Universidad Politécnica de Madrid

Máster Universitario en Ingeniería Web

Patrones de diseño II

Jesús Bernal Bermúdez

Colecciones. java.util.Iterator<E>

- Es un objeto que te permite recorrer una colección
  - hasNext(). Devuelve true si existen más elementos
  - next(). Devuelve el siguiente elemento
  - remove(). Borrar el elemento de la colección. Es una acción opcional

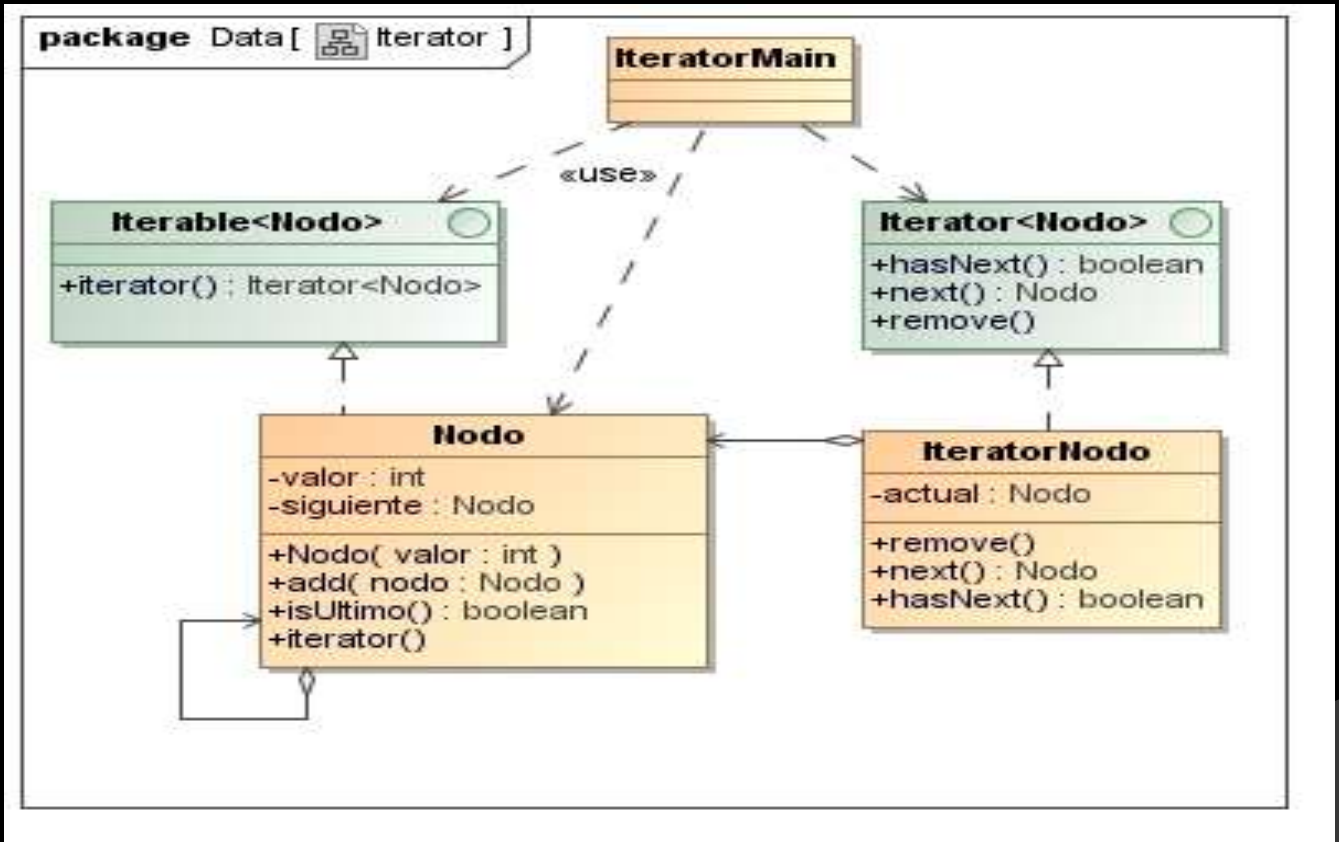
```
public final class PruebaIterator {
    private PruebaIterator() {}
    public static void main(String[] args) {
        Collection<String> lista = new ArrayList<String>();
        lista.add("uno"); lista.add(""); lista.add("dos"); lista.add("tres");
        IO.out.println("--------- for each------------");
        for (String item : lista) {
            IO.out.println(item);
        }
        IO.out.println("--------- Iterator------------");
        Iterator<String> it = lista.iterator();
        while (it.hasNext()) {
            String s = it.next();
            if ("".equals(s)) {
                it.remove(); //función opcional
            } else {
                IO.out.println(s);
            }
        }
    }
}
```

Iterator (Iterador)

- También conocido
  - Cursor
- Motivación
  - En las estructuras de datos, como por ejemplo un árbol, nos da una forma de recorrerlo sin conocer su estructura. Además, pueden existir varias formas de recorrerlo o poder realizar varios recorridos simultáneos. Todo esto nos permite procesar los datos en clases ajenas a lo datos en sí
- Propósito
  - Proporciona un modo de acceder secuencialmente a los elementos de un objeto sin exponer su representación interna
- Interface: java.lang.Iterable<T>, java.util.Iterator<E>

Iterator. Implementación

- Proporciona un modo de acceder secuencialmente a los elementos de un objeto sin exponer su representación interna



Iterator. Implementación

```
public class Nodo implements Iterable<Nodo> {
    private int valor;
    private Nodo siguiente;
    public Nodo(int valor) {
        this.valor = valor;
        this.siguiente = null;
    }
    public boolean isUltimo() {
        return this.siguiente == null;
    }
    public int getValor() {
        return this.valor;
    }
    public Nodo getSiguiente() {
        return siguiente;
    }
    public void insertar(Nodo subNodo) {
        Nodo anterior = this.siguiente;
        this.siguiente = subNodo;
        subNodo.siguiente = anterior;
    }
    @Override
    public Iterator<Nodo> iterator() {
        return new IteratorNodo(this);
    }
}
```

Iterator. Implementación

```
public class IteratorNodo implements Iterator<Nodo> {
    private Nodo actual;
    public IteratorNodo(Nodo actual) {
        this.actual = actual;
    }
    @Override public boolean hasNext() {
        return this.actual != null;
    }
    @Override public Nodo next() {
        if (!this.hasNext()) {
            throw new NoSuchElementException();
        }
        Nodo ant = this.actual;
        this.actual = this.actual.getSiguiente();
        return ant;
    }
    @Override public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Iterator. Test

```
public class NodoTest {
    Nodo n0, n1;
    @Before public void ini() {
        this.n0 = new Nodo(0);
        this.n1 = new Nodo(1);
        this.n0.insertar(this.n1);
    }
    @Test
    public void testIsUltimoCierto() {
        assertTrue(this.n1.isUltimo());
    }
    @Test
    public void testIsUltimoFalso() {
        assertFalse(this.n0.isUltimo());
    }
    @Test
    public void testInsertar() {
        assertSame(this.n0.getSiguiente(), n1);
    }
    @Test
    public void testInsertarIntermedio() {
        Nodo n2 = new Nodo(2);
        n0.insertar(n2);
        assertSame(n2.getSiguiente(), n1);
    }
    @Test
    public void iterador() {
        Iterator<Nodo> iti = n0.iterator();
        int v = 0;
        while (iti.hasNext()) {
            assertEquals(v++, iti.next().getValor());
        }
        assertEquals(2, v);
    }
}
```

Iterator. Ejercicio. Iterator<Integer>

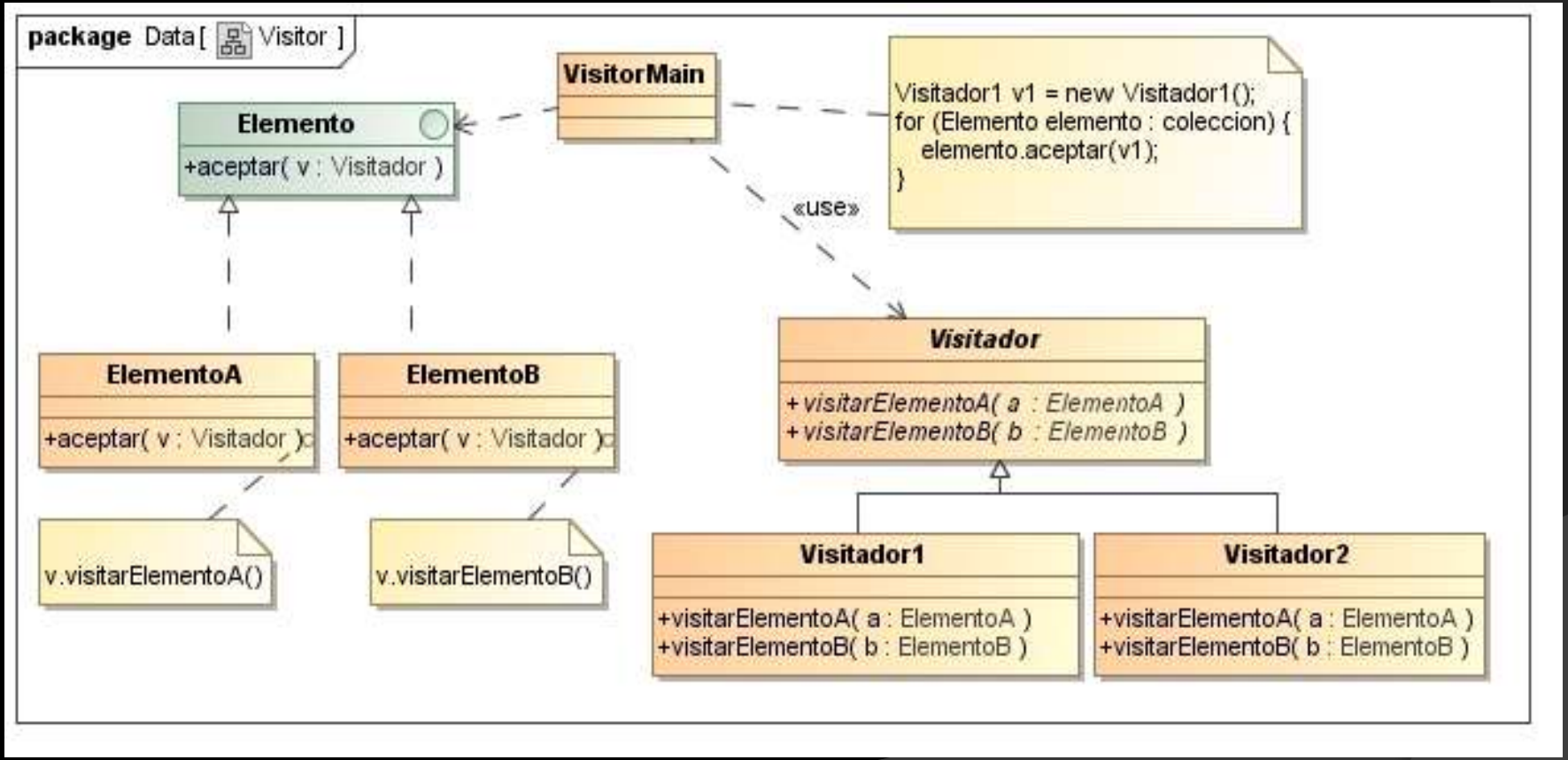
- Mejorar el ejercicio anterior para realizarlo:
  - Iterator<Integer>

Visitor (Visitante)

- Motivación
  - Dada un compilador con programas representados con arboles sintácticos. Se necesitan realizar múltiples operaciones sobre los datos y se pretende separar las operaciones de los datos
- Propósito
  - Definir un conjunto de operaciones sobre una estructura de datos de forma independiente

Visitor. Implementación

- Definir un conjunto de operaciones sobre una estructura de datos de forma independiente



Visitor. Implementación

```
public interface Elemento {
    void aceptar(Visitador v);
}
```

```
public class ElementoA implements Elemento {
    private String atributoA;

    public ElementoA() {
        this.setAtributoA("A");
    }

    public String getAtributoA() {
        return atributoA;
    }

    public void setAtributoA(String atributoA) {
        this.atributoA = atributoA;
    }

    @Override
    public void aceptar(Visitador v) {
        v.visitarElementoA(this);
    }
}
```

Visitor. Implementación

```
public class ElementoB implements Elemento {
    private String atributoB;
    public ElementoB() {
        this.setAtributoB("B");
    }
    public String getAtributoB() {
        return atributoB;
    }
    public void setAtributoB(String atributoB) {
        this.atributoB = atributoB;
    }
    @Override
    public void aceptar(Visitante v) {
        v.visitarElementoB(this);
    }
}
```

```
public interface Visitante {
    void visitarElementoA(ElementoA e);

    void visitarElementoB(ElementoB e);
}
```

Visitor. Implementación

```
public class Visitador1 implements Visitador {
    @Override
    public void visitarElementoA(ElementoA e) {
        IO.out.println("Visitador 1 --> elemento: " + e.getAtributoA());
    }
    @Override
    public void visitarElementoB(ElementoB e) {
        IO.out.println("Visitador 1 --> elemento: " + e.getAtributoB());
    }
}
```

```
public final class MainVisitor {
    private Collection<Elemento> coleccion = new ArrayList<Elemento>();
    public MainVisitor() {
        coleccion.add(new ElementoA()); coleccion.add(new ElementoA());
        coleccion.add(new ElementoB()); coleccion.add(new ElementoA());
    }
    public void visitador1() {
        Visitador1 v1 = new Visitador1();
        for (Elemento elemento : coleccion) {
            elemento.aceptar(v1);
        }
    }
    public static void main(String[] args) {
        IO.in.addController(new MainVisitor());
    }
}
```

Visitor. Ejercicio. Visitor2

- Ampliar el ejemplo anterior creando el Visitor2. Este cuenta las veces que aparecen los diferentes elementos

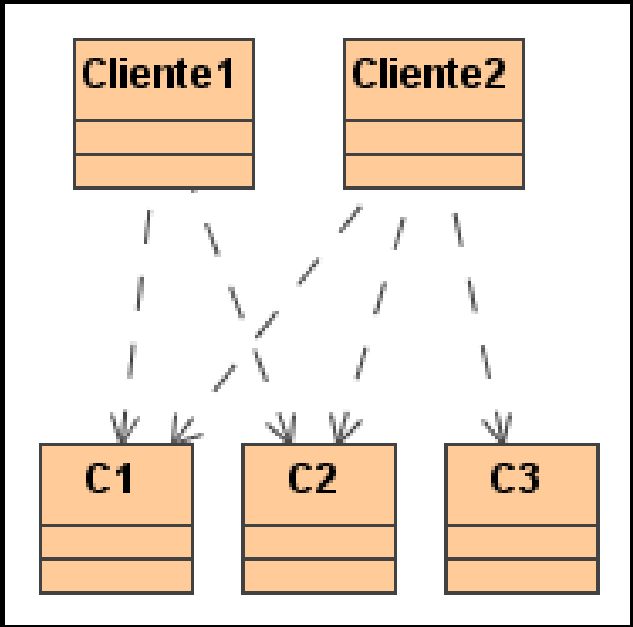
```
public class Visitador2Test {
    private Collection<Elemento> coleccion = new ArrayList<Elemento>();
    @Before
    public void ini() {
        coleccion.add(new ElementoA());
        coleccion.add(new ElementoA());
        coleccion.add(new ElementoB());
        coleccion.add(new ElementoA());
        coleccion.add(new ElementoB());
    }
    @Test
    public void testVisitorAs() {
        Visitador2 v2 = new Visitador2();
        for (Elemento elemento : coleccion) {
            elemento.aceptar(v2);
        }
        assertEquals(3, v2.getAs());
    }
    @Test
    public void testVisitorBs() {
        Visitador2 v2 = new Visitador2();
        for (Elemento elemento : coleccion) {
            elemento.aceptar(v2);
        }
        assertEquals(2, v2.getBs());
    }
}
```

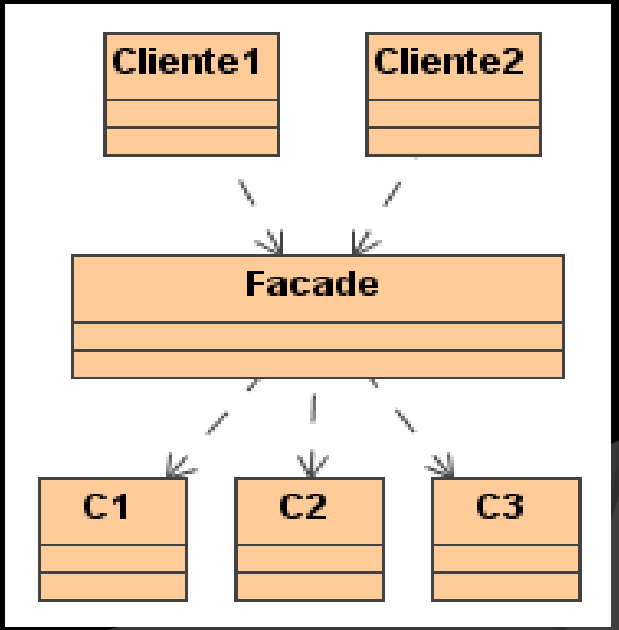
Facade (Fachada)

- Motivación
  - Dado un subsistema complejo, se quiere ofertar una interface única y simplificada que ayude a dar servicios generales
  - Cuando se quiera estructurar varios subsistemas en capas, y se requiere simplificar el punto de entrada en cada nivel
- Propósito
  - Proporciona un interface unificado para un conjunto de interfaces de un subsistema

Facade (Fachada)

- Propósito
  - Proporciona un interface unificado para un conjunto de interfaces de un subsistema

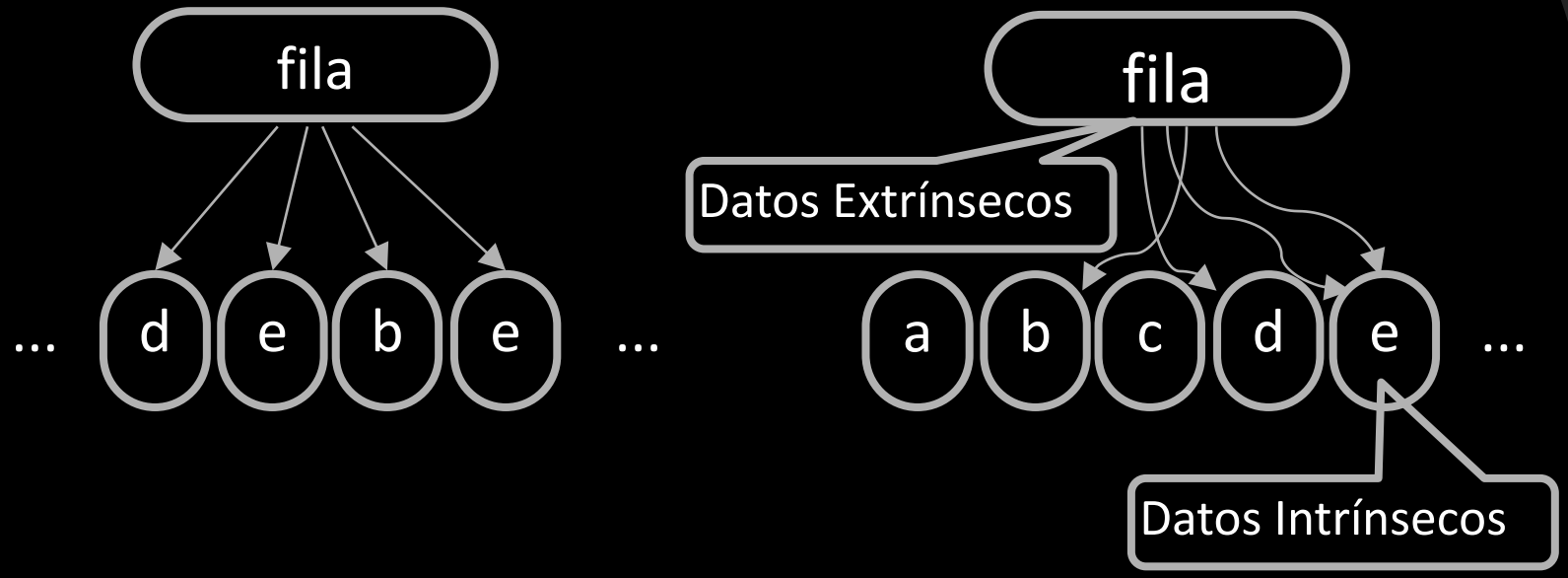




Flyweight (Peso ligero)

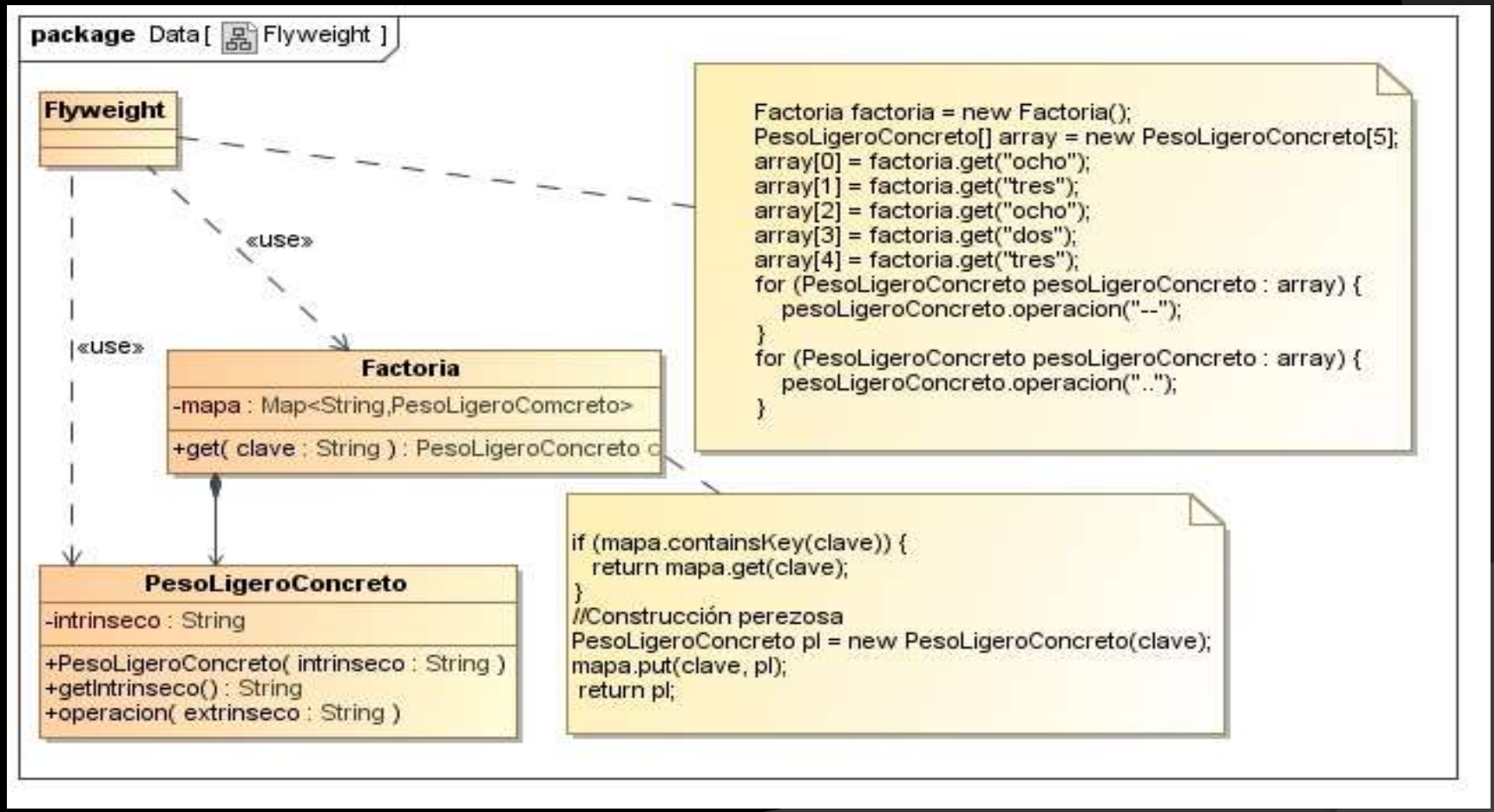
- Motivación
  - Eliminar o reducir la redundancia cuando tenemos gran cantidad de objetos que contienen información idéntica
  - En un editor de texto se podría plantear los caracteres como objetos, pero su número hace inviable esta solución. Una alternativa es plantear los caracteres como referencias de objetos carácter compartidos. Se distinguen los datos intrínsecos que son compartidos y se sitúan en el objeto compartido y los datos extrínsecos dependientes del contexto
- Propósito
  - Compartir objetos de grado fino de forma eficiente

Flyweight. Descripción



Flyweight. Implementación

- Compartir objetos de grado fino de forma eficiente



Flyweight. Implementación

```
public class Factoria {
    private final Map<String, PesoLigeroConcreto> mapa =
        new HashMap<String, PesoLigeroConcreto>();
    public PesoLigeroConcreto get(String clave) {
        if (mapa.containsKey(clave)) {
            return mapa.get(clave);
        }
        //Construcción perezosa
        PesoLigeroConcreto pl = new PesoLigeroConcreto(clave);
        mapa.put(clave, pl);
        return pl;
    }
}
```

```
public class PesoLigeroConcreto {
    private String intrinseco;
    public PesoLigeroConcreto(String intrinseco) {
        this.intrinseco = intrinseco;
    }
    public String getIntrinseco() {
        return this.intrinseco;
    }
    public void operacion(String extrinseco) {
        IO.out.println(extrinseco + this.intrinseco + extrinseco);
    }
}
```

Flyweight. Implementación

```
public final class FlyweightMain {
    public static void main(String[] args) {
        Factoria factoria = new Factoria();
        PesoLigeroConcreto[] array = new PesoLigeroConcreto[5];
        array[0] = factoria.get("ocho");
        array[1] = factoria.get("tres");
        array[2] = factoria.get("ocho");
        array[3] = factoria.get("dos");
        array[4] = factoria.get("tres");

        for (PesoLigeroConcreto pesoLigeroConcreto : array) {
            pesoLigeroConcreto.operacion("--");
        }

        for (PesoLigeroConcreto pesoLigeroConcreto : array) {
            pesoLigeroConcreto.operacion("..");
        }

    }
}
```

Ejercicio. Printer

- Se parte del interface Printer, y se disponen de tres implementaciones PrinterA, printerB y printerC
- Se pretende tener un solo objeto por tipo de impresora y deben estar accesibles para toda la aplicación
- Realizar una creación temprana
- Realizar el diseño de clases en UML
- Crear las clases en Java

```
public interface Printer {
    public abstract void print(String msg);
}

public class PrinterA implements Printer {
    @Override
    public void print(String msg) {
        IO.out.println("PrinterA >>> " + msg);
    }
}
```

Ejercicio. Texto (Singleton, Composite y Flyweight)

- Se pretende realizar un mini editor de texto (se aplican 3 patrones). Es un texto que se compone por párrafos y éste por caracteres
  - Clase ***Caracter***: representa un carácter
    - Se pretender un objeto carácter, por cada carácter diferente. La clave para recuperarlo es el char que representa el carácter (***FactoriaCaracter.getCaracter(char c)*** ;)
    - Se puede imprimir el carácter normal o forzado a mayúsculas
  - Clase ***Parrafo***: contiene un conjunto de caracteres
    - Se pueden añadir y borrar caracteres. Si se intenta añadir otro párrafo o texto se debe lanzar una ***UnsupportedOperationException***
    - Se puede forzar a escribirlo en mayúsculas
    - Se puede imprimir el párrafo completo. Al final se añade un salto de línea
  - Clase ***Texto***: es un conjunto de párrafos o de otros textos
    - Se pueden añadir o borrar párrafos o textos. Si se intenta añadir un carácter se debe lanzar una ***UnsupportedOperationException***
    - Se puede forzar a escribirlo en mayúsculas
    - Se puede imprimir el texto completo. Al final se añade un salto de línea y una raya de separador
  - Clase ***Componente***. Representa un ***Caracter***, un ***Parrafo*** o un ***Texto***. Si se intenta añadir a un ***Componente*** de tipo ***Carácter*** otro Componente de tipo Carácter, debe ignorarse
- Realizar el diseño de clases en UML
- Crear las clases de Java

Ejercicio. Texto. TextoTest(1-2)

```
public class TextoTest {
    private Componente H, o, l, a, pf, txt;
    @Before  public void ini() {
        H = FactoriaCaracter.getFactoria().get('H');
        o = FactoriaCaracter.getFactoria().get('o');
        l = FactoriaCaracter.getFactoria().get('l');
        a = FactoriaCaracter.getFactoria().get('a');
        pf = new Parrafo();
        pf.add(H);
        pf.add(o);
        pf.add(l);
        pf.add(a);
        txt = new Texto();
        txt.add(pf);
        txt.add(pf);
    }

    @Test public void testCaracterFlyweight() {
        assertSame(H, FactoriaCaracter.getFactoria().get('H'));
    }
    @Test public void testCaracterAddCaracter() {
        H.add(o);
    }
    @Test public void testDibujarCaracterNormal() {
        assertEquals("o", o.dibujar(false));
    }
    @Test public void testDibujarCaracterMayusculas() {
        assertEquals("O", o.dibujar(true));
    }
```

Ejercicio. Texto. TextoTest(2-2)

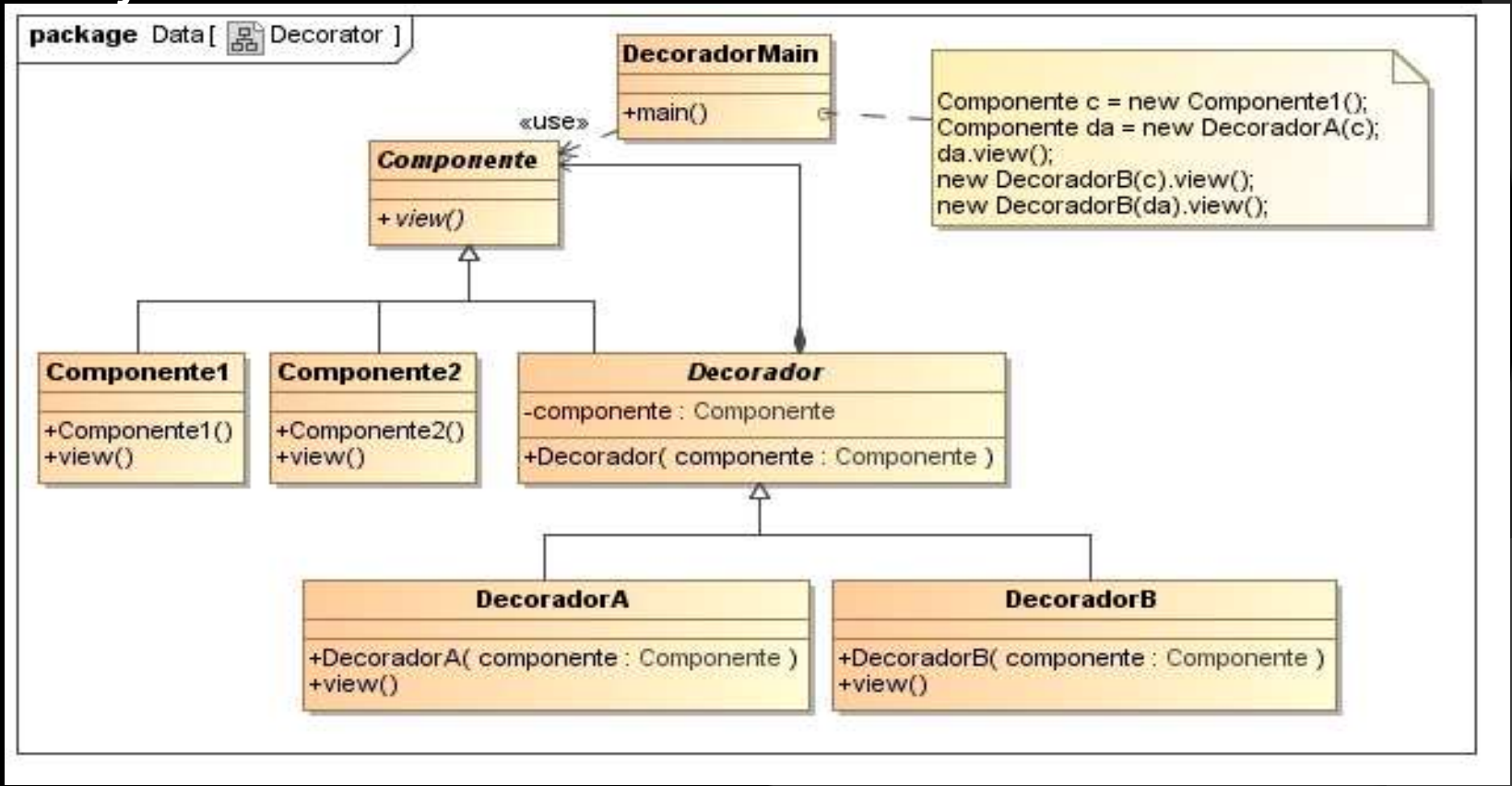
```
    @Test public void testParrafoNormal() {
        assertEquals("Hola\n", pf.dibujar(false));
    }
    @Test public void testParrafoMayusculas() {
        assertEquals("HOLA\n", pf.dibujar(true));
    }
    @Test public void testParrafoNoAddParrafo() {
        try {
            pf.add(pf);
            fail("");
        } catch (UnsupportedOperationException ignored) {
            ignored.toString();
        }
    }
    // ------------------------------------------------
    @Test public void testTextoNormal() {
        assertEquals("Hola\nHola\n---o---\n",txt.dibujar(false));
    }
    @Test public void testTextoMayusculas() {
        assertEquals("HOLA\nHOLA\n---o---\n",txt.dibujar(true));
    }
    @Test public void testTextoNoAddCaracter() {
        try {
            txt.add(H);
            fail();
        } catch (UnsupportedOperationException ignored) {
            ignored.toString();
        }
    }
}
```

Decorator (Decorador)

- También conocido
  - Wrapper (Envoltorio)
- Motivación
  - Se pretende añadir una nueva responsabilidad a un objeto, pero no a su clase. Un ejemplo sería añadir barras de Scroll a un componente visual
- Propósito
  - Asigna responsabilidades de forma dinámica a objetos

Decorator. Implementación

- Asigna responsabilidades de forma dinámica a objetos



Decorator. Implementación

```
public abstract class Componente {
    public abstract void view();
}
```

```
public class Componente1 extends Componente {
    @Override
    public void view() {
        IO.out.print("Concreto");
    }
}
```

```
public abstract class Decorador extends Componente {
    private Componente c;
    public Decorador(Componente c) {
        this.c = c;
    }
    public Componente getC() {
        return c;
    }
}
```

Decorator. Implementación

```
public class DecoradorA extends Decorador {
    public DecoradorA(Componente c) {
        super(c);
    }
    @Override public void view() {
        IO.out.print(">>> ");
        this.getC().view();
    }
}
```

```
public class DecoradorB extends Decorador {
    public DecoradorB(Componente c) {
        super(c);
    }
    @Override public void view() {
        IO.out.println();
        IO.out.println("***************");
        this.getC().view();
        IO.out.println();
        IO.out.println("***************");
    }
}
```

Decorator. Implementación

```
public final class MainDecorator {
    public static void main(String[] args) {
        Componente c = new Componente1();
        Componente da = new DecoradorA(c);
        da.view();
        new DecoradorB(c).view();
        new DecoradorB(da).view();
    }
}
```

Decorator. Ejercicio Vehiculo

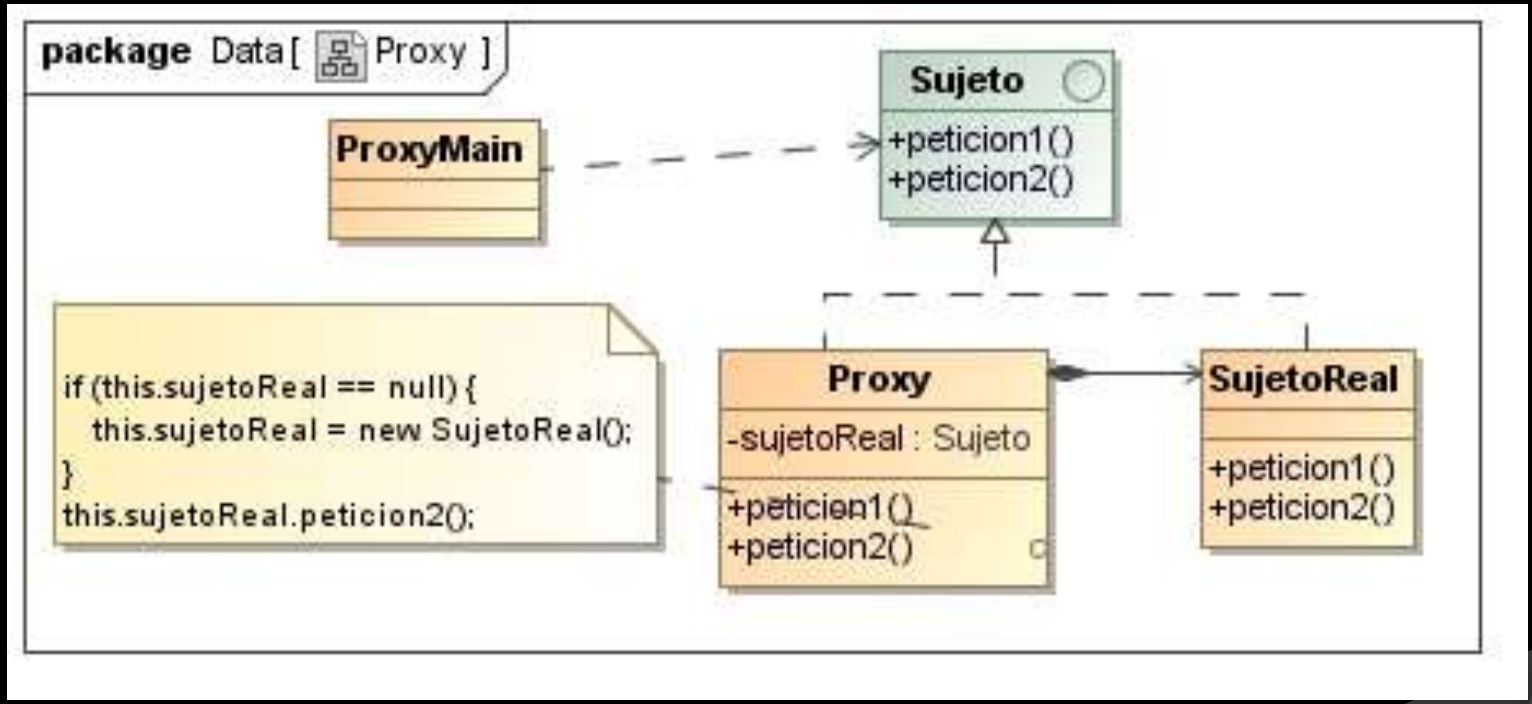
- Se pretende realizar la gestión de vehículos. Se Parte de un vehículo con modelo y precio
- A los diferentes vehículos se les puede añadir extras: GPS, MP3, EDS... Cada extra tiene un precio y una descripción
- Finalmente, al objeto se le puede consultar su descripción (debe informar de los extras incorporados) y el precio final
  - public String getDescripcion();
  - Public int getPrecio();

Proxy (Apoderado)

- También conocido: Surrogate (Sustituto)
- Motivación. Supongamos que el coste de un objeto es muy grande. Se pretende retrasar la creación del mismo con otro objeto que puede responder a ciertas peticiones más sencillas
- Propósito. Se proporciona un objeto más versátil como representante de otro objeto
- Aplicabilidad
  - Proxy virtual
  - Proxy remoto. Un representante local de otro remoto
  - Proxy de protección. Se controla el acceso al objeto original
  - Referencia inteligente. Se realizan operaciones adicionales

Proxy. Implementación

- Se proporciona un objeto representante de otro objeto



Proxy. Implementación

```
public abstract class Sujeto {
    public abstract void peticion();

    public abstract void peticion2();
}
```

```
public class SujetoReal extends Sujeto {

    @Override
    public void peticion() {
        IO.out.println("Peticion 1: responde el real");
    }

    @Override
    public void peticion2() {
        IO.out.println("Peticion 2: responde el real");
    }
}
```

Proxy. Implementación

```
public class Proxy extends Sujeto {
    private Sujeto sujetoReal;
    public Proxy() {
        this.sujetoReal = null;
    }

    @Override
    public void peticion() {
        if (this.sujetoReal == null) {
            this.sujetoReal = new SujetoReal();
        }
        this.sujetoReal.peticion();
    }

    @Override
    public void peticion2() {
        if (this.sujetoReal == null) {
            IO.out.println("Peticion 2: responde proxy");
        } else {
            this.sujetoReal.peticion2();
        }
    }

}
```

Proxy. Ejercicio ServerLog

- Ser parte de la clase Server. Dispone del método service() que nos proporciona un servicio
- Se pretende crear un log cada vez que se accede al servidor con la siguiente información:
  - Fecha de acceso
    - *new java.util.Date()*
  - Tiempo de respuesta del servidor en ms
    - new *java.util.Date().getTime()* - new *java.util.Date().getTime()*

Proxy. Ejercicio ServerLog

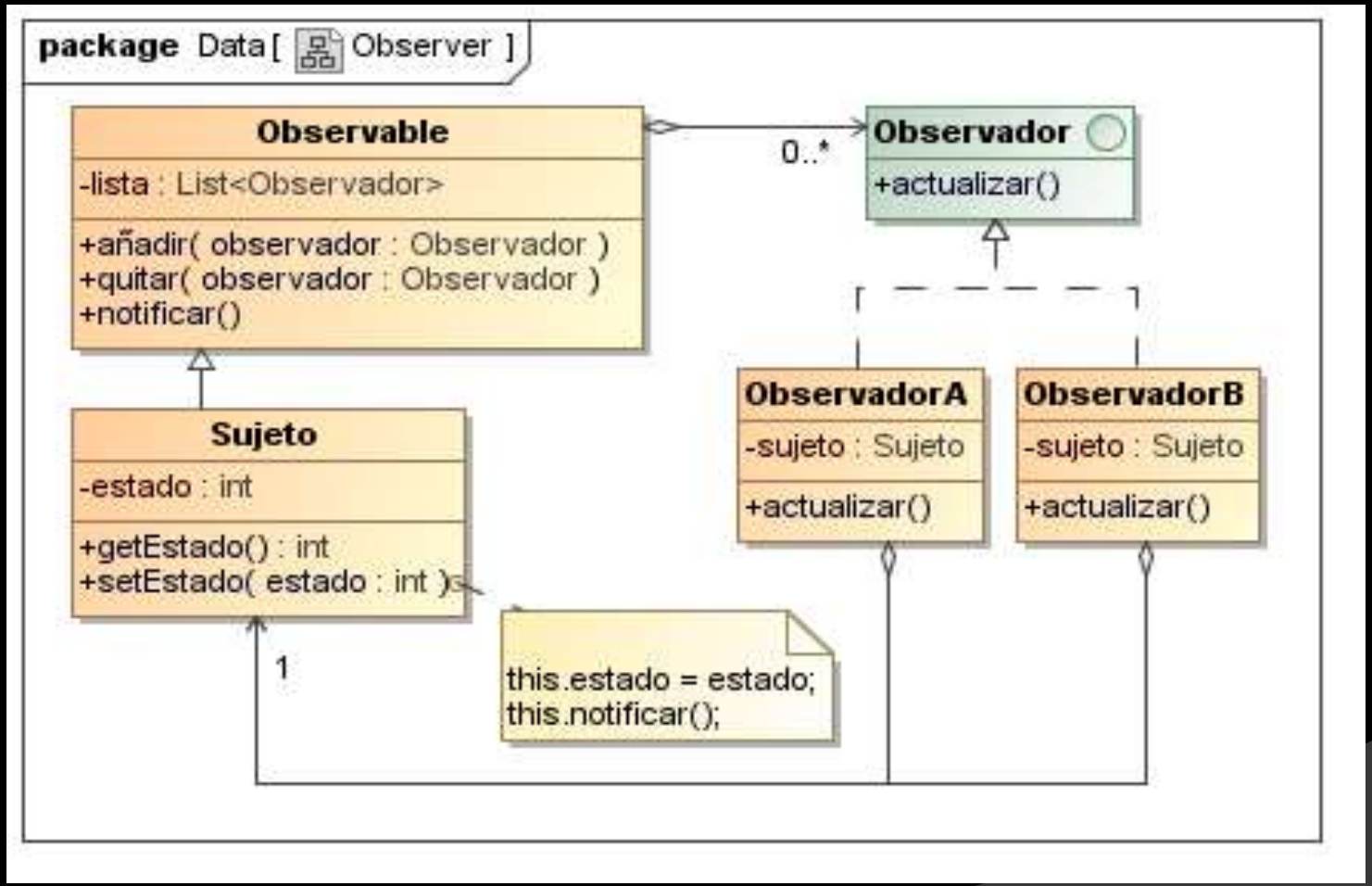
```
public interface Service {
    public void service();
}
================================================================================
public class Server implements Service{
    private Random random;

    public Server() {
        this.random = new Random();
    }

    public void service() {
        try {
            Thread.sleep(Math.round(this.random.nextDouble() * 3000));
        } catch (InterruptedException ignored) {
            ignored.toString();
        }
    }
}
```

```
[Server:service] Thu Sep 13 09:28:42 CEST 2012 >>> 1486ms
[Server:service] Thu Sep 13 09:28:43 CEST 2012 >>> 1037ms
[Server:service] Thu Sep 13 09:28:44 CEST 2012 >>> 2639ms
[Server:service] Thu Sep 13 09:28:47 CEST 2012 >>> 2003ms
...
```

Observer (Observador)

- También conocido
  - Dependents (Dependiente), Publish-Subscribe (Publicar-Suscribir)
- Propósito
  - Se define una dependencia entre uno a muchos, del tal manera, que cuando cambie avise a todos los objetos dependientes
- Motivación
  - Muchas veces se separa los datos en si de su representación, pudiendo tener varias representaciones de un mismo dato

Observer. Implementación



Observer. Implementación

```
public interface Observador {
    void actualizar();
}
```

```
public class Observable {
    private final Set<Observador> lista = new HashSet<Observador>();
    public void aniadir(Observador observador) {
        this.lista.add(observador);
    }
    public void quitar(Observador observador) {
        this.lista.remove(observador);
    }
    public void notificar() {
        for (Observador observador : lista) {
            observador.actualizar();
        }
    }
}
```

```
public class Sujeto extends Observable {
    private int estado;
    public int getEstado() {
        return estado;
    }
    public void setEstado(int estado) {
        this.estado = estado;
        this.notificar();
    }
}
```

Observer. Implementación

```
public class ObservadorA implements Observador {
    private Sujeto sujeto;
    public ObservadorA(Sujeto sujeto) {
        this.sujeto = sujeto;
        this.sujeto.aniadir(this);
    }
    @Override
    public void actualizar() {
        IO.out.println("A: sujeto a cambiado: " + this.sujeto.getEstado());
    }
}
```

```
public final class MainObserver {
    public static void main(String[] args) {
        Sujeto sujeto = new Sujeto();
        new ObservadorA(sujeto);
        new ObservadorB(sujeto);
        IO.out.println("Leo sujeto... " + sujeto.getEstado());
        IO.out.println("Cambio sujeto...");
        sujeto.setEstado(2);
    }
}
```

Observer. Ejercicio. Persona

- Se parte de la clase *Persona* con los atributos “nombre”, “edad” y “movil” y sus correspondientes getters & setters
- Se pretende que la clase IO sea la vista de la clase *Persona*, y siempre muestre los valores actuales de *Persona*
- Nos apoyaremos en los métodos de IO siguientes:
  - IO.out.setStatusBar(String s). Saca por la barra de estado una cadena, aquí se mostrará los valores del objeto persona
  - IO.in.addModel(Objecto o). Asocia una ficha del objeto *Persona* donde se podrá ejecutar los métodos disponibles

Ticket de supermercado

```
Fri Oct 08 10:54:22 CEST 2010
Ticket: 1

Azucar 1kg Azucarera .... +100
Sal yodada 1Kg Marina .... +10
Varios... .... +100
(Anul.) Sal yodada 1Kg Marina .... -10
(6X) Azucar 1kg Azucarera (100) .... +600
(3X) Sal yodada 1Kg Marina (10) .... +30
(3x2) Azucar 1kg Azucarera ....  -100
(3x2) Azucar 1kg Azucarera ....  -100

Total.....630

Gracias por su visita
```

Ticket de supermercado

- Las ofertas pueden cambiar con facilidad a lo largo del tiempo: 3x2, 2ª al 70%...
- Se debe minimizar el tráfico entre los puestos y el ordenador central
- En cualquier momento se puede actualizar un producto y se debe reflejar en todos los puestos
- Existe una impresora, pero dos modos de impresión: normal y para mayores