Ejercicios Patrones de Diseño

Patrón Factory Method

Consiste en extraer la funcionalidad de crear los objetos a una clase constructora (la factoría) que nos creará diferentes objetos.

```
public static void main(String[] args) {
	try {
	UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
		Config c=Config.getInstance();
		if (isLocal)
			facadeInterface=new FacadeImplementation();
		else {
			final String serverRMI = c.getServerRMI();
			// Remote service name
			String serviceName = "/"+c.getServiceRMI();
			//System.setSecurityManager(new RMISecurityManager());
			// RMI server port number
			int portNumber = Integer.parseInt(c.getPortRMI());
			// RMI server host IP IP
			facadeInterface = (ApplicationFacadeInterface)
Naming.lookup("rmi://"+ serverRMI + ":" + portNumber + serviceName);
			}
	}
	catch (Exception e) {
	//System.out.println(e.toString());
		e.printStackTrace();
	}
	int x=0;
	JFrame a = new StartWindow();
	a.setVisible(true);
}
```

En el código original, podemos ver que en función del valor de isLocal crea un tipo de lógica de negocio u otro.
Para aplicar el patrón Factory Method, lo primero que haremos será crear **una nueva clase Factory que será la encargada de crear una lógica de negocio u otra** en función del parámetro que le mandemos, si le mandamos un 1 creara una lógica de negocio local y si le mandamos un 2 una remota. De esta forma si queremos extender nuestra aplicación y añadir un nuevo tipo de lógica de negocio, solo tendremos que añadir una nueva condición en nuestra factoría.
Clases:

- Creator: BusinessLogicFactory
- Product: ApplicationFacadeInterface
- ConcreteProduct:
    - FacadeImplementation

```
public class BusinessLogicFactory {
	public ApplicationFacadeInterface getBusinessLogicFactory(int type)
			throws RemoteException, InstantiationException,
IllegalAccessException, ClassNotFoundException, SQLException,
MalformedURLException, NotBoundException{
```

```
                    Config c=Config.getInstance();
                    ApplicationFacadeInterface facadeInterface = null;

                    if(type == 1){//Si es 1 será local
                          facadeInterface=new FacadeImplementation();
                    }

                    else if (type == 2){//Si es 2 será distribuida
                          final String serverRMI = c.getServerRMI();
                          // Remote service name
                          String serviceName = "/"+c.getServiceRMI();
                          //System.setSecurityManager(new
RMISecurityManager());
                          // RMI server port number
                          int portNumber = Integer.parseInt(c.getPortRMI());
                          // RMI server host IP IP
                          facadeInterface = (ApplicationFacadeInterface)
Naming.lookup("rmi://"+ serverRMI + ":" + portNumber + serviceName);
                    }
                    return facadeInterface;
      }
}
```

Luego tendremos que modificar la clase StartWindow para que creé la lógica de negocio mediante la factoría

```
public static void main(String[] args) {
            try {
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
                  BusinessLogicFactory bf = new BusinessLogicFactory();

                  if (isLocal)
                        facadeInterface=bf.getBusinessLogicFactory(1);
                  else {
                        facadeInterface=bf.getBusinessLogicFactory(2);
                  }

            } catch (Exception e) {
            //System.out.println(e.toString());
                  e.printStackTrace();
            }
            int x=0;
            JFrame a = new StartWindow();
            a.setVisible(true);
      }
```

Patrón Iterador

Este patrón nos da una interfaz mediante la que podremos **acceder de forma secuencial a los objetos de una lista o colección**

Modificar el método getAllRuralHouses de FacadeImplementation...

```
public Vector<RuralHouse> getAllRuralHouses() throws RemoteException,
        Exception {
                ObjectContainer db=DB4oManager.getContainer();

                 try {
                        RuralHouse proto = new RuralHouse(0,null,null,null);
                        ObjectSet result = db.queryByExample(proto);
                        Vector<RuralHouse> ruralHouses=new Vector<RuralHouse>();
                        while(result.hasNext())
                                ruralHouses.add((RuralHouse)result.next());
                        return ruralHouses;
               } finally {
                   //db.close();
               }
        }
```

... Por la Siguiente Signatura

```
public ExtendedIterator<RuralHouse> ruralHouseIterator() throws
RemoteException,
        Exception {
                ObjectContainer db=DB4oManager.getContainer();

                 try {
                        RuralHouse proto = new RuralHouse(0,null,null,null);
                        ObjectSet result = db.queryByExample(proto);
                        Vector<RuralHouse> ruralHouses=new Vector<RuralHouse>();
                        while(result.hasNext()) {
                                ruralHouses.add((RuralHouse)result.next());
                        }
                        IteradorCasasRurales itrRuralHouses = new
IteradorCasasRurales(ruralHouses);
                        return itrRuralHouses;
               } finally {
                   //db.close();
               }
        }
```

Al realizar el cambio en la signatura, tendremos que realizar algunos cambios en las clases que utilizaban este método QueryAvaliabilityGUI & BookRuralHouseGUI. Estas clases tienen un método en el que esperan un vector de casas rurales, por lo que nos crearemos un método que coja todos las casas rurales que nos da el iterador y los guarde en un vector de casas rurales.

Lo primero que haremos será crear el método en la interfaz ApplicationFacadeInterface

```
public Vector<RuralHouse> getAllRuralHouses() throws RemoteException,
Exception;
```

Y luego lo implementamos en la clase FacadeImplementation

```
	public Vector<RuralHouse> getAllRuralHouses() throws RemoteException,
			Exception {
		Vector<RuralHouse> rhs = new Vector<RuralHouse>();
		ExtendedIterator<RuralHouse> itr = this.ruralHouseIterator();
		while(itr.hasNext()){
			rhs.add(itr.next());
		}
		return rhs;
	}
Ahora modificamos las clases que daban errores
public class BookRuralHouseGUI extends JFrame {
private void jbInit() throws Exception{
...
    ApplicationFacadeInterface facade=StartWindow.getBusinessLogic();
      //Vector<RuralHouse> ruralHouses=facade.ruralHouseIterator();
    Vector<RuralHouse> ruralHouses=facade.getAllRuralHouses();
      jComboBox1 = new JComboBox(ruralHouses);
...}
public class QueryAvailabilityGUI extends JFrame {
private void jbInit() throws Exception{
...
         ApplicationFacadeInterface facade=StartWindow.getBusinessLogic();
              //Vector<RuralHouse> rhs=facade.ruralHouseIterator();
         Vector<RuralHouse> rhs=facade.getAllRuralHouses();
           jComboBox1 = new JComboBox(rhs);
...
}
```

Para poder realizar esta modificación, lo primero que tenemos que hacer es crear el iterador de casas rurales. Para ello primero crearemos la interfaz Extended que nos proporcionará nuevas funcionalidades como recorrer la lista desde atrás hacia adelante y luego nuestro Iterador concreto será el que implemente esa interfaz

```
public interface ExtendedIterator <T> extends Iterator<T>{
	//devuelve el elemento actual y pasa al anterior
	 //public Object previous();
	 public T previous();

	 //true si existe el elemento anterior
	 public boolean hasPrevious();

	 //Se posiciona en el primer elemento
	 public void goFirst();

	 //Se posiciona en el último elemento
	 public void goLast();
}
```

Después creamos el Iterador de casas rurales e implementamos el que acabamos de crear

```
public class IteradorCasasRurales implements ExtendedIterator <RuralHouse>{
	Vector<RuralHouse> vector;
	int position = 0;

	public IteradorCasasRurales(Vector<RuralHouse> vector){
		this.vector = vector;
	}
	@Override
	public boolean hasNext() {
		if(position >= vector.size()){
			return false;
		}
		else{
			return true;
		}
	}
	@Override
	public RuralHouse next() {
		RuralHouse rh = vector.get(position);
		position = position + 1;
		return rh;
	}
	@Override
	public void remove() {
		vector.remove(position);

	}
	@Override
	public RuralHouse previous() {
		RuralHouse rh = vector.get(position);
		position = position - 1;
		return rh;
	}
	@Override
	public boolean hasPrevious() {
		if(position < 0){
			return false;
		}
		else{
			return true;
		}}
	@Override
	public void goFirst() {
		position = 0;

	}

	@Override
	public void goLast() {
		position = vector.size()-1;

	}
```

Ejemplo de Ejecución

Crearemos un método para ver como recorre el Iterador la lista de casas rurales, primero en orden y luego en orden inverso

```
public static void main(String[] args) {
		BusinessLogicFactory bf = new BusinessLogicFactory();
		try{
```

```
            ApplicationFacadeInterface facadeInterface =
bf.getBusinessLogicFactory(1);
            ExtendedIterator<RuralHouse>
i=facadeInterface.ruralHouseIterator();
            RuralHouse rh;
            System.out.println("Imprimimos las casas en orden inverso");
            i.goLast();
            while (i.hasPrevious()){
                  rh=i.previous();
                  rh.print();
            }
            System.out.println("Imprimimos las casas en orden");
            i.goFirst();
            while (i.hasNext()){
                  rh=i.next();
                  rh.print();
            }
            facadeInterface.close();
            }catch(Exception e){
                  System.out.println("Error");
            }}
```

Resultado de la Ejecución

```
kkkk
DataBase Initialized
Imprimimos las casas en orden inverso
Numero: 6Descripcion: Casa de veranoDueño: AnaCiudad: Donosti
Numero: 5Descripcion: CasaDueño: KevinCiudad: Vitoria
Numero: 4Descripcion: Casa de veranoDueño: KevinCiudad: Malaga
Numero: 3Descripcion: Casa de PlayaDueño: AlfredoCiudad: Benidorm
Numero: 2Descripcion: Eskiatzeko etxeaDueño: JonCiudad: Jaca
Numero: 1Descripcion: Ezkioko etxeaDueño: JonCiudad: Ezkio
Imprimimos las casas en orden
Numero: 1Descripcion: Ezkioko etxeaDueño: JonCiudad: Ezkio
Numero: 2Descripcion: Eskiatzeko etxeaDueño: JonCiudad: Jaca
Numero: 3Descripcion: Casa de PlayaDueño: AlfredoCiudad: Benidorm
Numero: 4Descripcion: Casa de veranoDueño: KevinCiudad: Malaga
Numero: 5Descripcion: CasaDueño: KevinCiudad: Vitoria
Numero: 6Descripcion: Casa de veranoDueño: AnaCiudad: Donosti
DataBase closed
```

Patrón Adapter

Queremos mostrar los datos de las casas rurales de un usuario en una tabla mediante un JTable. Esta clase tiene un constructor mediante el que le podemos pasar un Objeto de la clase TableModel pero los datos que nosotros podemos conseguir en la clase Owner (sin modificarla) son de tipo String e int, por tanto necesitaremos **adaptar esos datos para poder crear una tabla.** Para ello necesitamos una clase que haga la función de adaptador que nos coja los datos y nos genere un objeto del tipo TableModel que nos permita crear la tabla. Para ello lo primero que tenemos que hacer es crear la clase adaptadora que implemente la Interfaz TableModel e implementamos lo métodos que nos obliga dicha interfaz

```
public class OwnerAdapter implements TableModel{
	private Owner owner;
	private ArrayList<TableModelListener> observadores;

	public OwnerAdapter(Owner ow){
		owner = ow;
		observadores = new ArrayList<TableModelListener>();
	}

	@Override
	public void addTableModelListener(TableModelListener arg0) {
		observadores.add(arg0);
	}

	@Override
	public Class<?> getColumnClass(int arg0) {
		// TODO Auto-generated method stub
		switch (arg0){
		case 0: //Columna 0 el número de la casa es un int pero la clase será Integer
		return
((Integer)owner.getRuralHouses().elementAt(0).getHouseNumber()).getClass();
		case 1: //Columna 1 la descripación de la casa String
			return
(owner.getRuralHouses().elementAt(0).getDescription()).getClass();
		case 2: //Columna 2 la ciudad de la casa String
			return (owner.getRuralHouses().elementAt(0).getCity()).getClass();
		default: //Si no devolvemos la clase Object que es la más genérica
			return Object.class;
		}
	}

	@Override
	public int getColumnCount() {
		return 3; //RuralHouse number, description and city
	}

	@Override
	public String getColumnName(int arg0) {
		switch (arg0){
		case 0: //Columna 0 el número de la casa es un int pero la clase será Integer
			return "HouseNumber";
		case 1: //Columna 1 la descripación de la casa String
			return "HouseDescription";
		case 2: //Columna 2 la ciudad de la casa String
			return "HouseCity";
		default: //Si no devolvemos la clase Object que es la más genérica
			return "Column";
		}
	}

	@Override
	public int getRowCount() {
		return owner.getRuralHouses().size();
	}
```

```
	@Override
	public Object getValueAt(int row, int col) {
		switch (col){
		case 0: //Columna 0 el numero de la casa de la fila row
			return owner.getRuralHouses().get(row).getHouseNumber();
		case 1: //Columna 1 la descripación de la casa de la fila row
			return owner.getRuralHouses().get(row).getDescription();
		case 2: //Columna 2 la ciudad de la casa de la fila row
			return owner.getRuralHouses().get(row).getCity();
		default: //Si no devolvemos null
			return null;
		}
	}

	@Override
	public boolean isCellEditable(int arg0, int arg1) {
		return false;
	}

	@Override
	public void removeTableModelListener(TableModelListener arg0) {
		observadores.remove(arg0);
	}

	@Override
	public void setValueAt(Object param, int row, int col) {
		// TODO Auto-generated method stub
		switch (col){
		case 0: //Columna 0 el numero de la casa de la fila row
			owner.getRuralHouses().get(row).setHouseNumber((int) param);;
		case 1: //Columna 1 la descripación de la casa de la fila row
			owner.getRuralHouses().get(row).setDescription((String) param);;
		case 2: //Columna 2 la ciudad de la casa de la fila row
			owner.getRuralHouses().get(row).setCity((String) param);;
		default: //Si no devolvemos null
			System.out.println("No se ha podido realizar la operacion");
	}
	}
}
```

La interfaz TableModel nos obliga a implementar algunos métodos que luego utilizará la clase JTable para acceder a los datos y mostrarlos en una tabla. En esta clase adaptadora tenemos un atributo que es un Owner (un atributo del tipo de Objeto que queremos adaptar) del que obtendremos los datos con los diferentes métodos que nos obliga a implementar la Interfaz TableModel

Una vez creada la clase adaptadora, crearemos una nueva ventana para hacer ver como se visualizan los datos

```
public class VentanaTabla extends JFrame{
	private Owner ow;
	private JTable tabla;

	public VentanaTabla(Owner owner){
```

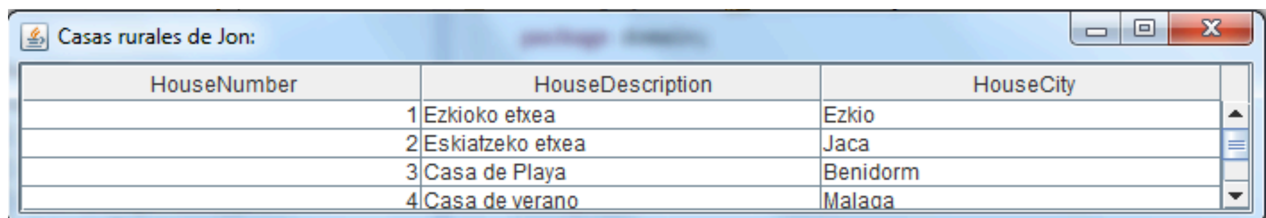
```
            super("Casas rurales de "+owner.getName()+":");
            this.setBounds(100, 100, 700, 100);
            ow = owner;
            OwnerAdapter adapt = new OwnerAdapter(ow);
            tabla = new JTable(adapt);
            tabla.setPreferredScrollableViewportSize(new Dimension(500,
70));
            //Creamos un JscrollPane y le agregamos la JTable
             JScrollPane scrollPane = new JScrollPane(tabla);
             //Agregamos el JScrollPane al contenedor
             getContentPane().add(scrollPane, BorderLayout.CENTER);
      }}
```

En esta clase Tenemos un constructor que tiene un Owner como parámetro, este será el que utilicemos para crear la clase adaptadora y utilizarla para crear la tabla JTable.

Finalmente creamos un método para ejecutar esa ventana

```
public static void main(String[] args) {
      try{
      FacadeImplementation fi = new FacadeImplementation();
      Owner ow = fi.getOwners().elementAt(0);
      fi.close();
      VentanaTabla vt = new VentanaTabla(ow);
      vt.setVisible(true);
      }catch (Exception e){
            e.printStackTrace();
      }
}
```

Y podemos ver el resultado de la ejecución



Patrón Observer

Al aplicar este patrón, tendremos una serie de objetos (**observadores)** que pueden estar **observando a uno o más objetos observables,** de manera **que cuando uno de estos cambie se lo notificará a los observadores** que estén esperando a que este haga cambios. En nuestro caso concreto podemos ver que el objeto observable es la casa rural, y los observadores son la agencia de viajes y el usuario particular. Estos estarán subscritos a una serie de casas, de manera que cuando se cree una oferta para una de esas casas aparecerá una ventana con los datos de la nueva oferta

Para aplicar este patrón, lo primero que haremos será hacer que la casa rural pueda ser observable, para ello haremos que extienda de la clase Observable

```
public class RuralHouse extends Observable implements Serializable {}
```

El objeto que es observable, cuando cambie tendrá que avisar a los que le están observando, y para ello primero usaremos el método setChanged() para que marque el objeto observable como que ha cambiado y luego el método notifyObservers() que se encargará de llamar al método update() de los observadores. La casa rural cambiará cuando se le añada una oferta, y entonces se le notificará a los observers que estén subscritos a esta casa rural
Para que el objeto observable pueda avisar de que ha habido cambios al observer, primero tendremos que subscribir el objeto como observador del observable (tendrá una lista de observadores a la que le añadiremos objetos observer)

```
public Offer createOffer(Date firstDay, Date lastDay, float price)  {
         Offer off=new Offer(this,firstDay,lastDay,price);
         offers.add(off);
         //Para el patron observer
          this.setChanged();
          this.notifyObservers();
         return off;
       }
```

Lo siguiente que tenemos que hacer es hacer que las clases AgenciaDeViajes y UsuarioParticular implementen la interfaz Observer. Esta interfaz, nos obliga a implementar el método update, que será al que llame el notifyObservers para avisarnos de que ha habido cambios, para ello usaremos el método addObserver(Observer o)

```
public class UsuarioParticular implements Observer{
       private String name;
       private Vector<Offer> ofertas;

       public UsuarioParticular(String name){
              this.name = name;
              ofertas = new Vector<Offer>();
       }
       public void activarAlerta(RuralHouse casa){
              casa.addObserver(this);
              System.out.println("Se ha registrado el observador");
       }

       @Override
```

```
	public void update(Observable rh, Object arg0) {
		RuralHouse casa = (RuralHouse) rh;
		Offer oferta = casa.offers.lastElement();
		ofertas.add(oferta);
		VentanaAvisoOferta vao = new VentanaAvisoOferta(oferta, name);
	}
}

public class AgenciaDeViajes implements Observer{

	private String nombre;
	private Vector<Offer> ofertas;

	public AgenciaDeViajes(String name){
		nombre = name;
	}


	public void activarAlerta(RuralHouse casa){
		casa.addObserver(this);
		System.out.println("Se ha registrado el observador");
	}

	@Override
	public void update(Observable rh, Object arg1) {
		RuralHouse casa = (RuralHouse) rh;
		Offer oferta = casa.offers.lastElement();

		ofertas.add(oferta);
		VentanaAvisoOferta vao = new VentanaAvisoOferta(oferta, nombre);
	}
}
```

En el método update() lo que hacemos es crear una nueva ventana en la que se visualizarán los datos de la oferta creada, de manera que casa vez que se creé una oferta para una de las casas a las que la AgenciaDeViajes o el UsuarioParticular estén subscritos, se creará una nueva ventana indicado que se ha creado una nueva oferta para ellos y los datos de esa oferta

```
public class VentanaAvisoOferta extends JFrame{
	private Offer oferta;
	public VentanaAvisoOferta(Offer oferta, String name){
		super("Aviso Oferta");
		this.oferta = oferta;
		this.setBounds(100, 100, 450, 100);
		this.setContentPane(new JPanel());
		((JComponent) this.getContentPane()).setBorder(new 
EmptyBorder(5, 5, 5, 5));
		this.getContentPane().setLayout(null);
		String ofert = "Se ha creado una nueva oferta para el usuario: 
"+name;
		JLabel lblNewLabel = new JLabel(ofert);
		lblNewLabel.setBounds(29, 26, 365, 14);
		String datos = "Oferta: "+oferta.getOfferNumber()
				+"Día Inicio: "+oferta.getFirstDay()
				+"Día Fin: "+oferta.getLastDay()
				+"Precio: "+oferta.getPrice();
		System.out.println(datos);
		JLabel lblNewLabel2 = new JLabel(datos);
		lblNewLabel2.setBounds(29, 40, 365, 14);

		this.getContentPane().add(lblNewLabel);
		this.getContentPane().add(lblNewLabel2);

		this.setVisible(true);
	}
}
```

Para ver el resultado creamos un método main en el que subscribimos a la casa1 un usuario y una agencia de viajes y a la casa 2 un usuario

```
public static void main(String args[]){
		ApplicationFacadeInterface facadeInterface= null;
		try {

	UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
			BusinessLogicFactory bf = new BusinessLogicFactory();
			if (isLocal)
				facadeInterface=bf.getBusinessLogicFactory(1);
			else {
				facadeInterface=bf.getBusinessLogicFactory(2);
			}
		} catch (Exception e) {
			System.out.println(e.toString());
			e.printStackTrace();
		}
		try{

		StartWindow sw = new StartWindow(facadeInterface);
		sw.setVisible(true);
		ExtendedIterator<RuralHouse> itr = 
facadeInterface.ruralHouseIterator();

		/*while(itr.hasNext()){
			System.out.println(itr.next().getHouseNumber());
		}*/
```

```
            Vector<RuralHouse> rhs = facadeInterface.getAllRuralHouses();

            RuralHouse casa = rhs.elementAt(0);
            RuralHouse casa2 = rhs.elementAt(1);
            UsuarioParticular up = new UsuarioParticular("Juan");
            AgenciaDeViajes av = new AgenciaDeViajes("Viajes el corte
Ingles");
            up.activarAlerta(casa);
            av.activarAlerta(casa);
            up.activarAlerta(casa2);

            }catch(Exception e){
                  e.printStackTrace();
            }
      }
```

Y si ejecutamos el programa, podemos ver que si creamos una oferta para la casa 1 aparece la siguiente ventana.