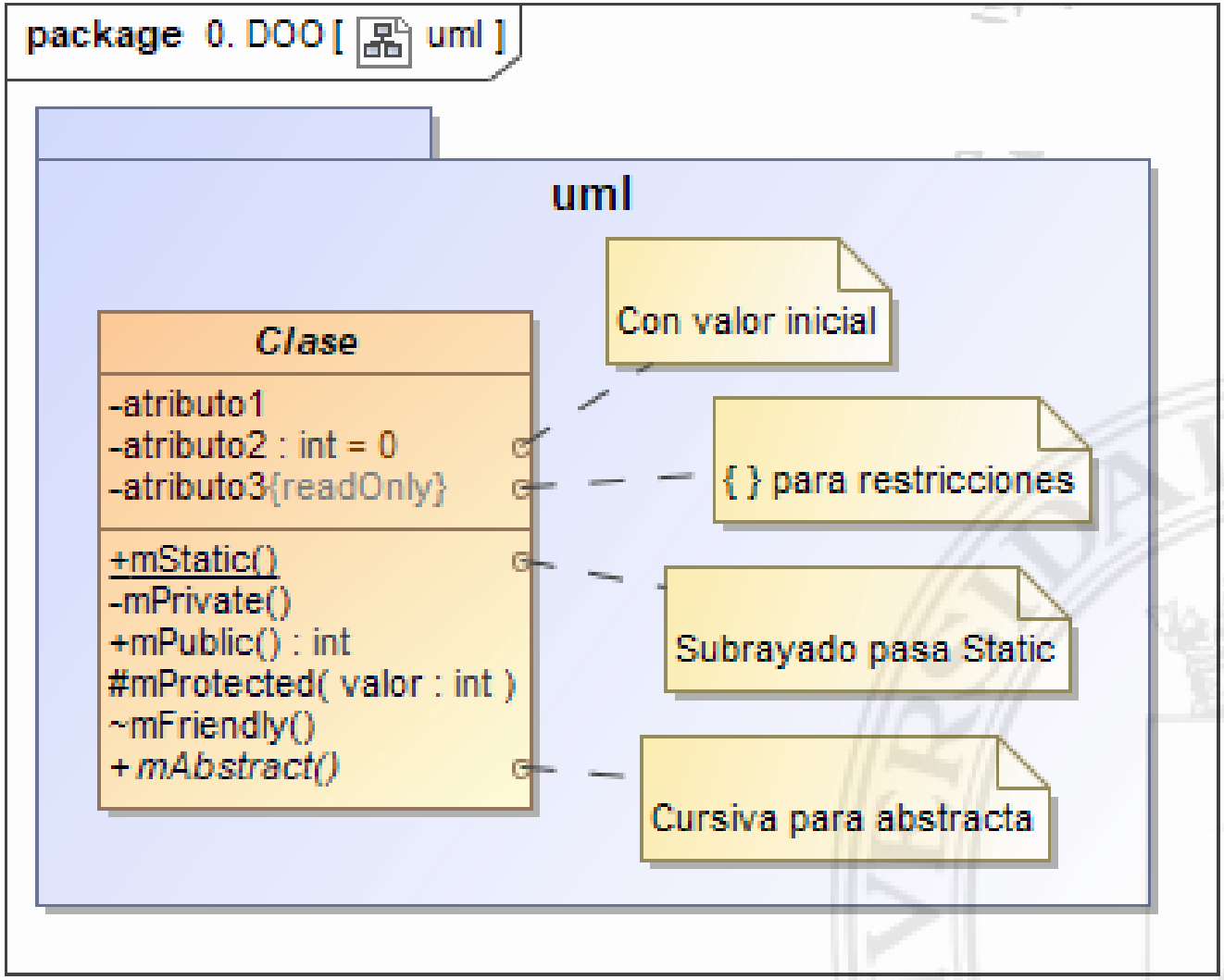
Arquitectura y Patrones para Aplicaciones Web

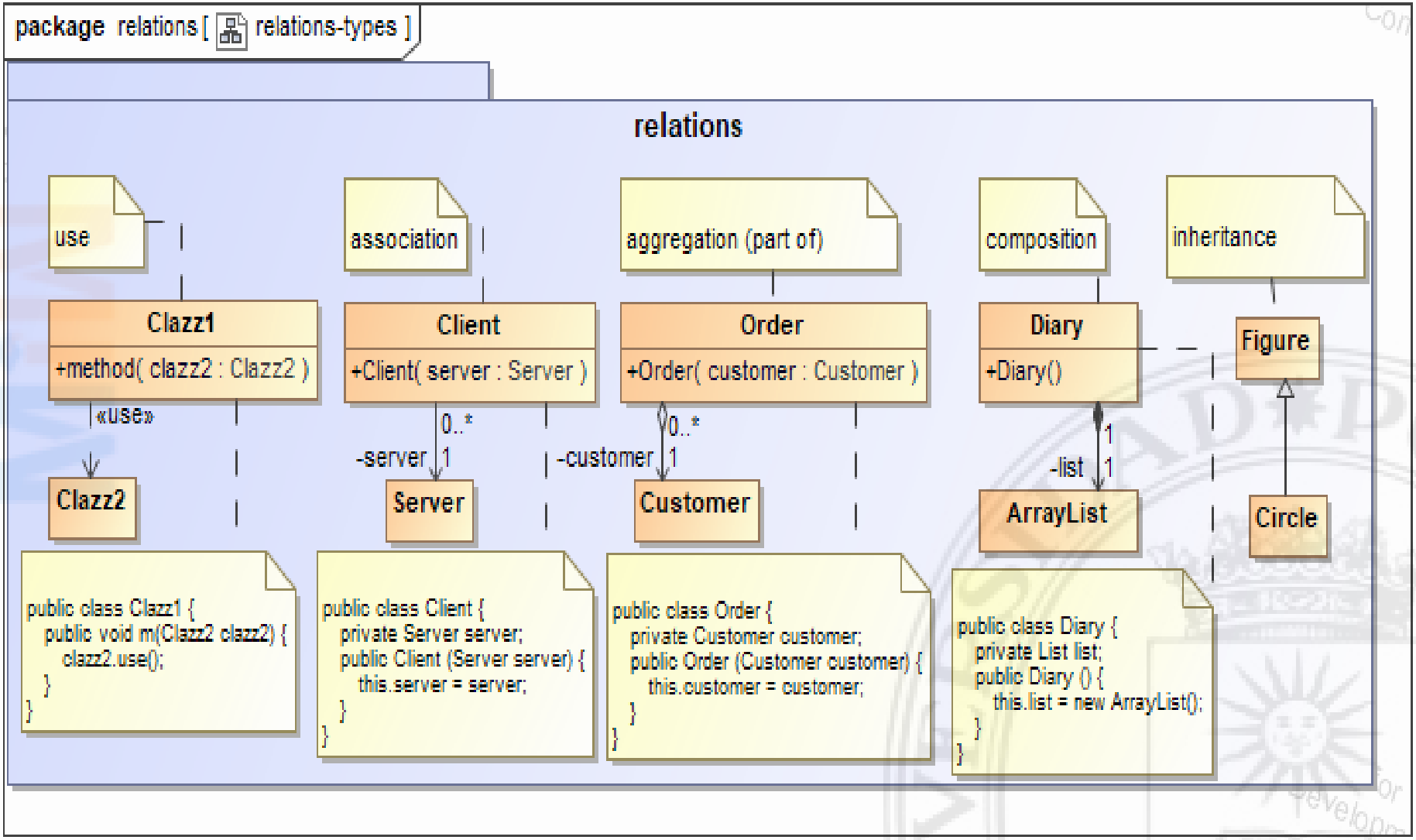
Patrones de Diseño

Jesús Bernal Bermúdez

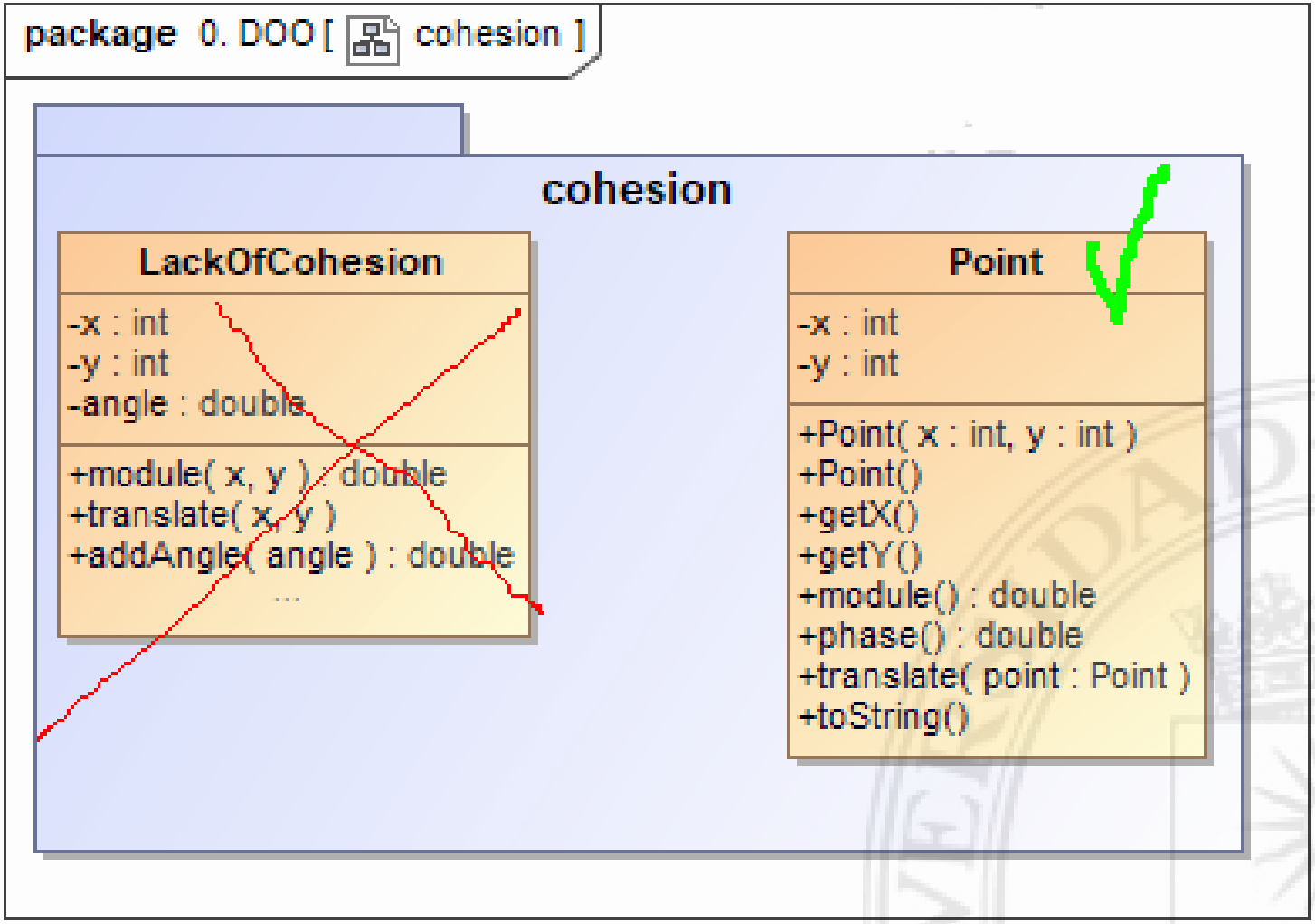
Relaciones. UML



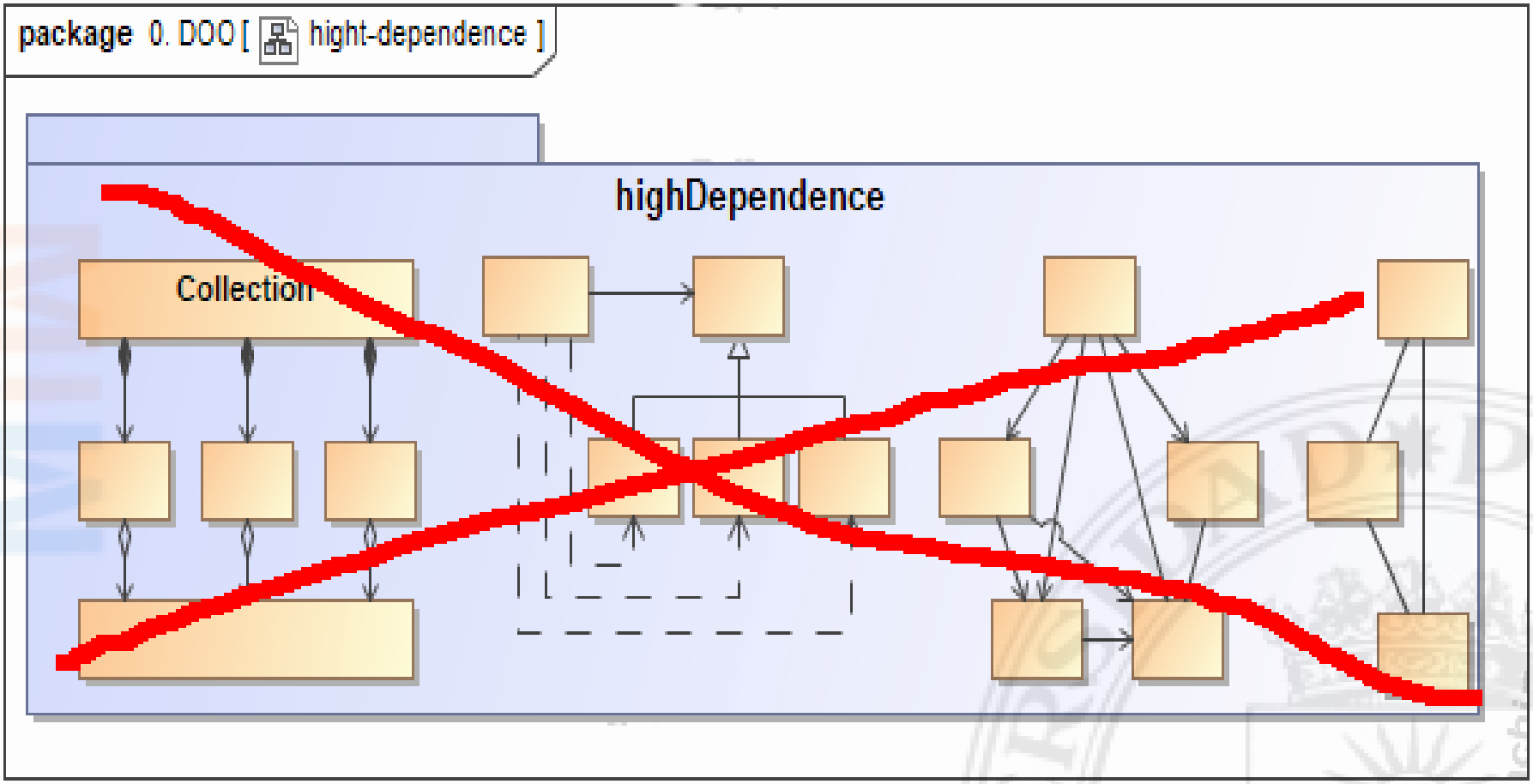
Relaciones. UML



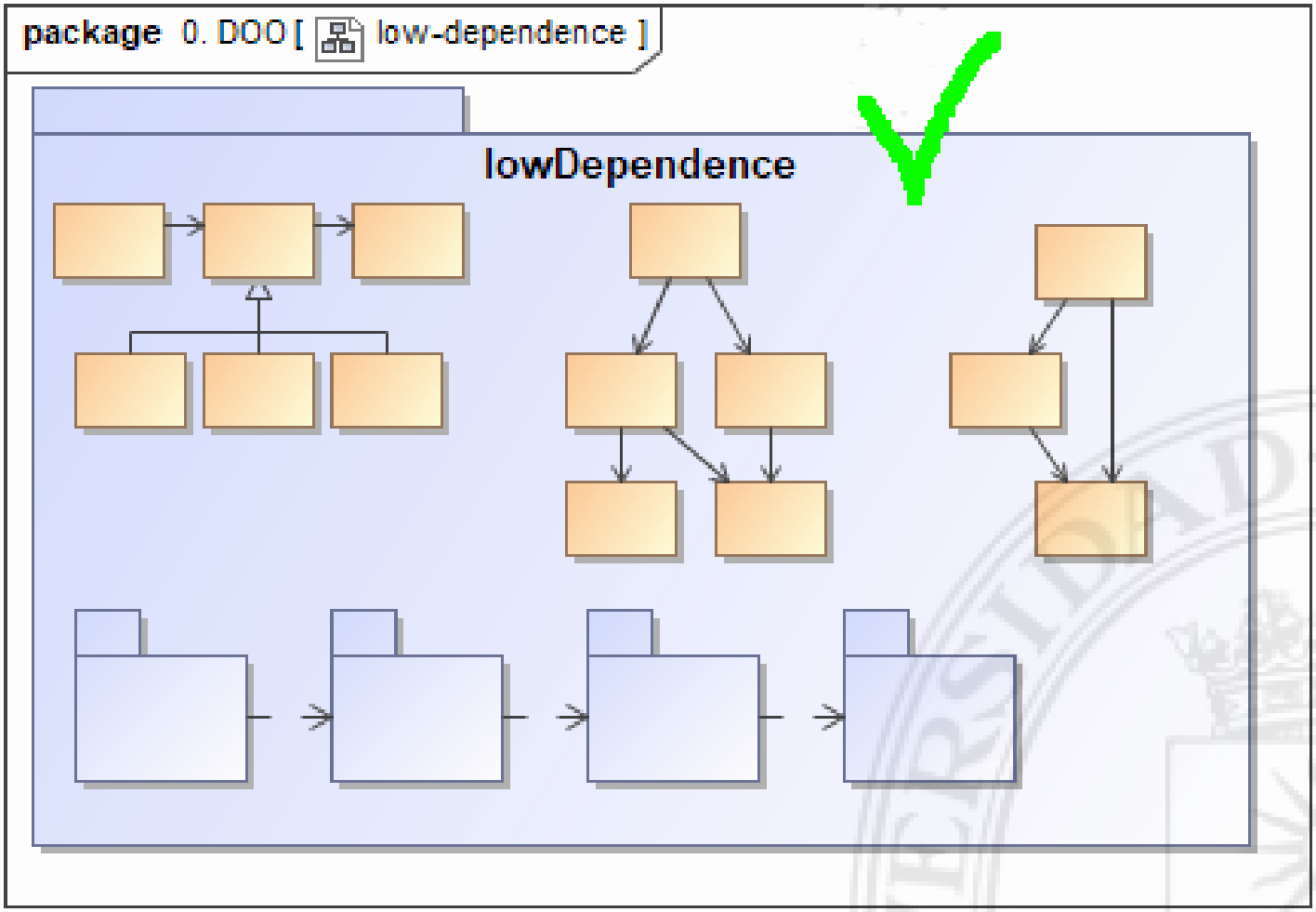
Cohesión


package 0. DOO [cohesion]
cohesion
LackOfCohesion
-x : int
-y : int
-angle : double
+module(x, y) : double
+translate(x, y)
+addAngle(angle) : double
...
Point
-x : int
-y : int
+Point(x : int, y : int)
+Point()
+getX()
+getY()
+module() : double
+phase() : double
+translate(point : Point)
+toString()

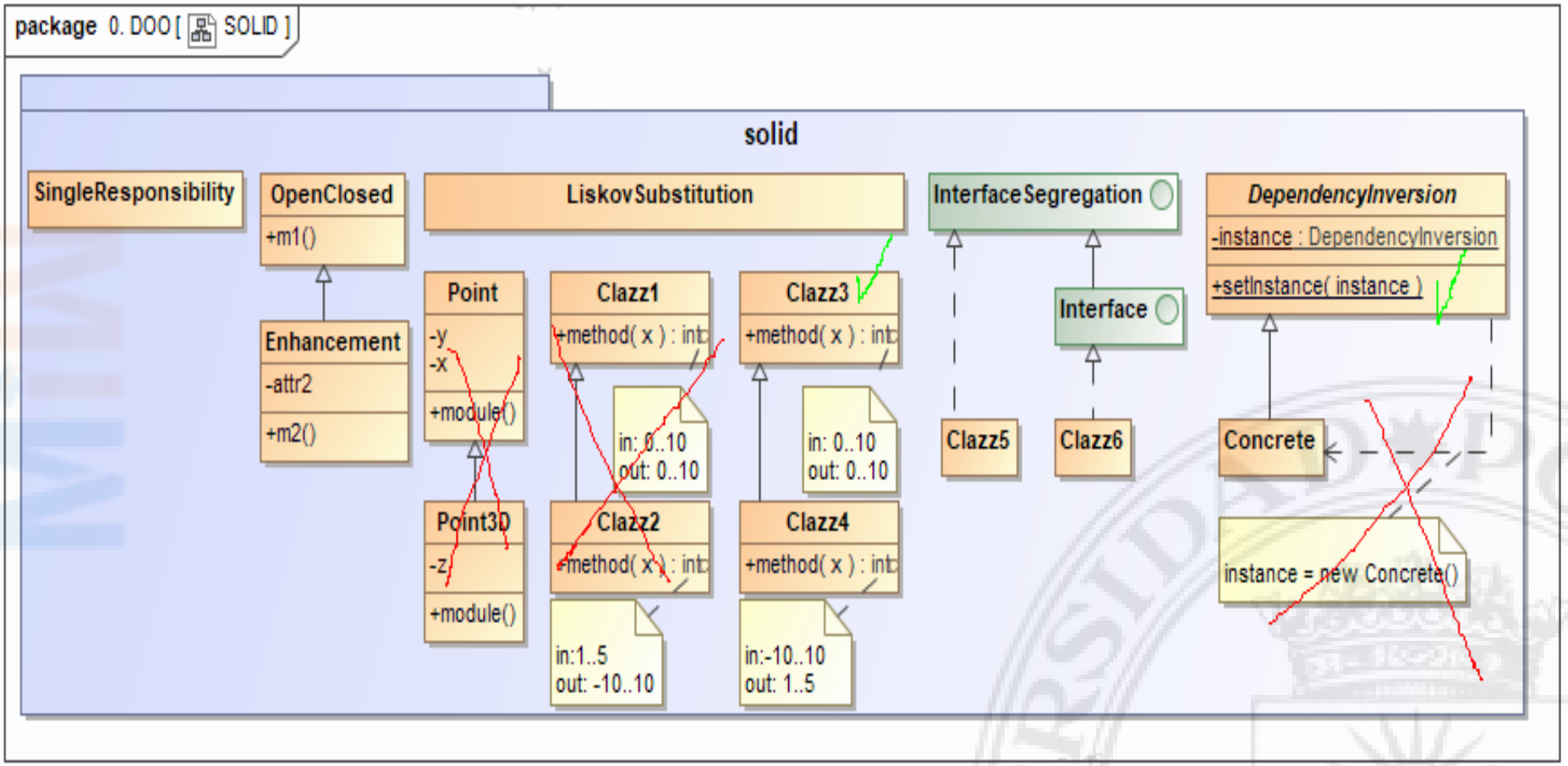
Alto acoplamiento



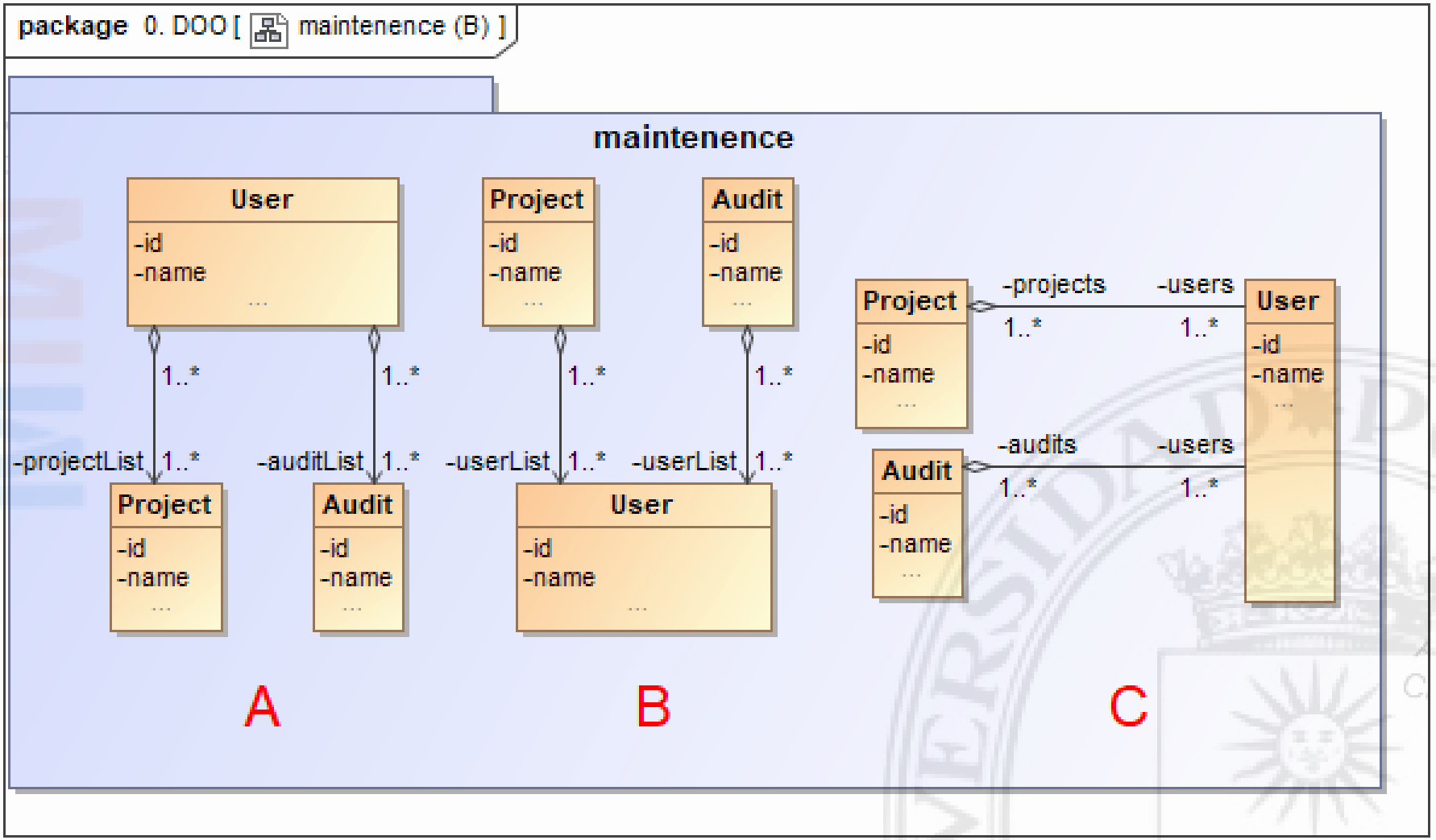
Bajo acoplamiento



Principios SOLID



Test


package 0. DOO [maintenence (B)]
maintenence
User
-id
-name
...
Project
-id
-name
...
Audit
-id
-name
...
1..*
-projectList
-auditList
-userList
-userList
A
B
C
-projects
-users
-audits
-users

Introducción

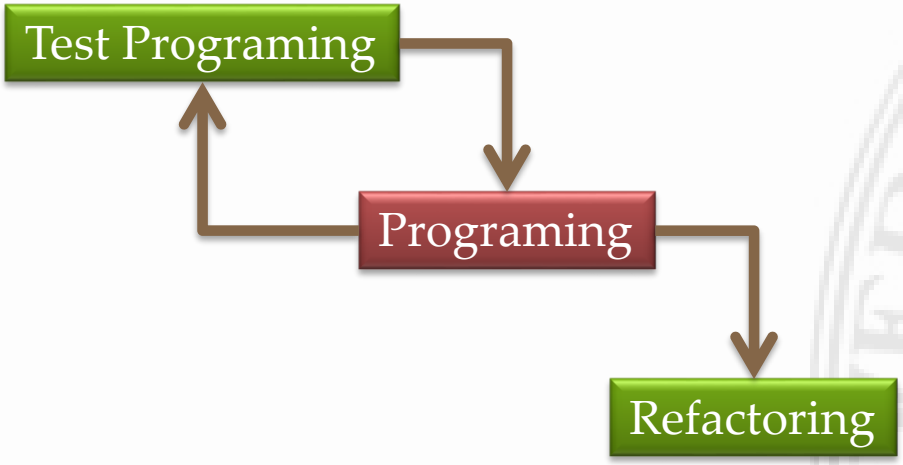
- **Antipatrones (Anti-patterns)**
  - **Cut & paste**
  - Magic numbers
  - Blob
  - Lava flow
  - Spaghetti code
  - Blind faith
  - Poltergeist
  - Golden Hammer
  - Programming by permutation…
  - Sequential coupling
  - Checking type instead of interface
  - Busy spin
  - …

Calidad del código

- Formateo
  - Herramienta del IDE
  - Líneas en blanco
  - Sin comentarios
  - Nombres de clases, métodos, atributos, parámetros y variables
- Sencillez del código
  - Estructuras anidadas: <3
  - Complejidad ciclomática: <8-12
- Métricas
  - Paquete: <20 clases
  - Clases: <500-200 líneas, <20 métodos
  - Métodos: <3-4 parámetros, <15 líneas
- Eliminar redundancias (*copy & paste*)
- Eliminar código muerto
- Tratamiento de errores

Metodologías ágiles

- Programación Extrema (Extreme Programming - XP)
- Desarrollo dirigido por pruebas (Test Driven Development – TDD)
- Integración continua (Continuous Integration – CI)
- **Refactorización (Refactoring)**
- Diseño simple: *You aren't gonna need it (YAGNI)*



Introducción

- Qué es un patrón: "es una solución a un problema que ocurre de forma repetida en nuestro entorno"
- Partes esenciales
  - ***Nombre del patrón***. Enriquece el vocabulario y permite comunicarse mejor entre diseñadores
  - ***Problema***. Explica el problema y su contexto
  - ***Solución***. Describe el diseño de la solución, sus relaciones y su responsabilidades
  - ***Consecuencias***. Describe las ventajas e inconvenientes de aplicar el patrón
- Referencia. *Patrones de Diseño, GoF (Gang Of Four), E. Gamma, R. Helm, R. Johnson y J. Vlissides*

Catálogo de patrones. Propósito: Creación

- Ámbito: ***Objeto***
  - ***Singleton***. Se garantiza que una clase sólo tenga una instancia y se proporciona un acceso global a ella
  - ***Builder***. Separa la construcción de objeto complejo de su representación, permitiendo diferentes construcciones
  - ***Abstract Factory***. Proporciona un interface para crear familias de objetos relacionadas
  - ***Prototype***. Especifica y crea objetos por medio de un prototipo mediante la clonación
- Ámbito: ***Clase***
  - ***Factory Method***. Define una abstracción para crear objetos, y son las subclases que deciden la clase concreta a instanciar

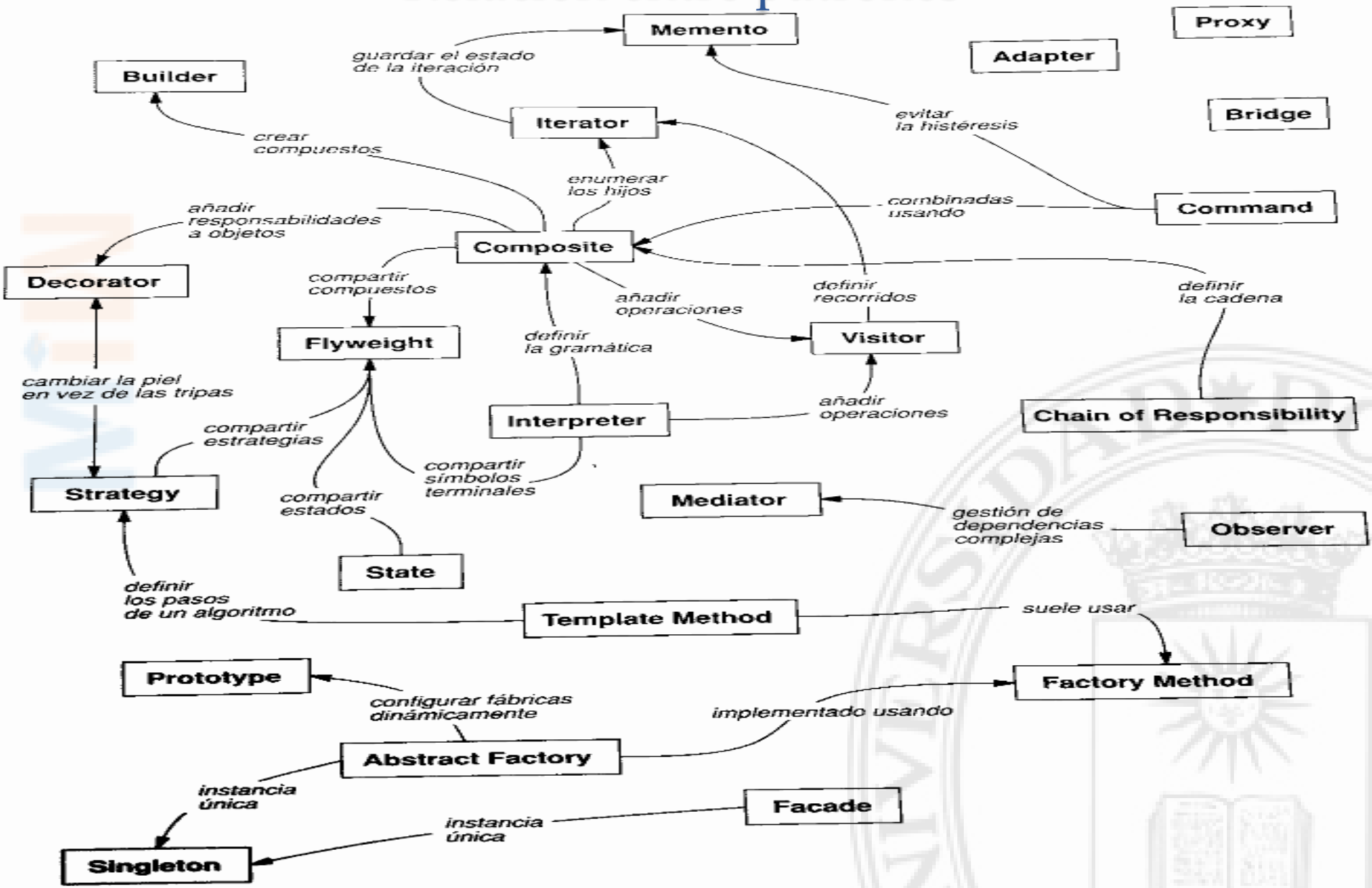
Catálogo de patrones. Propósito: Estructural

- Ámbito: ***Objeto***
  - ***Composite***. Permite estructuras en árbol tratando por igual a las hojas que a los elementos compuestos
  - ***Decorator***. Asigna responsabilidades de forma dinámica a objetos, proporcionando una alternativa flexible a la herencia
  - ***Flyweight***. Compartir objetos de grado fino de forma eficiente
  - ***Facade***. Proporciona un interface unificado para un conjunto de interfaces de un subsistema
  - ***Proxy***. Se proporciona un sustituto o representante para controlar el acceso a un objeto
  - ***Bridge***. Desacopla una abstracción de su implementación, permitiendo modificaciones independientes de ambas
- Ámbito: ***Clase y Objeto***
  - ***Adapter***. Convierte una interface de una clase en otra que es la que esperan los clientes

Catálogo de patrones. Propósito: Comportamiento

- Ámbito: ***Objeto***
  - ***State***. Permite que un objeto cambie su comportamiento cada vez que cambie su estado interno
  - ***Visitor***. Definir un conjunto de operaciones sobre una estructura de datos de forma independiente
  - ***Observer***. Se define una dependencia entre uno a muchos, del tal manera, que cuando cambie avise a todos los objetos dependientes
  - ***Command***. 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
  - ***Memento***. Externaliza el estado interno de un objeto sin violar la encapsulación
  - ***Iterator***. Proporciona un modo de acceder secuencialmente a los elementos de un objeto sin exponer su representación interna
  - ***Strategy***. Define un conjunto de algoritmo haciéndolos intercambiables dinámicamente
  - ***Mediator***. Define un objeto que encapsula como interactúan un conjunto de ellos
- Ámbito: ***Clase***
  - ***Interpreter***. Define una representación de la gramática de un lenguaje con un intérprete
  - ***Template Method***. Define una operación en el esqueleto de un algoritmo, delegando en las subclases los detalles

Relación entre patrones



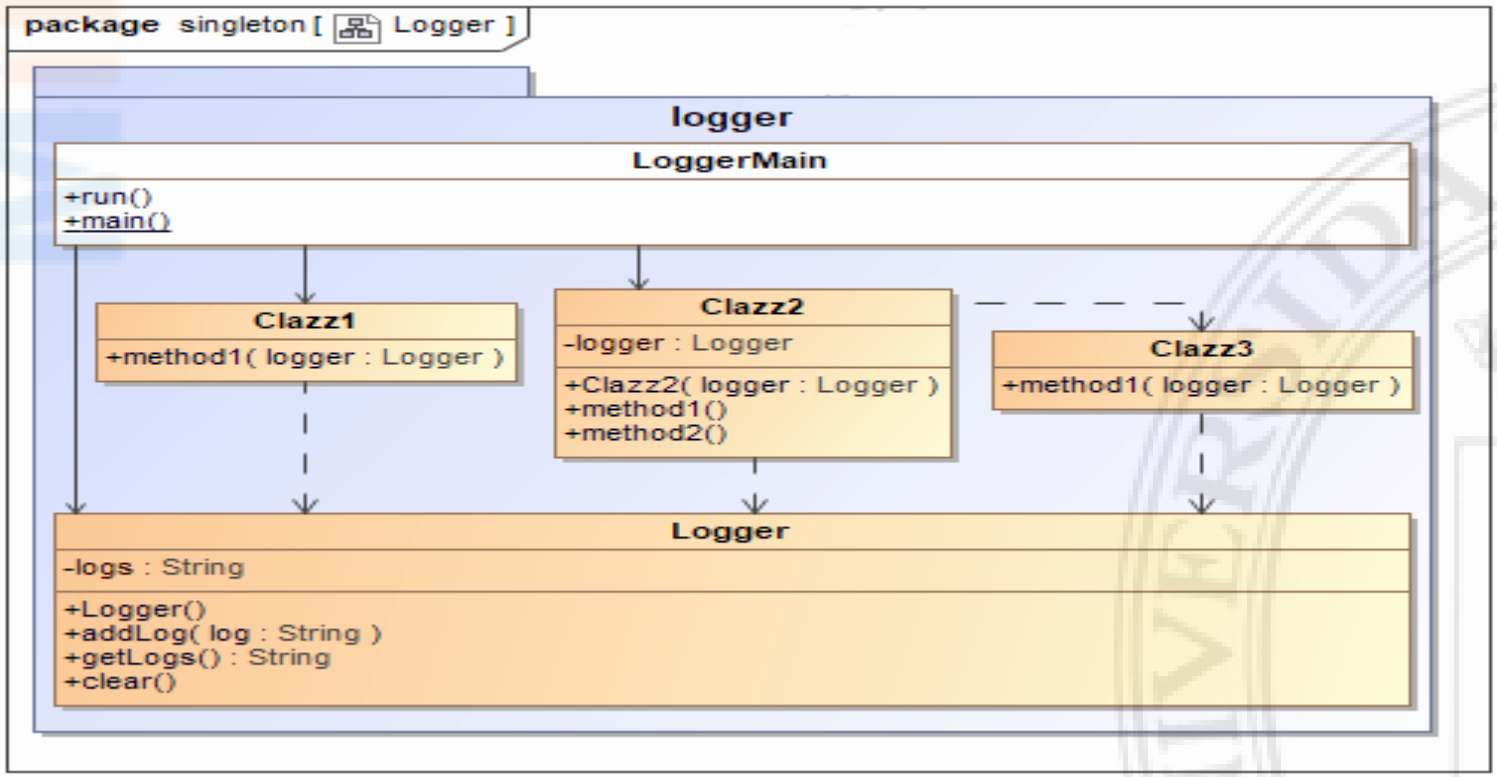
Seleccionar un patrón

- Los patrones soluciona problemas de diseño: diseñar para el cambio
- Releer los propósitos de los patrones
- Analizar las relaciones entre patrones
- Estudiar patrones similares
- Estudiar las causas del rediseño
- Pensar en los posibles cambios y si el diseño está preparado

- *GitHub: https://github.com/miw-upm/apaw*

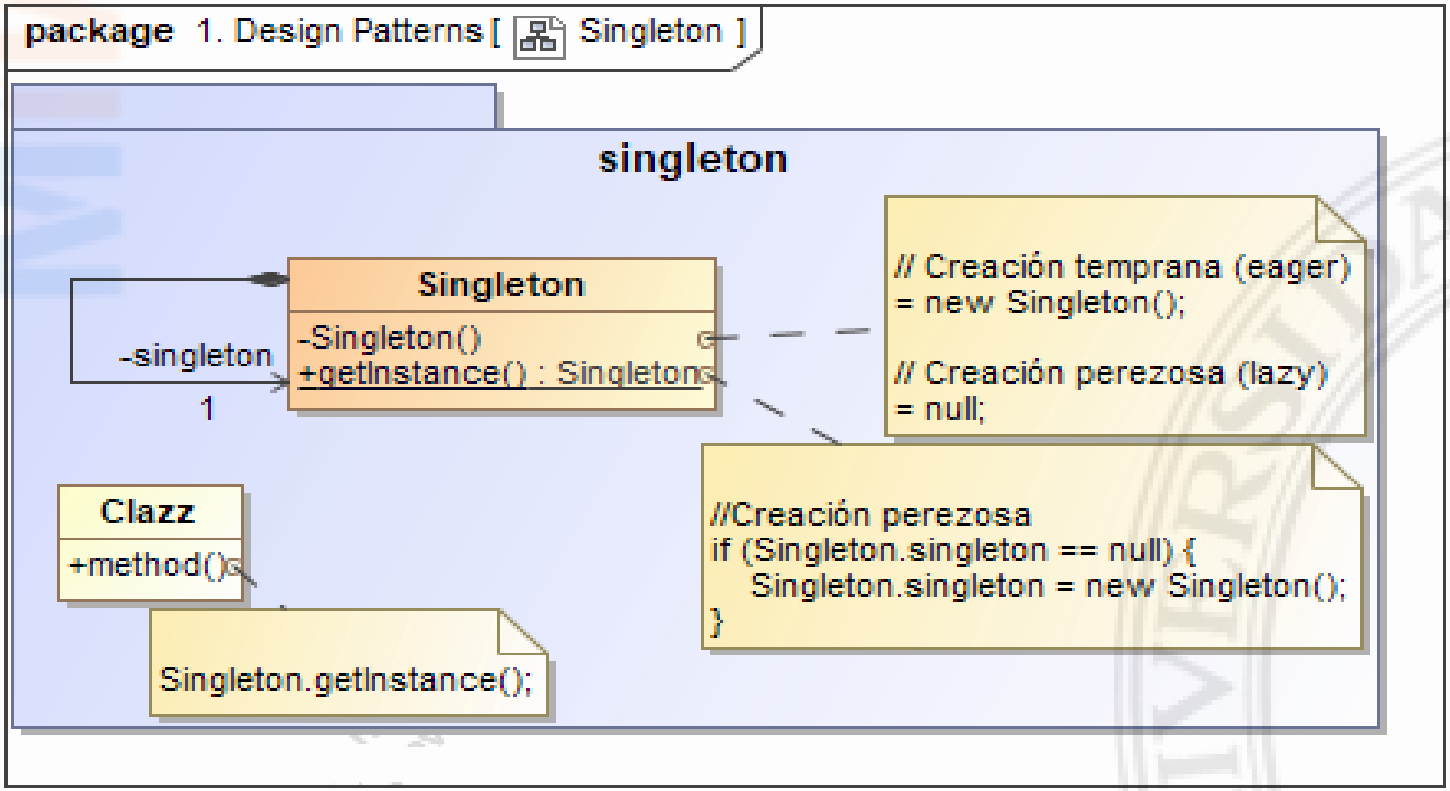
Singleton (Único)

- Motivación
  - En una aplicación se necesita que exista un solo objeto de una clase, Gestor de Ficheros, Cola de Impresión, Gestor de Registros, Gestor de BD..., y además, sea accesible desde el resto de clases de la aplicación
- Fuentes: paquete singleton



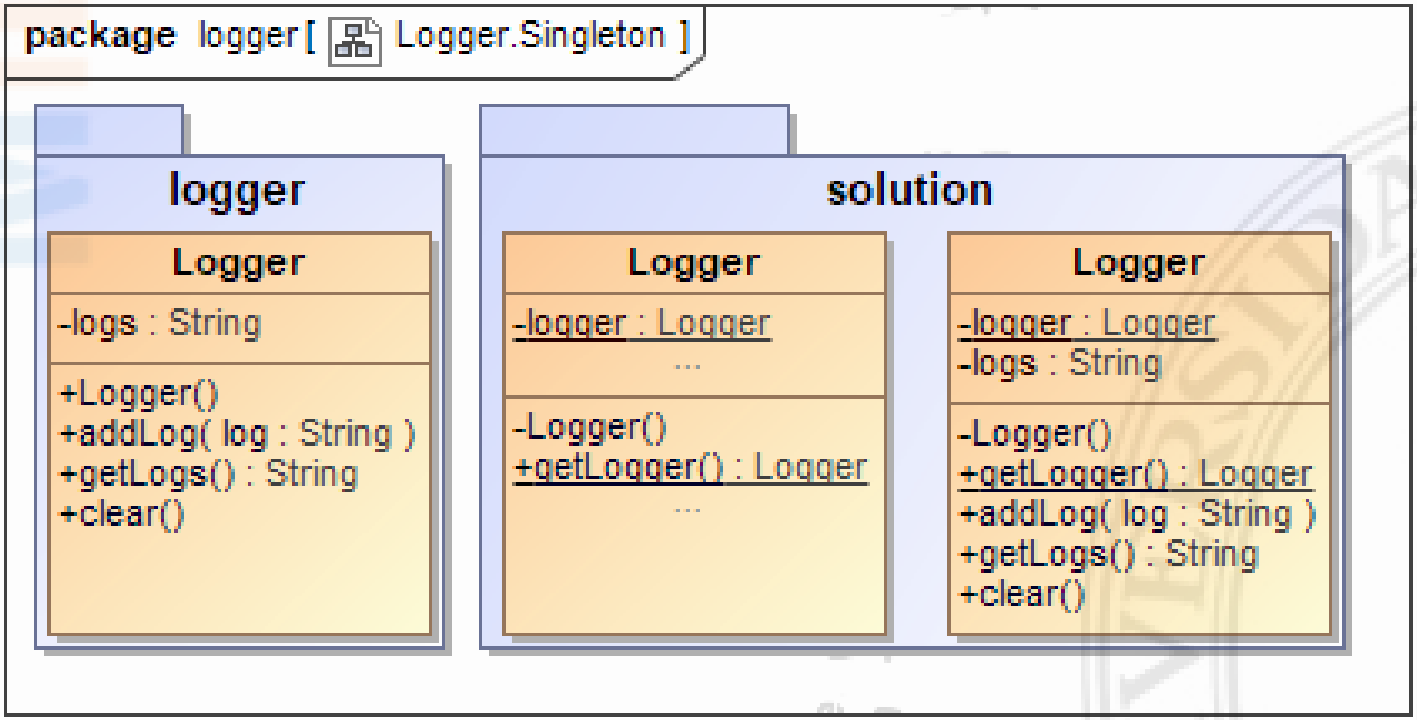
Singleton (Único)

- Propósito
  - Se garantiza que una clase sólo tenga una instancia y se proporciona un acceso global a ella
- Estructura



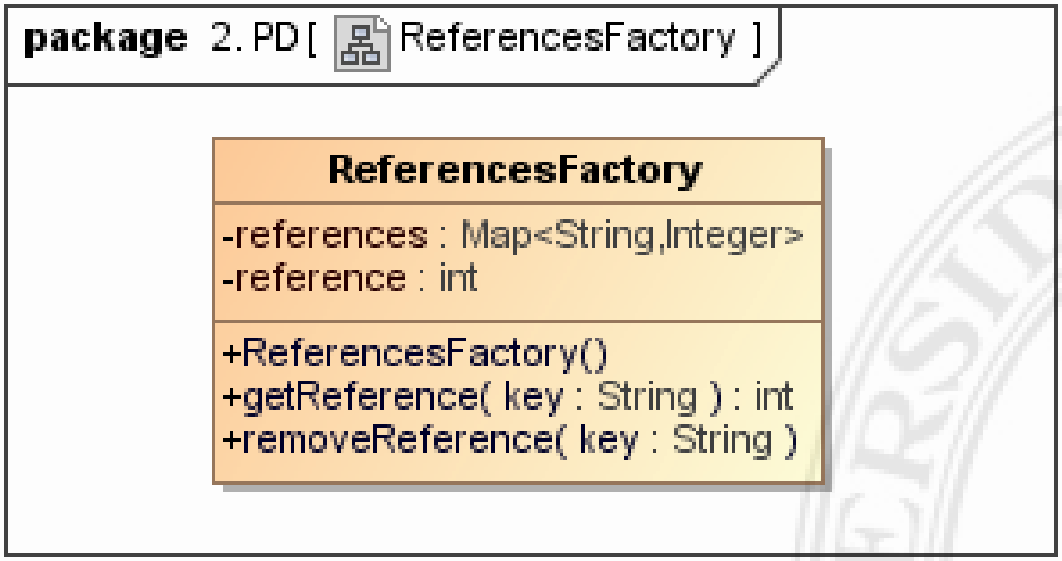
Logger

- Roles:
  - Singleton: Logger
    - + Atributo privado estático de la Clase
    - + Constructor privado
    - + Método *get* estático y público, para devolver el atributo



Factory

- Aplicar el patrón Singleton a ReferencesFactory, con creación temprana
  - Refactorizar ***ReferencesFactory*** respetando la funcionalidad original

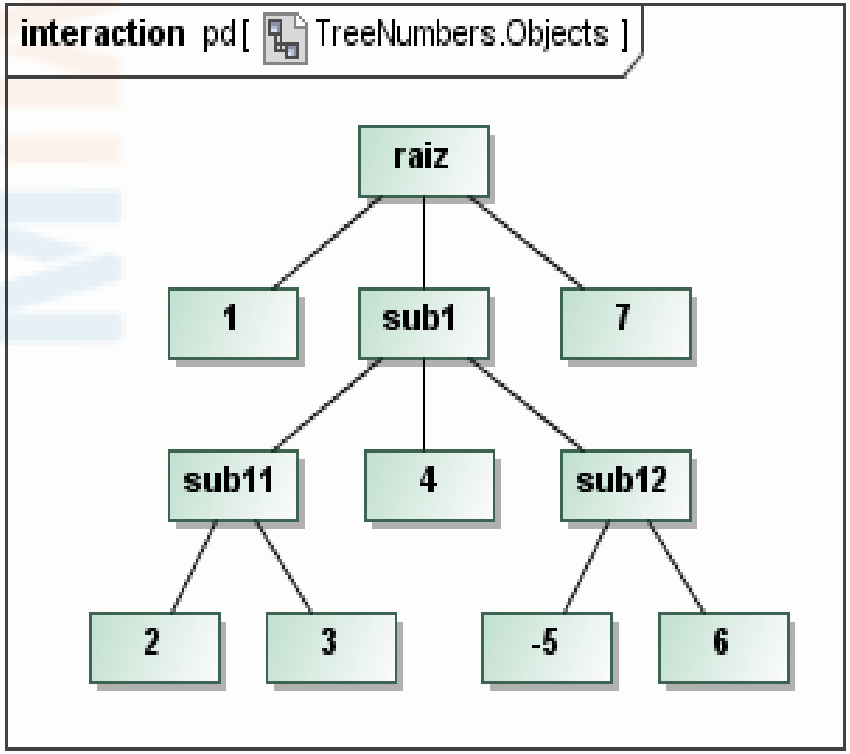


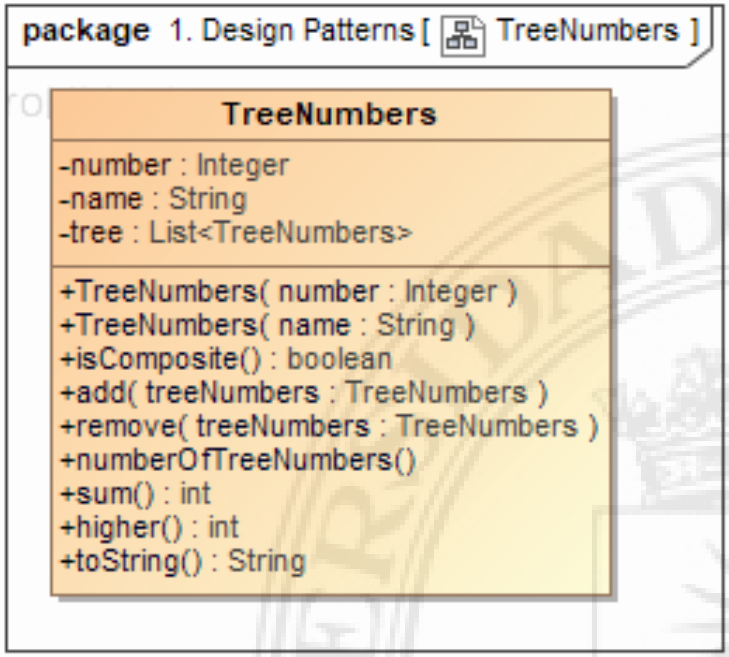
Composite (Compuesto)

- Motivación
  - En las aplicaciones gráficas, se permiten realizar dibujos por la agrupación de elementos simples y otros elementos agrupados
- Fuentes: paquete composite
- Árbol de números
  - Se quiere construir una estructura de árbol con valores numéricos. Existen nodos con valores numéricos (hojas) y nodos que contienen otros nodos (compuestos)
  - Si se intenta añadir a un nodo hoja, se debe lanzar la excepción: *UnsupportedOperationException*. Si se intenta borrar nodos a una hoja no se hace nada y no genera excepción
  - Si se intenta añadir o borrar con parámetro *null*, no hace nada y no genera excepción
  - Se deben crear las operaciones *numberOfNodes*, *sum* y *higher*, se opera sobre si mismo y todos los nodos que dependen de él

Composite (Compuesto)

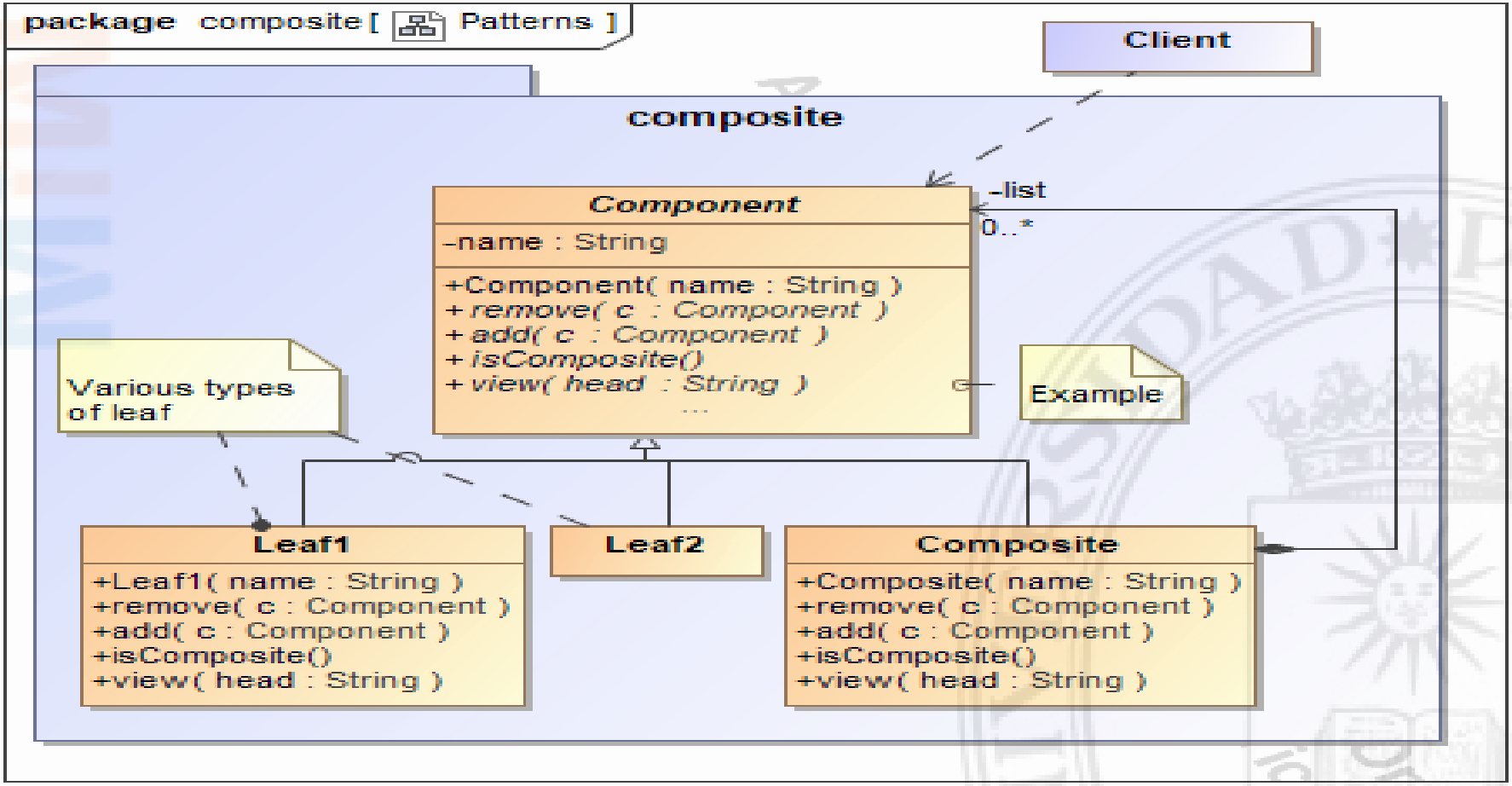
- Clase *TreeNumbers*, Clase *TreeNumbersTest*
- ¿Tiene alta cohesión? ¿Cumple los principios del DOO?
  - **SOLID** (*Single responsibility*), **Anti-Pattern**(*Spaghetti code*)
- ¿Y si se quiere añadir un nuevo tipo de hoja ☹?





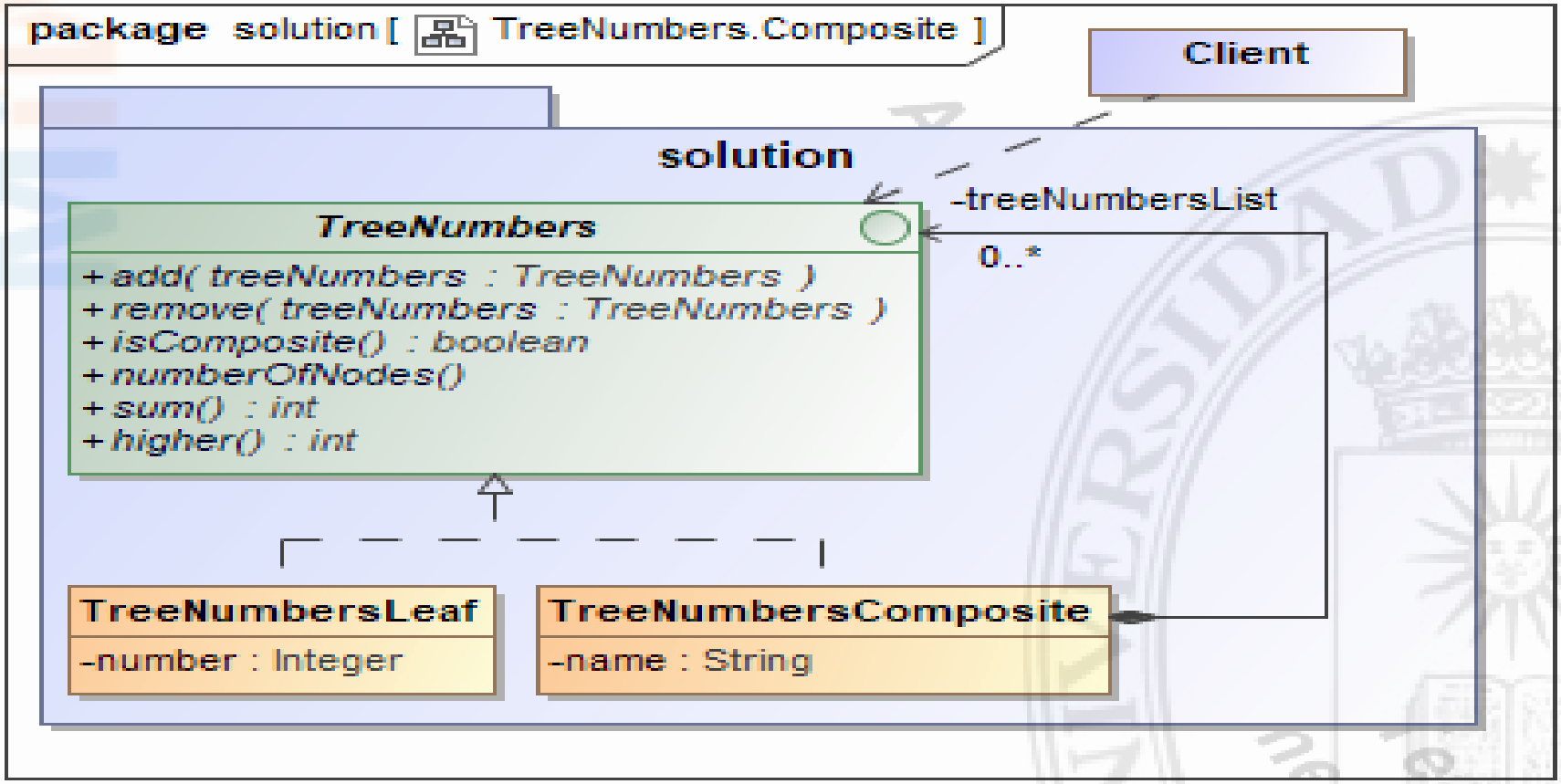
Composite (Compuesto)

- Propósito
  - Permite estructuras en árbol tratando por igual a las hojas que a los elementos compuestos



TreeNumbers

- Roles:
  - Component: TreeNumbers
  - Composite: TreeNumbersComposite
  - Leaf: TreeNumbersLeaf

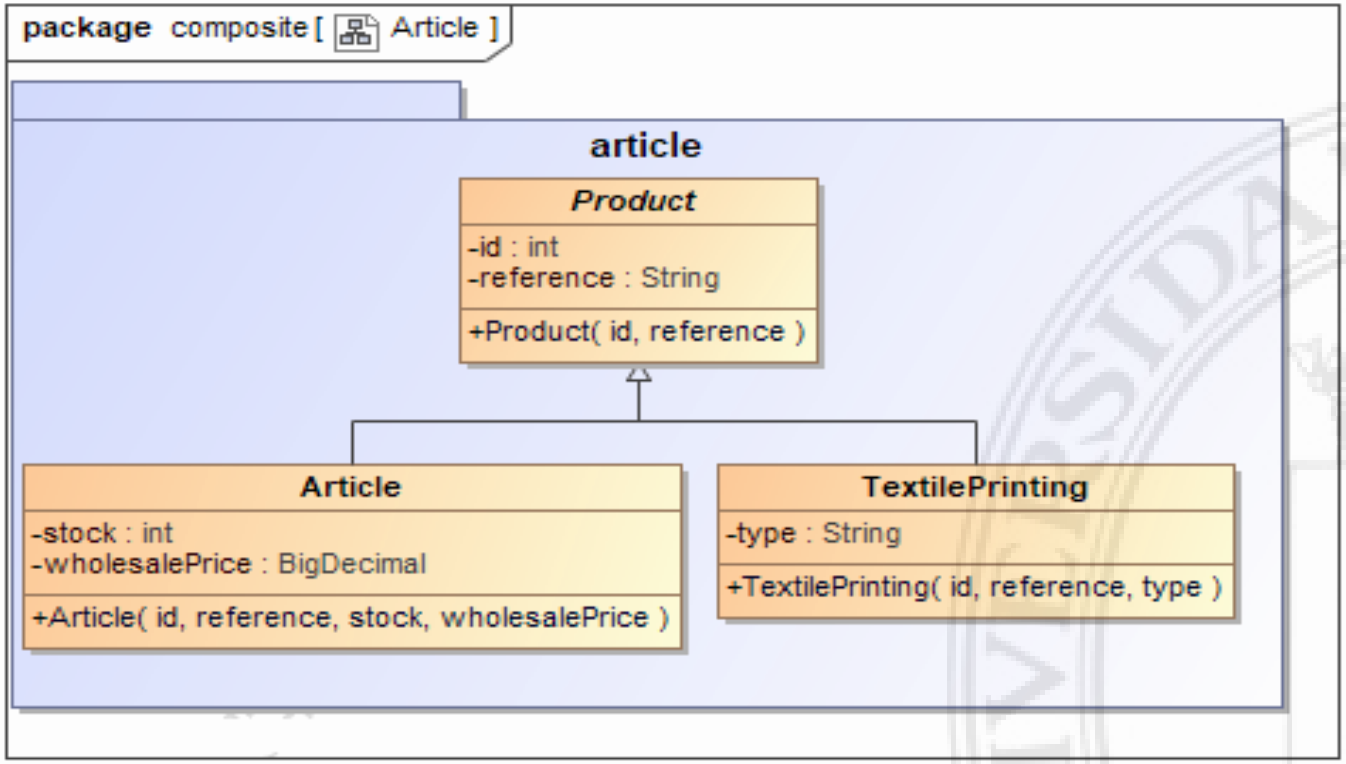


Expression

- Se quiere construir un editor de expresiones matemáticas con valores enteros. Especificar el diagrama de clases que permita representar las expresiones
- Una expresión válida estará formada o bien por un número, o bien por la suma, resta, división o multiplicación de dos expresiones
- Ejemplos de expresiones válidas:
  - 5
  - (1+(8*3))
  - ((7+3) * (1+5))

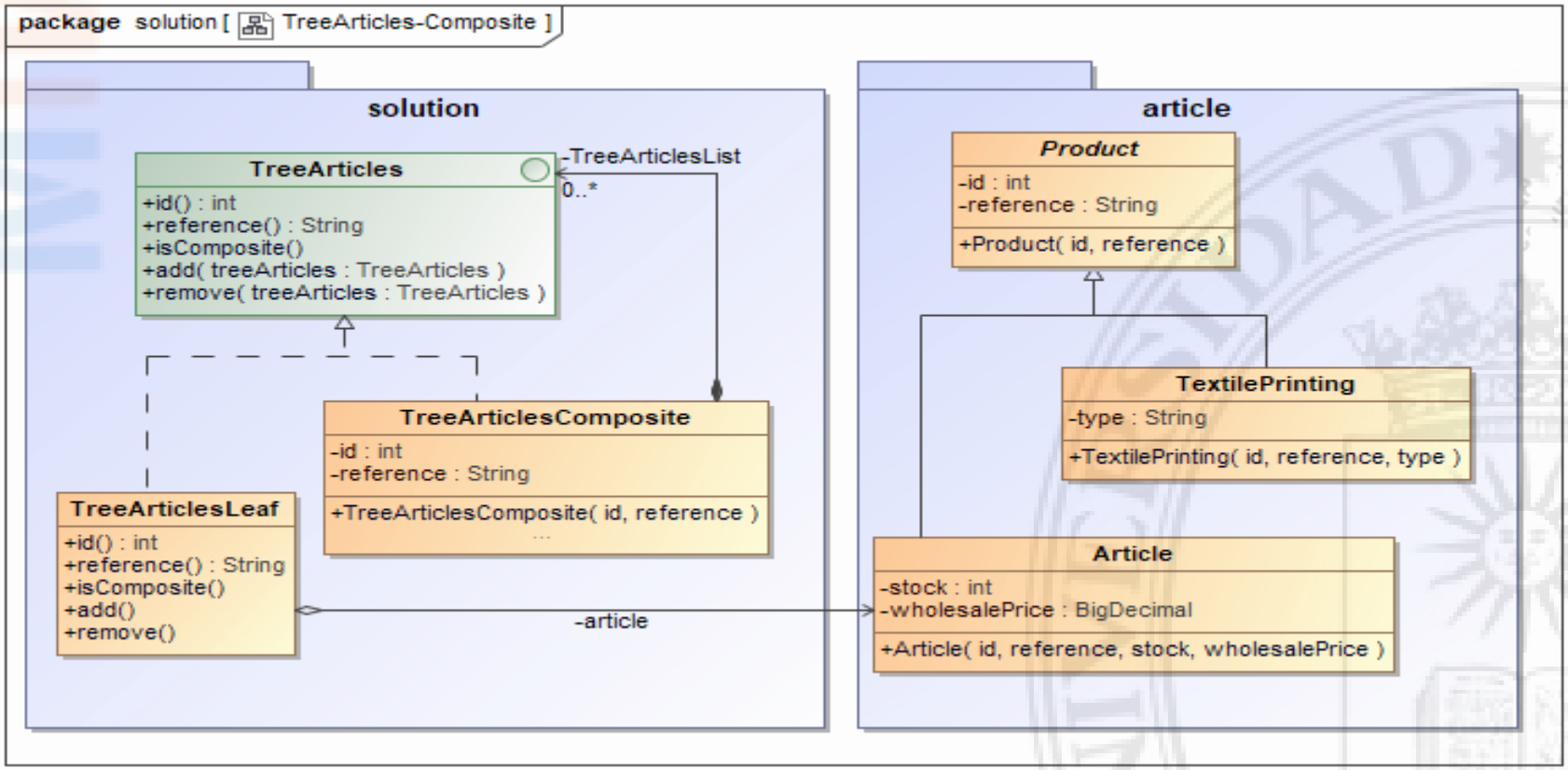
Article

- Se parte de un código en producción, y **no se quieren alterar las clases existentes**
- Aplicar el patrón compuesto para realizar agrupaciones en árbol de solo artículos



Article

- Se parte de un código en producción, y **no se quieren alterar las clases existentes**
- Aplicar el patrón compuesto para realizar agrupaciones en árbol de solo artículos

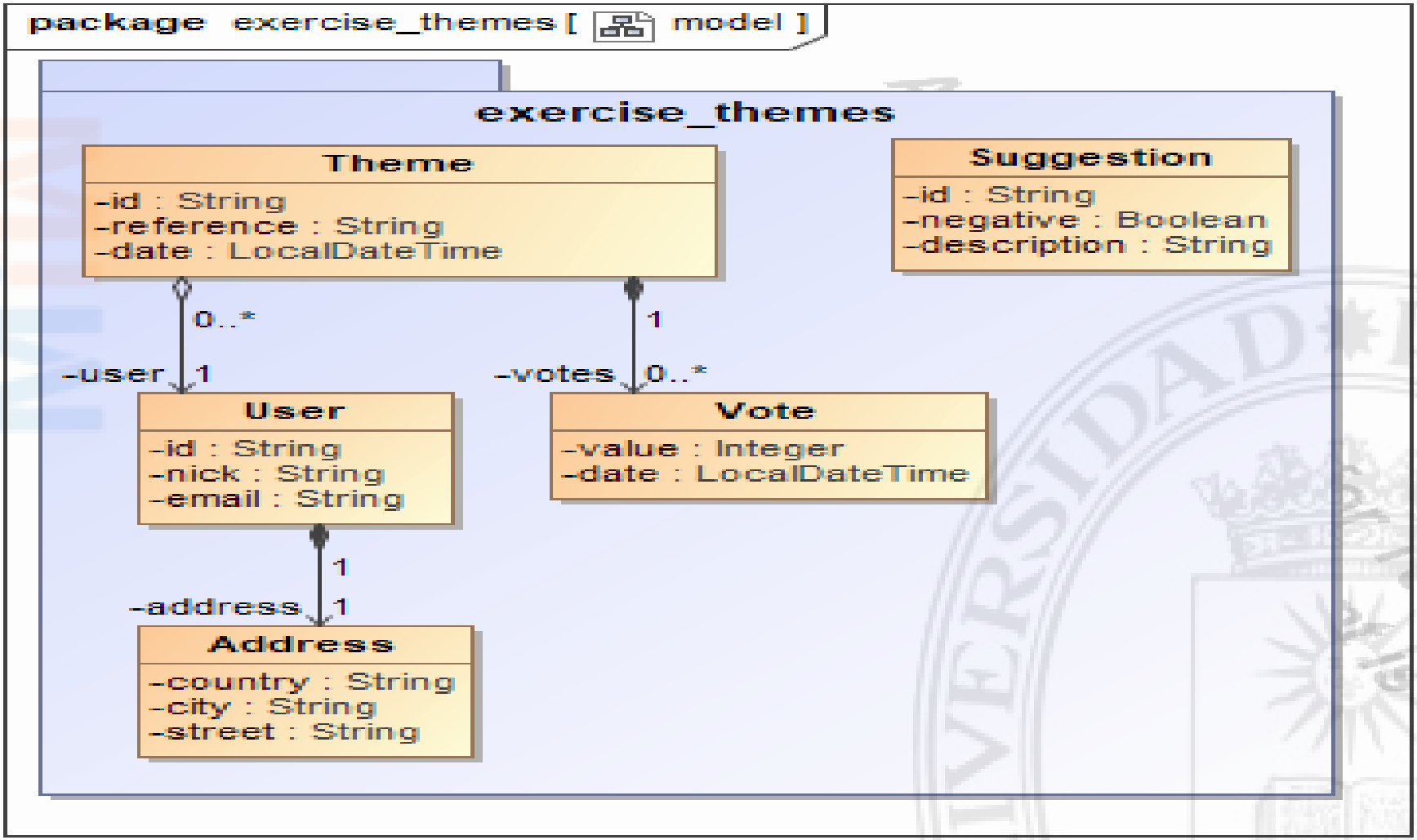


Práctica de APAW

1. Establecer el esquema UML entre las entidades
   - Cinco entidades, con un mínimo de 2 atributos, y si se guarda en BD, otro atributo *Id* de tipo *String*.
   - Un mínimo de 14 atributos entre todas las entidades (incluidos *id*).
   - Debe existir alguna entidad con relación de *n..1*.
   - Debe existir alguna entidad con relación de *1..n*.
   - Debe existir alguna entidad con relación *1..1* o *n..n*.
   - Alguna entidad no debe tener relaciones
   - Algún atributo con:
     - LocalDateTime
     - Boolean
     - Integer

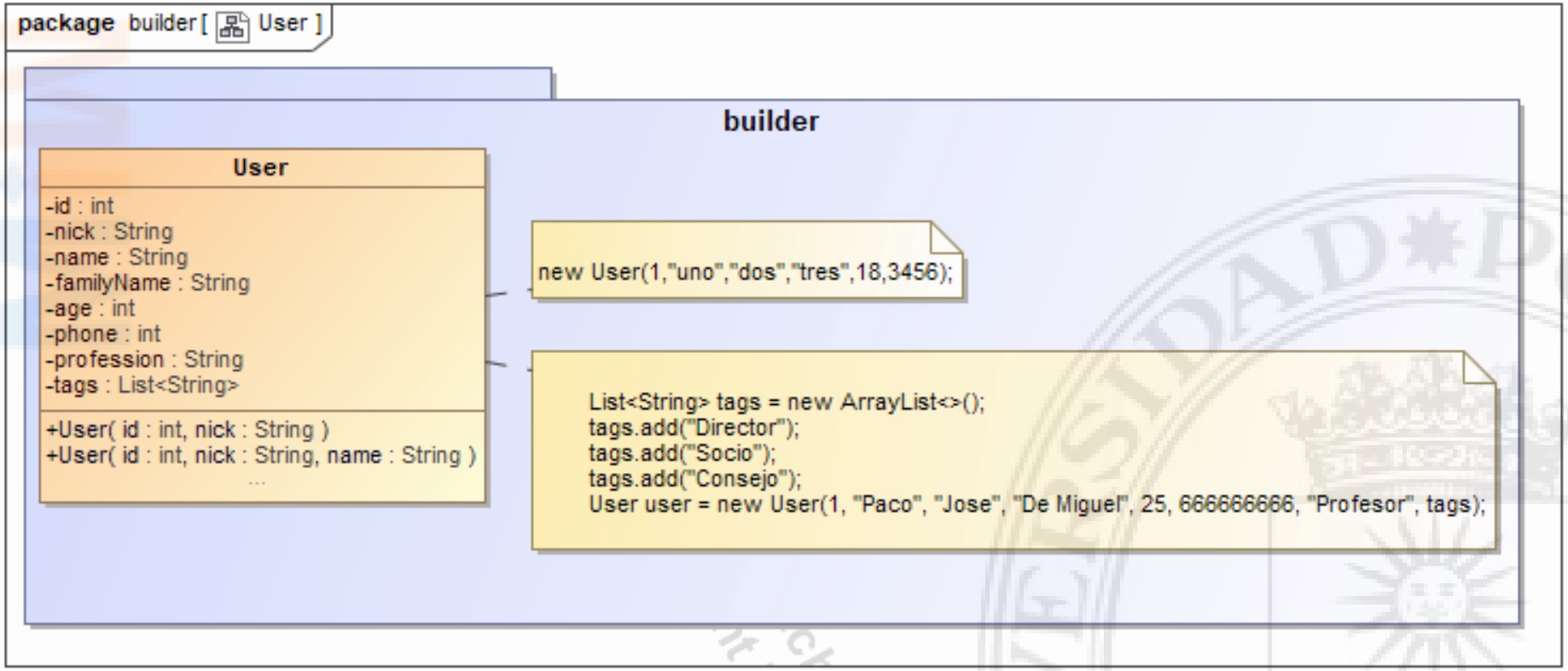
Práctica de APAW

- Práctica a modo de ejemplo: *Themes*



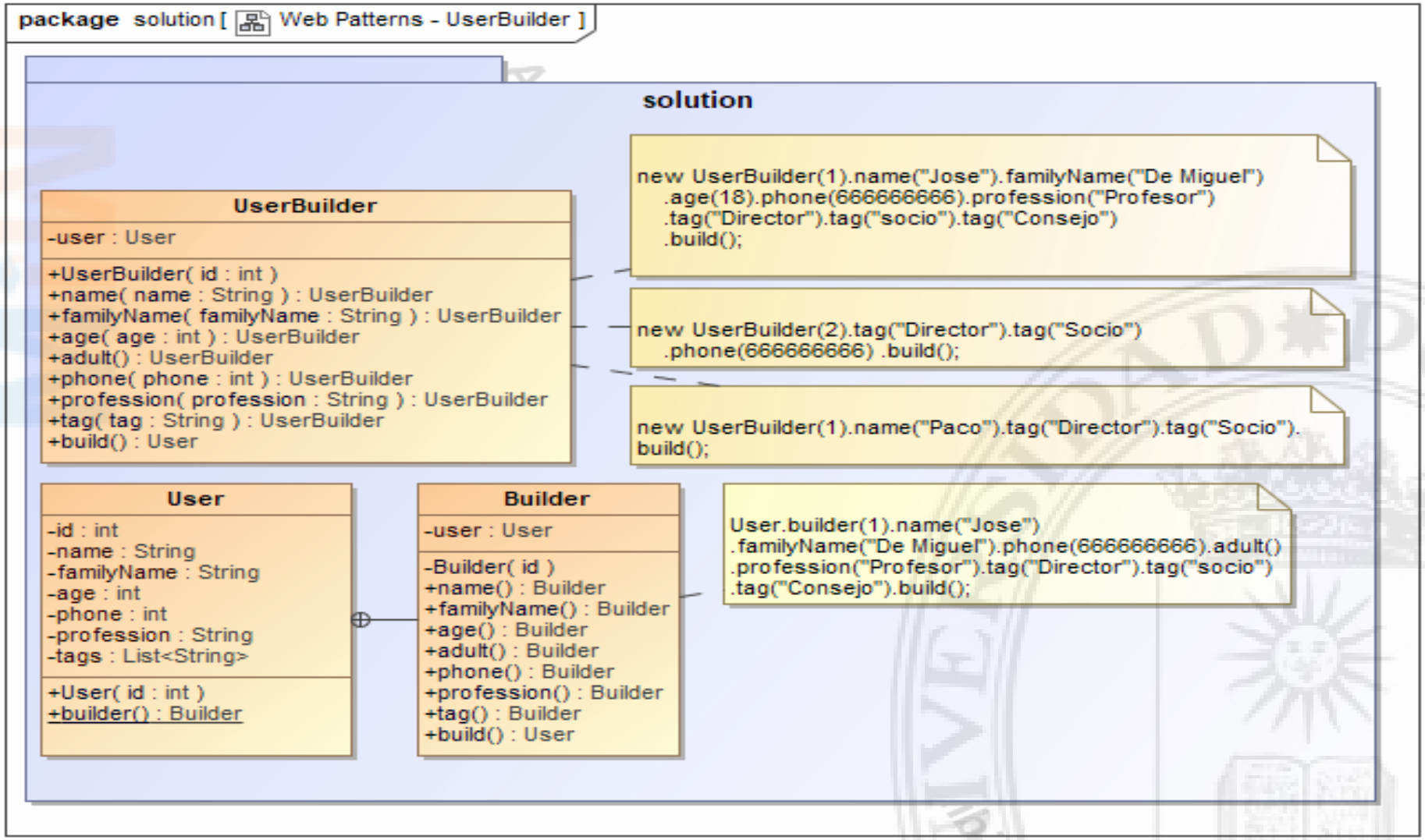
Builder (Constructor)

- Motivación
  - Este patrón permite tener una política general para la creación de objetos, centralizándolo en una clase constructora



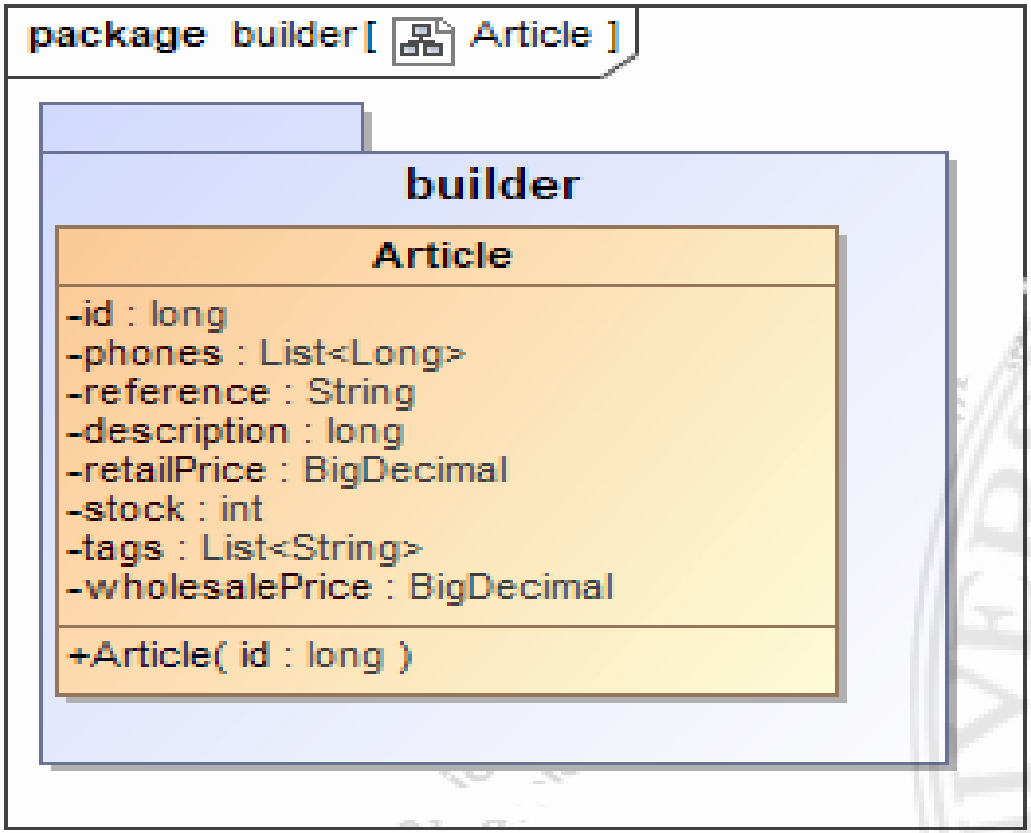
Builder. user

- UserBuilder



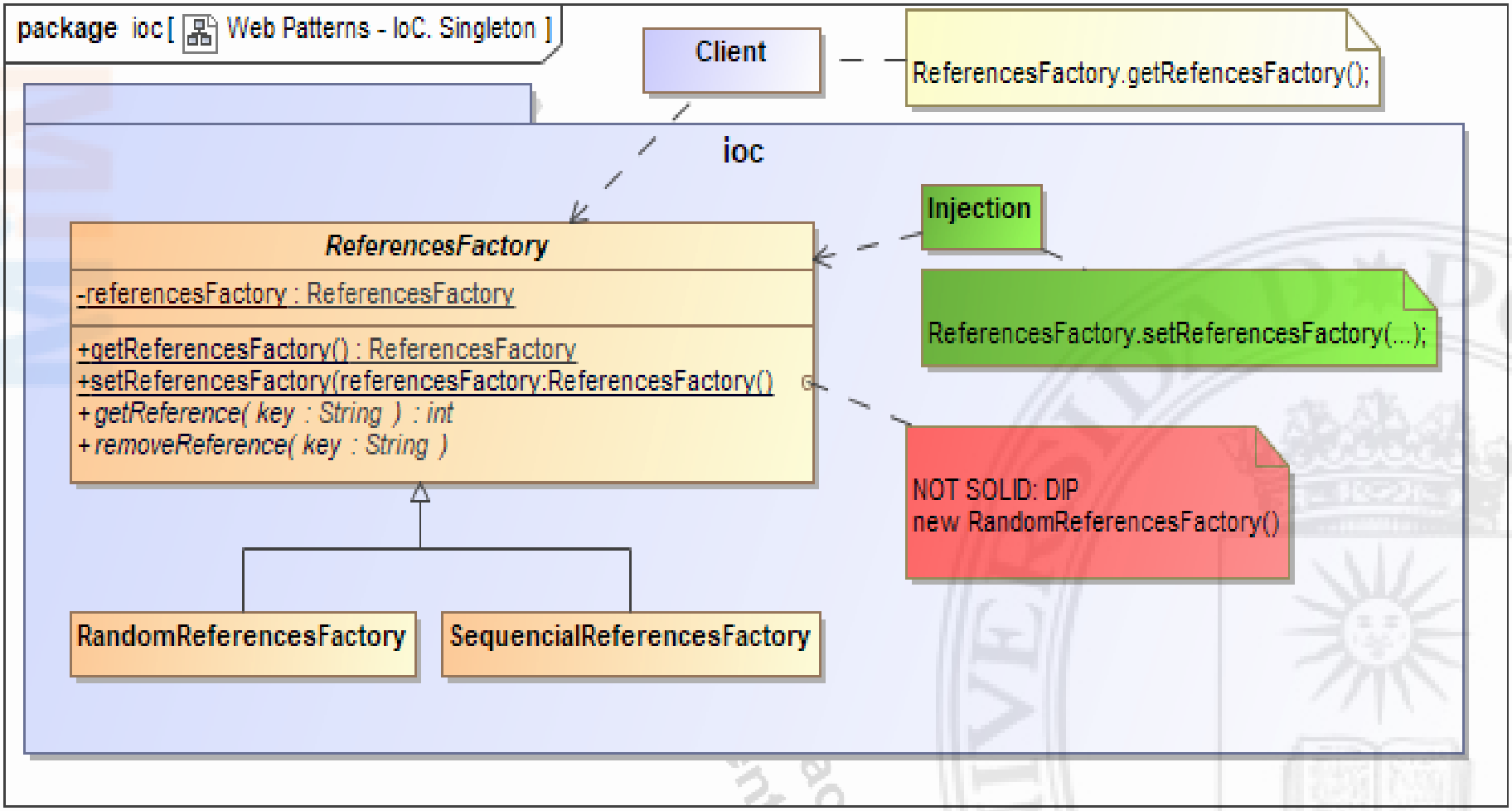
Article

- Aplicar el patrón *Builder* a *Article*
  - *Article.builder(id).build();*
  - *Article.builder(id).tag("tag1").build();*
  - *Article.builder(id).phone(555555555).reference("referencia").tag("tag1").tag("tag2").build();*



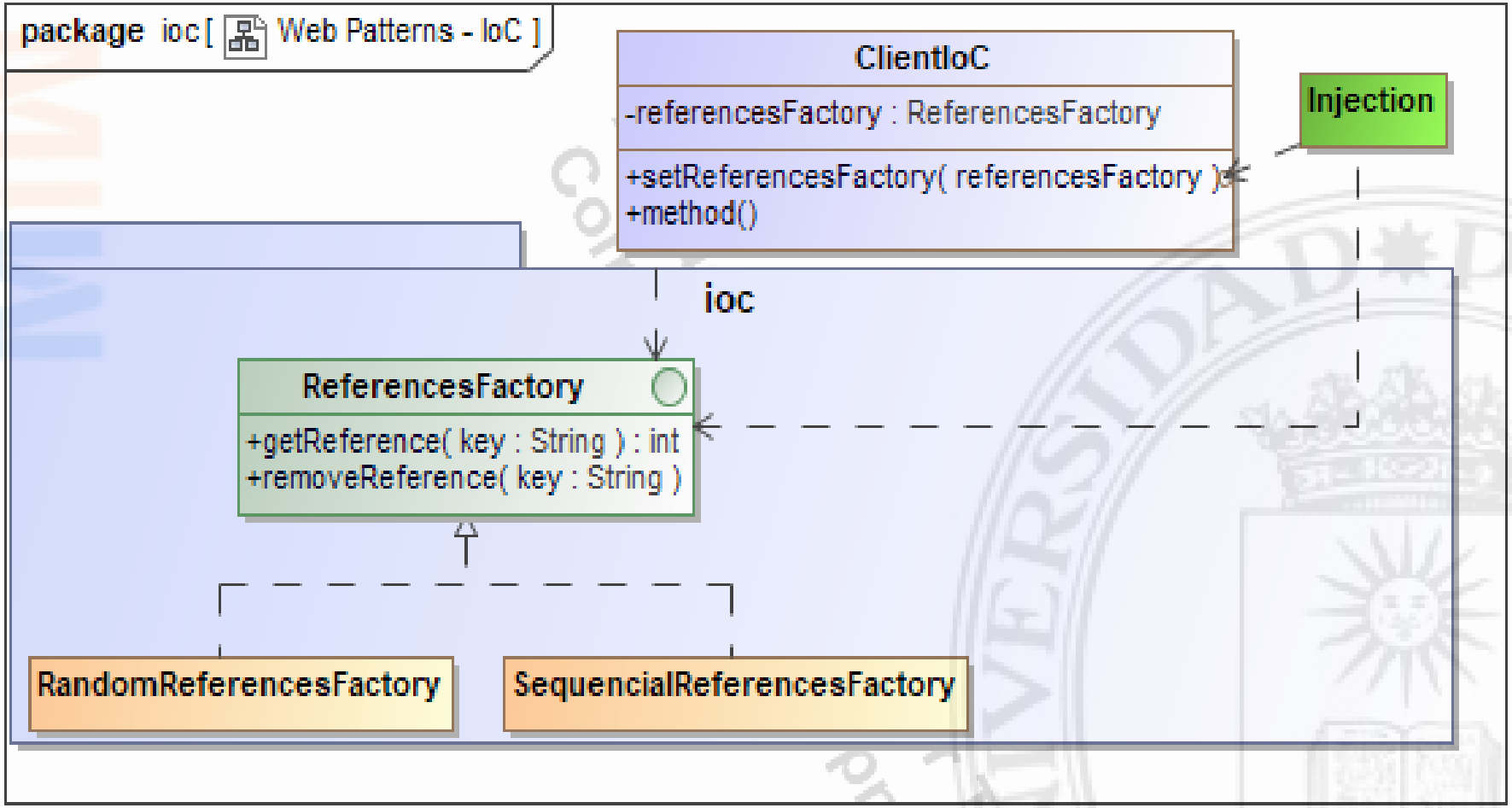
Inversion of control

- En una aplicación se necesita que exista una solo objeto de una clase *ReferencesFactory (abstract class)*



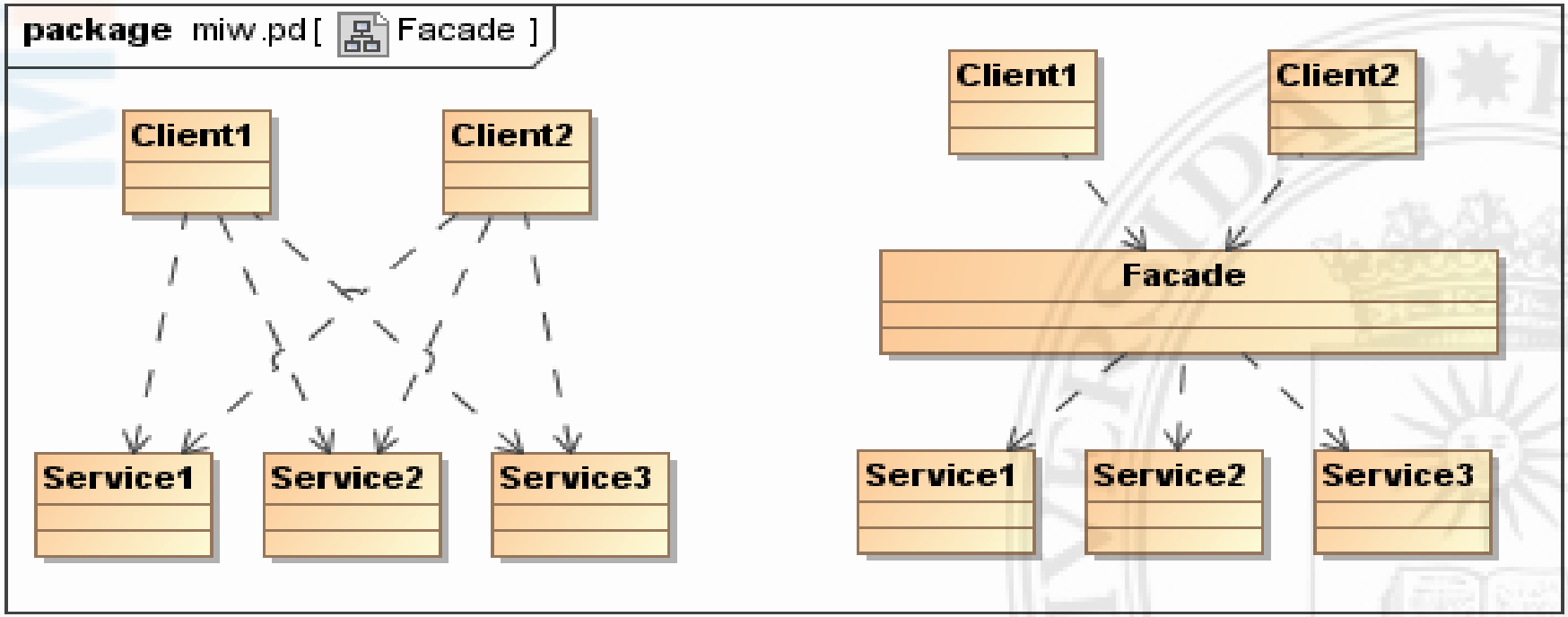
Inversion of control

- En una aplicación se necesita que exista una solo objeto de del tipo *ReferencesFactory (interface)*



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
- *En cualquier librería de terceros, se debe aplicar una Fachada*

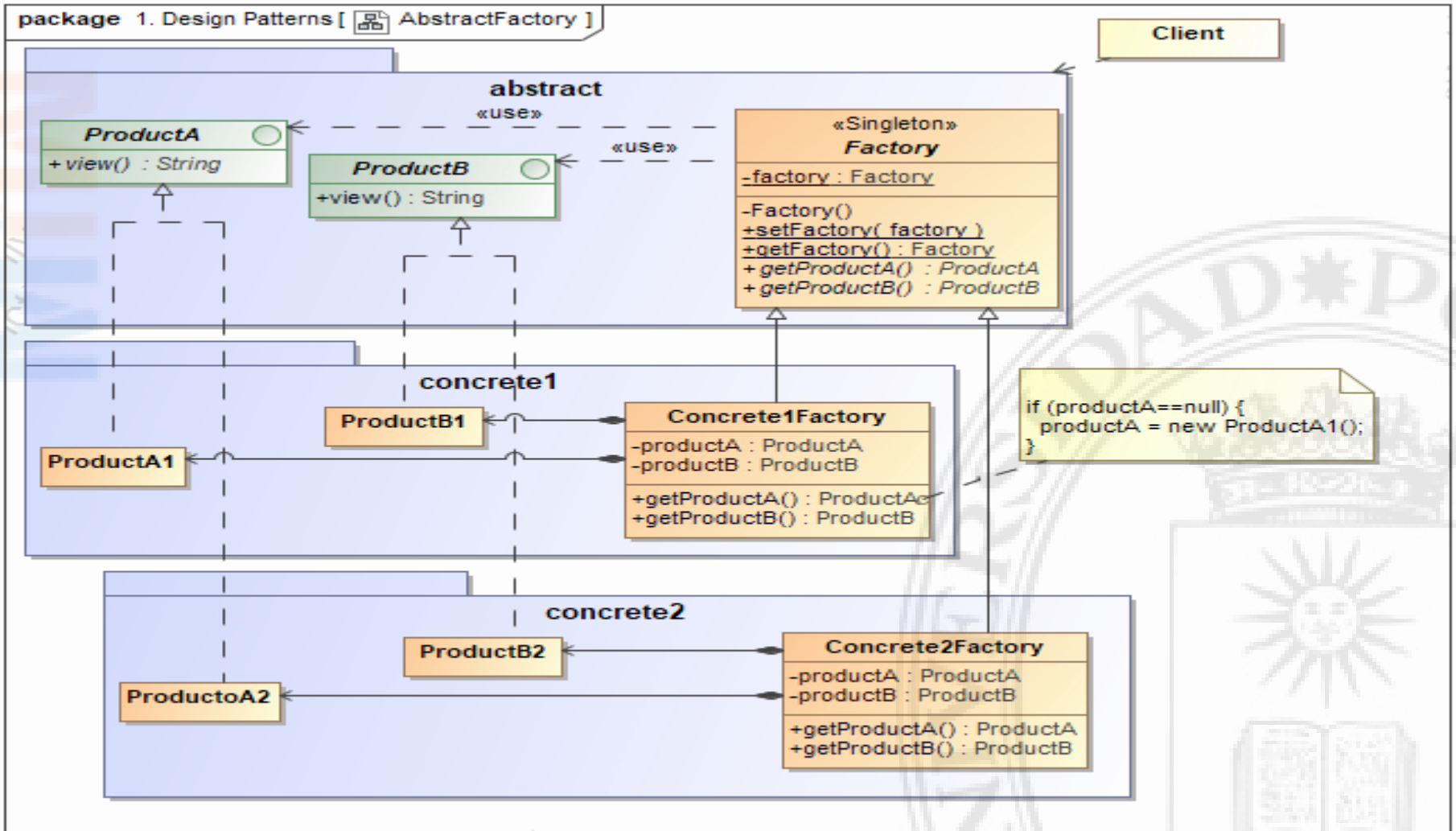


Abstract Factory (Fábrica abstracta)

- También conocido
  - Kit
- Motivación
  - Se permiten múltiples interface de usuario
- Fuentes: paquete abstractFactory

Abstract Factory. Implementación

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

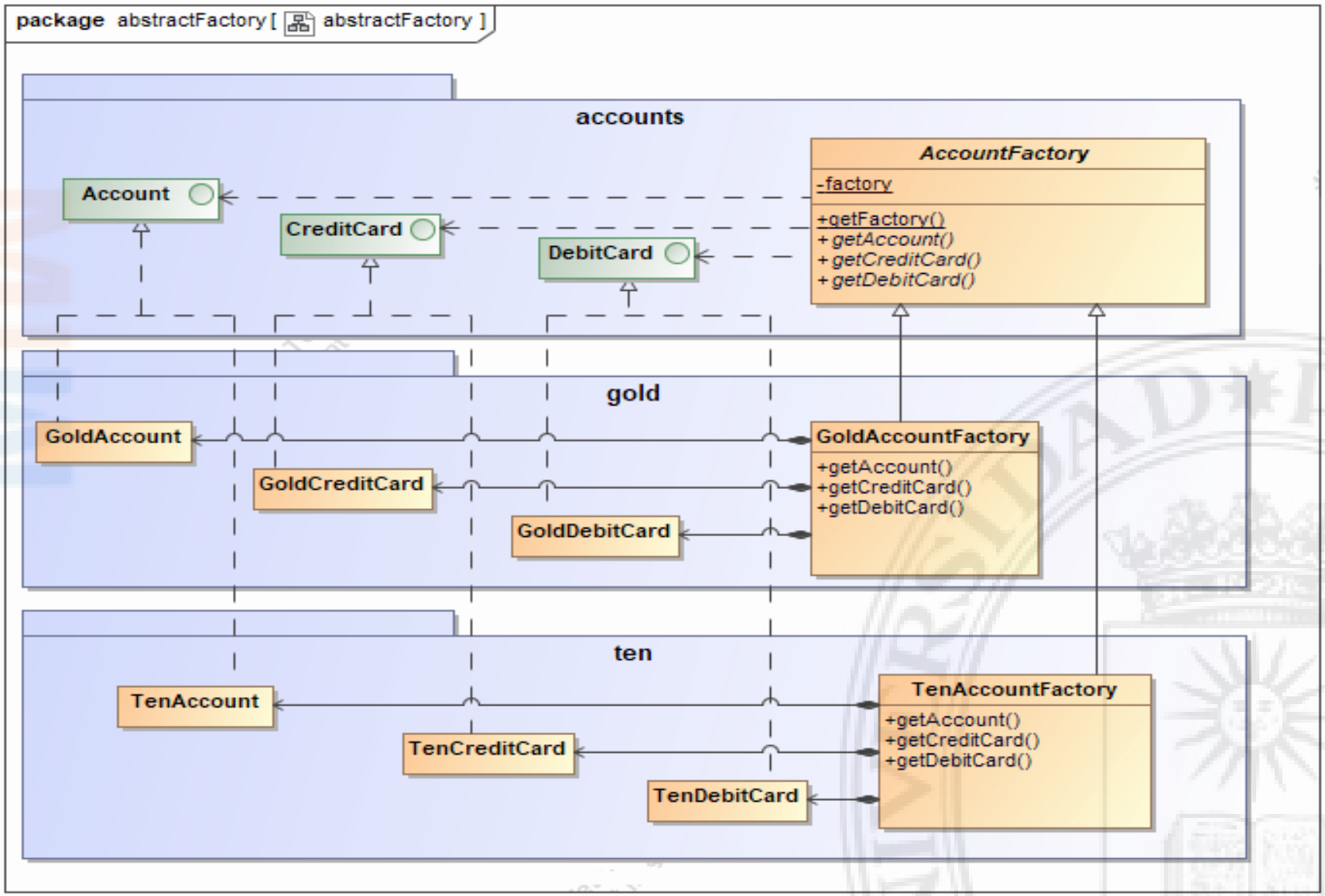


Cuentas

- Fabricas: ***Joven, 10*** y ***Oro***
- Productos: ***Cuenta, Tarjeta débito*** y ***Tarjeta crédito***
- Métodos de productos: *show(): String*
- *Total: 16 entre clases e interfaces*

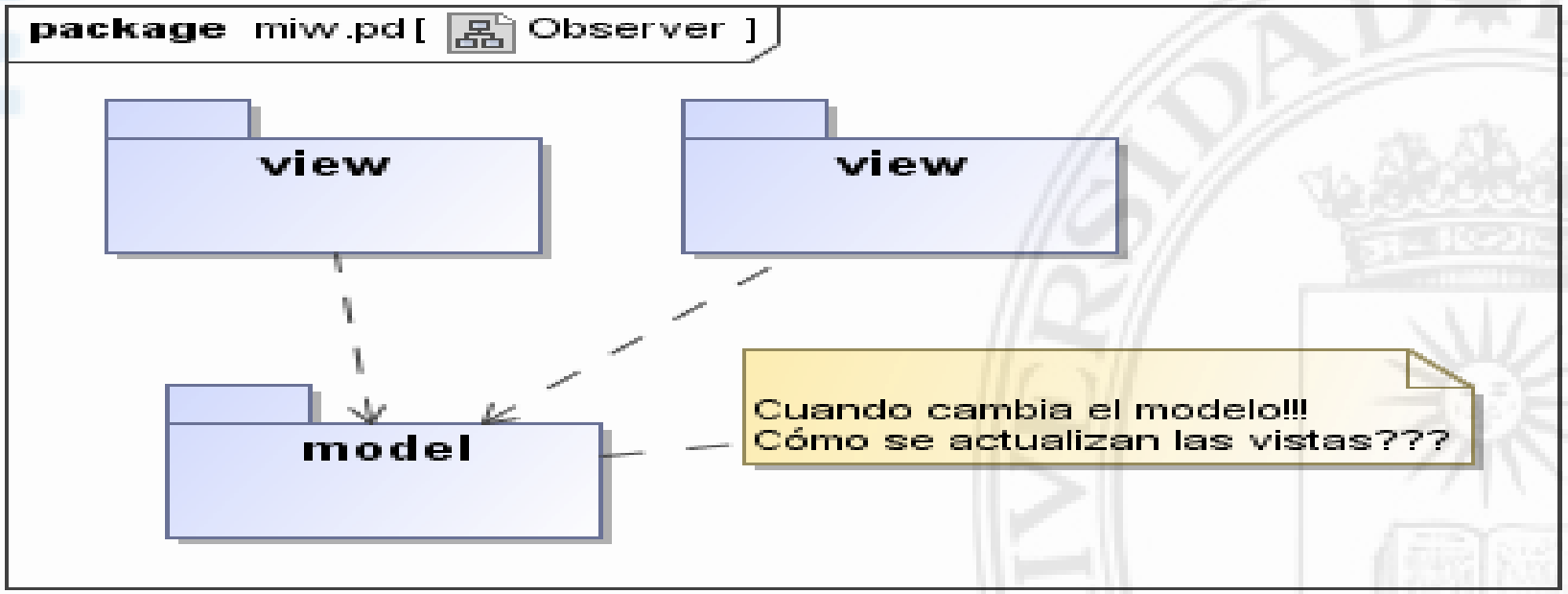
Tipo Cuenta	Cuenta	Tarjeta débito	Tarjeta crédito
Joven	1%	Gratuita	Gratuita Max.600
10	1´5%	Gratuita	10 € Max. 2000€
Oro	2%	5 €	20€ Max. 4000€

Cuentas


package abstractFactory [abstractFactory]
accounts
AccountFactory
-factory
+getFactory()
+getAccount()
+getCreditCard()
+getDebitCard()
Account
CreditCard
DebitCard
gold
GoldAccount
GoldCreditCard
GoldDebitCard
GoldAccountFactory
+getAccount()
+getCreditCard()
+getDebitCard()
ten
TenAccount
TenCreditCard
TenDebitCard
TenAccountFactory
+getAccount()
+getCreditCard()
+getDebitCard()

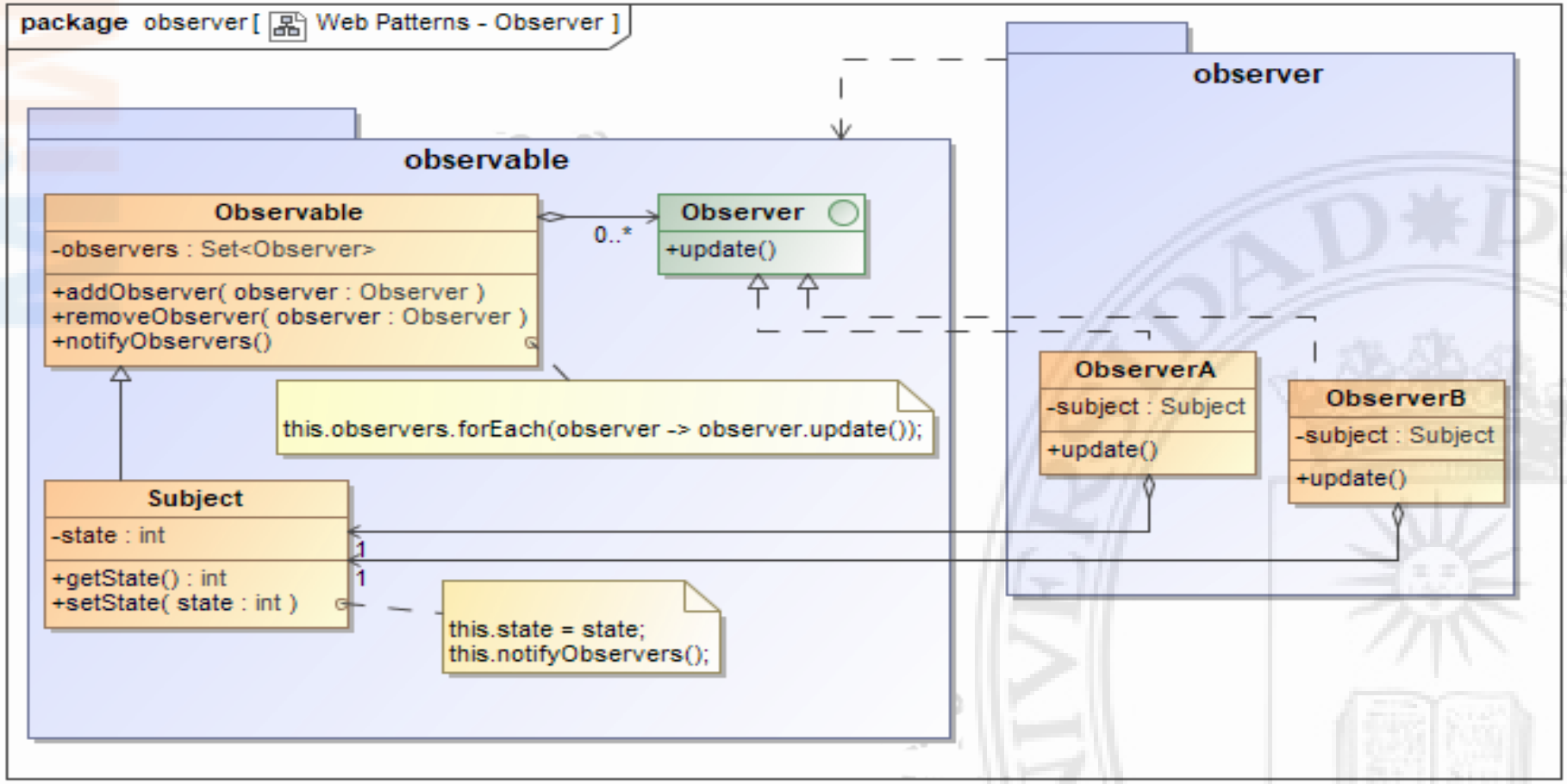
Observer (Observador)

- También conocido
  - Dependents (Dependiente), Publish-Subscribe (Publicar-Suscribir)
- Motivación
  - Muchas veces se separa los datos en si de su representación (MVC), pudiendo tener varias representaciones de un mismo dato
- Fuentes: paquete observer



Observer (Observador)

- Propósito
  - Se define una dependencia entre uno a muchos, del tal manera, que cuando cambie avise a todos los objetos dependientes



Lambda

```
public class Lambda {

    public Consumer<String> functionConsumer() { // accept(T)
        return msg -> LogManager.getLogger(this.getClass()).info( S: "Consumer: " + msg);
    }

    public Function<String, Integer> function() { // apply(T):R
        return msg -> Integer.parseInt(msg);
    }

    public Function<String, Integer> function2() { // apply(T):R
        return Integer::new;
    }

    public Predicate<String> functionPredicate() { // test(T):boolean
        return "two"::equals;
    }

    public Predicate<String> functionPredicate2() { // test(T):boolean
        return msg -> "two".equals(msg);
    }

    public Supplier<String> functionSupplier() { // get(): T
        return () -> {
            String prefix = "...";
            return prefix.concat("-");
        };
    }
}
```

Lambda

```
class LambdaTest {
    @Test
    void testFunctionConsumer() {
        Stream.of("one", "two", "three").forEach(new Lambda().functionConsumer());
        Stream.of("one", "two", "three").forEach(msg -> LogManager.getLogger(this.getClass()).info( S: "Consumer: " + msg));
    }
    @Test
    void testFunction() {
        List<Integer> list1 = Stream.of("1", "2", "3").map(new Lambda().function()).collect(Collectors.toList());
        List<Integer> list2 = Stream.of("1", "2", "3").map(Integer::new).collect(Collectors.toList());
        LogManager.getLogger(this.getClass()).info( S: "Function: " + list1 + ", " + list2);
    }
    @Test
    void testPredicate() {
        List<String> list1 = Stream.of("one", "two", "three")
                .filter(new Lambda().functionPredicate()).collect(Collectors.toList());
        List<String> list2 = Stream.of("one", "two", "three")
                .filter("two"::equals).collect(Collectors.toList());
        LogManager.getLogger(this.getClass()).info( S: "Predicate: " + list1 + ", " + list2);
    }
    @Test
    void testSupplier() {
        List<String> list1 = Stream.generate(new Lambda().functionSupplier()).limit(3).collect(Collectors.toList());
        List<String> list2 = Stream.generate(() -> {
            String prefix = "...";
            return prefix.concat("-");
        }).limit(3).collect(Collectors.toList());
        LogManager.getLogger(this.getClass()).info( S: "Supplier: " + list1 + ", " + list2);
    }
}
```

Publisher-Subscribe

- Proyecto: Reactor
  - https://projectreactor.io/docs/core/release/reference
- Publisher
  - Un **Mono <T>** es un publicador
    - Emite de forma asíncrona 0 o 1 elemento y termina con una señal ***onComplete.***
    - Termina con una señal ***onError***.
  - Un **Flex<T>** es un publicador
    - Emite una secuencia asíncrona de 0 a N elementos y termina con una señal ***onComplete.***
    - Termina con una señal ***onError***.
- Subscribe
  - subscribe(
    - **Consumer<? super T> consumer,**
    - **Consumer<? super Throwable> errorConsumer,**
    - **Runnable completeConsumer**
    - Consumer<? super Subscription> subscriptionConsumer);

Publisher-Subscribe

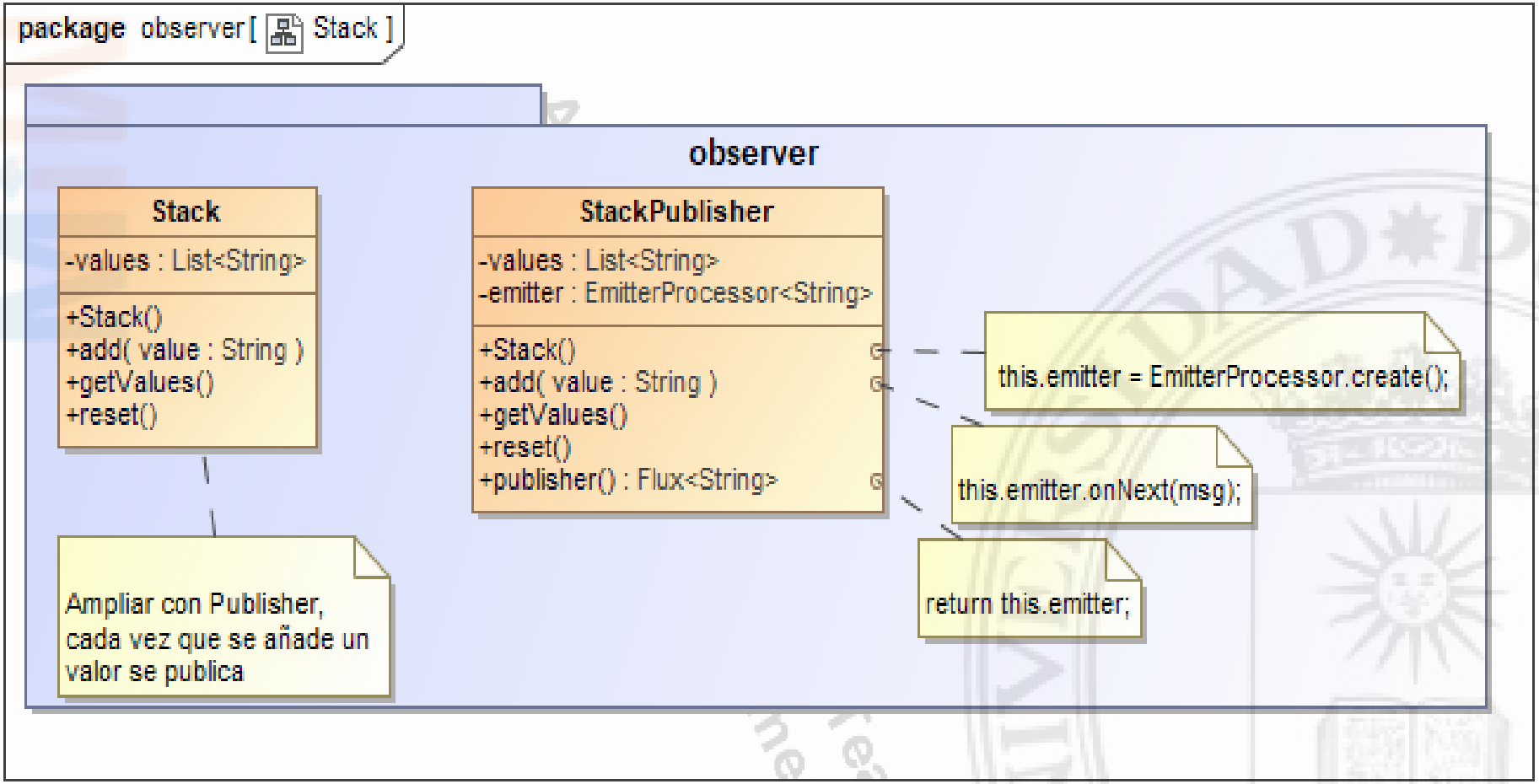
- Creación
  - *Mono.empty();*
  - *Mono.just("one");*
  - *Mono.error(new RuntimeException("mono-error"));*
  - *Mono.delay(Duration. ofSeconds(2)).map(value -> "One");*
  - *Flux.just("one","two","three");*
  - *Flux.interval(Duration.ofMillis(200)).map(value -> "A" + value).take(8);*
- Subscripción
  - *publisher.subscribe(onNext, onError, onCompleted, onSubscribe);*
  - *publisher.subscribe(msg -> System.out.println(msg));* ***// asynchronous***
  - *publisher.block();* ***// synchronous***
- Unión
  - *publisher=publisher1.mergeWith(publisher2);*
- Sincronización
  - *publisher=Mono.when(publisher1, publisher2);*

Publisher-Subscribe

```
class ReactiveProgrammingTest {
    @Test
    void testMonoEmpty() {
        new ReactiveProgramming().monoEmpty().subscribe(
                msg -> LogManager.getLogger(this.getClass()).info( s: "Consumer: " + msg), //onNext
                throwable -> LogManager.getLogger(this.getClass()).info( s: "Error: " + throwable.getMessage()), //onError
                () -> LogManager.getLogger(this.getClass()).info( s: "Completed") //onComplete
        );
    }
    @Test
    void testMonoEmptyBlock() {
        new ReactiveProgramming().monoEmpty().block();
        LogManager.getLogger(this.getClass()).info( s: "Finish... testMonoEmptyBlock");
    }
    @Test
    void testMonoOne() {
        new ReactiveProgramming().monoOne().subscribe(
                msg -> LogManager.getLogger(this.getClass()).info( s: "Consumer: " + msg)
        );
    }
    @Test
    void testMonoError() {
        new ReactiveProgramming().monoError().subscribe(
                msg -> LogManager.getLogger(this.getClass()).info( s: "Consumer: " + msg),
                throwable -> LogManager.getLogger(this.getClass()).info( s: "Error: " + throwable.getMessage()),
                () -> LogManager.getLogger(this.getClass()).info( s: "Completed")
        );
    }
    @Test
    void testMonoDelay() throws InterruptedException {
        new ReactiveProgramming().monoDelay().subscribe(
                msg -> LogManager.getLogger(this.getClass()).info( s: "Consumer: " + msg),
                throwable -> LogManager.getLogger(this.getClass()).info( s: "Error: " + throwable.getMessage()),
                () -> LogManager.getLogger(this.getClass()).info( s: "Completed")
        );
        LogManager.getLogger(this.getClass()).info( s: "----------- Subscribed -------------");
        TimeUnit.SECONDS.sleep( timeout: 5);
    }
```

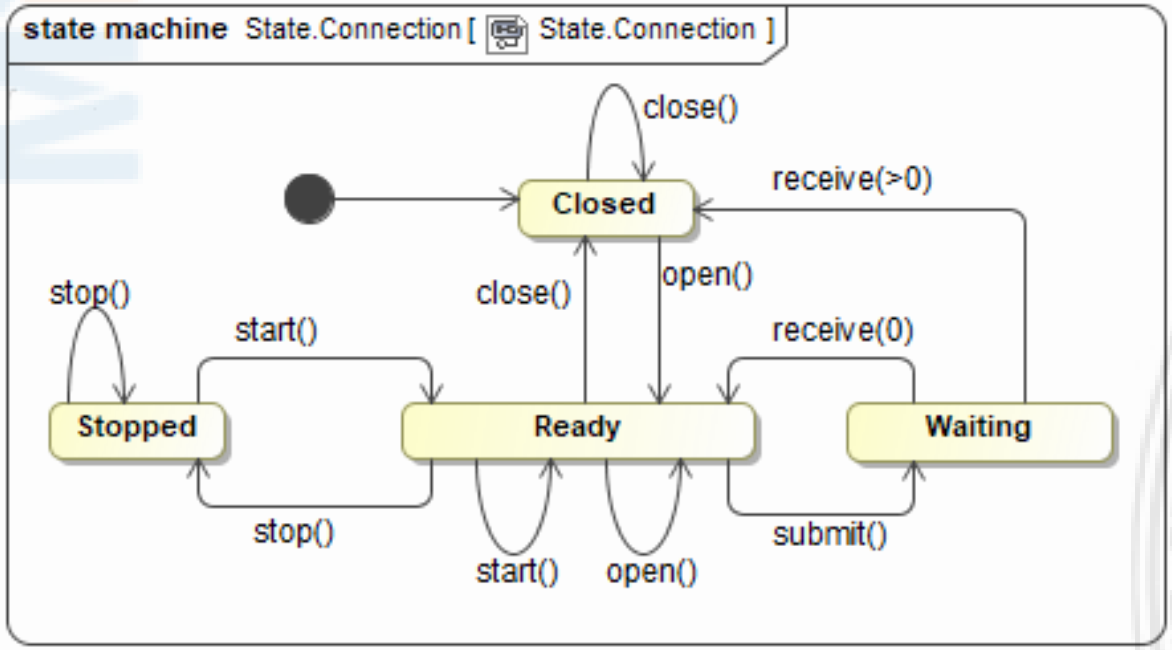
Publisher

- Propósito
  - Se define una dependencia entre uno a muchos, del tal manera, que cuando cambie avise a todos los objetos dependientes



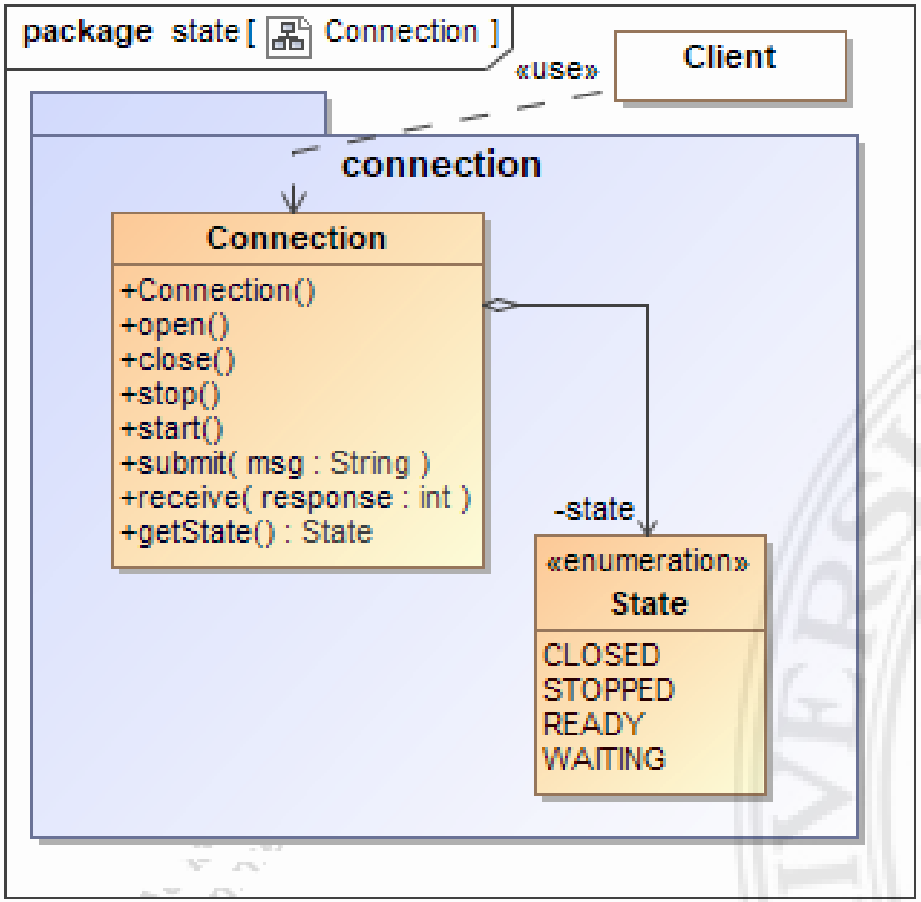
State (Estado)

- También conocido
  - Objects for States (Estados como Objetos)
- Fuentes: paquete state, paquete test state
- Motivación
  - Partiendo de una clase que representa una conexión TCP, la acción enviar debe tener diferentes respuestas dependiendo del estado de la conexión. Ello nos lleva al antipatrón: código espagueti



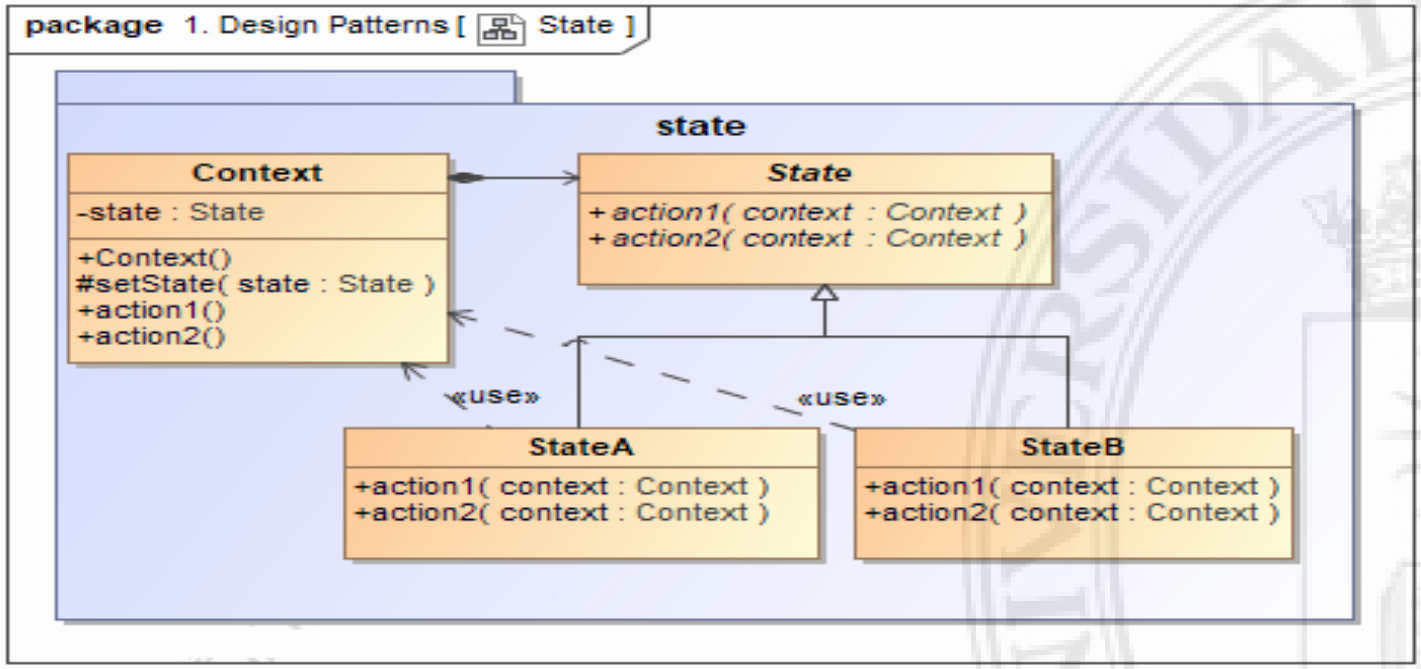
State (Estado)

- Sólo se permiten las acciones marcada, el resto debe lanzar la excepción: *UnsupportedOperationException*



State (Estado)

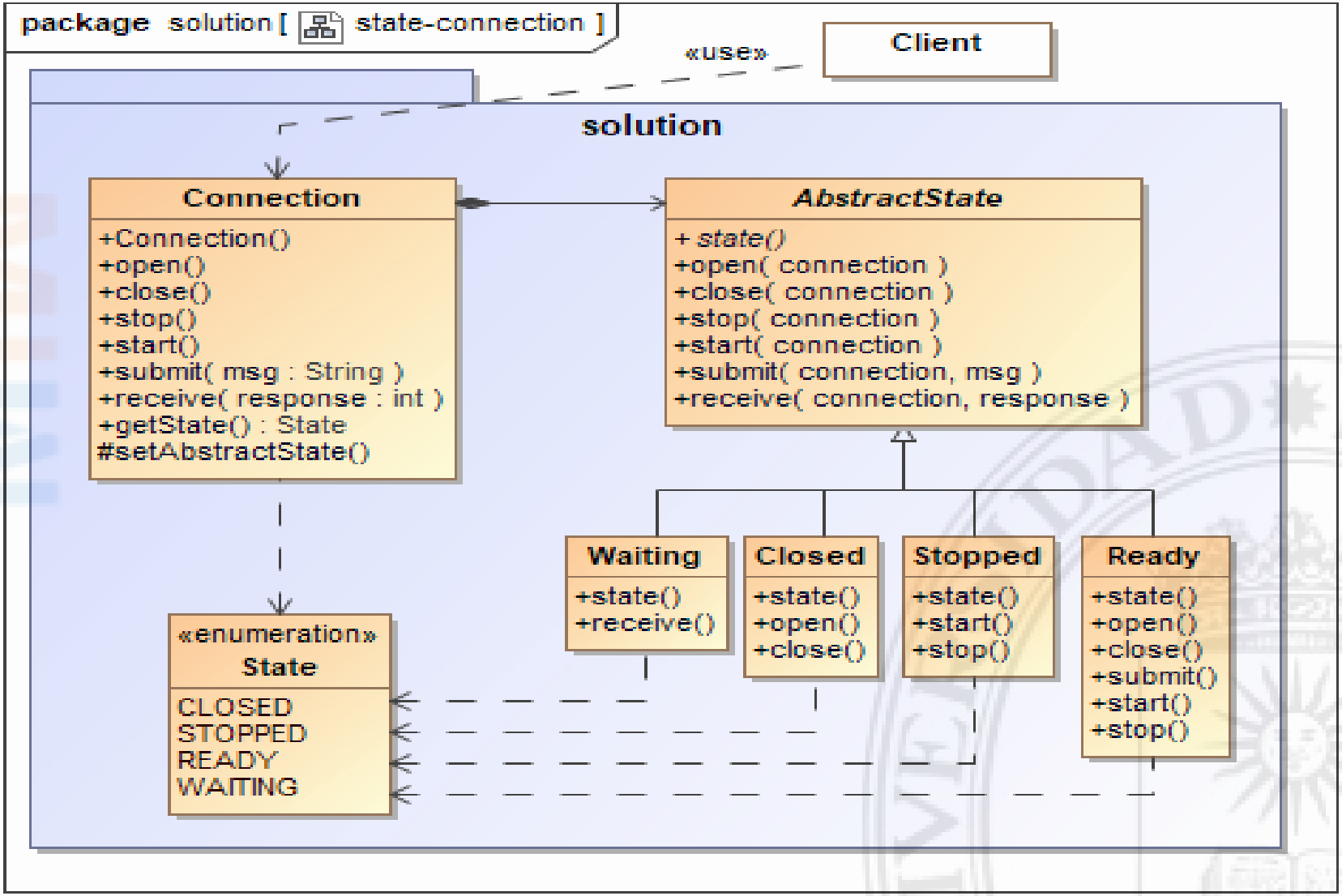
- Propósito
  - Permite que un objeto cambie su comportamiento cada vez que cambie su estado interno
- Creación de objetos:
  - Crearlos nuevos cada vez que se necesite
  - Crearlos al principio y no destruirlos



Connection

- Refactorizar la clase ***Connection*** para aplicar el patrón ***State***
- Se dispone de *ConnectionTest*

Connection

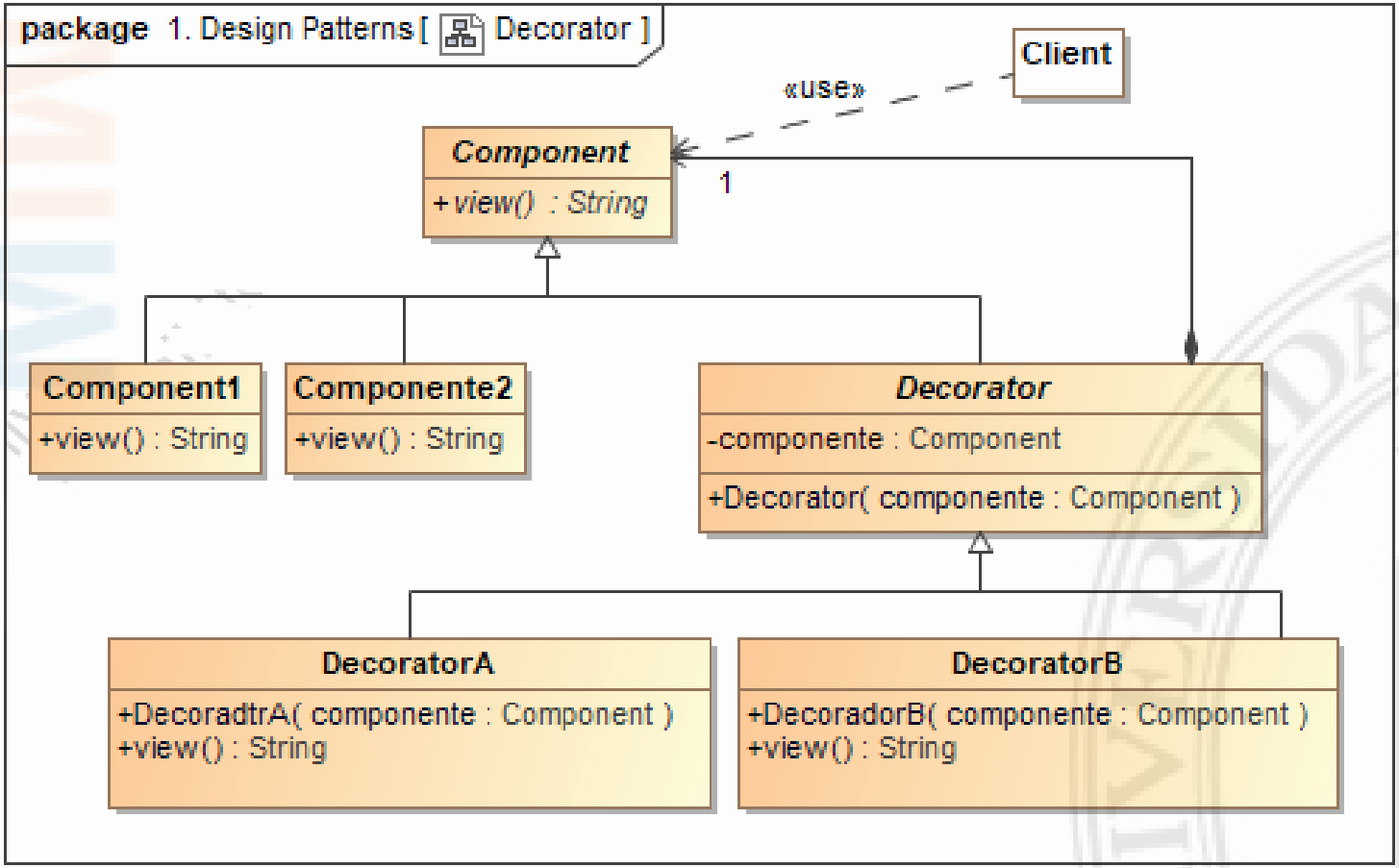

package solution [state-connection]
«use»
Client
solution
Connection
+Connection()
+open()
+close()
+stop()
+start()
+submit(msg : String)
+receive(response : int)
+getState() : State
#setAbstractState()
AbstractState
+ state()
+open(connection)
+close(connection)
+stop(connection)
+start(connection)
+submit(connection, msg)
+receive(connection, response)
Waiting
+state()
+receive()
Closed
+state()
+open()
+close()
Stopped
+state()
+start()
+stop()
Ready
+state()
+open()
+close()
+submit()
+start()
+stop()
«enumeration»
State
CLOSED
STOPPED
READY
WAITING

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
- Fuentes: paquete decorator

Decorator (Decorador)

- Propósito
  - Asigna responsabilidades de forma dinámica a objetos, proporcionando una alternativa flexible a la herencia

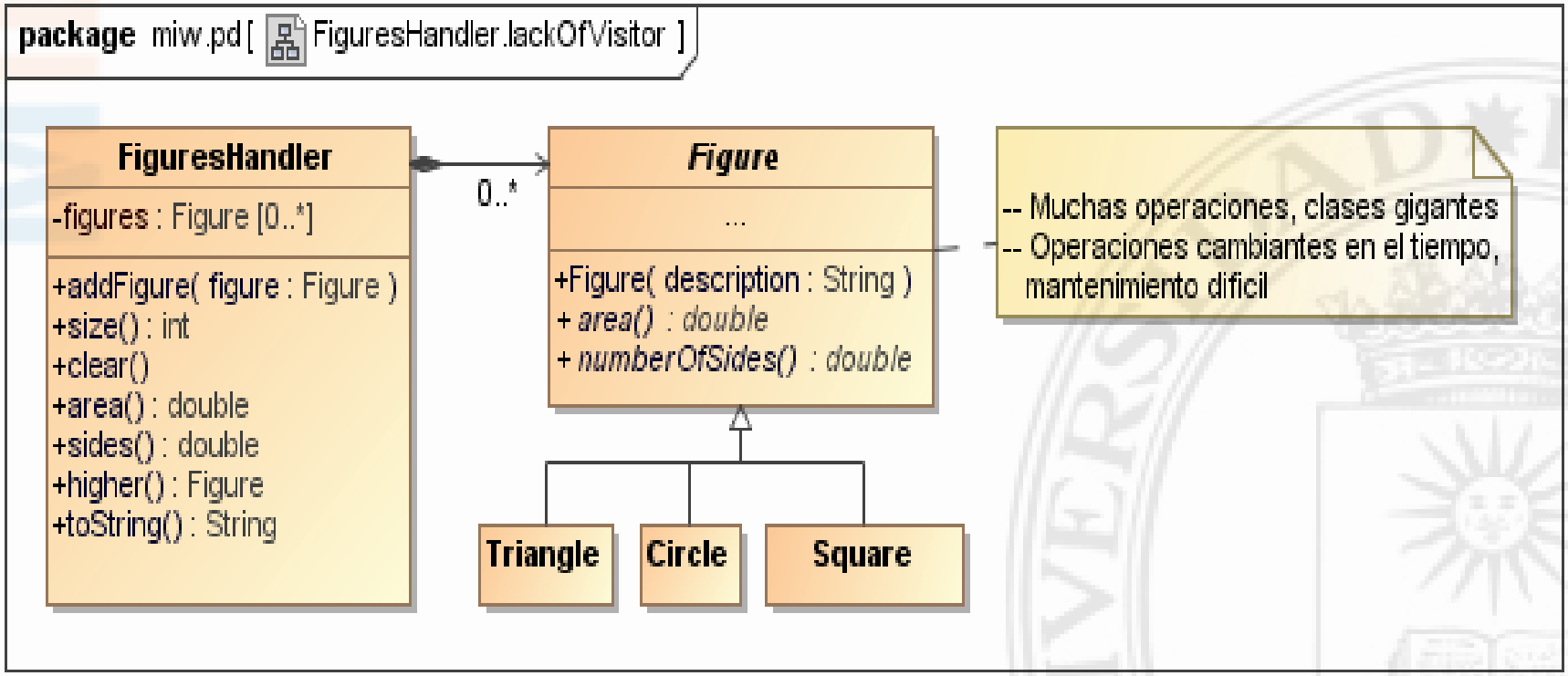


Vehículo

- 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 descripcionFinal();
  - public int precioFinal();

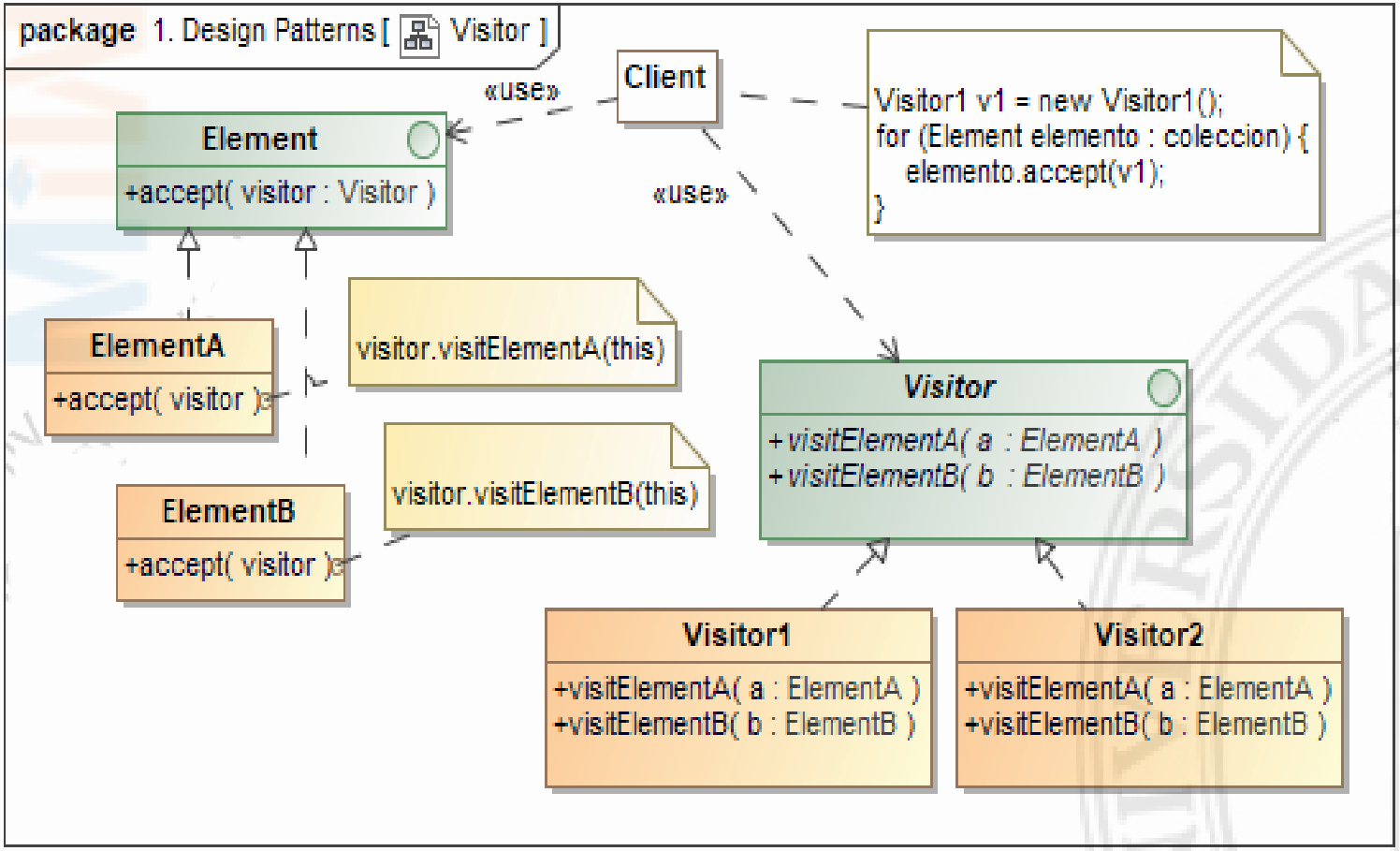
Visitor (Visitante)

- 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
- Fuentes: paquete visitor



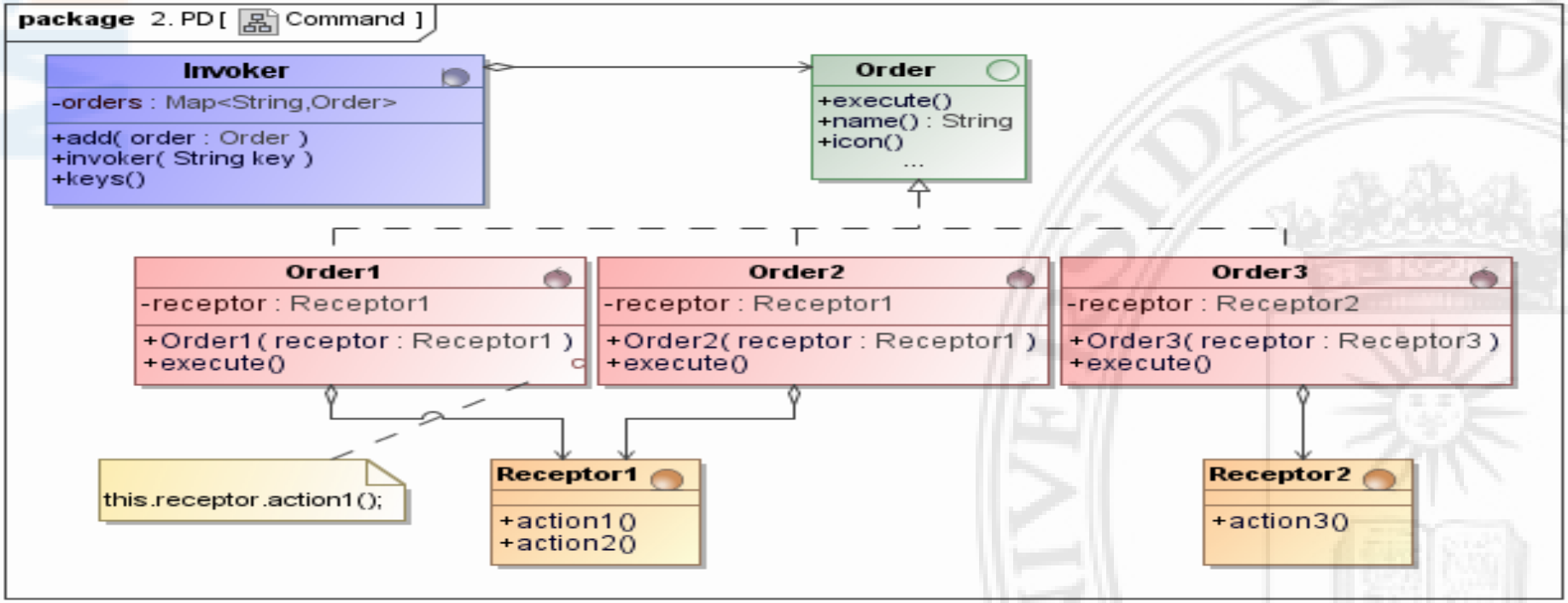
Visitor. Implementación

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



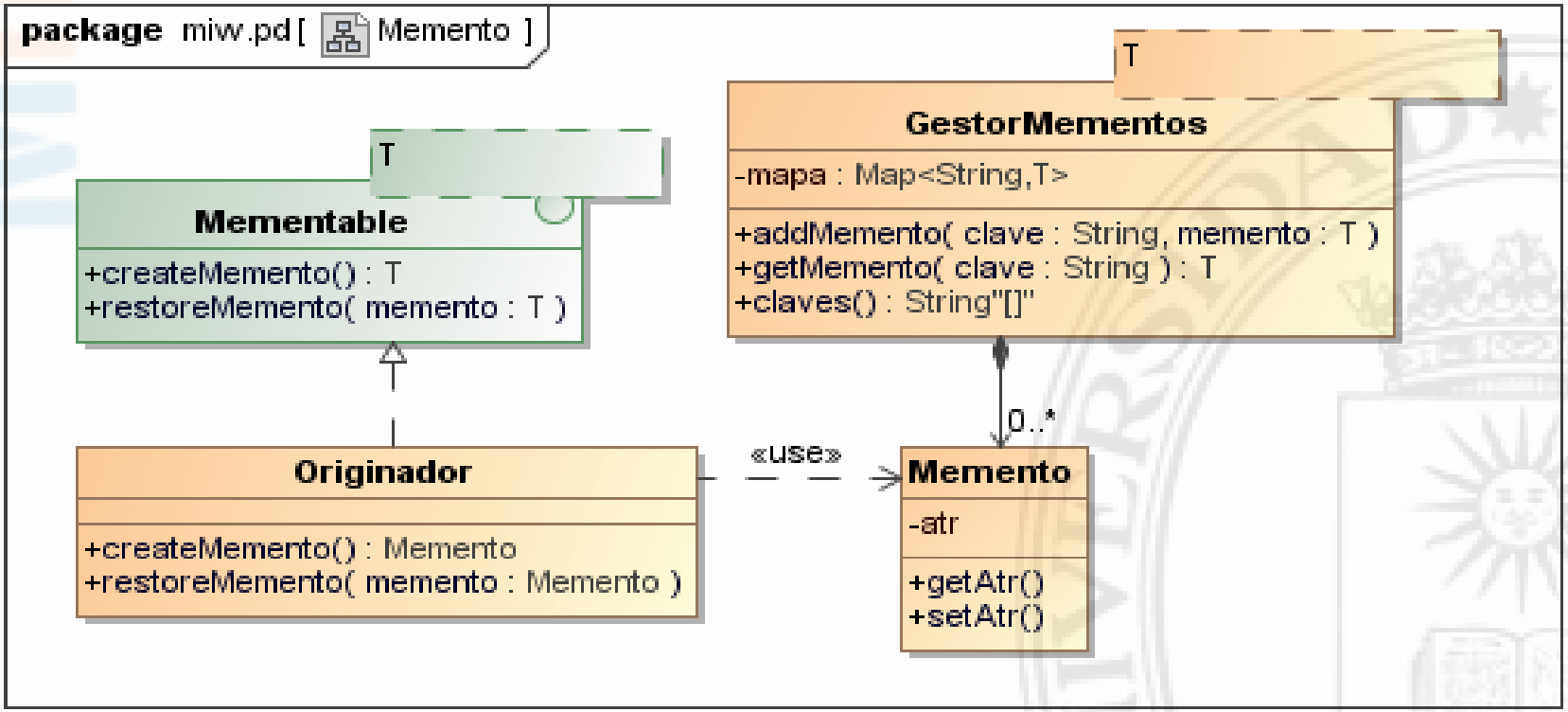
Command (Orden)

- 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



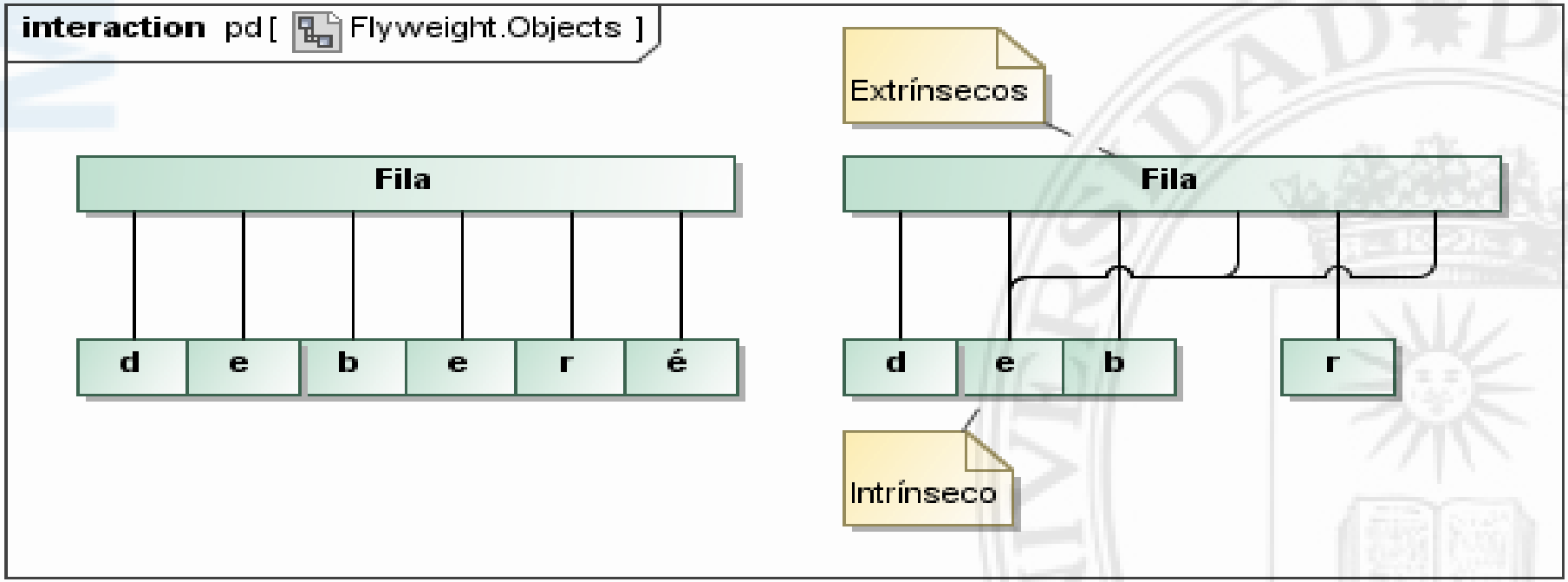
Memento (Recuerdo)

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



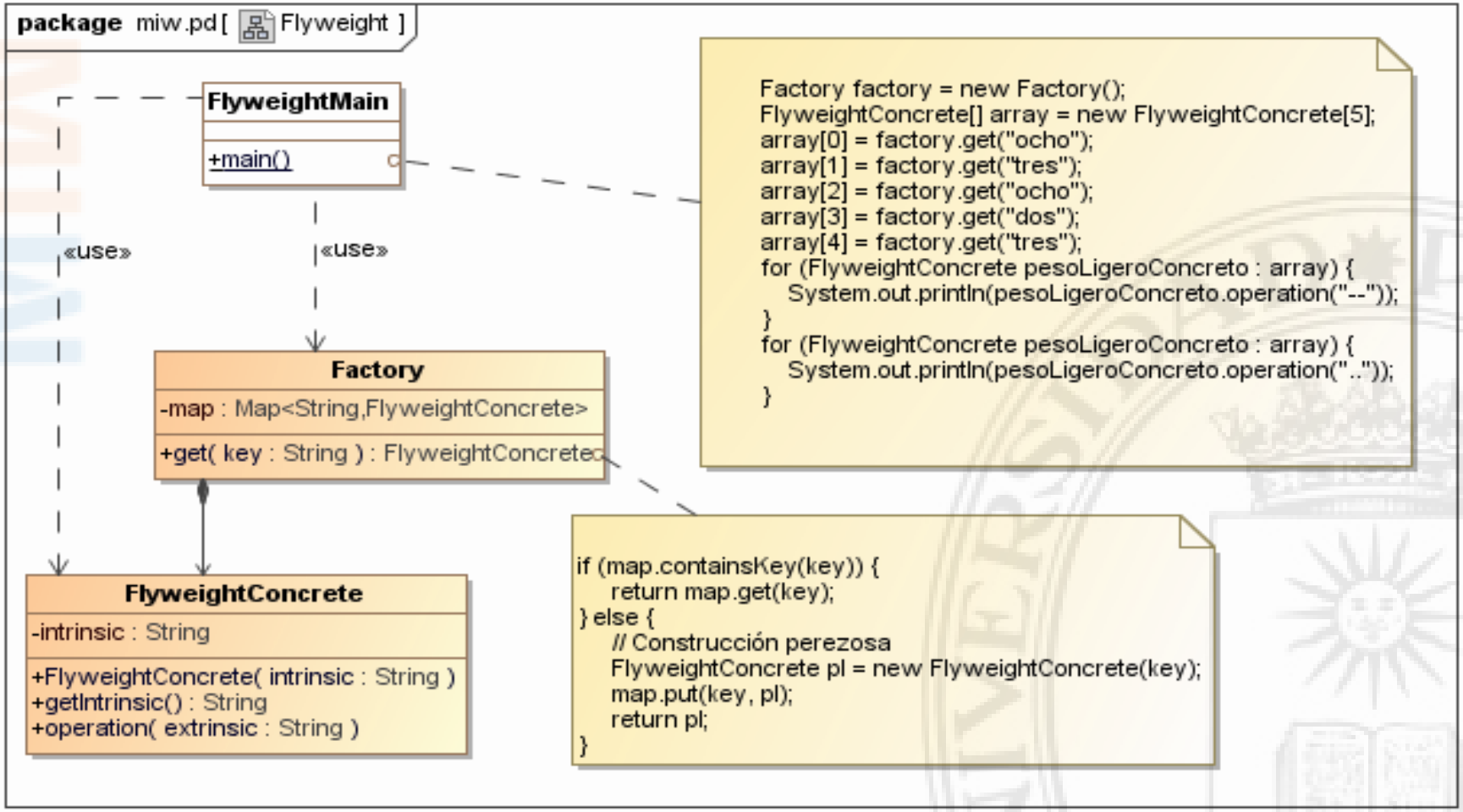
Flyweight (Peso ligero)

- 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
- Fuentes: paquete flyweight



Flyweight. Implementación

- Propósito
  - Compartir objetos de grado fino de forma eficiente



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
- Buscar por Internet un ejemplo para comprenderlo

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

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 sustituto o representante para controlar el acceso a un 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

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

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