Universidad Politécnica de Madrid

Máster Universitario en Ingeniería Web

Patrones de diseño III

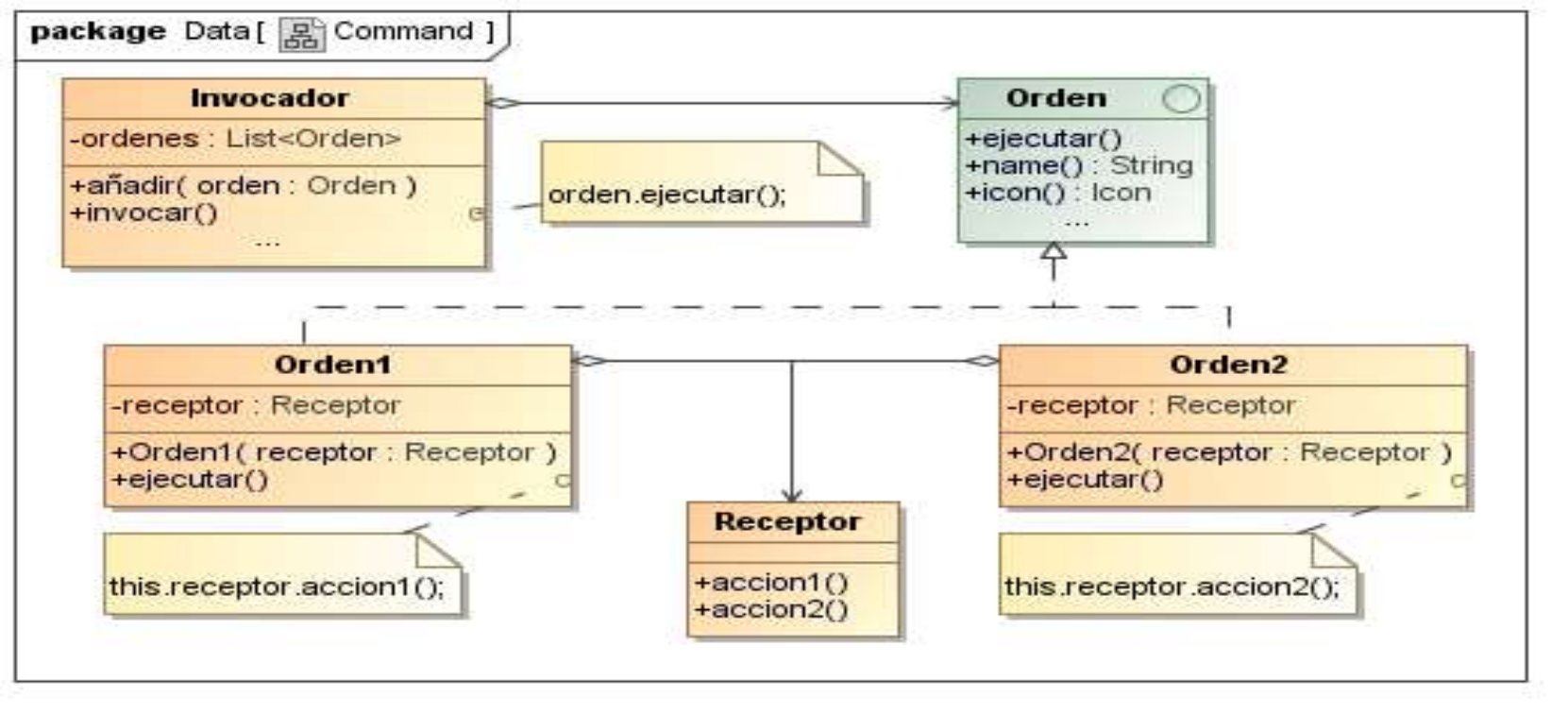
Jesús Bernal Bermúdez

Command (Orden)

- También conocido
  - Action (Acción), Transaction (Transacción)
- Motivación
  - A veces es necesario realizar peticiones a objetos sin conocer la petición ni a quien va dirigida
- Propósito
  - Se desacopla el objeto que invoca a la operación asociada, mediante un objeto. Ello permite realizar ordenes compuestas (patrón composite) o llevar una cola y deshacer operaciones

Command. Implementación

- Se encapsula en un objeto las peticiones, permitiendo llevar una cola o deshacer operaciones



Command. Implementación

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

```
public class JButtonInvocador extends JButton implements ActionListener {
    private static final long serialVersionUID = 1L;

    private Orden orden;

    public JButtonInvocador(Orden orden) {
        super(orden.name());
        this.orden = orden;
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        this.orden.ejecutar();
    }

}
```

Command. Implementación

```
public class OrdenConcreta1 implements Orden {
    private Receptor receptor;
    public OrdenConcreta1(Receptor receptor) {
        this.receptor = receptor;
    }
    @Override public void ejecutar() {
        this.receptor.accion1();
    }
}
```

```
public class OrdenConcreta2 implements Orden {
    private Receptor receptor;
    public OrdenConcreta2(Receptor receptor) {
        this.receptor = receptor;
    }
    @Override public void ejecutar() {
        this.receptor.accion2();
    }
}
```

```
public class Receptor {
    public void accion1() {
        IO.out.println("Acción 1");
    }
    public void accion2() {
        IO.out.println("Acción 2");
    }
}
```

Command. Implementación

```
public final class MainCommand {
    private Orden orden1, orden2;
    public MainCommand() {
        Receptor receptor = new Receptor();
        this.orden1 = new OrdenConcreta1(receptor);
        this.orden2 = new OrdenConcreta2(receptor);
    }
    public void invocador() {
        Orden[] ordenes = new Orden[2];
        ordenes[0] = this.orden1;
        ordenes[1] = this.orden2;
        Orden orden = (Orden) IO.in.select(ordenes);
        orden.ejecutar();
    }
    public void invocador1() {
        this.orden1.ejecutar();
    }
    public void invocador2() {
        this.orden2.ejecutar();
    }
    public static void main(String[] args) {
        IO.in.addController(new MainCommand());
    }
}
```

Command. Ejercicio. Calculadora

- Receptor Calculadora
  - Se pretende desacoplar la vista con las acciones del receptor
  - También se prepara para realizar un deshacer
- Tiene que haber un objeto Comando por cada acción del receptor (Calculadora) (en total cuatro)
  - Esto nos permite gestionar una colección de ordenes, pudiendo realizar macros, deshacer y generalizar el tratamiento de las ordenes

Command. Ejercicio. Calculadora

```
public class Calculadora {
    private int total;

    public Calculadora() {
        this.iniciar();
    }

    public int getTotal() {
        return total;
    }

    protected void setTotal(int total) {
        this.total = total;
    }

    public void sumar(int valor) {
        this.setTotal(this.total + valor);
    }

    public void restar(int valor) {
        this.setTotal(this.total - valor);
    }

    public void iniciar() {
        this.setTotal(0);
    }
}
```

Command. Ejercicio. Calculadora

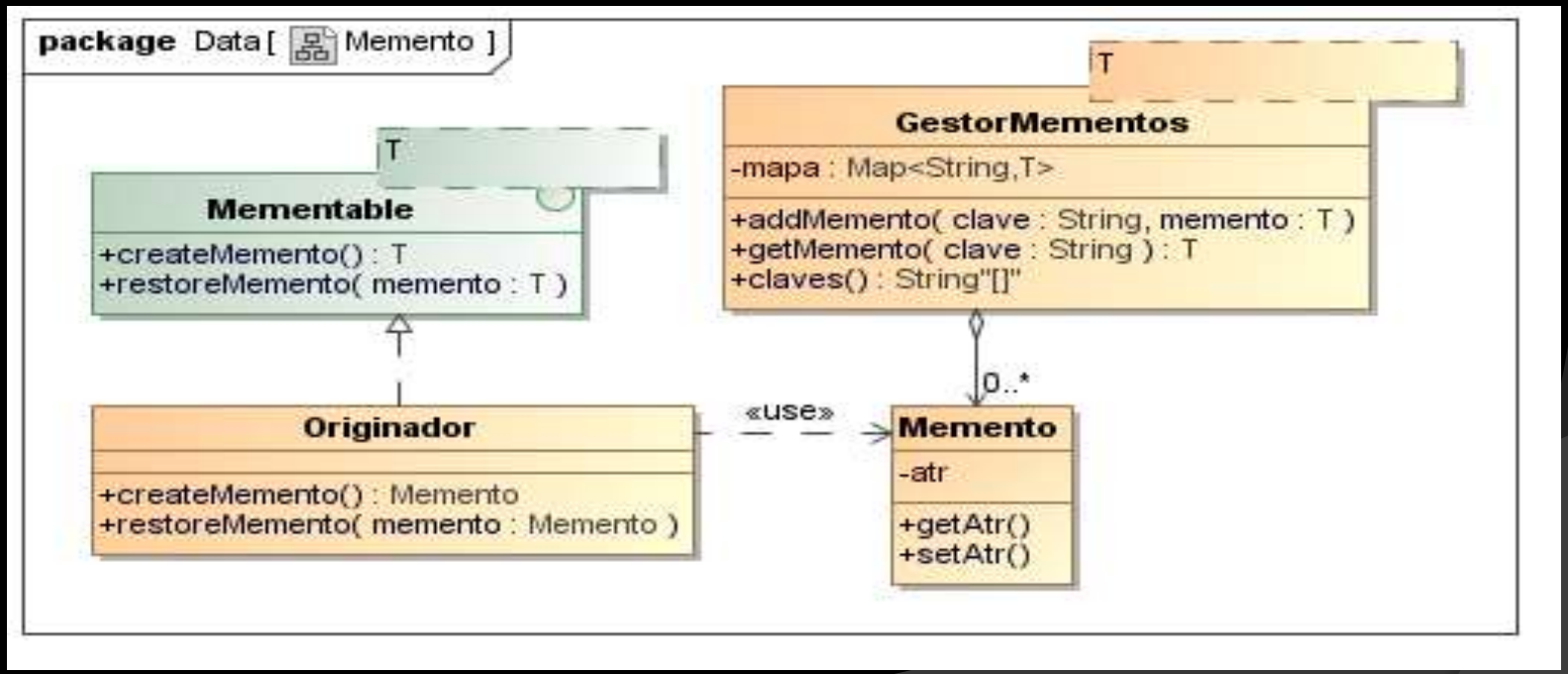
```
public class MainGestorComandos {
    private Set<Comando> comandos = new HashSet<Comando>();

    public void add(Comando comando) {
        this.comandos.add(comando);
    }
    public void ejecutar() {
        Comando comando = (Comando) IO.in.select(this.comandos.toArray());
        comando.execute();
    }

    public static void main(String[] args) {
        Calculadora calc = new Calculadora();
        MainGestorComandos gestor = new MainGestorComandos();
        gestor.add(new ComandoSumar(calc));
        gestor.add(new ComandoRestar(calc));
        gestor.add(new ComandoIniciar(calc));
        gestor.add(new ComandoImprimir(calc));
        IO.in.addController(gestor);
    }
}
```

Memento (Recuerdo)

- También conocido
  - Token
- Motivación
  - Cuando se requiere volver a situaciones anteriores
- Propósito
  - Externaliza el estado interno de un objeto sin violar la encapsulación

Memento (Recuerdo). Implementación

- Propósito
  - Externaliza el estado interno de un objeto sin violar la encapsulación



Memento (Recuerdo). Implementación

```
public interface Mementable<T> {
    T createMemento();
    void restoreMemento(T memento);
}
```

```
public class Originador implements Mementable<Memento> {
    private int id;
    private int valor;
    private String cadena;
    public Originador(int id) {
        this.id = id;
    }
    public int getId() {return id;}
    public int getValor() {return valor;}
    public void setValor(int valor) {this.valor = valor;}
    public String getCadena() {return cadena;}
    public void setCadena(String cadena) {this.cadena = cadena;}
    @Override public Memento createMemento() {
        return new Memento(this.valor, this.cadena);
    }
    @Override public void restoreMemento(Memento memento) {
        this.setValor(memento.getValor());
        this.setCadena(memento.getCadena());
    }
    @Override public String toString() {
        return "Originador[" + id + "," + cadena + "," + valor + "]";
    }
}
```

Memento (Recuerdo). Implementación

```
public class Memento {
    private int valor;
    private String cadena;
    public Memento(int valor, String cadena) {
        this.valor = valor;
        this.cadena = cadena;
    }
    public int getValor() {
        return this.valor;
    }
    public String getCadena() {
        return this.cadena;
    }
}
```

```
public class GestorMementos<T> {
    private SortedMap<String, T> lista = new TreeMap<String, T>();
    public void addMemento(String key, T memento) {
        this.lista.put(this.lista.size() + ":" + key, memento);
    }
    public T getMemento(String key) {
        return this.lista.get(key);
    }
    public String[] keys() {
        return this.lista.keySet().toArray(new String[0]);
    }
}
```

Memento (Recuerdo). Implementación

```
public final class MainMemento {
    private GestorMementos<Memento> gm;
    private Mementable<Memento> o;
    private MainMemento() {
        this.gm = new GestorMementos<Memento>();
        this.o = new Originador(666);
        IO.in.addModel(this.o);
        IO.in.addController(this);
    }
    public void createMemento() {
        this.gm.addMemento(
                    IO.in.readString("Nombre del Memento"), o.createMemento());
    }
    public void restoreMemento() {
        this.o.restoreMemento(
              this.gm.getMemento((String) IO.in.select(gm.keys(), "Restaurar")));
    }
    public static void main(String[] args) {
        new MainMemento();
    }
}
```

Memento (Recuerdo). Ejercicio. Calculadora

- Añadir al ejercicio de la calculadora dos comandos:
  - Guardar: guarda un punto de restauración
  - Deshacer: recupera un punto de restauración
- Guía, crear las clases:
  - MementoCalculadora
  - CalculadoraMementable
  - ComandoGuardar
  - ComandoDeshacer

Memento (Recuerdo). Ejercicio. Calculadora

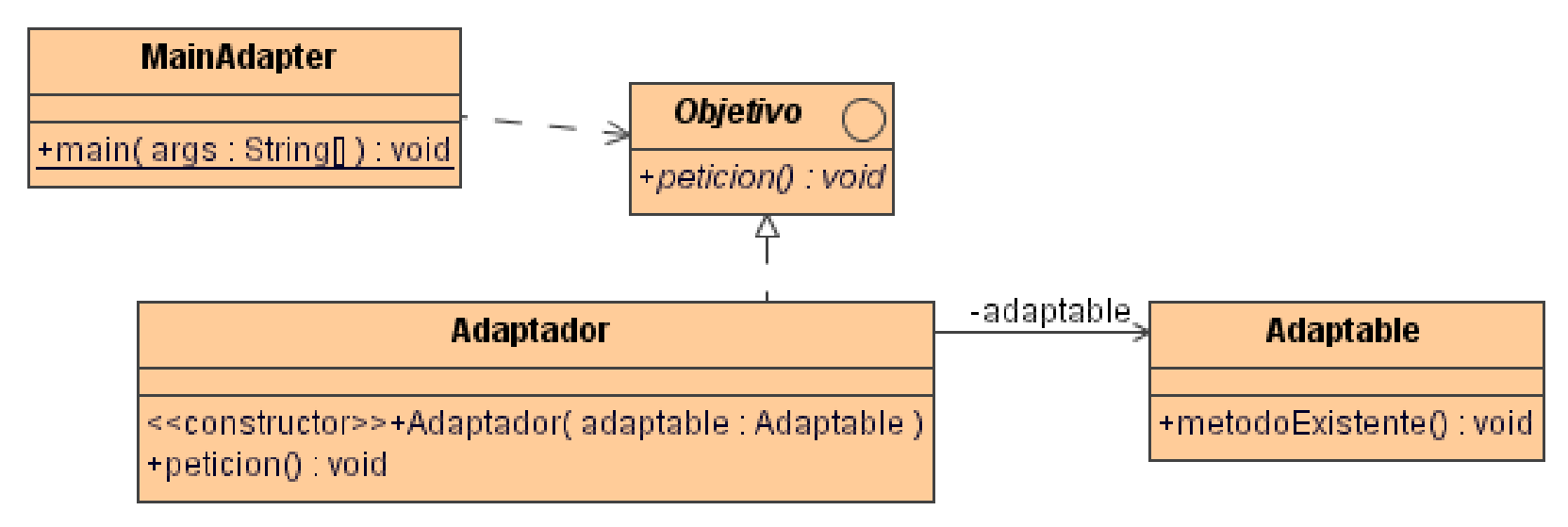
```
public class MainGestorComandos2 {
    private Set<Comando> comandos = new HashSet<Comando>();
    public void add(Comando comando) {
        this.comandos.add(comando);
    }
    public void ejecutar() {
        Comando comando = (Comando) IO.in.select(this.comandos.toArray());
        comando.execute();
    }
    public static void main(String[] args) {
        CalculadoraMementable calc = new CalculadoraMementable();
        MainGestorComandos2 gestorCom = new MainGestorComandos2();
        GestorMementos<MementoCalculadora> gestorMem =
                                        new GestorMementos<MementoCalculadora>();
        gestorCom.add(new ComandoSumar(calc));
        gestorCom.add(new ComandoRestar(calc));
        gestorCom.add(new ComandoIniciar(calc));
        gestorCom.add(new ComandoImprimir(calc));

        gestorCom.add(new ComandoGuardar(calc, gestorMem));
        gestorCom.add(new ComandoDeshacer(calc, gestorMem));

        IO.in.addController(gestorCom);
        IO.out.setLog(Log.DEBUG);
    }
}
```

Adapter (Adaptador)

- También conocido
  - Wrapper (Envoltorio)
- Motivación
  - Reduce la dependencia entre clases. A veces, se requiere reutilizar una clase en otra aplicación, pero poseen dominios diferentes
- Propósito
  - Convierte una interface de una clase en otra que es la que esperan los clientes

Adapter. Implementación

- Convierte una interface de una clase en otra que es la que esperan los clientes



Adapter. Implementación

```
//Clase existente
public class Adaptable {
    public void metodoExistente() {
        IO.out.println("Ya existe...");
    }
}
```

```
//Se necesita este interface
public interface Objetivo {
    void peticion();
}
```

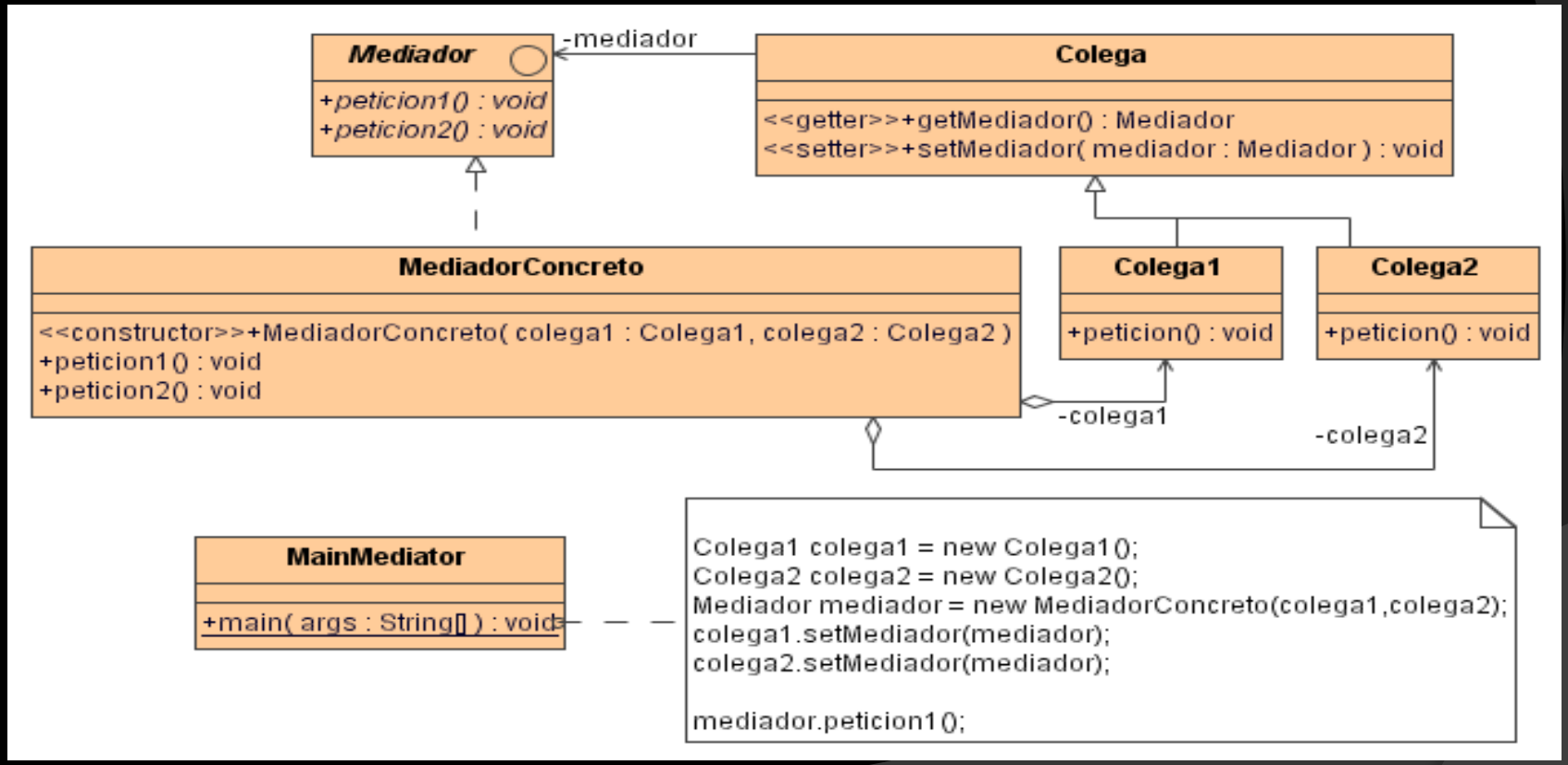
```
public class Adaptador implements Objetivo {
    private Adaptable adaptable;
    public Adaptador(Adaptable adaptable) {
        this.adaptable = adaptable;
    }
    @Override
    public void peticion() {
        this.adaptable.metodoExistente();
    }

}
```

Mediator (Mediador)

- Propósito
  - Se encapsula en un objeto cómo interactúan una serie de objetos
- Motivación
  - La colaboración de objetos pude llegar a ser compleja y difícil de reutilizar. Una solución es crear un objeto que realice operaciones de mayor nivel abstrayendo de las conexiones entre objetos

Mediator. Implementación

- Se encapsula en un objeto cómo interactúan una serie de objetos



Mediator. Implementación

```
public class Colega {
    private Mediador mediador;
    public Mediador getMediador() {
        return mediador;
    }
    public void setMediador(Mediador mediador) {
        this.mediador = mediador;
    }
}
```

```
public class Colega1 extends Colega {
    public void peticion() {
        IO.out.println("Petición al colega 1. Realiza petición 2 al mediador");
        this.getMediador().peticion2();
    }
}
```

```
public class Colega2 extends Colega {
    public void peticion() {
        IO.out.println("Petición a colega 2");
    }
}
```

Mediator. Implementación

```
public interface Mediator {
    void peticion1();
    void peticion2();
}
```

```
public class Mediator1 implements Mediator {
    private Colega1 colega1 = new Colega1();
    private Colega2 colega2 = new Colega2();
    public Mediator1(Colega1 colega1, Colega2 colega2) {
        this.colega1 = colega1;
        this.colega2 = colega2;
    }
    @Override
    public void peticion1() {
        IO.out.println("Mediador: petición 1... petición a colegas 1 y 2");
        this.colega1.peticion();
        this.colega2.peticion();
    }
    @Override
    public void peticion2() {
        IO.out.println("Mediador: petición 2... petición a colega 2");
        this.colega2.peticion();
    }
}
```

Mediator. Implementación

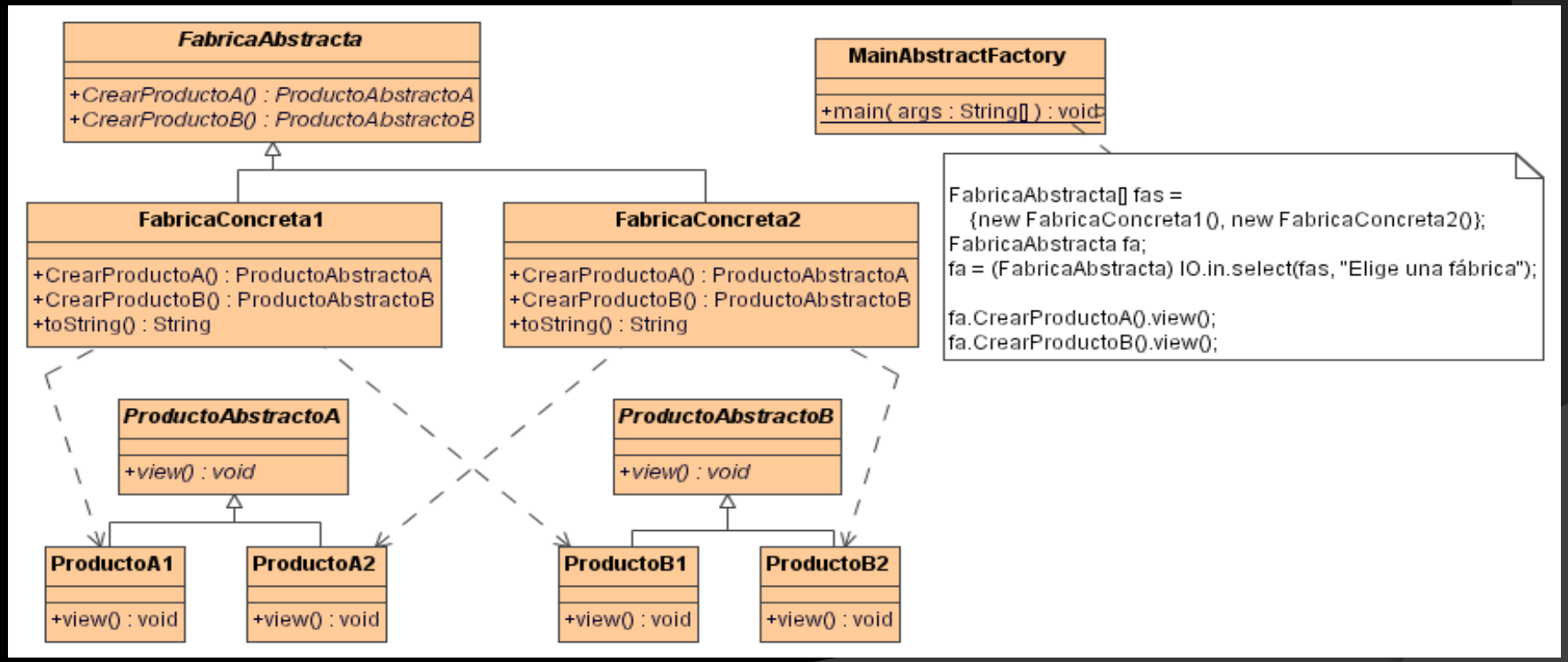
```
public final class MainMediator {
    private MainMediator() {}

    public static void main(String[] args) {
        Colega1 colega1 = new Colega1();
        Colega2 colega2 = new Colega2();
        Mediator mediador = new Mediator1(colega1, colega2);
        colega1.setMediador(mediador);
        colega2.setMediador(mediador);

        mediador.peticion1();
    }

}
```

Abstract Factory (Fábrica abstracta)

- También conocido
  - Kit
- Propósito
  - Proporciona un interface para crear familias de objetos relacionadas
- Motivación
  - Se permiten múltiples interface de usuario

Abstract Factory. Implementación

- Propósito
  - Proporciona un interface para crear familias de objetos relacionadas



Abstract Factory. Implementación

```
public abstract class ProductoAbstractoA {
    public abstract void view();
}
public class ProductoA1 extends ProductoAbstractoA {
    @Override public void view() {
        IO.out.println("ProductoA1");
    }
}
public class ProductoA2 extends ProductoAbstractoA {
    @Override public void view() {
        IO.out.println("ProductoA2");
    }
}
```

```
public abstract class ProductoAbstractoB {
    public abstract void view();
}
public class ProductoB1 extends ProductoAbstractoB {
    @Override public void view() {
        IO.out.println("ProductoB1");
    }
}
public class ProductoB2 extends ProductoAbstractoB {
    @Override public void view() {
        IO.out.println("ProductoB2");
    }
}
```

Abstract Factory. Implementación

```
public abstract class FabricaAbstracta {
    public abstract ProductoAbstractoA crearProductoA();
    public abstract ProductoAbstractoB crearProductoB();
}
```

```
public class FabricaConcreta1 extends FabricaAbstracta {
    @Override public ProductoAbstractoA crearProductoA() {
        return new ProductoA1();
    }
    @Override public ProductoAbstractoB crearProductoB() {
        return new ProductoB1();
    }
    @Override public String toString() {
        return "FabricaConcreta1";
    }
}
```

```
public class FabricaConcreta2 extends FabricaAbstracta {
    @Override public ProductoAbstractoA crearProductoA() {
        return new ProductoA2();
    }
    @Override public ProductoAbstractoB crearProductoB() {
        return new ProductoB2();
    }
    @Override public String toString() {
        return "FabricaConcreta2";
    }
}
```

Abstract Factory. Implementación

```
public class MainAbstractFactory {
    private FabricaAbstracta fa;
    private FabricaAbstracta[] fas =
                        {new FabricaConcreta1(), new FabricaConcreta2()};

    public void tipoFabrica() {
        fa = (FabricaAbstracta) IO.in.select(fas, "Elige una fábrica");
    }

    public void fabricarProductos() {
        fa.crearProductoA().view();
        fa.crearProductoB().view();

    }

    public static void main(String[] args) {
        IO.in.addController(new MainAbstractFactory());
    }
}
```

Abstract Factory. Ejercicio. Cuentas

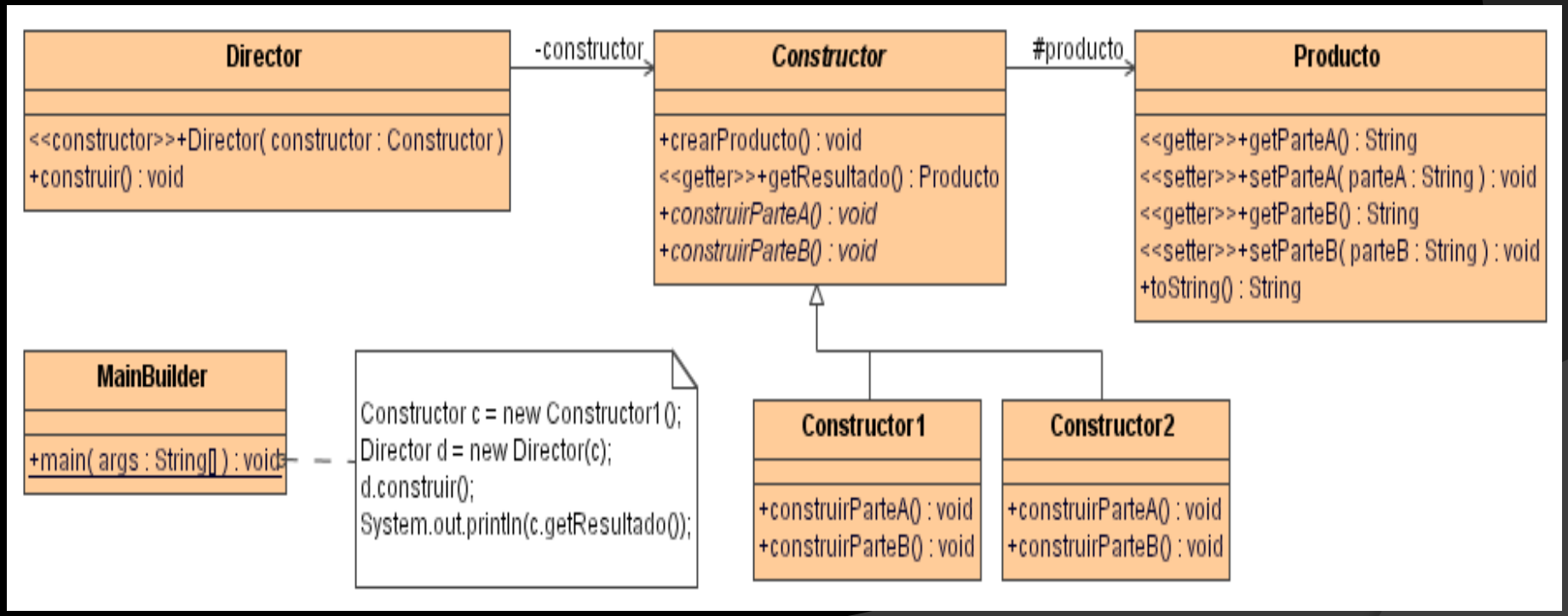
Tipo Cuenta	Cuenta	Tarjeta débito	Tarjeta crédito
Joven	1%	Gratuita	No
10	1´5%	Gratuita	10 € Max. 2000€
Oro	2%	5 €	20€ Max. 4000€

Builder (Constructor)

- Motivación
  - Este patrón permite tener una política general para la creación de objetos, centralizandolo en una clase Builder
- Propósito
  - Centraliza la construcción de un objeto complejo, permitiendo diferentes representaciones del mismo

Builder. Implementación

- Propósito
  - Centraliza la construcción de un objeto complejo, permitiendo diferentes representaciones del mismo



Builder. Implementación

```
public class Producto {
    private String parteA;
    private String parteB;

    public String getParteA() {
        return parteA;
    }
    public void setParteA(String parteA) {
        this.parteA = parteA;
    }
    public String getParteB() {
        return parteB;
    }
    public void setParteB(String parteB) {
        this.parteB = parteB;
    }
    @Override
    public String toString() {
        return "Producto[" + parteA + "," + parteB + "]";
    }
}
```

Builder. Implementación

```
public abstract class Constructor {
    private Producto producto;
    public Producto getProducto() {
        return producto;
    }
    public void crearProducto() {
        this.producto = new Producto();
    }
    public Producto getResultado() {
        return this.producto;
    }
    public abstract void construirParteA();
    public abstract void construirParteB();
}
```

```
public class Constructor1 extends Constructor {
    @Override
    public void construirParteA() {
        this.getProducto().setParteA("Parte A");
    }
    @Override
    public void construirParteB() {
        this.getProducto().setParteB("Parte B");
    }
}
```

Builder. Implementación

```
public class Director {
    private Constructor constructor;
    public Director(Constructor constructor) {
        this.constructor = constructor;
    }
    public void construir() {
        constructor.crearProducto();
        constructor.construirParteA();
        constructor.construirParteB();
    }
}
```

```
public final class MainBuilder {
    private MainBuilder() {}
    public static void main(String[] args) {
        Constructor c = new Constructor1();
        Director d = new Director(c);
        d.construir();
        System.out.println(c.getResultado());
    }
}
```

Bridge (Puente)

- También conocido
  - Handle, Body (Manejador, Cuerpo)
- Propósito
  - Desacopla una abstracción de su implementación, permitiendo modificaciones independientes de ambas
- Motivación
  - Cuando una estructura de abstracciones puede tener varias implementaciones, por ejemplo, plantillas diferentes de apariencia o para sistemas diferentes; la herencia no suele ser una buena solución

Strategy (Estrategia)

- También conocido
  - Policy (Política)
- Propósito
  - Define un conjunto de algoritmo haciéndolos intercambiables dinámicamente
- Motivación
  - Existen muchos algoritmos de ordenación, dependiendo su eficacia del número de elementos a ordenar

Chain of Responsibility (Cadena de responsabilidad)

- Propósito
  - Se establecer una cadena de objetos receptores a través de los cuales se pasa una petición formulada por un objeto emisor. Cualquiera de los objetos receptores puede responder a la petición en función de un criterio establecido
- Motivación
  - En un servicio de ayuda sensible al contexto

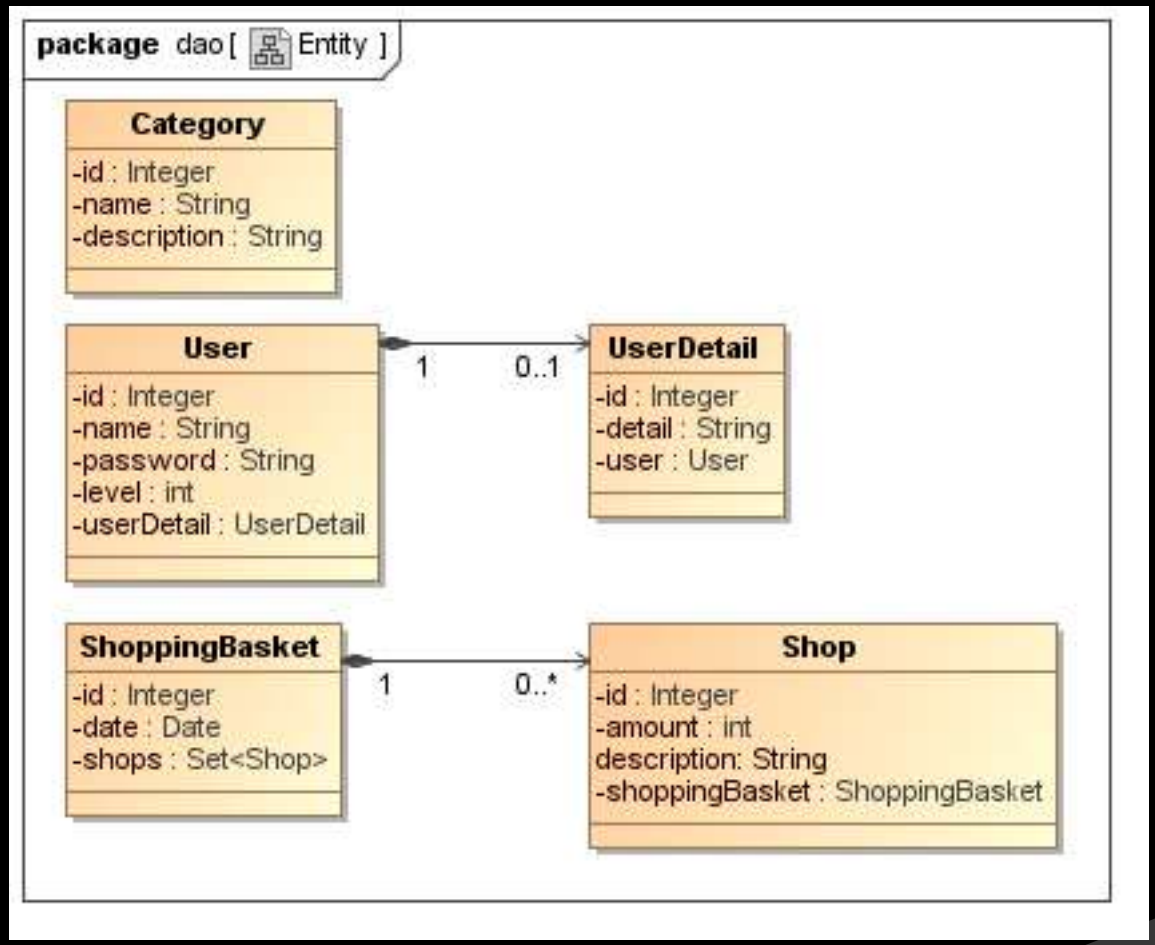
Prototype (Prototypo)

- Propósito
  - Crear los objetos a partir de prototipos mediante la clonación
- Motivación
  - Decidir dinámicamente el tipo de objeto a crear dependiendo del contexto

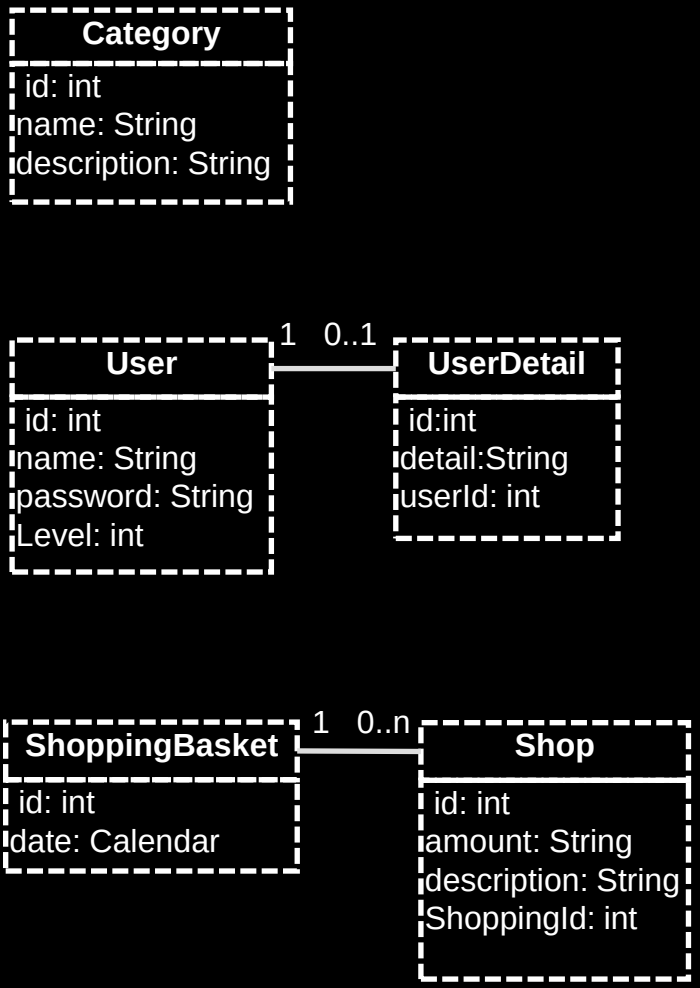
Persistencia. DAO (Data Access Object)

- Objetivo: abstraer y encapsular el acceso a la capa de persistencia (Bases de Datos), permitiendo desacoplar la capa de negocio con la capa de persistencia
- Adapta el modelo relacional de BD con el modelo de objetos
- Permite el intercambio de implementaciones de persistencia sin afectar a la capa de negocio

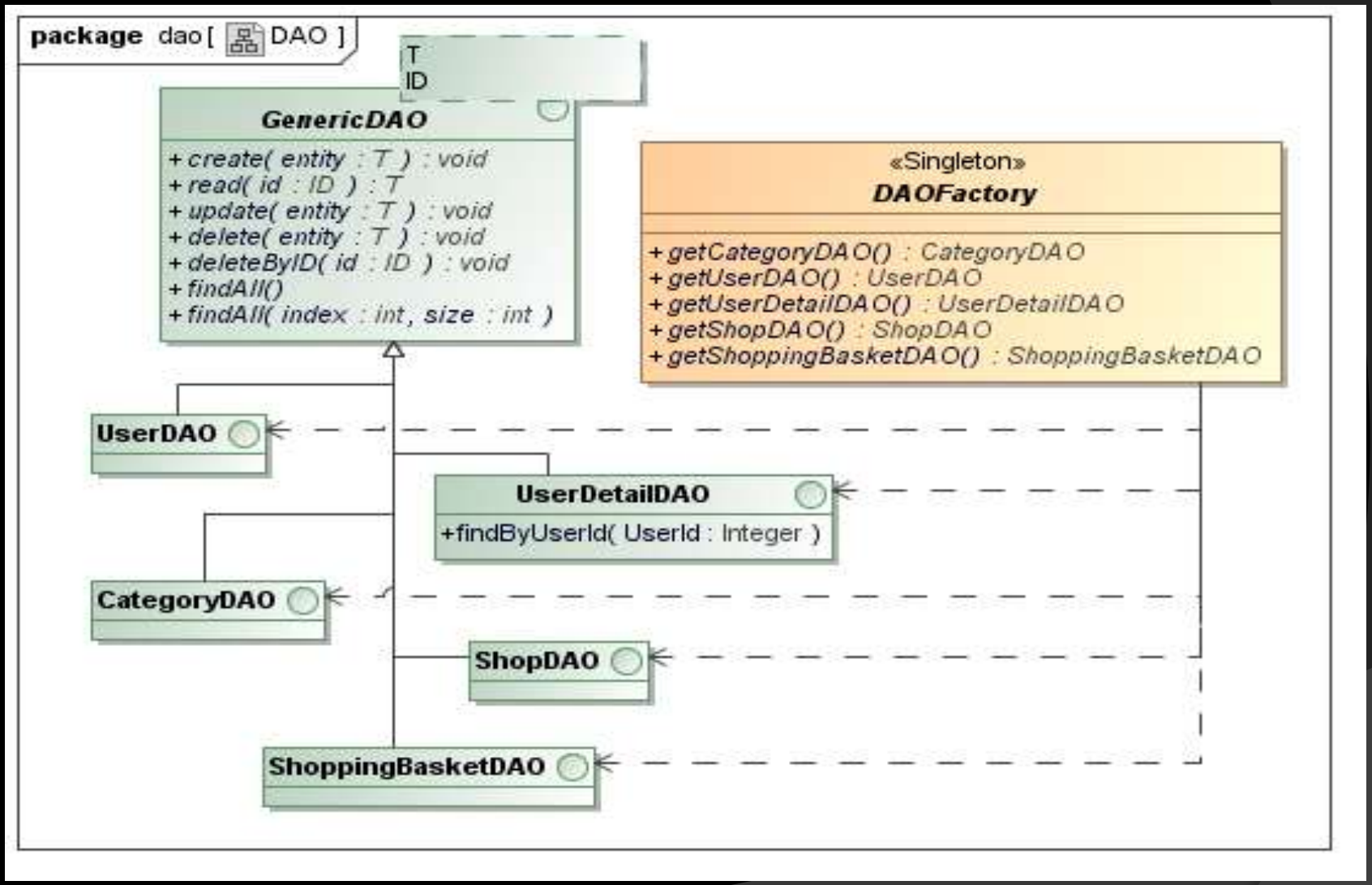
Persistencia. DAO



Persistencia. Tablas relacionales



Persistencia. DAO



Persistencia. DAO