación:



El enunciado queda como sigue:

Queremos una GUI con tres botones: Visualizar, Reservar y Buscar. Podemos suponer que tales botones permiten operar con una serie de hoteles. La existencia de botones ya sugiere la presencia de un patrón Command.

Cuando se inicia la aplicación, cualquiera de los tres botones puede ser pulsado, sin embargo, una vez que se pulse uno, éste tiene que quedar desactivado y lógicamente los otros dos deben quedar activados. La idea es que implementemos esta lógica en la clase Mediator, quedando liberadas las propias clases comando de tal lógica de control.

Veamos el código. Comenzamos por la jerarquía de comandos

Command.java

```
package comportamiento.mediator.gui;


public interface Command {
  void execute();
}
```

Las clases que implementan la interfaz Command siguen un esquema similar:

- Extienden de JButton e implementan Command.
- Al extender de JButton pueden registrarse como productores de eventos de tipo Action. Tales eventos se notificarán a los listeners correspondientes, que para nuestro caso es la clase que implementa la GUI (la cliente cliente).
- Al implementar Command, cada subclase comando debe definir un método execute(). Tal método delegará en la clase Mediator, pues ya se dijo que la lógica del control de la GUI estaría encapsulada en el Mediator.
- Definen un atributo IMediator para almacenar la referencia al Mediator recibida en su constructor. De esta manera pueden registrarse en el Mediator y que así éste pueda manejar las interacciones entre los diversos componentes de la GUI (activar tal botón, escribir cierto texto en una etiqueta, etc).

BtnBuscar.java

```
package comportamiento.mediator.gui;


import java.awt.event.ActionListener;
import javax.swing.JButton;


// Clase colega
public class BtnBuscar extends JButton implements Command {
```

```
      private static final long serialVersionUID = 1L;
      private IMediator mediator;

    public BtnBuscar(ActionListener oyente, IMediator mediator) {
        super("Buscar");
        addActionListener(oyente);
        this.mediator = mediator;
        this.mediator.registrarBuscar(this);
    }

    @Override
    public void execute() {
      mediator.buscar();
    }

}
```

BtnReservar.java

```
package comportamiento.mediator.gui;

import java.awt.event.ActionListener;
import javax.swing.JButton;

// Clase colega
public class BtnReservar extends JButton implements Command {

      private static final long serialVersionUID = 1L;
      private IMediator mediator;
```

```
    public BtnReservar(ActionListener oyente, IMediator mediator) {
        super("Reserva");
        addActionListener(oyente);
        this.mediator = mediator;
        this.mediator.registrarReservar(this);
    }

    @Override
    public void execute() {
      mediator.reservar();
    }

}
```

BtnVisualizar.java

```
package comportamiento.mediator.gui;

import java.awt.event.ActionListener;
import javax.swing.JButton;

// Clase colega
public class BtnVisualizar extends JButton implements Command {

      private static final long serialVersionUID = 1L;
      private IMediator mediator;

    public BtnVisualizar(ActionListener oyente, IMediator mediator) {
        super("Visualizar");
        addActionListener(oyente);
        this.mediator = mediator;
```

```
        this.mediator.registrarVisualizar(this);
    }

    @Override
    public void execute() {
      this.mediator.visualizar();
    }

}
```

Hay una cuatro componente gráfico: la etiqueta que mostrará el mensaje de inicio de la aplicación y, posteriormente, el correspondiente a cada botón seleccionado. La etiqueta lógicamente no es un comando, aunque sí debe registrarse en el Mediator, ya que su comportamiento también será controlado por éste.

Etiqueta.java

```
package comportamiento.mediator.gui;

import java.awt.Font;
import javax.swing.JLabel;

public class Etiqueta extends JLabel {

      private static final long serialVersionUID = 1L;
      public Etiqueta(IMediator mediator) {
        super("Aplicacion iniciada...");
        mediator.registrarEtiqueta(this);
        setFont(new Font("Arial", Font.BOLD, 24));
    }
 }
```

Veamos ahora el código para el Mediator. Primero la interfaz:

IMediator.java

```java
package comportamiento.mediator.gui;

//Abstract Mediator
public interface IMediator {
    void reservar();
    void visualizar();
    void buscar();
    void registrarReservar(BtnReservar objReservar);
    void registrarVisualizar(BtnVisualizar objVisualizar);
    void registrarBuscar(BtnBuscar objBuscar);
    void registrarEtiqueta(Etiqueta objEtiqueta);
}
```

Mediator.java

```java
package comportamiento.mediator.gui;

//Concrete mediator
public class Mediator implements IMediator {

    BtnVisualizar btnVisualizar;
    BtnBuscar btnBuscar;
    BtnReservar btnReservar;
    Etiqueta etiqueta;

    @Override
    public void registrarVisualizar(BtnVisualizar v) {
        btnVisualizar = v;
    }

    @Override
```

```
public void registrarBuscar(BtnBuscar b) {
    btnBuscar = b;
}

@Override
public void registrarReservar(BtnReservar r) {
    btnReservar = r;
}

@Override
public void registrarEtiqueta(Etiqueta e) {
    etiqueta = e;
}

@Override
public void reservar() {
    btnReservar.setEnabled(false);
    btnVisualizar.setEnabled(true);
    btnBuscar.setEnabled(true);
    etiqueta.setText("Reservando...");
}

@Override
public void visualizar() {
    btnVisualizar.setEnabled(false);
    btnBuscar.setEnabled(true);
    btnReservar.setEnabled(true);
    etiqueta.setText("Visualizando...");
}

@Override
```

```
    public void buscar() {
        btnBuscar.setEnabled(false);
        btnVisualizar.setEnabled(true);
        btnReservar.setEnabled(true);
        etiqueta.setText("Buscando...");
    }

}
```

Notad en el código anterior, que la implementación del Mediator es muy simple, ya que básicamente tenemos dos tipos de métodos: los que sirven para registrar componentes gráficos (y así luego poder referenciarlos) y los métodos que definen el comportamiento de toda la GUI cuando se selecciona un control gráfico.

Por último, veamos el código para el programa cliente, el cual tiene un papel importante, ya que:

- Se encarga de crear la GUI (extiende de JFrame).
- Es responsable de instanciar a todos los objetos del programa.
- Implementa la interfaz ActiionListener, por lo que define un método actionPerformed() para recibir los eventos de tipo ActionEvent acaecidos en la GUI. Este código es particularmente interesante, ya que es genérico (mediante el patrón Command) y permite ejecutar el método execute() del objeto comando correspondiente.
- En su constructor se puede observar que cuando se crean los controles gráficos, para el caso de los botones pasa en el constructor de éstos una referencia de si misma (this) y la referencia al Mediator. La referencia this sirve para que los objetos comandos puedan registrar a la clase cliente como oyente de sus eventos de pulsación, es decir, que cuando se pulse alguno de los botones, tal botón pueda notificarlo a la clase cliente (al método actionPerformed).

MainClient.java

```java
package comportamiento.mediator.gui;

import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class MainClient extends JFrame implements ActionListener {

      private static final long serialVersionUID = 1L;
      private IMediator mediator = new Mediator();

    public MainClient() {
        JPanel p = new JPanel();
        p.add(new BtnVisualizar(this, mediator));
        p.add(new BtnReservar(this, mediator));
        p.add(new BtnBuscar(this, mediator));
        getContentPane().add(new Etiqueta(mediator), "North");
        getContentPane().add(p, "South");
        setSize(400, 200);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    @Override
    public void actionPerformed(ActionEvent evt) {
        Command command = (Command) evt.getSource();
        command.execute();
    }
```

```
    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            @Override
                  public void run() {
                        MainClient instancia = new MainClient();
                  instancia.setLocationRelativeTo(null);

                        instancia.setVisible(true);
                  }
            });
    }

}
```

Problemas específicos e implementación

Omitir o incluir una clase abstracta para el Mediator.

Si las clases colegas van a trabajar sólo con una única clase Mediator, entonces no hay necesidad de crear una clase Abstracta Mediator, ya que no hay comportamiento común alguno.

Alternativas para implementar la comunicación entre el Mediator y colegas

- Para enviar mensajes al Mediator puede utilizarse simple delegación (las clases colegas mantienen una referencia al Mediator), a la vez que se define en el Mediator una interfaz pública de métodos adecuada: métodos para poder registrar colegas y métodos para notificar cambios en el estado de las clases colegas.
- Uno métodos muy empleado es el patrón Observer. En este caso, el Mediator es el Observer y los colegas implementan una interfaz Observable. Cada vez que se produce un cambio en el estado de un objeto Observable, el Observer (Mediator) es informado, encargándose de notificar al resto de colegas.

- En implementaciones complejas pueden definirse mensajes asíncronos que serán añadidos a una cola de mensajes. El Mediator selecciona mensajes de esta cola para su procesamiento.

Complejidad del Mediator

El Mediator maneja toda la interacción entre los objetos colegas. Debido a esto, un problema potencial es la complejidad que puede llegar a adquirir el Mediator cuando crece el número de objetos participantes.

Patrones relacionados

- Facade: El patrón Facade se diferencia del patrón Mediator en que abstrae la complejidad de un subsistema, proporcionando una interfaz más conveniente o fácil de utilizar. Su protocolo es unidireccional, es decir, Facade hace peticiones al subsistema pero no al revés. En contraste, Mediator es bidireccional, ya que las clases colegas envían mensajes al Mediator y éste a ellas.
- Observer: Es uno de los mecanismos que utiliza Mediator para realizar la comunicación de los participantes.