Abstracción–caso

"Abstraction–occurrence", se encuentra en diagramas de clases que forman parte de un modelo de dominio del sistema. Se aplica a conjuntos de objetos relacionados: "casos"

- *Contexto*
- *Problema*
- *Fuerzas*
- *Solución* item *Ejemplos*
- *Anti–patrones*
- *Patrones relacionados*
- *Referencias*:generalización del patrón `Title-Item` publicado inicialmente en [Eriksson and Penker, 2000].

Abstracción–caso–2

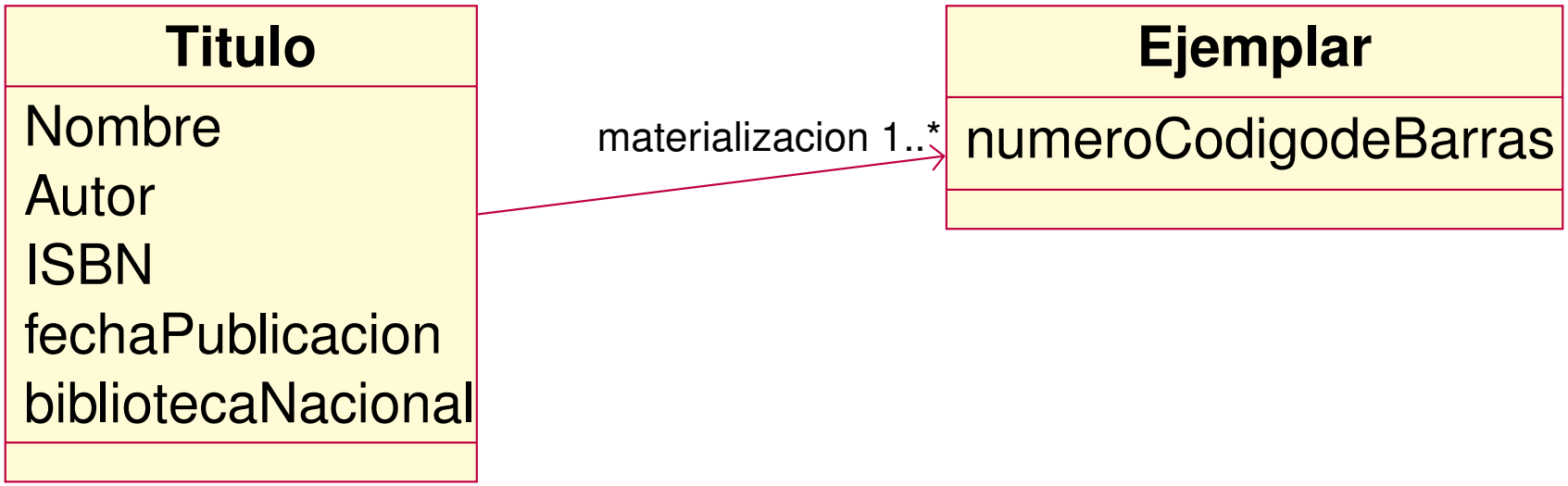
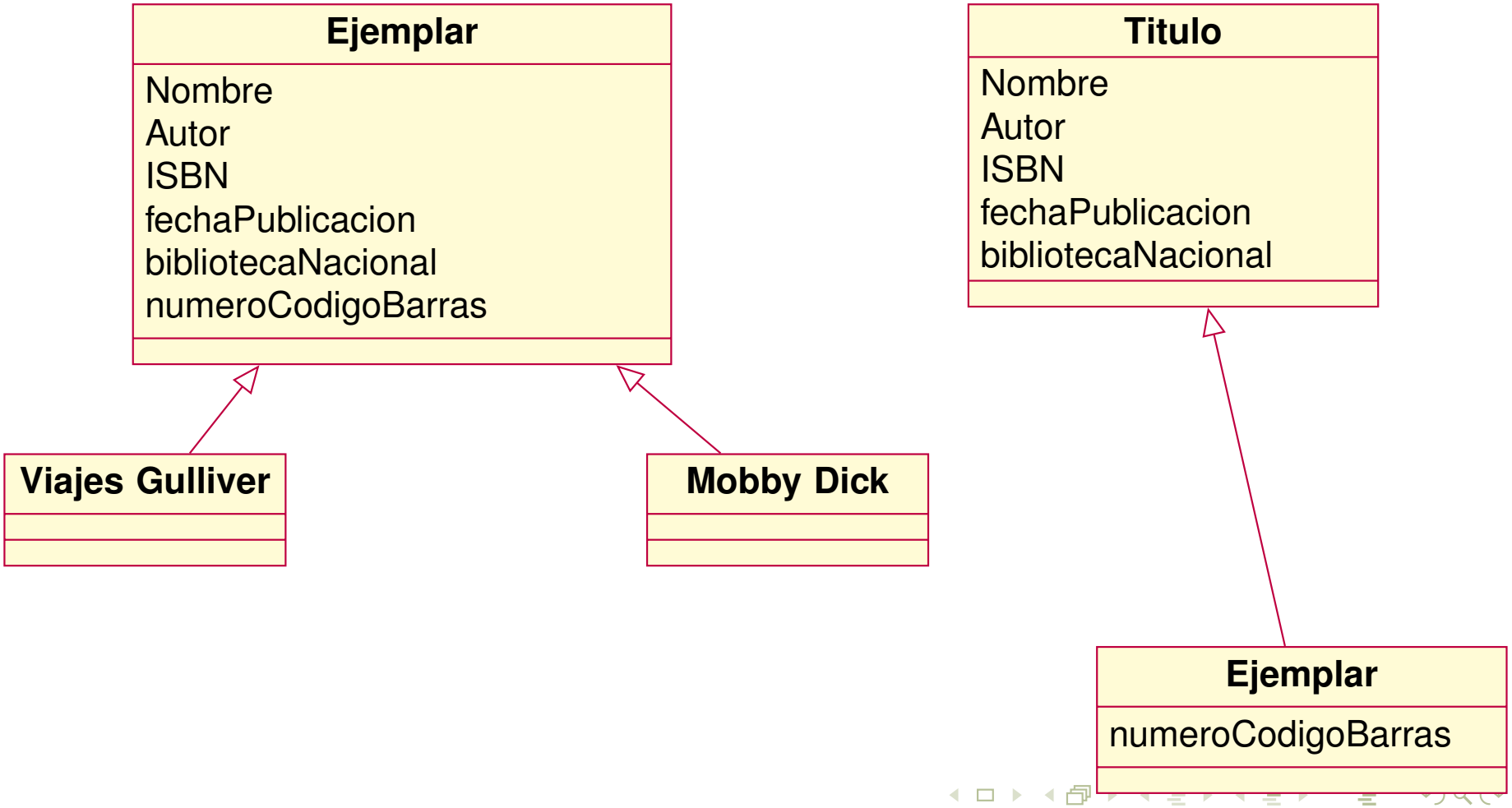


Figura: Plantilla–ejemplo del patrón *Abstracción–caso*

Abstracción–caso-3



Patrones relacionados

- El catálogo GoF propone los patrones: *Flyweight* y *Prototype*, que pueden considerarse relacionados con *Abstracción–caso*
- *Flyweight* utiliza la *compartición* del estado intrínseco de objetos que poseen una granularidad fina para evitar la creación masiva de estos objetos en una determinada aplicación.
- *Prototype*: especifica la tipología de los objetos a crear utilizando una clase–plantilla (*prototipo*) de estos, de tal forma que los nuevos objetos se crean copiando el prototipo.

Flyweight

- Un gran número de objetos de baja granularidad
- Almacenarlos todos sin compartir estado: mucha replicación de datos
- Posible encapsulación del *estado extrínseco*

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

Flyweight–2

Figura: Diagrama de clases del patrón `Flyweight`

Prototype

Cuando resulta más práctico *clonar* los objetos que instanciar una clase determinada para crear objetos en determinadas aplicaciones.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

Prototype–2

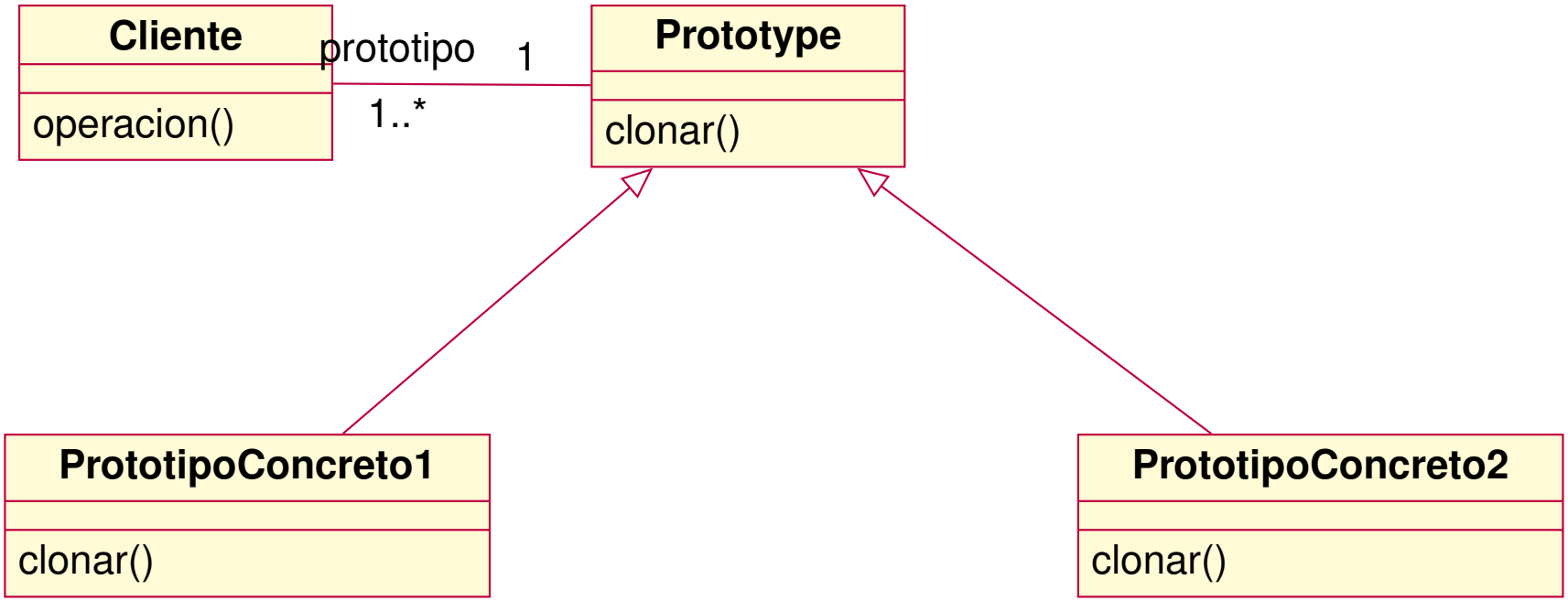


Figura: Diagrama de clases del patrón `Prototype`

Singleton

La idea de este patrón viene de la necesidad de los sistemas software de encontrar clases para las cuales sólo se necesita crear una instancia. A este tipo de clases se les denomina *singleton*.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en Lethbridge [Lethbridge and Laganiere, 2005]

Singleton–2

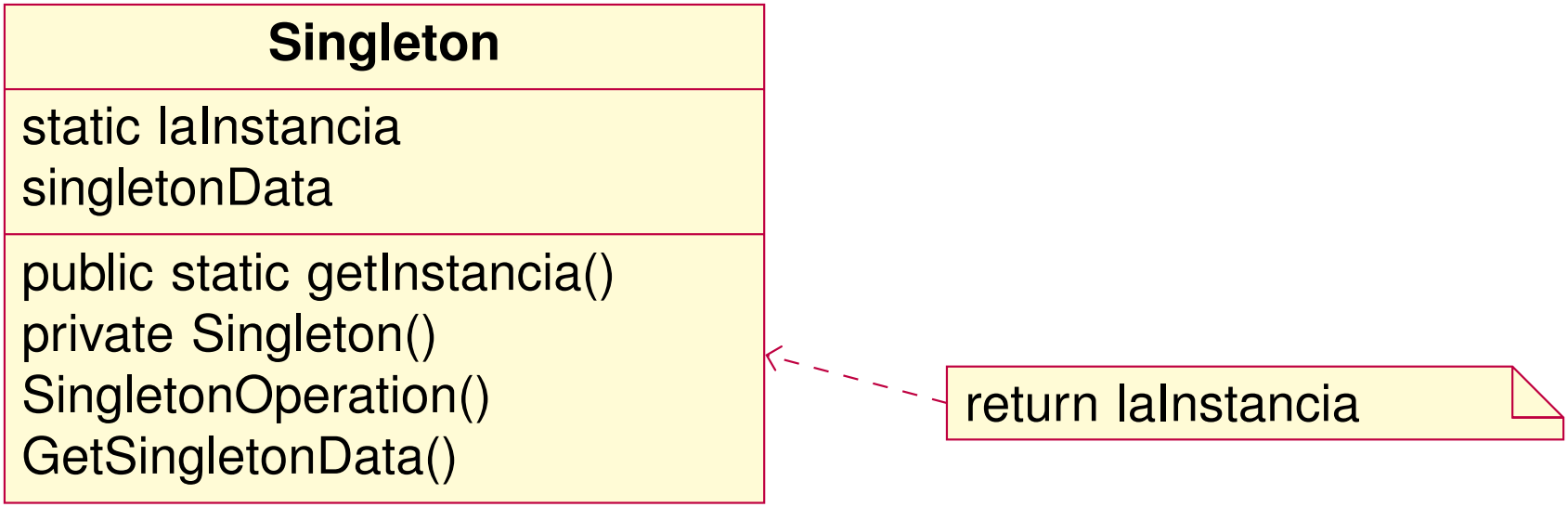


Figura: Diagrama de clases del patrón Singleton

Factoría

Necesitamos crear objetos dentro de un marco de trabajo pero desconocemos la clase de estos objetos ya que depende de la aplicación concreta. Proporcionar interfaces para crear familias de objetos sin especificar las clases a las que pertenecerán.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

factoria–2

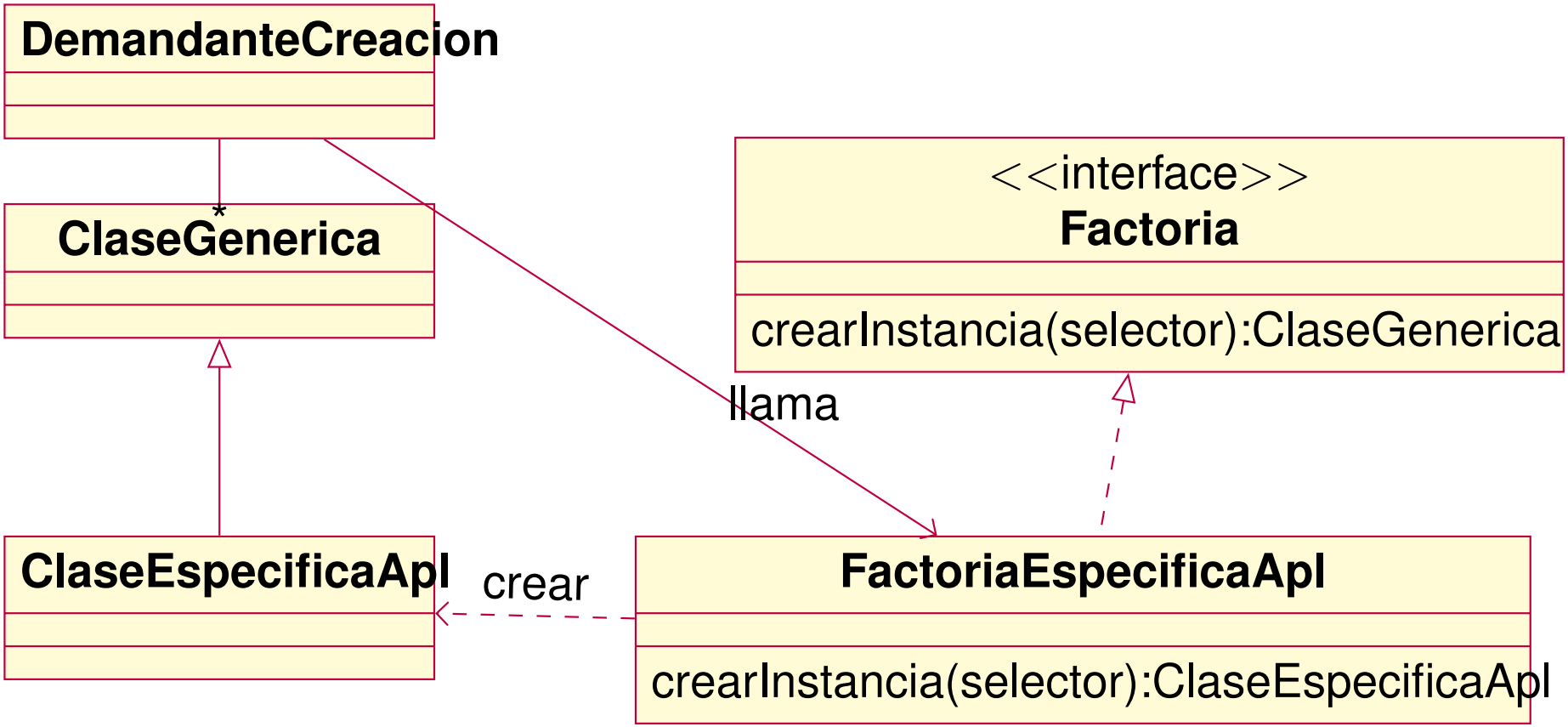


Figura: Diagrama de clases del patrón `Factoria`

Método Factoría

Cuando el método para crear los objetos está incluido en una clase abstracta pero el conocimiento para crearlos está fuera del marco actual de trabajo.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

Método Factoría–2

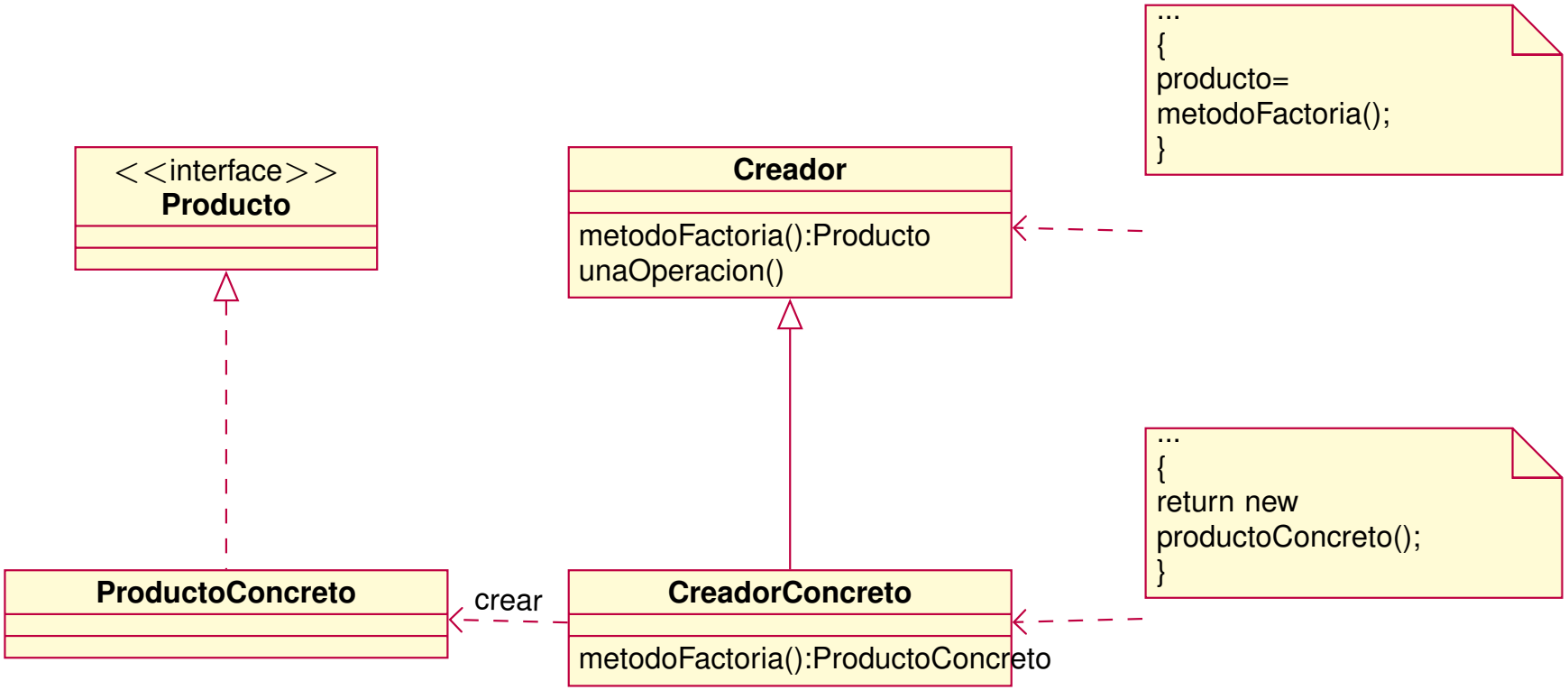


Figura: Diagrama de clases del patrón `MetodoFactoria`

Método Factoría vs Factoría Abstracta

El método factoría puede ser redefinido en una subclase.

```
Class A{
  public void hacerAlgo(){
     Foo f= hacerFoo();//metodo factoria
     f.loQueSea();
  }
  protected Foo hacerFoo(){
    return new FooNormal();
  }
}
Class extends A{
  protected Foo hacerFoo(){//Redefine el metodo-
    return new FooSuper();//factoria y devuelve algo
        diferente
  }
}
```

Método Factoría vs Factoría Abstracta–2

Una clase delega la creación del objeto a otra clase a través de *delegación*

```
Class A{
  private Factoria factoria;
  public A(Factoria factoria){
    this.factoria= factoria;
  }
  public void hacerAlgo(){
     Foo f= factoria.hacerFoo();//metodo factoria
     f.loQueSea();
  }
}
Interface Factoria{
  Foo hacerFoo();
  Barra hacerBarra();
}
```

Bridge

El objetivo de este patrón es conseguir desacoplar una abstracción de su implementación, de tal manera que ambas puedan variar independientemente.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

Bridge–2

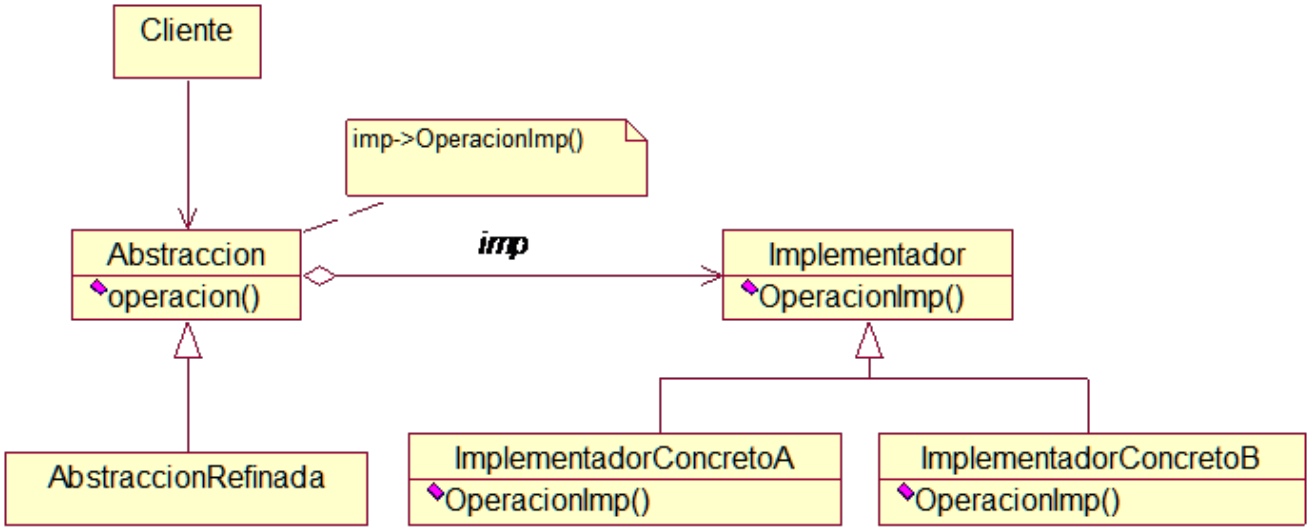


Figura: Diagrama de clases del patrón `Bridge`

Jerarquía General

Cuando se necesita representar una jerarquía de objetos en la que algunos de estos pueden tener subordinados y otros no.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

Jerarquía General–2

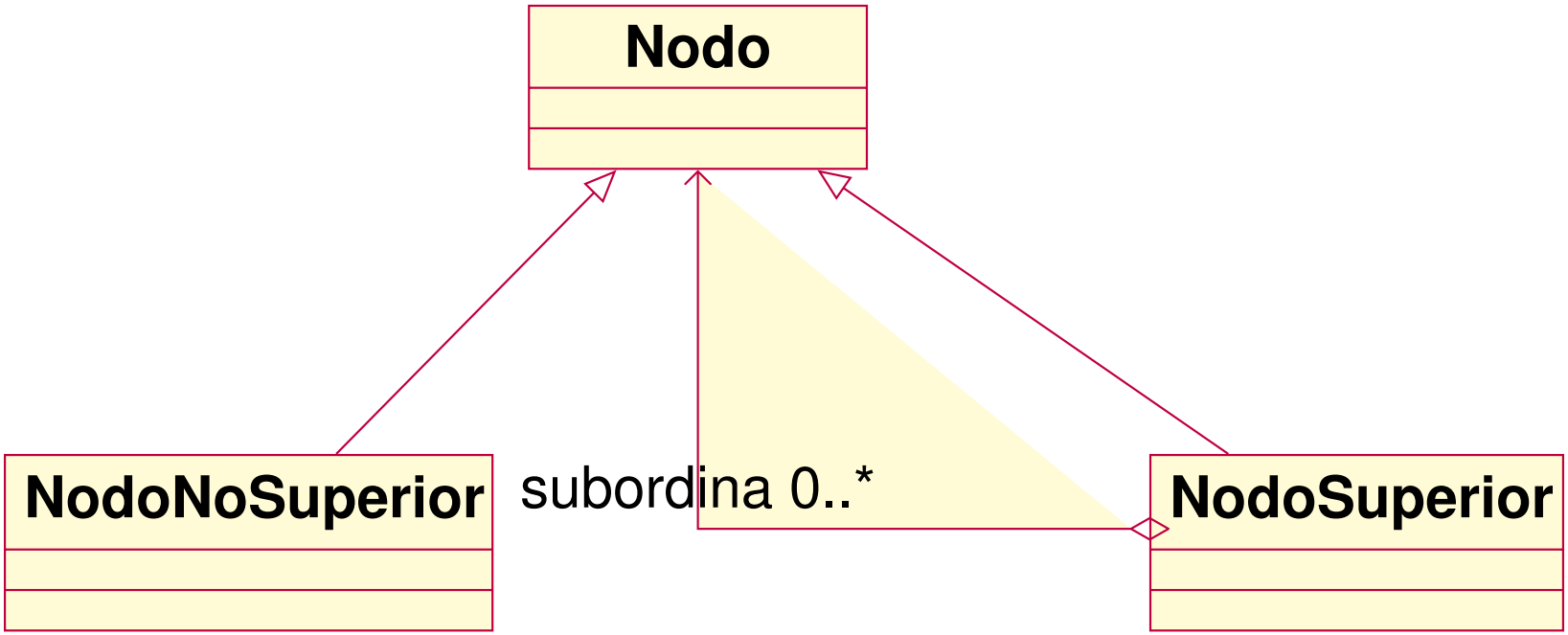


Figura: Diagrama de clases del patrón `Jerarquía General`

Composite

Mediante este patrón un grupo de objetos se puede manipular de la misma forma que un solo objeto. Se entiende como una *clase abstracta* que representa tanto a entidades como a sus contenedores.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

Composite–2

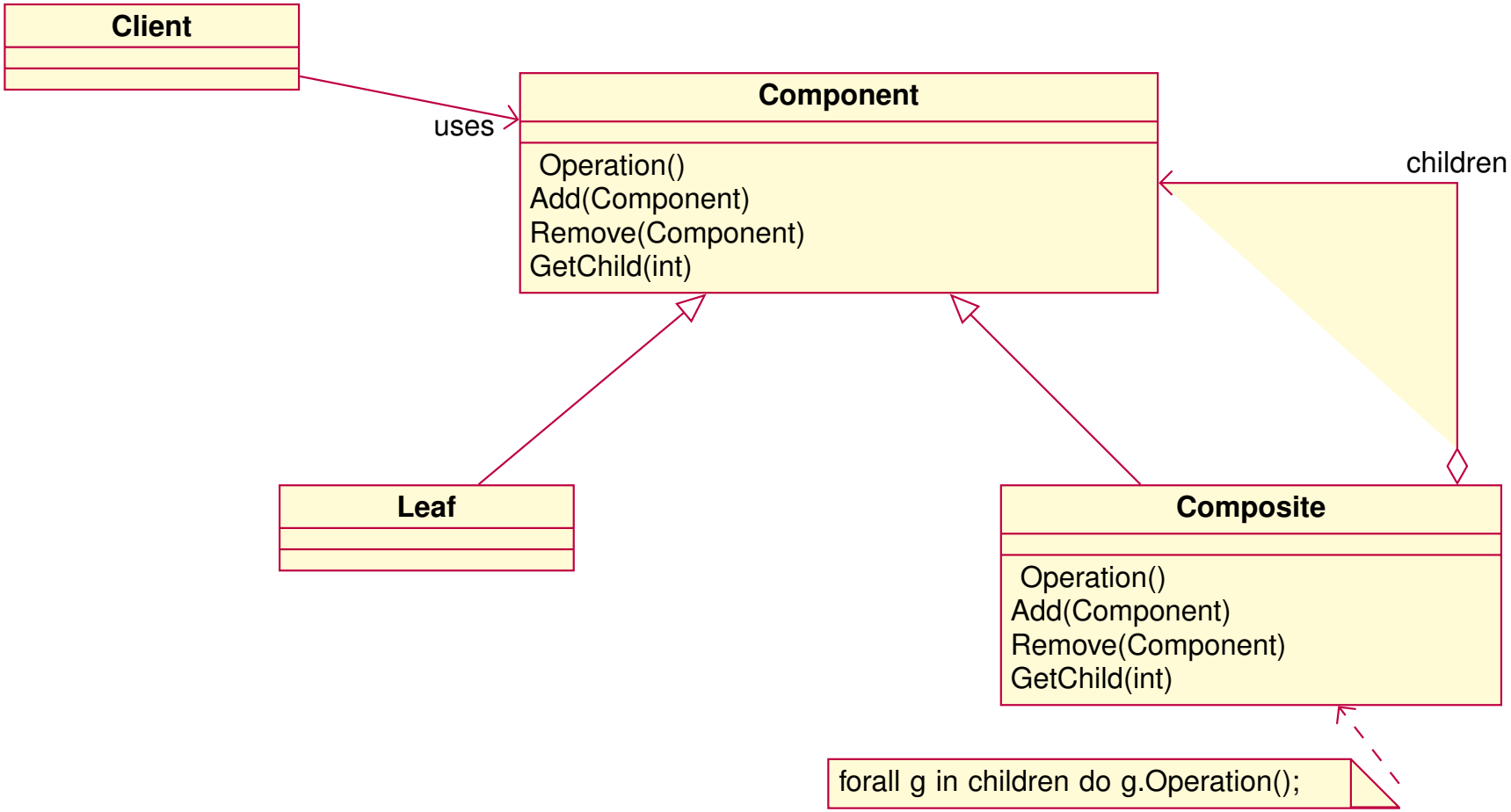


Figura: Diagrama de clases del patrón `Composite`

Proxy

Es de utilidad para reducir la carga en memoria de objetos pesados cuando la aplicación no los necesita inmediatamente.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

Proxy–2

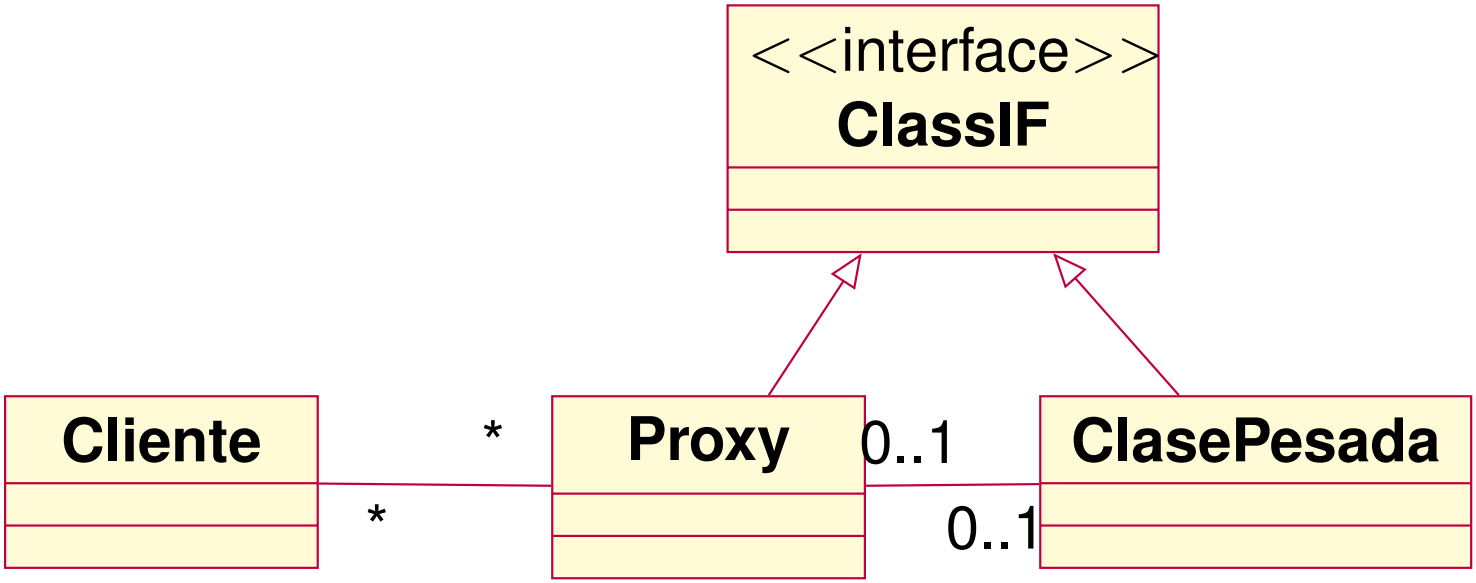


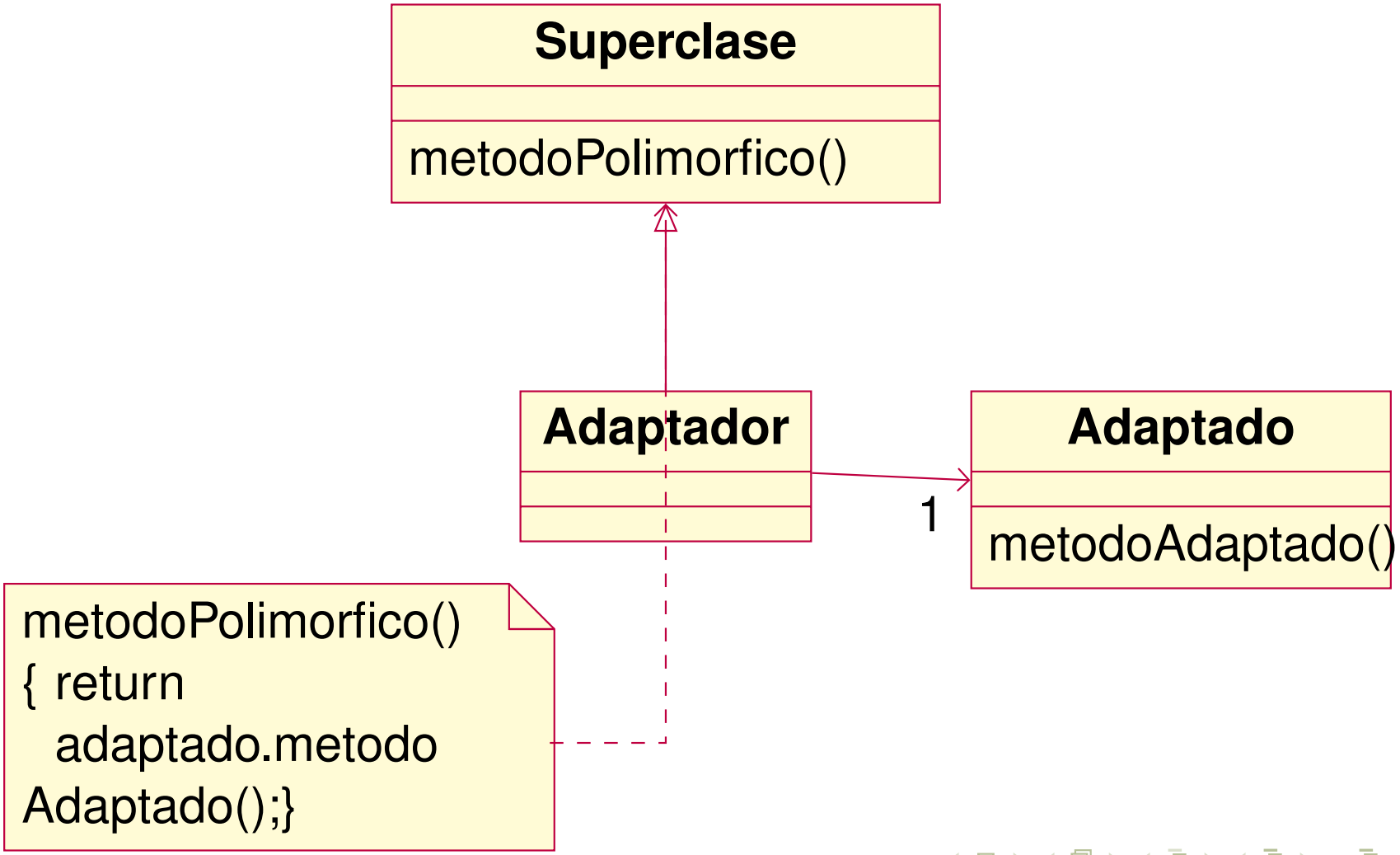
Figura: Diagrama de clases del patrón `Proxy`

Adaptador

Para reutilizar una clase pre-existente en una jerarquía de clases, con la que no guarda ninguna relación, dentro de la aplicación que estamos desarrollando.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

Adaptador–2


Superclase
metodoPolimorfico()
Adaptador
Adaptado
1
metodoAdaptado()
metodoPolimorfico()
{ return
adaptado.metodo
Adaptado();}

Adaptador–3

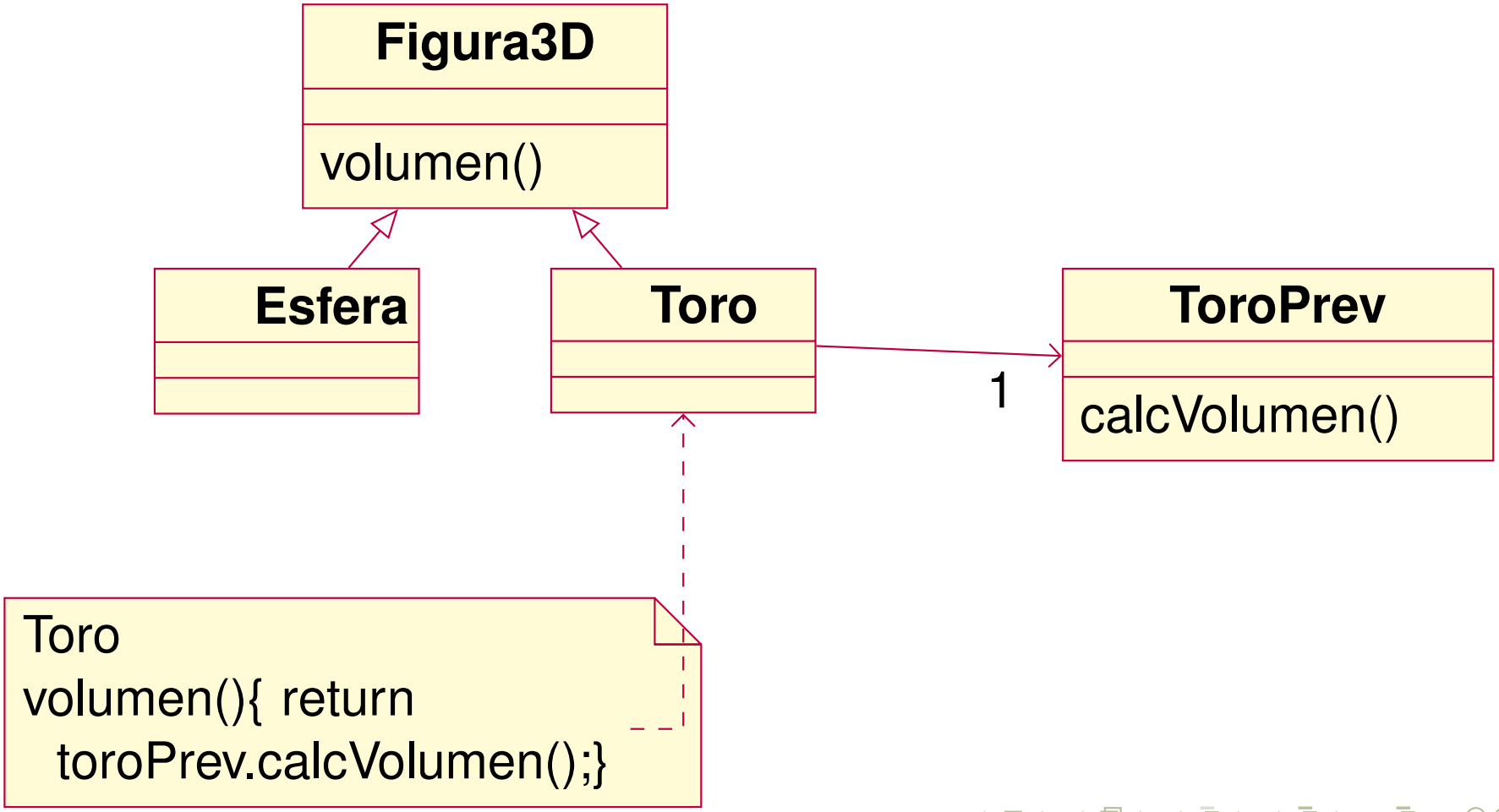


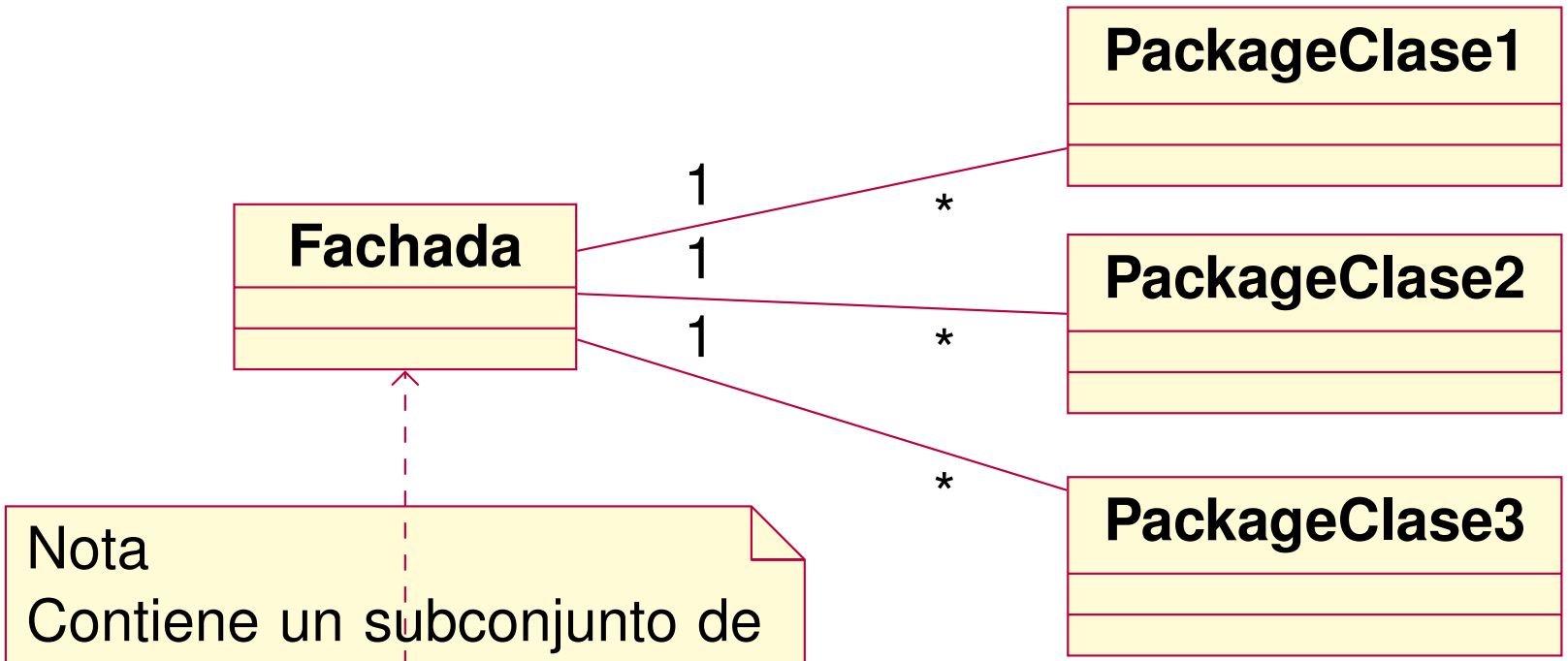
Figura3D
volumen()
Esfera
Toro
ToroPrev
1
calcVolumen()
Toro
volumen(){ return
toroPrev.calcVolumen();}

Fachada

Se utiliza para simplificar la utilización de un paquete de clases complejo *"fabricando una interfaz"* más fácil de utilizar.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

Fachada–2


PackageClase1
Fachada
1
1
1
*
*
*
PackageClase2
PackageClase3
Nota
Contiene un subconjunto de métodos públicos tal que los otros subsistemas no tendrán que acceder a las clases dentro de los paquetes.

Extension Interface

Para definir componentes evolutivos, que sufren modificaciones tanto en su implementación como en su interfaz.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

Extension Interface–2

Los clientes se diseñan para que puedan a los componentes software a través de interfaces separadas, una por cada rol que pueda jugar el componente.

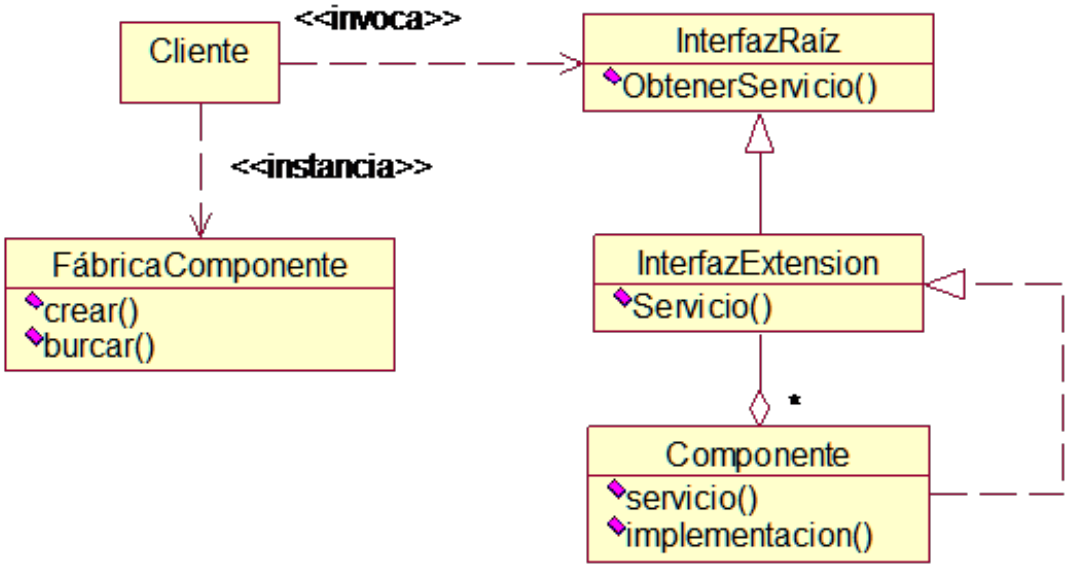


Figura: Diagrama de clases del patrón `Extension Interface`

Inmutable

Cuando se necesitan objetos cuyo estado no puede cambiar después de ser creados.

```
public final class User {
  private final String nombreusuario;
  private final String clave;
  public User(String nombreusuario, String clave) {
    this.nombreusuario = nombreusuario;
    this.clave = clave;
  }
  public String getnombreusuario() {
    return nombreusuario;
  }
  public String getclave() {
    return clave;
  }
}
```

Inmutable–2

El patrón `Builder` (GoF [Gamma et al., 2009]) puede utilizarse para construir el patrón *Inmutable*.

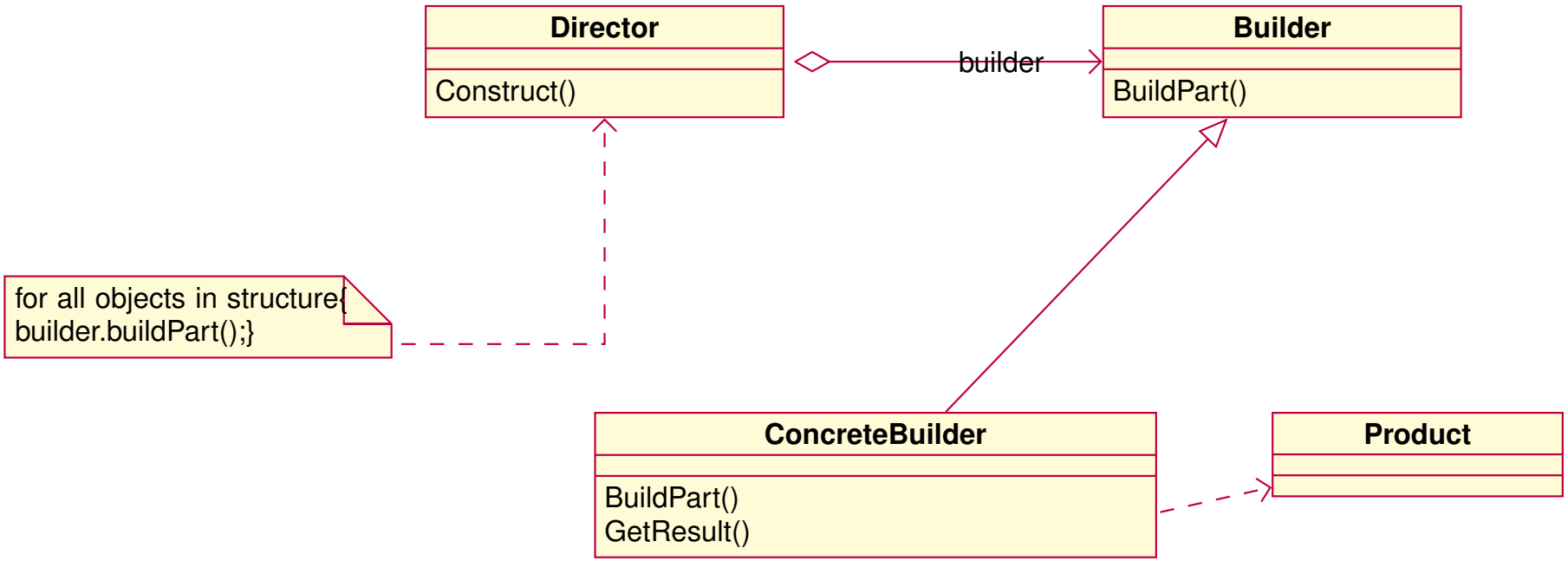


Figura: Diagrama de clases del patrón `Builder`

Inmutable–3

Programación que utiliza una clase *Inmutable* para garantizar que no se cambian los datos de un usuario, una vez creado.

```
public class UsuarioInmutable {
  private final String nombreusuario;
  private final String clave;
  private final String nombre;
  private final String apellidos;
  private final String email;
  private final Date fechaCreacion;
  private UsuarioInmutable(BuilderUsuario builder) {
    this.nombreusuario = builder.nombreusuario;
    this.clave = builder.clave;
    this.fechaCreacion = builder.fechaCreacion;
    this.nombre = builder.nombre;
    this.apellidos = builder.apellidos;
    this.email = builder.email;
  }  ...
```

Inmutable–4

```
//... continua la clase Inmutable
 public static class BuilderUsuario {
    private final String nombreusuario;
    private final String clave;
    private final Date fechaCreacion;
    private String nombre;
    private String apellidos;
    private String email;
    public BuilderUsuario(String nombreusuario, String
        clave) {
      this.nombreusuario = nombreusuario;
      this.clave = clave;
      this.fechaCreacion = new Date();
    }
    public BuilderUsuario nombre(String firsname) {
      this.nombre = firsname;
      return this;
    }
```

Inmutable–5

```
//... continua la clase Inmutable
    public BuilderUsuario apellidos(String apellidos) {
      ...
    }
    public BuilderUsuario email(String email) {
      ...
    }
    public UsuarioInmutable build() {
      return new UsuarioInmutable(this);
    }
  }//acaba la clase interna BuilderUsuario
  public String getnombreusuario() {
    return nombreusuario;
  }
  //getclave(), getnombre(), getapellidos(), getemail(),
      getfechaCreacion()
}//acaba la clase UsuarioInmutable
```

Inmutable–6

Programación del cliente de la clase `UsuarioInmutable`

```
public static void main(String[] args) {
  UsuarioInmutable user = new UsuarioInmutable.
      BuilderUsuario("manuel","DS2016In").
  nombre("manuel").apellidos("capel").email("
      manuel@gmail.com").build();}
```

- Patrones relacionados: `SoloLectura`
- Referencias: aparece incialmente en el libro de [Grand, 1999]

Interfaz SoloLectura

Para conseguir que clases *sin privilegio* puedan modificar el estado de los objetos de una clase. La clase `SoloLectura` sería inmutable para las clases sin privilegio.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en el libro [Grand, 1999]

SoloLectura–2

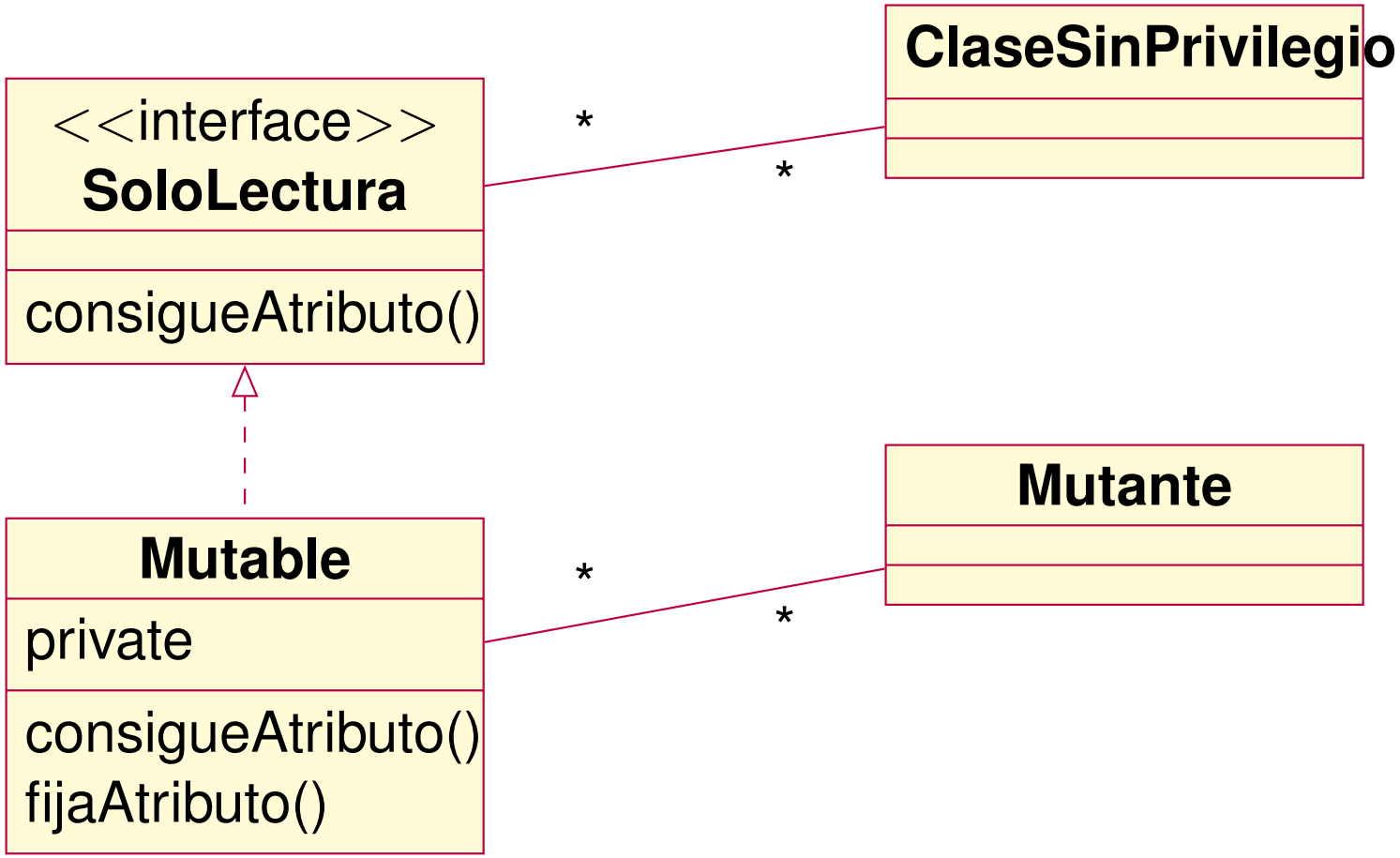


Figura: Diagrama de clases del patrón `SoloLectura`

Strategy

Permite intercambiar algoritmos después de hacerlos independientes (mediante “encapsulación”) de las aplicaciones que los utilizarán. Los algoritmos encapsulados puede modificarse independientemente de sus clientes. Cada algoritmo encpasulado en una clase se le llama *estrategia*

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

Strategy–2

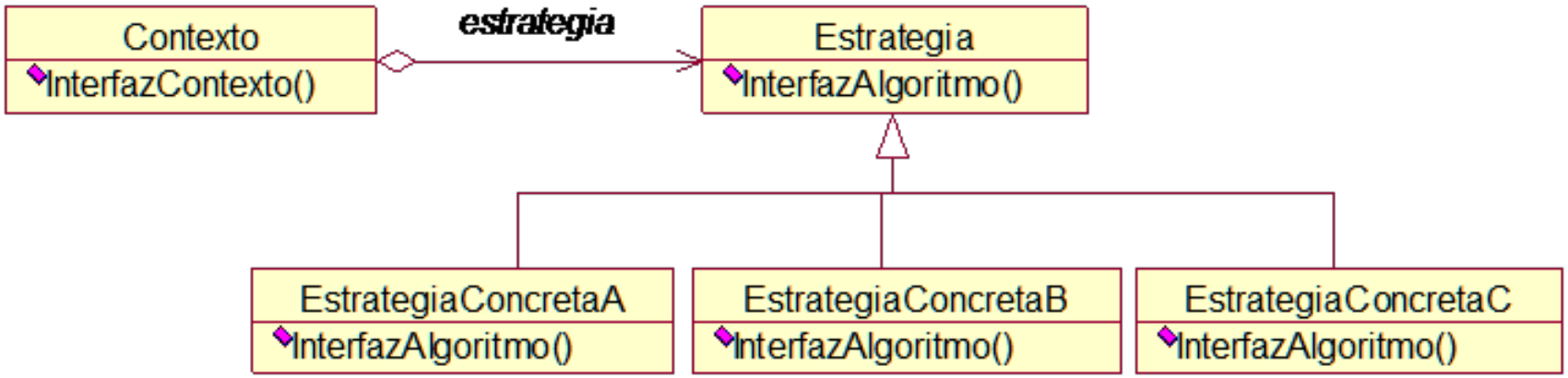


Figura: Diagrama de clases del patrón `Strategy`

Actor–papel

Cuando determinados objetos pueden adoptar comportamientos/funcionalidad diferentes durante la ejecución de una aplicación y no queremos utilizar herencia múltiple.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en el libro [J. Rumbaugh, 2004]

Actor–papel–2

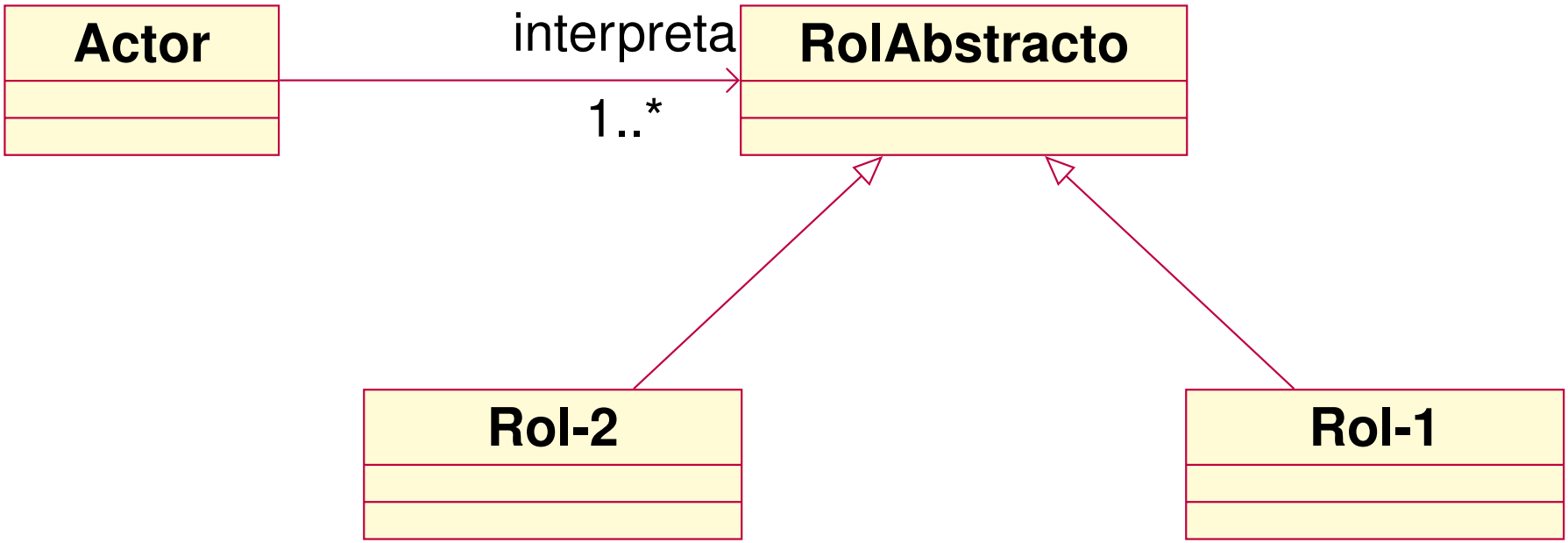


Figura: Diagrama de clases del patrón Actor–papel

Delegacion

Necesitamos utilizar una operación programada como un método de una clase externa pero no queremos heredar de esa clase.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en [Lethbridge and Laganiere, 2005]

Delegacion–2

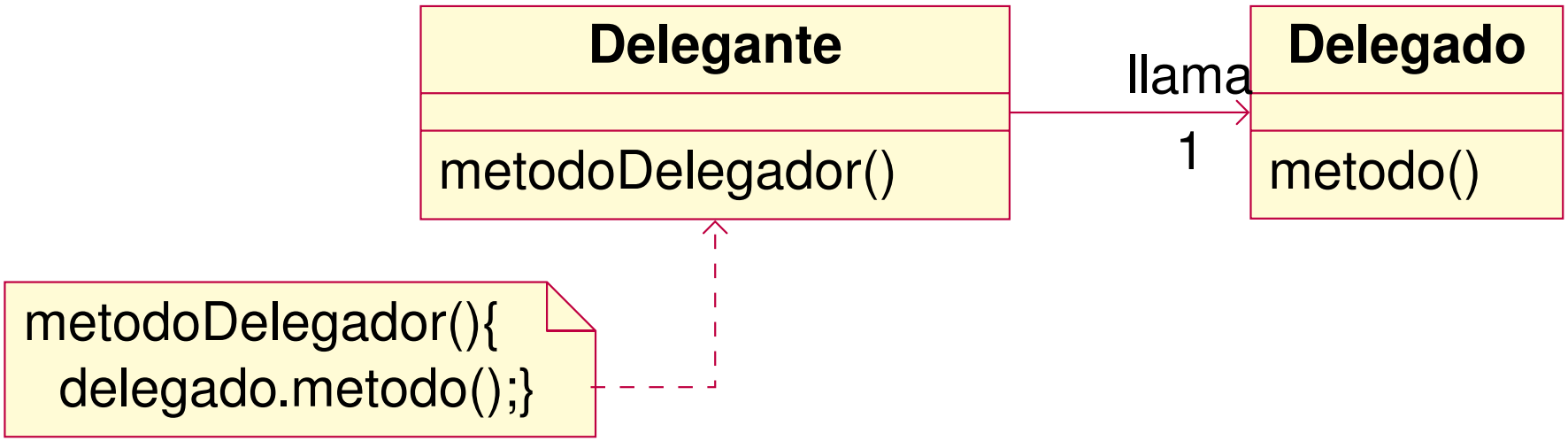


Figura: Diagrama de clases del patrón `Delegacion`

Delegacion–3

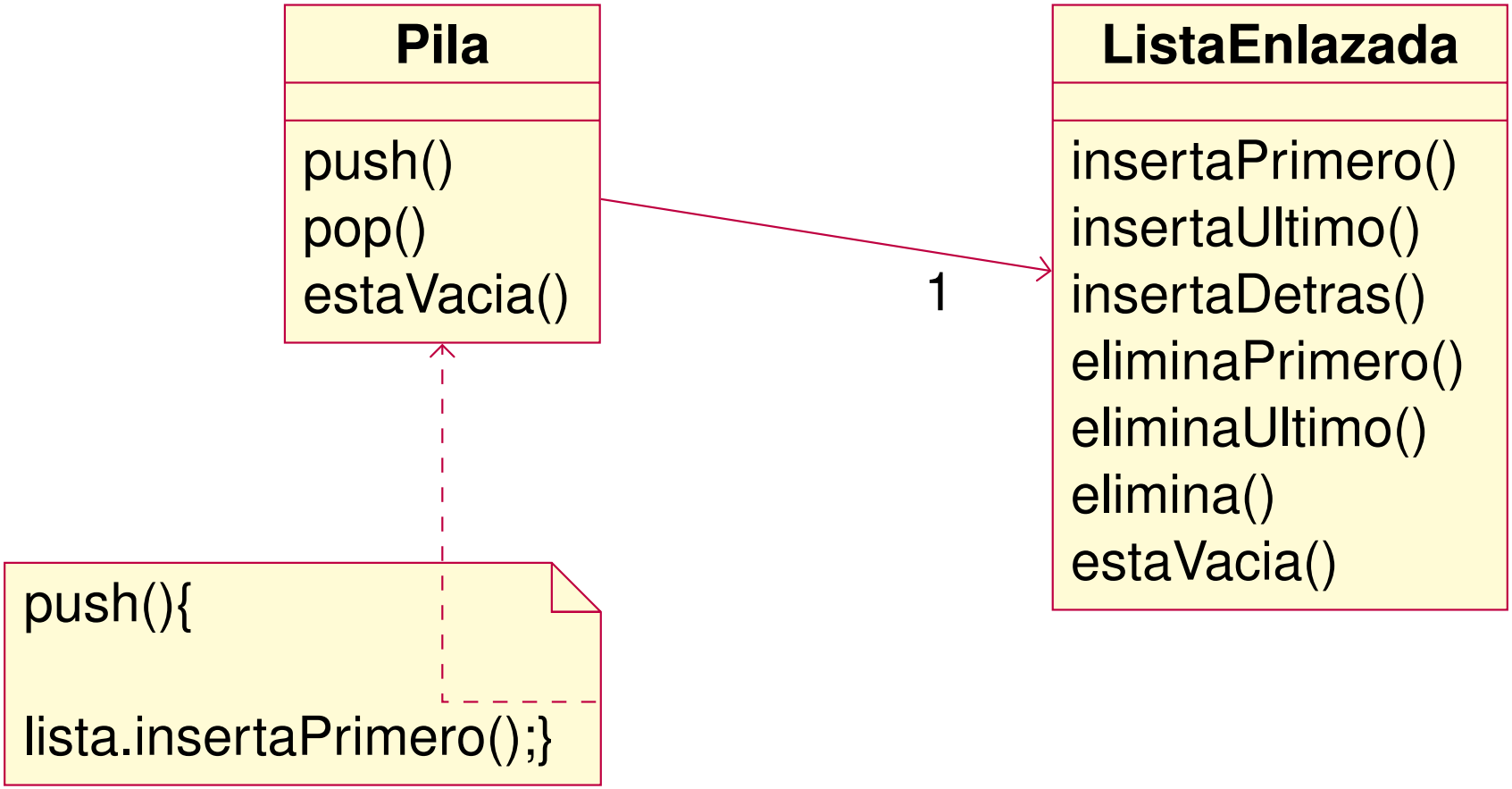


Figura: Ejemplo de uso del patrón de diseño Delegación

Command

Las solicitudes de los clientes se convierten en objetos, permitiendo generalizar a los clientes dependiendo sólo de su tipo de solicitud, encolar o registrar solicitudes y convertir en reversibles las operaciones.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

Command–2

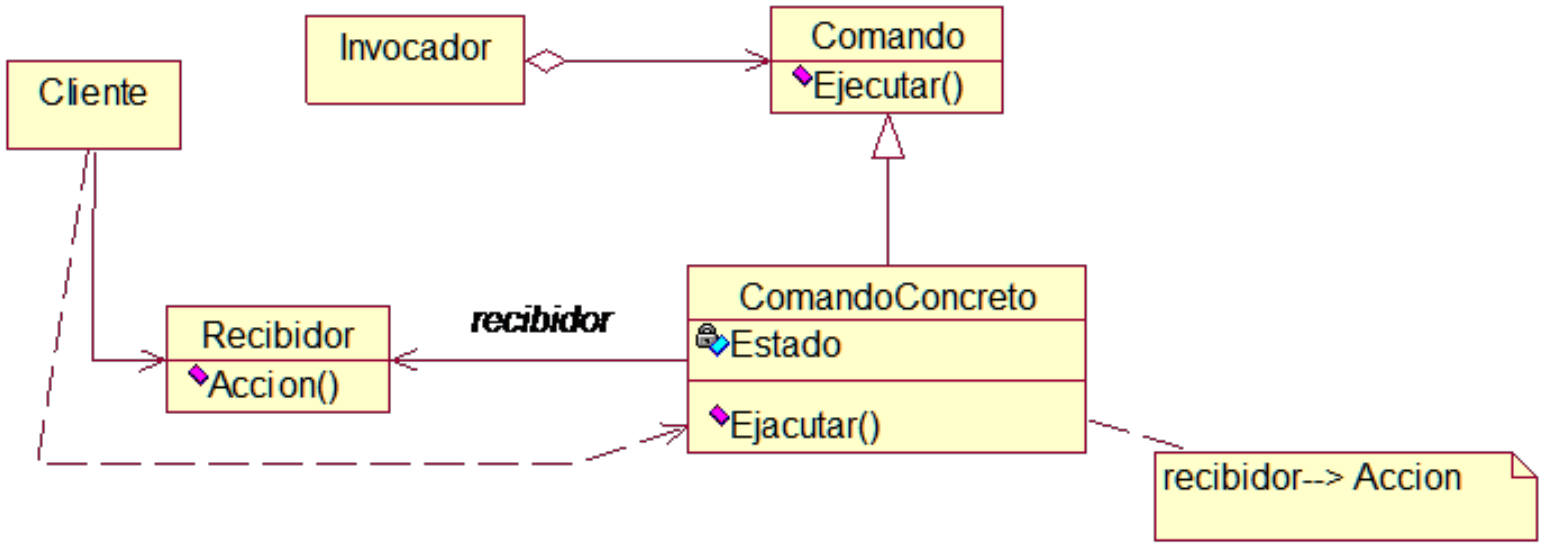


Figura: Diagrama de clases del patrón `Command`

Observable–Observador

Reducir la interdependencia entre clases cuando una de ellas modifica información y las otras se suscriben a dicha actualización.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: en GoF [Gamma et al., 2009] se le denomina: *Publicar y Suscribir*

Observable–Observador–2

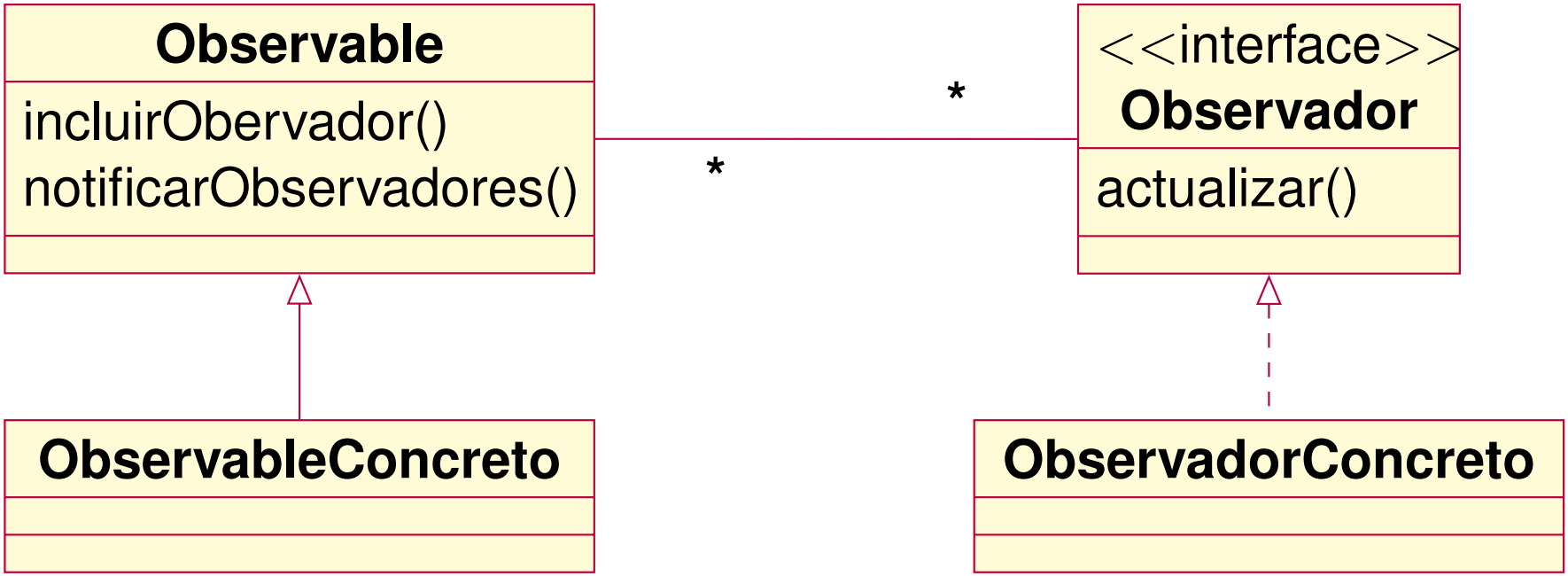


Figura: Diagrama de clases del patrón Observable–observador

Observable–Observador–3

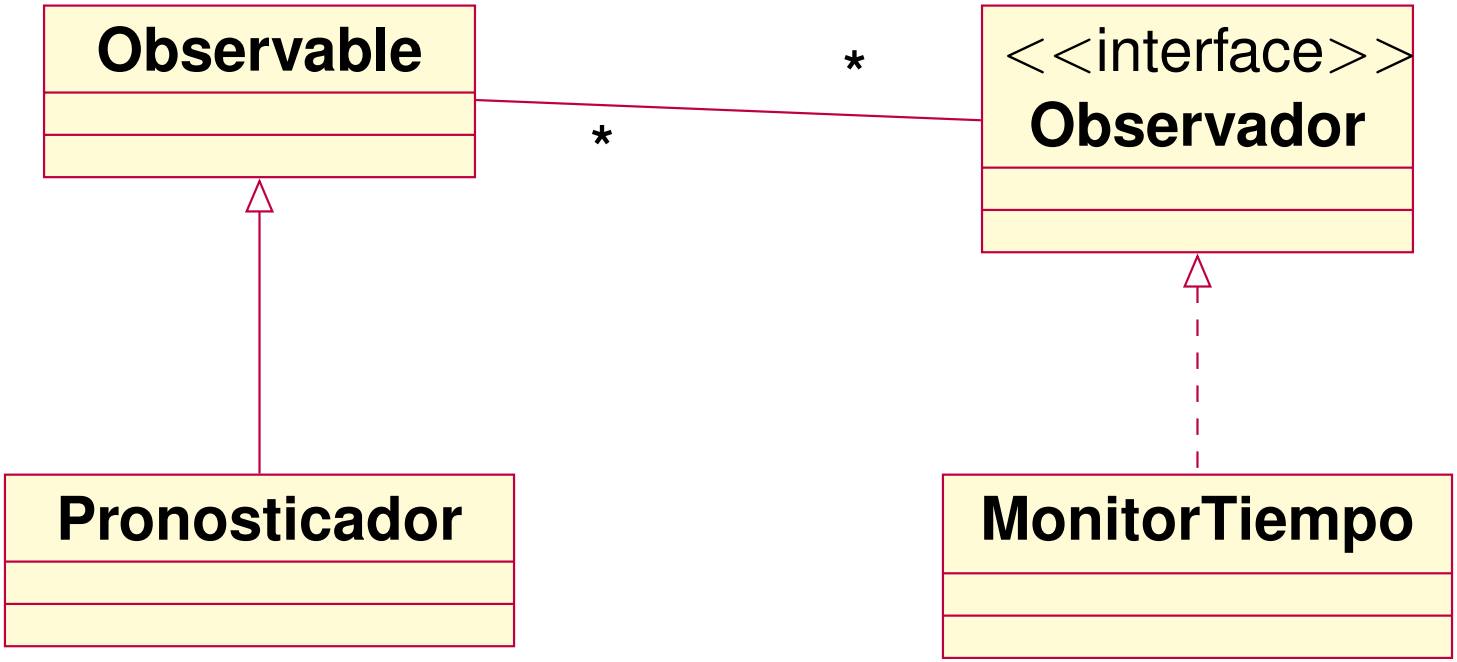


Figura: Ejemplo del patrón Observable–observador

Visitante

Para permitir a las aplicaciones realizar operaciones cuya ejecución necesitará recorrer una estructura jerárquica de datos sin modificar dichos datos o alterando la estructura o las relaciones de herencia de sus clases.

- Contexto
- Problema
- Fuerzas
- Solución
- Ejemplos
- Anti–patrones
- Patrones relacionados
- Referencias: inicialmente propuesto en GoF [Gamma et al., 2009]

Visitante–2

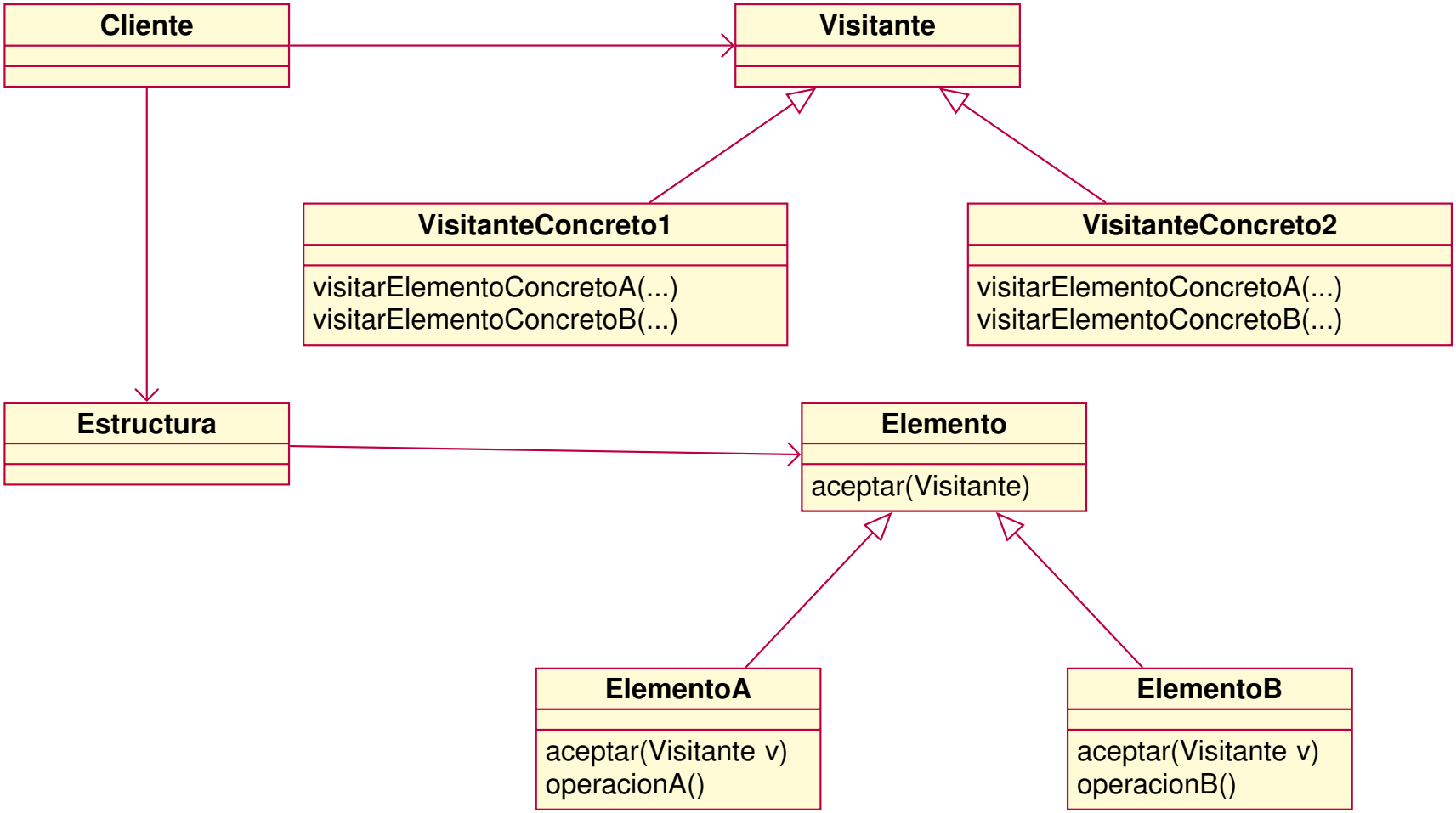


Figura: Diagrama de clases del patrón de diseño Visitante

Visitante–3

```
Cliente::{

VisitantePrecio vep= new VisitantePrecio();
Equipo t= new Tarjeta(); //elementos concretos...
Equipo b= new Bus();
Equipo d= new Disco();
...
t.aceptar(vep);//visita al objeto tarjeta
b.aceptar(vep);//visita al objeto bus
d.aceptar(vep);//visita al objeto disco
}
```

Fachada

Definition (Contexto)

- Mejorar la portabilidad de una aplicación al encapsular las APIs de bajo nivel del sistema operativo.
- Encapsular mecanismos o servicios proporcionados por APIs existentes, no orientadas a objeto.
- Aumentar la cohesión de componentes.

Fachada–2

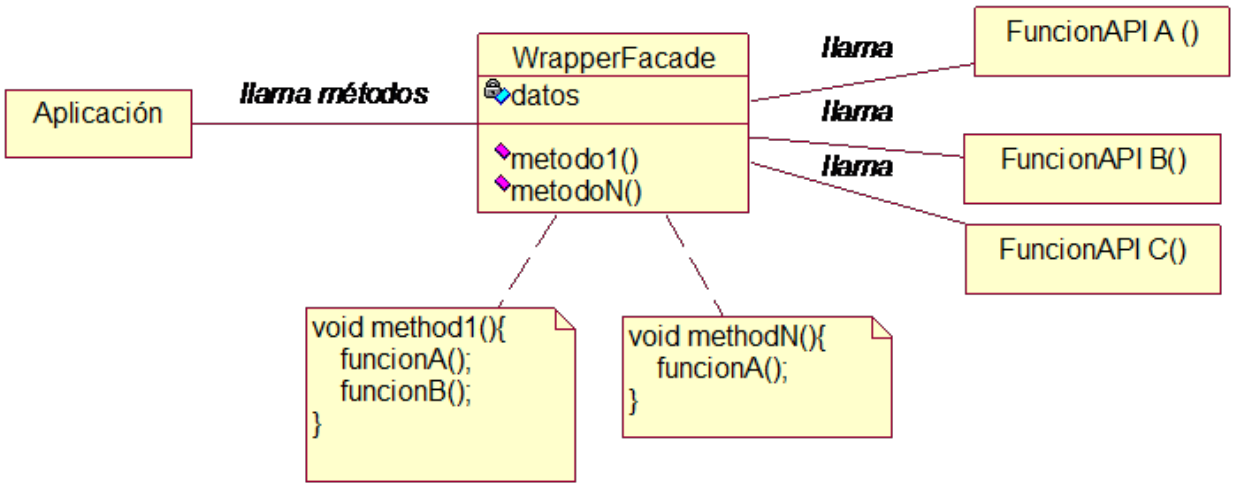


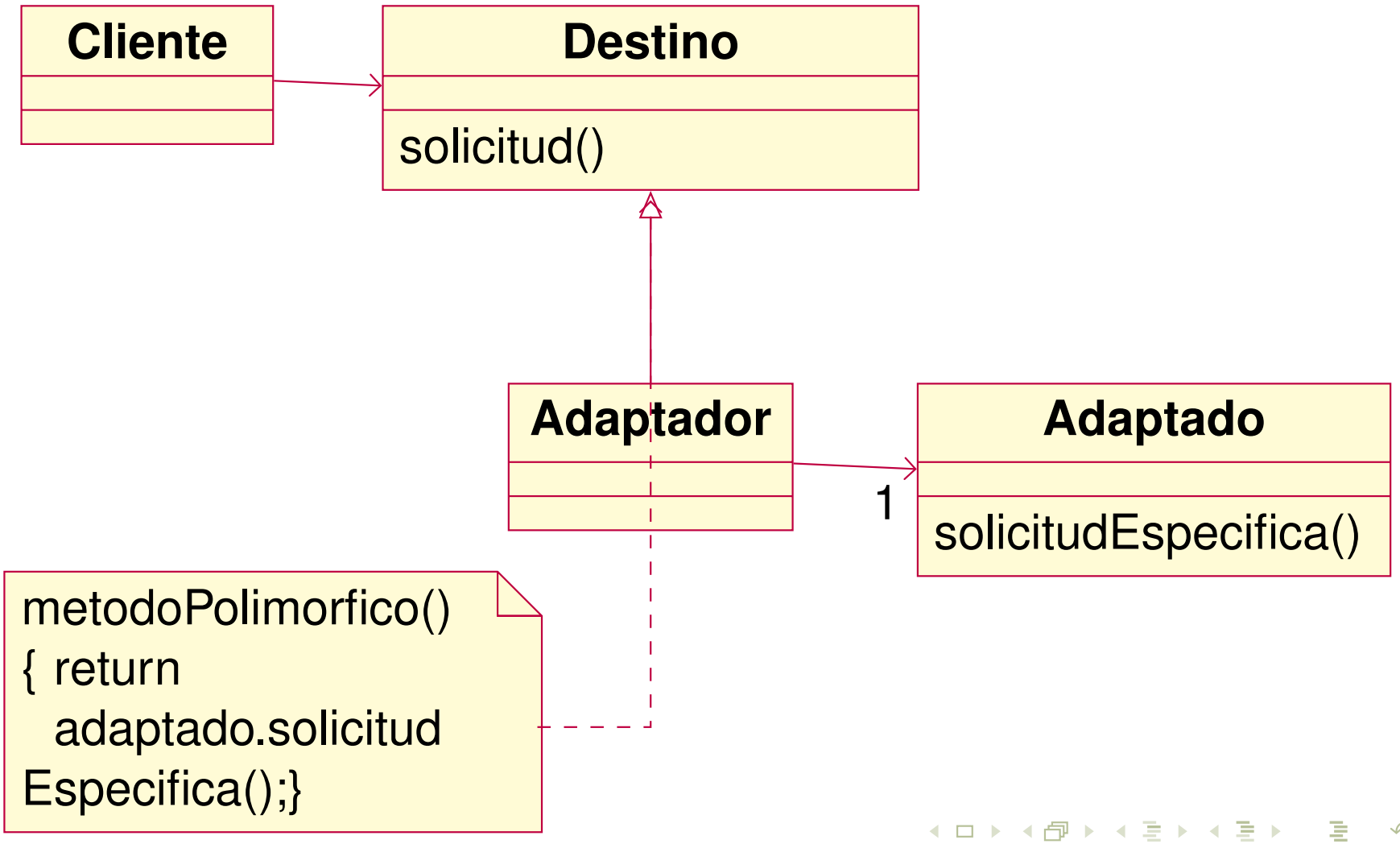
Figura: Diagrama de clases de Wrapper–Facade

Adaptador

Definition (Contexto)

- Reutilizar clases existentes cuya interfaz no es adecuada para lo que se necesita en la aplicación cliente
- Crear clases reutilizables que cooperen con otras clases
- Utilizar varias subclases de manera conjunta

Adaptador–2

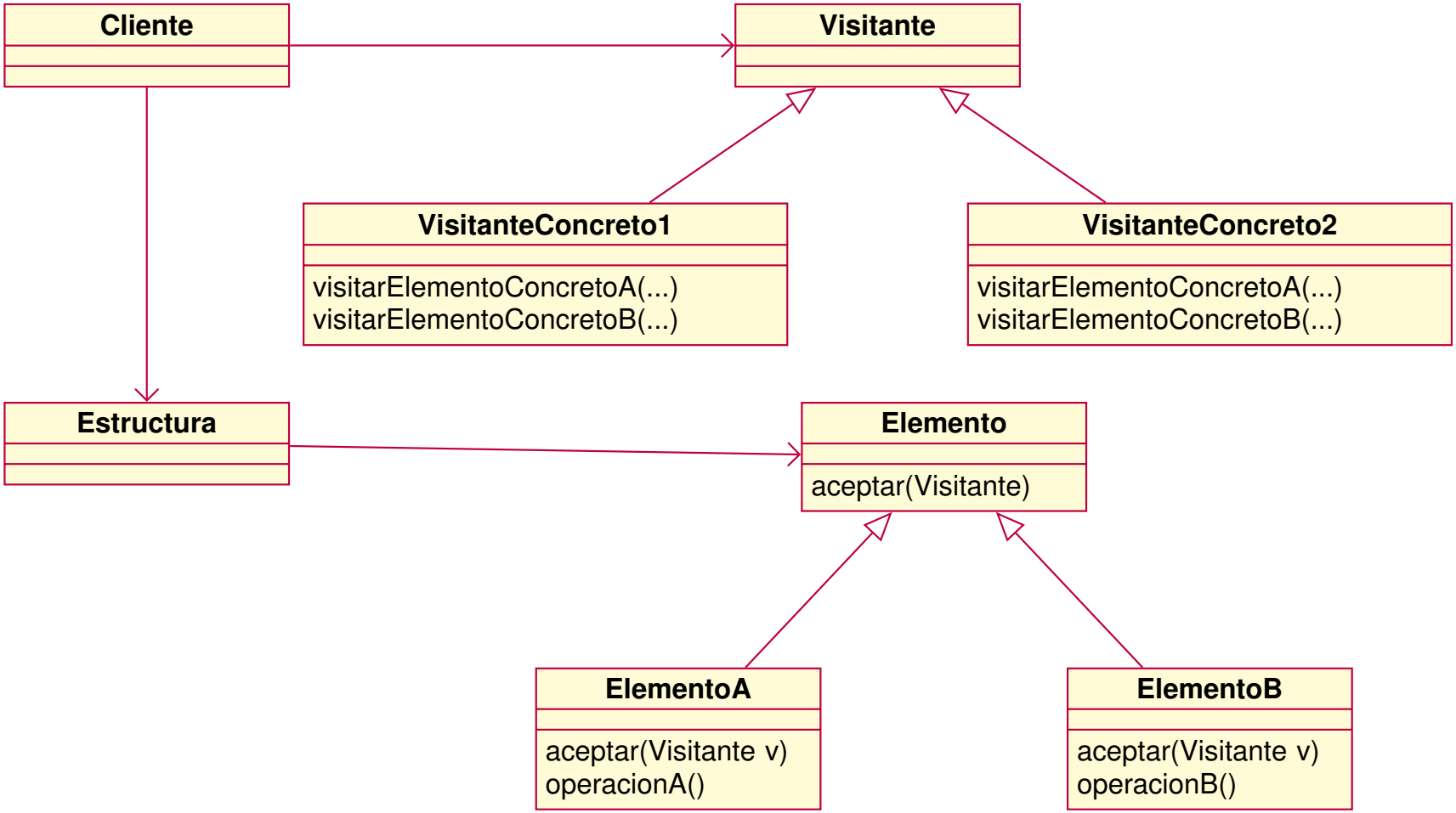

Cliente
Destino
solicitud()
Adaptador
Adaptado
1
solicitudEspecifica()
metodoPolimorfico()
{ return
adaptado.solicitud
Especifica();}

Visitante

Definition (Contexto)

- Estructuras de objetos de muchas clases y con interfaces diferentes
- Ejecutar operaciones distintas y no relacionadas sobre los objetos de la estructura
- Definir nuevas operaciones sobre los objetos

Visitante–2

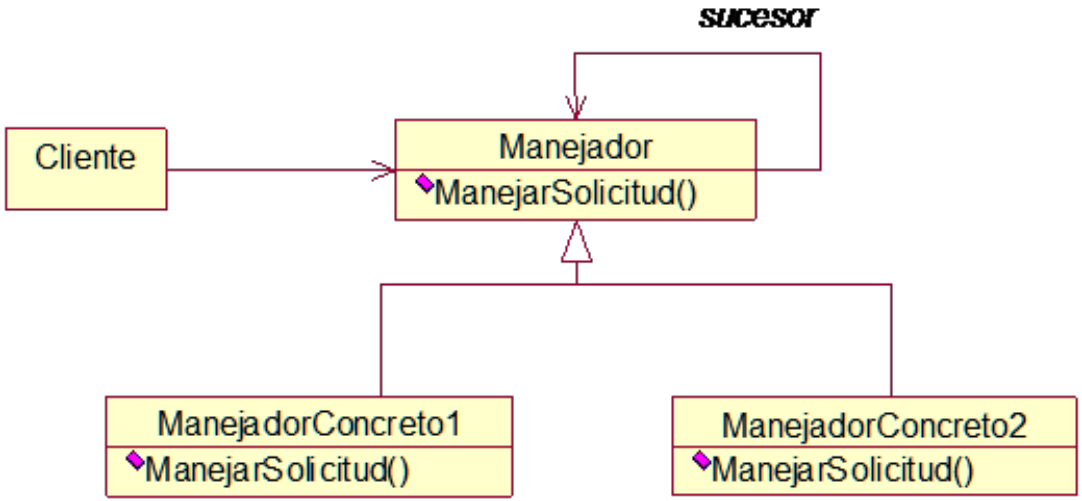

Cliente
Visitante
VisitanteConcreto1
visitarElementoConcretoA(...)
visitarElementoConcretoB(...)
VisitanteConcreto2
visitarElementoConcretoA(...)
visitarElementoConcretoB(...)
Estructura
Elemento
aceptar(Visitante)
ElementoA
aceptar(Visitante v)
operacionA()
ElementoB
aceptar(Visitante v)
operacionB()

Chain of responsability

Definition (Contexto)

- Objetos “manejadores” son asignados dinámicamente para que atiendan peticiones de aplicaciones–cliente
- Delegar el atender las peticiones a uno o varios objetos sin que estos estén determinados de antemano
- Inclusión dinámica de objetos manejadores
- Relacionado con el patrón `Composite`

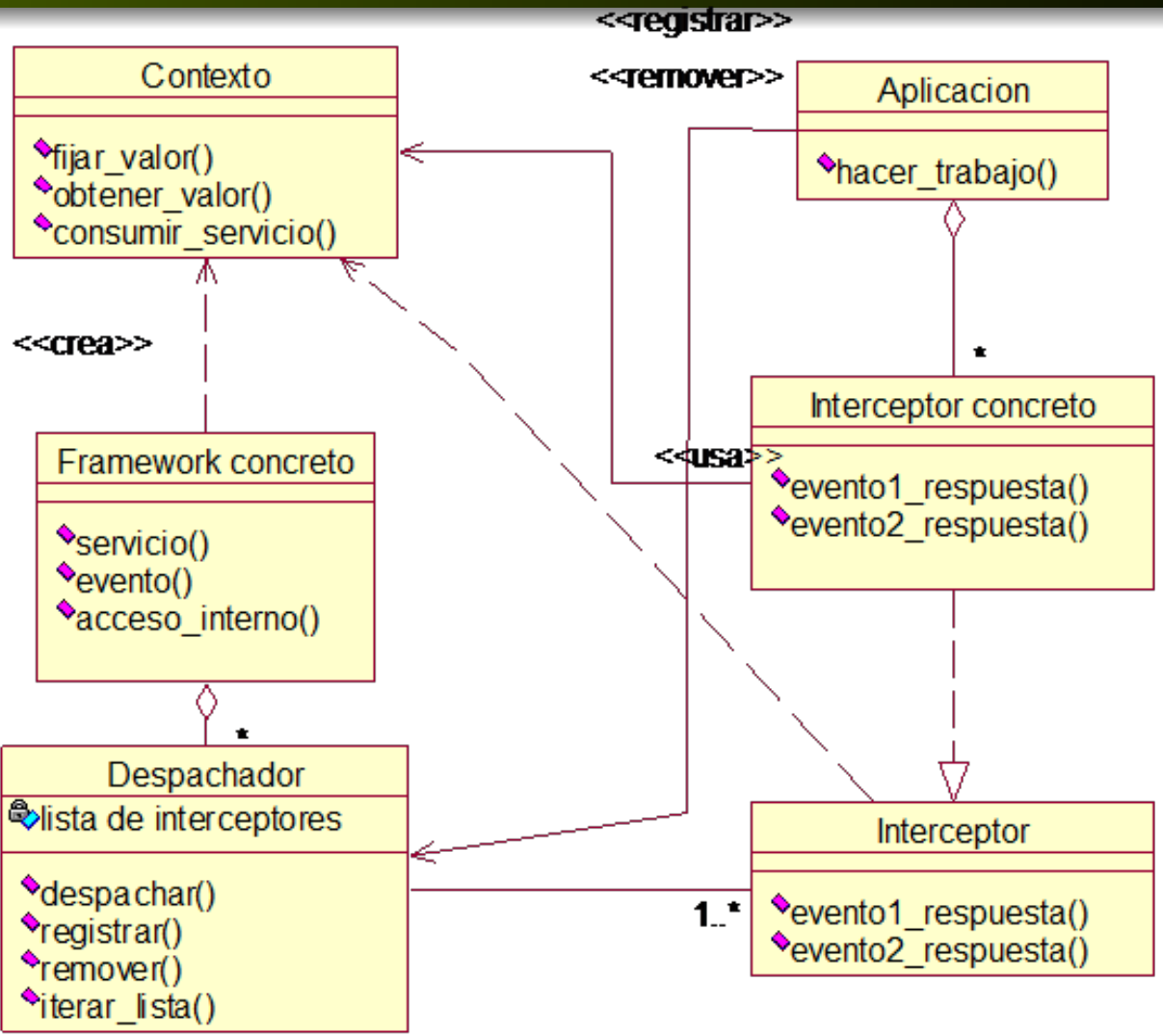
Chain of responsability



Patrón Interceptor

- Permite a determinados servicios ser añadidos de manera transparente a un marco de trabajo y ser disparados automáticamente cuando ocurren ciertos eventos
- Contexto: Desarrollo de marcos de trabajo que puedan ser extendidos de manera transparente
- Problema: Los marcos de trabajo, arquitecturas software, etc. ha de poder anticiparse a las demandas de servicios concretos que deben ofrecer a sus usuarios
- Integración dinámica de nuevos componentes sin afectar a la AS o a otros componentes
- Solución: Registro offline de servicios a través de una interfaz predefinida del marco de trabajo, posteriormente se disparan estos servicios cuando ocurran determinados eventos

Interceptor



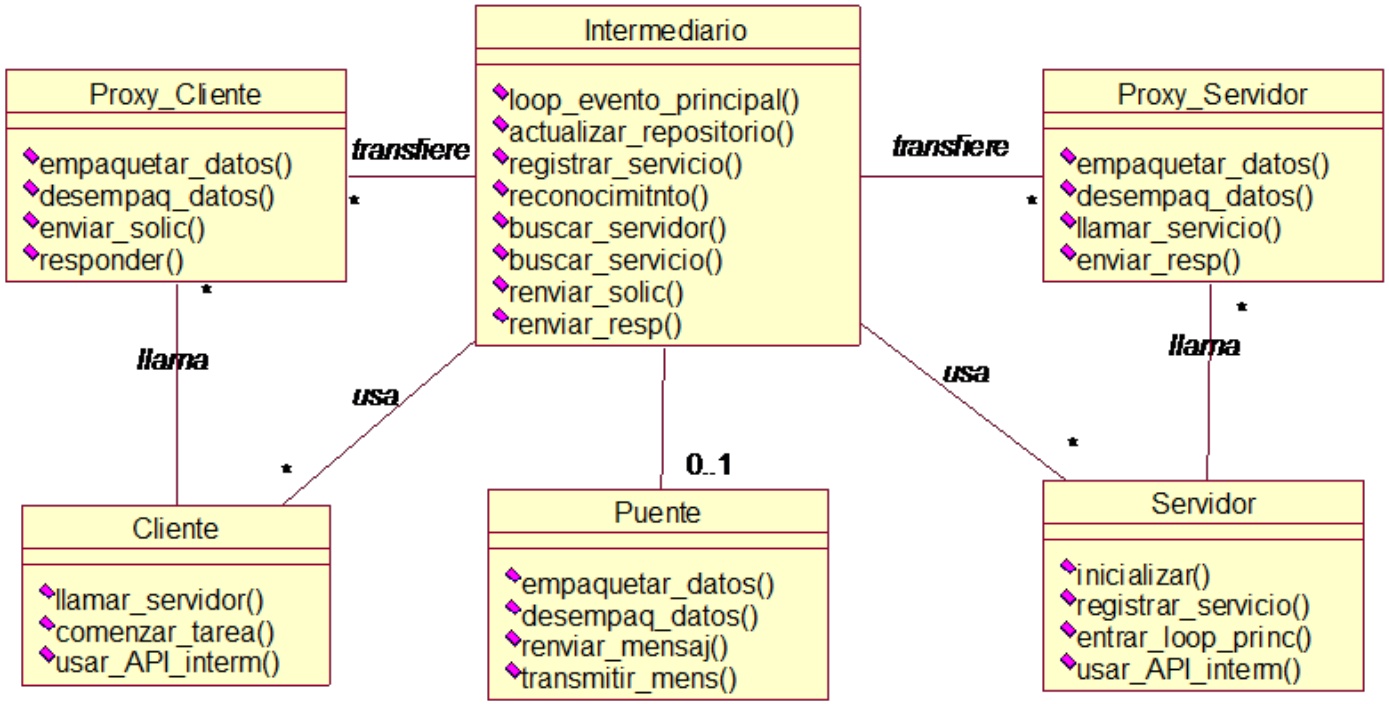
Estructura de clases de “Interceptor"

- Un *FrameworkConcreto* que instancia una arquitectura genérica y extensible
- Los *Interceptores* que son asociados con un evento particular
- *InterceptoresConcretos* que especializan las interfaces del interceptor e implementan sus métodos de enlace
- *Despachadores para* configurar y disparar interceptores concretos
- Una *Aplicación* que se ejecuta por encima de un *framework concreto* y utiliza los servicios que éste le proporciona

Patrón Broker

- Para modelar sistemas distribuidos compuesto de componentes software totalmente desacoplados
- Contexto: cualquier sistema distribuido y, posiblemente, heterogéneo con componentes cooperando dentro de un sistema de información
- Problema:¿Cómo estructurar componentes configurables dinámicamente e independientes de los mecanismos concretos de comunicación de un sistema distribuido?
- Solución: Introducir un componente *Broker* para mejor desacoplamiento entre clientes y servidores
- Las tareas de los Brokers incluyen la localización del servidor apropiado, envío de la petición al servidor y transmisión de los resultados

Broker

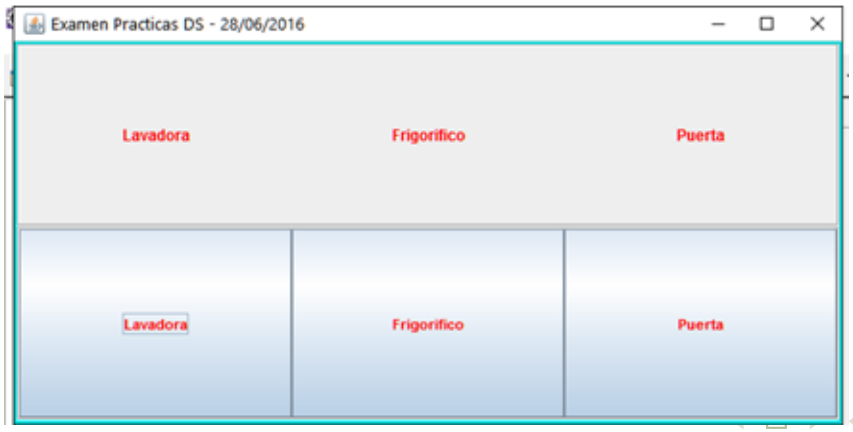

Intermediario
loop_evento_principal()
actualizar_repositorio()
registrar_servicio()
reconocimitnto()
buscar_servidor()
buscar_servicio()
renviar_solic()
renviar_resp()
Proxy_Cliente
empaquetar_datos()
desempaq_datos()
enviar_solic()
responder()
Proxy_Servidor
empaquetar_datos()
desempaq_datos()
llamar_servicio()
enviar_resp()
transfiere
transfiere
llama
usa
usa
llama
0..1
Cliente
llamar_servidor()
comenzar_tarea()
usar_API_interm()
Puente
empaquetar_datos()
desempaq_datos()
renviar_mensaj()
transmitir_mens()
Servidor
inicializar()
registrar_servicio()
entrar_loop_princ()
usar_API_interm()

Estructura de clases del patrón Broker

- *Intermediario*: admite las solicitudes, asigna los servidores y responde a las peticiones de los clientes
- *Servidor*: se registra en el *Intermediario* e implementa el servicio
- *Cliente*: accede a los servicios remotos
- *Proxy Cliente* y *Proxy Servidor*, que proporcionan transparencia, ocultando los detalles de implementación del patrón
- *Puente*: le proporciona interoperabilidad al *Intermediario*

Ejemplo de uso del patrón Broker

Ejemplo demostrativo con un sistema E-Home que implementa una red de dispositivos domésticos colaborativos.
El `Broker` se encargará de facilitar la compatibilidad entre el software y los datos del cliente con los controladores de los distintos dispositivos incluidos en esta red.
Los servicios que están situados en un control central estarán conectados a través de una red con el software que se ejecuta en el controlador de los dispositivos.



Ejemplo de uso del patrón Broker-II

Estructura de clases

- `Cliente`: envía una petición de activación de un dispositivo concreto a `Broker`
- `Broker`: accede a `ControlCentral` de la red y envía al servidor los mensajes que ha recibido de `Cliente`
- `ControlCentral`: recibe mensajes de `Broker` y contacta con los distintos dispositivos para configurarlos y activarlos
- `ControladorDispositivo`: realiza acciones específicas sobre el dispositivo del que es responsable

Ejemplo de uso del patrón Broker-III

ControladorDispositivo, además de su constructor, sólo define un método tal como el siguiente:

```
synchronized void realizarAccion(String m){
  System.out.println(m+ " realizara accion");
}
```

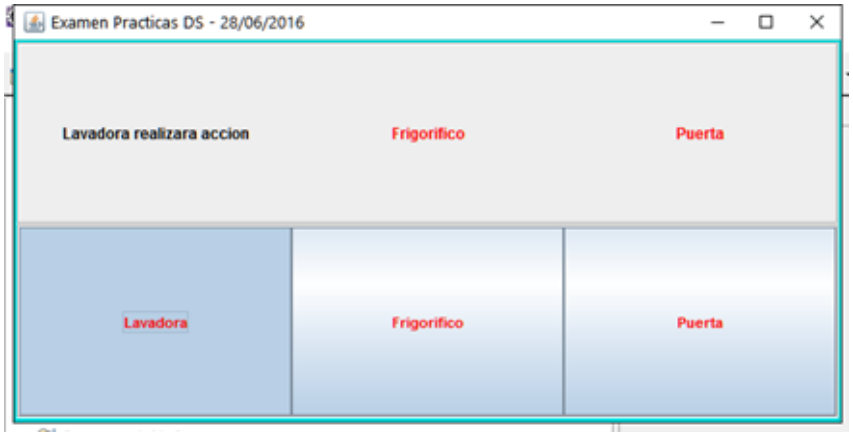
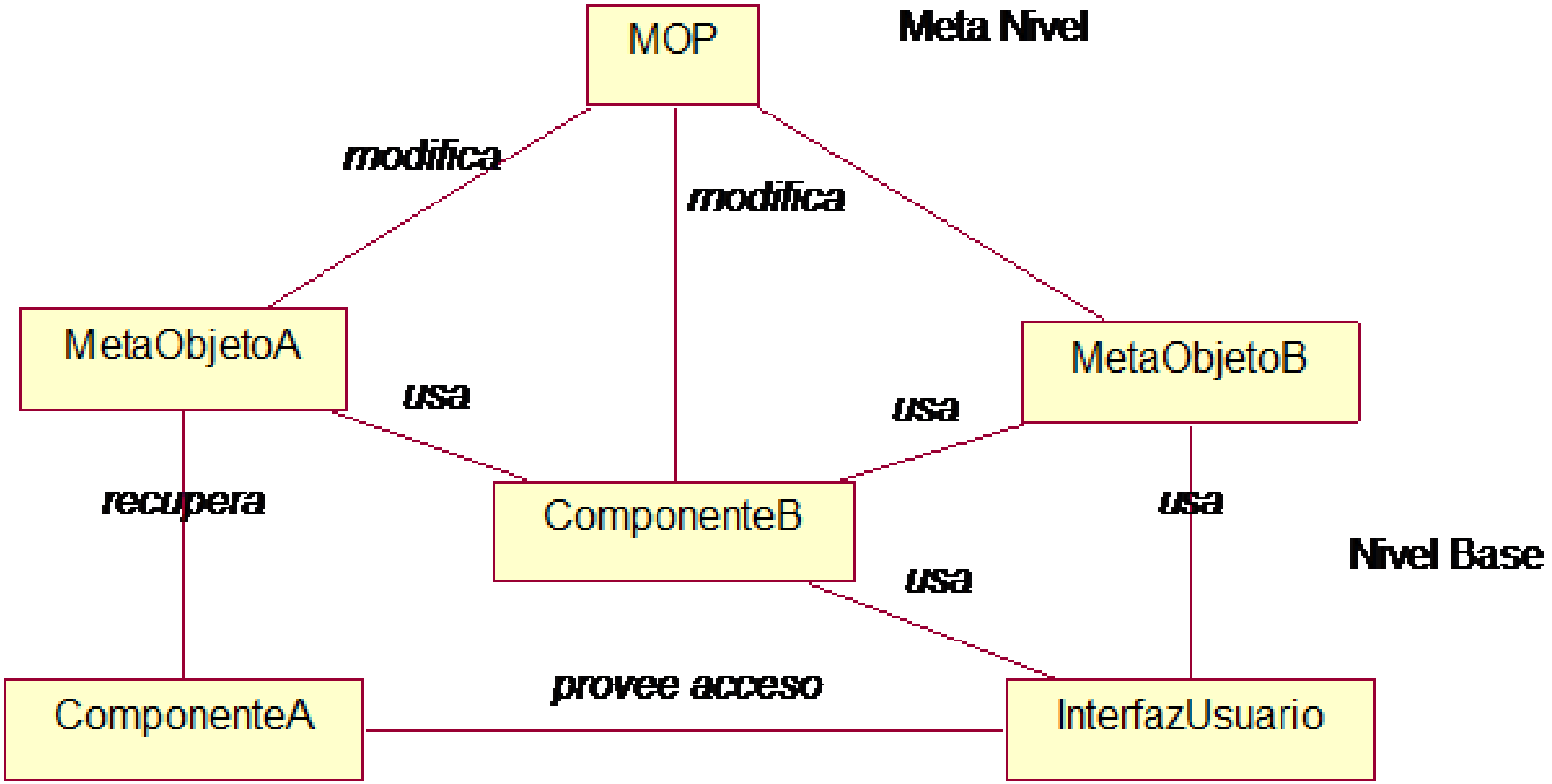


Figura: Aspecto de la interfaz después de pulsar el botón *Lavadora*

Patrón Reflection

- Proporciona un mecanismo para cambiar la estructura y comportamiento de sistemas software dinámicamente
- Soporta la modificación de aspectos fundamentales, tales como estructuras y mecanismos de llamadas a métodos.
- Contexto: Cualquier sistema que necesite soporte para realizar cambios propios y para conseguir persistencia de sus entidades
- Problema: ¿Cómo se puede modificar el comportamiento de los objetos de una jerarquía dinámicamente sin afectar a los propios objetos en su configuración actual?
- Solución: Hacer que el software sea “auto-consciente" de su función y comportamiento, haciendo que los aspectos seleccionados sean accesibles para su adaptación y cambio dinámico

Reflection


MOP
Meta Nivel
modifica
modifica
MetaObjetoA
MetaObjetoB
usa
usa
recupera
ComponenteB
usa
usa
Nivel Base
provee acceso
ComponenteA
InterfazUsuario

Estructura de clases del patrón Reflection

- *Meta Nivel*: autoconsciencia de la estructura y funcionamiento del software
- La implementación del *Meta Nivel* utiliza *Meta Objetos*
- *Meta Objetos*: encapsulan y representan información acerca del software
- *Nivel Base*: objetivo y relación con el metanivel
  - Los cambios realizados en el *Meta Nivel* afectan consecuentemente al comportamiento del *Nivel Base*
- Cambios en los metaobjetos y su efecto en los componentes y código del nivel base

Ejemplo de uso del patrón Reflection: *juego de la metamorfosis*

Ejemplo demostrativo del patrón *Reflection* que genera clases y métodos utilizando este patrón de diseño y las facilidades reflectivas de Java.
También incluye una interfaz de 4 botones programada con Swing/Java.

Estructura de la aplicación:

- Clase `Bicho`: métodos para fijar el tipo de bicho, obtener su descripción (aparecerá como una etiqueta en el panel inferior) y métodos para que un bicho se *"mueva"* (si puede hacerlo).

Ejemplo de uso del patrón Reflection: *juego de la metamorfosis*–II

La metamorfosis de un bicho puede estar en tres fases distintas: *Gusano ("worm")*, *Capullo ("cocoon")* y *Mariposa ("butterfly")*.

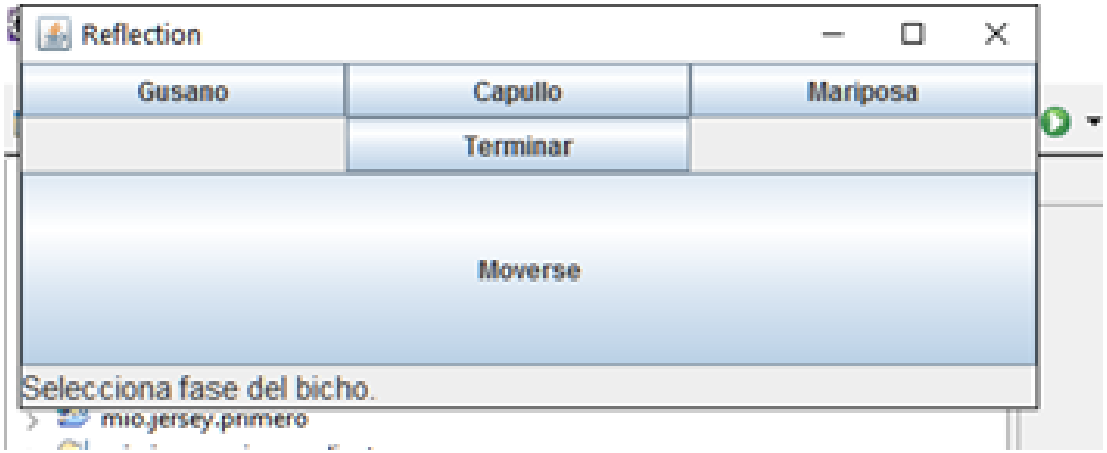


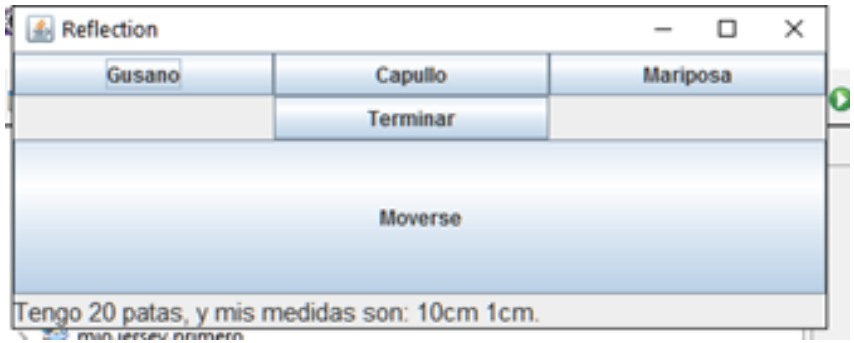
Figura: Aspecto de la interfaz al comenzar la aplicación

Ejemplo de uso del patrón Reflection: *juego de la metamorfosis*–III

```
public class Bicho {
  public String tipo;
  public void setTipo(String t){
    tipo= t;
    }
  public String getTipo(){
    return tipo;
    }
  public String descripcionGusano(int patas, int longitud, int ancho){
        return "Tengo: "+patas+" patas, y mis medidas son: "+longitud+"cm, "+ancho+"
             cm.";
        }
  public String descripcionCapullo(int radio){
        return "Mido: "+radio+"cm de radio.";
        }
  public String descripcionMariposa(int longitud, int ancho){
        return "Mido: "+longitud+"cm de longitud y "+ancho+"cm de ancho.";
        }
...
}
```

Ejemplo de uso del patrón Reflection: *juego de la vida*–IV

```
....
public String mover(int velocidad){
  if (tipo=="Gusano"){
        return "Voy caminando:"+velocidad+"Km/h.";}
   else if (tipo=="Capullo"){
        return "Soy un capullo, no me puedo mover.";}
      else{return "Voy volando:"+velocidad+"Km/h.";}
        }
}
```



Ejemplo de uso del patrón Reflection: *juego de la vida*–V

```
\\Clase Interfaz --parte Reflectiva
Class cls = Class.forName("com.ds_reflection.Bicho");
Object obj = cls.newInstance();
Method metodo;
```

Instalación de botones

```
public void actionPerformed(ActionEvent evt){//Utilizar para: new JButton("Gusano")
Class[] paramString = new Class[1];
Class[] paramGusano = new Class[3];
...
paramString[0]= String.class;
String tipo= "Gusano", salida1;
try {
   metodo = cls.getDeclaredMethod("setTipo", paramString);
   metodo.invoke(obj, tipo);
   metodo = cls.getDeclaredMethod("descripcionGusano", paramGusano);
   Integer[] pam = {20,10,1};
   salida1 = (String) metodo.invoke(obj, pam);
   textField.setText(salida1);
  }
  catch (NoSuchMethodException | SecurityException | IllegalAccessException|
       IllegalArgumentException | InvocationTargetException e) {
                                e.printStackTrace(); }
};//actionPerformed()
```

Ejemplo de uso del patrón Reflection: *juego de la vida*–VI

```
\\Clase Interfaz --parte Reflectiva
Class cls = Class.forName("com.ds_reflection.Bicho");
Object obj = cls.newInstance();
Method metodo;
```

Instalación del botón “Moverse”

```
public void actionPerformed(ActionEvent e){
Class[] paramVelocidad = new Class[1];
paramVelocidad[0] = Integer.TYPE;

...
try {
        metodo = cls.getDeclaredMethod("getTipo", null);
        String tipo  = (String)metodo.invoke(obj, null);
        metodo = cls.getDeclaredMethod("mover", paramVelocidad);
        int velocidad = 1;
        salida2 = (String) metodo.invoke(obj, velocidad);
        textField.setText(salida2);
        }
        catch (NoSuchMethodException | SecurityException | IllegalAccessException|
             IllegalArgumentException | InvocationTargetException e) {
                                e.printStackTrace(); }
};//actionPerformed()
```