Tema 5
Patrones de Diseño

Pablo Sánchez

Dpto. Matemáticas, Estadística y Computación
Universidad de Cantabria
Santander (Cantabria, España)
p.sanchez@unican.es

Objetivos

Objetivos

1. Comprender en concepto de patrón de diseño.
2. Saber aplicar los patrones GoF.
3. Saber aplicar ciertos patrones no GoF: patrón *mixin*.
4. Comprender el concepto de *antripatrón*.
5. Comprender el concepto de refactorización.
6. Saber aplicar ciertas refactorizaciones básicas.

Bibliografía Básica

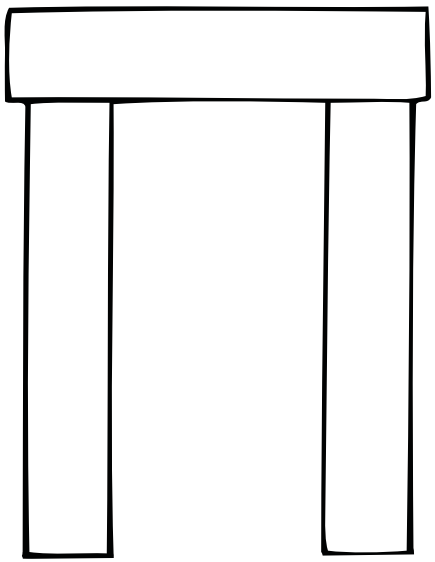
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison Wesley, November 1994.

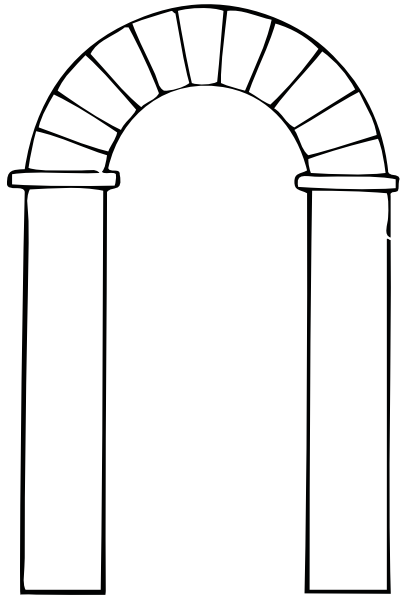
William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray.
AntiPatterns: Refactoring Software, Architectures and Projects in Crisis.
Wiley, April 1998.

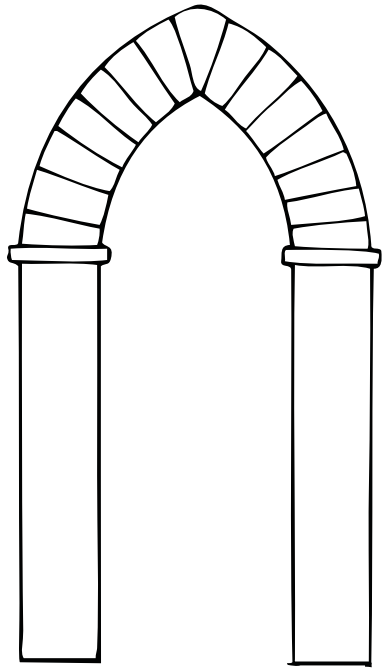
Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Addison Wesley, July 1999.

Índice

Motivación







Motivación

Motivación

Gol de Puyol a Alemania Sudáfrica 2010 (ver)

Índice

Patrón de Diseño Software

Patrón de Diseño Software

Solución probada y beneficiosa a problemas recurrentes que aparecen continuamente durante el diseño de un producto software.

Patrón de Diseño Software (Gamma et al, 1994)

Una descripción de como interconectar clases y objetos de forma que puedan resolver un problema general de diseño dentro de un contexto determinado.

Estructura de un Patrón

1. Nombre (y alias).
2. Clasificación.
3. Propósito.
4. Motivación.
5. Aplicabilidad.
6. Estructura.
7. Participantes.
8. Interacción entre participantes.
9. Consecuencias.
10. Implementación.
11. Ejemplos con código.
12. Usos conocidos.
13. Patrones relacionados.

Tipos de Patrón

Creacionales Relativos a problemas relacionados con la creación de objetos (ej. Factoría).

Estructurales Relativos a problemas relacionados con las relaciones y dependencias estáticas entre clases y objetos (ej. Peso mosca).

De comportamiento Relativos a problemas relacionados el comportamiento de clases y objetos en tiempo de ejecución (ej. Estrategia).

Índice

Patrones GoF (Gang of Four)

1. Singleton.
2. Factoría abstracta.
3. Adaptador.
4. Estrategia.
5. Estado.

Patrón Singleton

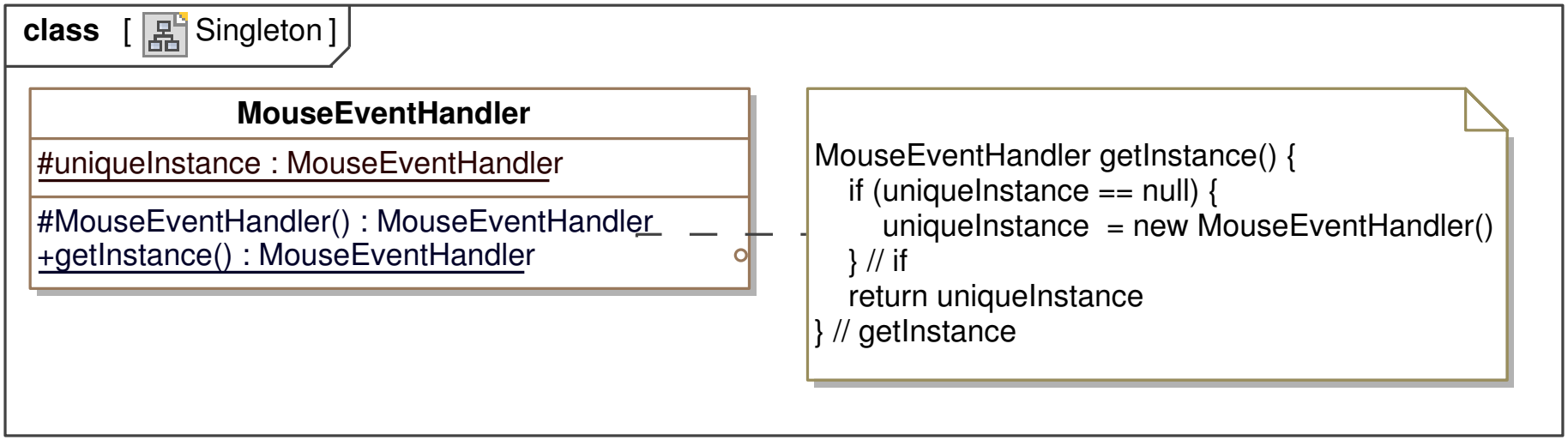
Problema

Sólo debe existir una instancia de una clase (ej. driver de ratón).

Aplicabilidad

Sólo debe existir una instancia por clase, la cual debe estar accesible desde un punto de entrada bien definido.

Patrón Singleton



Patrones GoF (Gang of Four)

1. Singleton.
2. Factoría abstracta.
3. Adaptador.
4. Estrategia.
5. Estado.

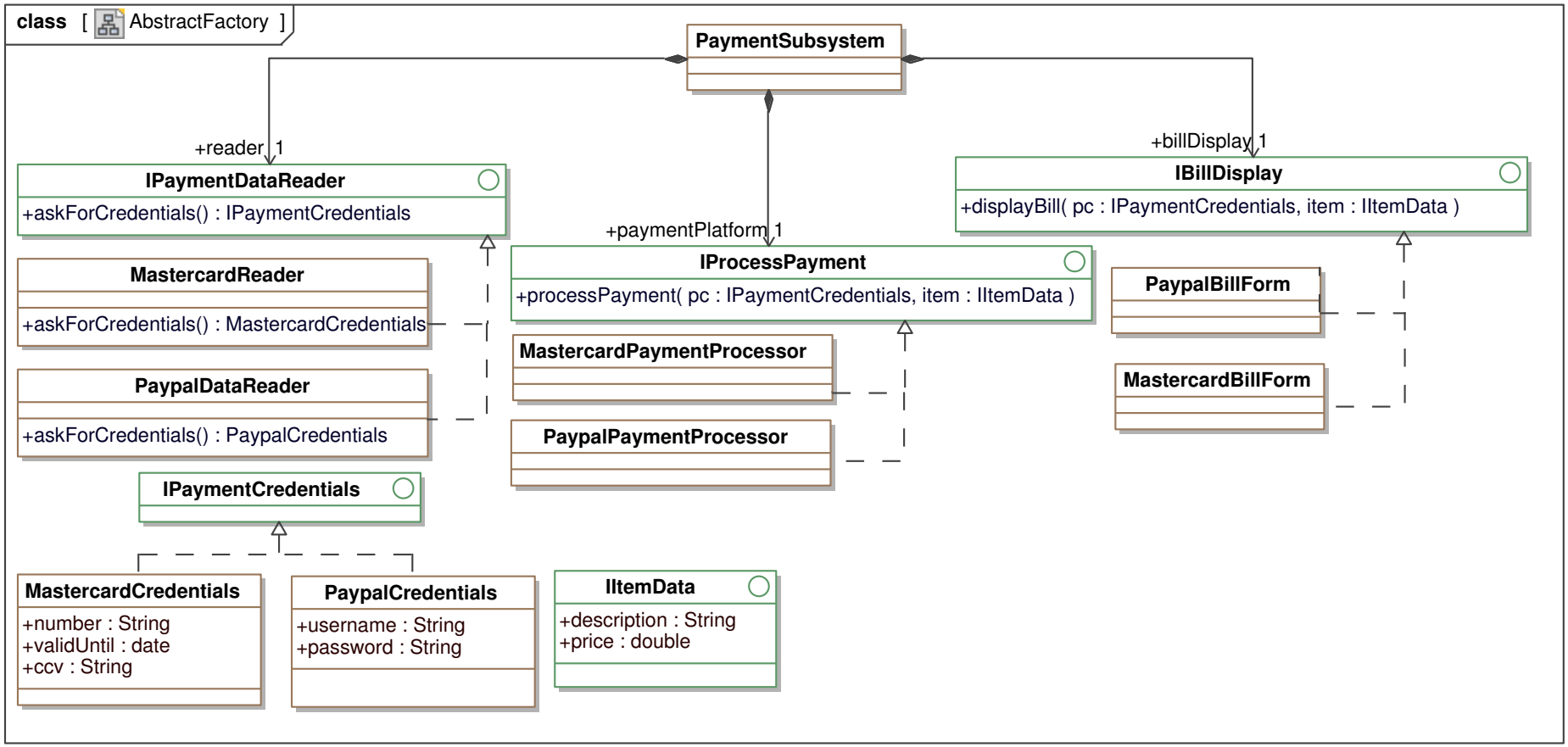
Patrón Factoría Abstracta

Problema

Dependencia de clases concretas.

Ej. `List<Persona> listaAsistentes = new ArrayList<Persona>`

Patrón Factoría Abstracta - Problema

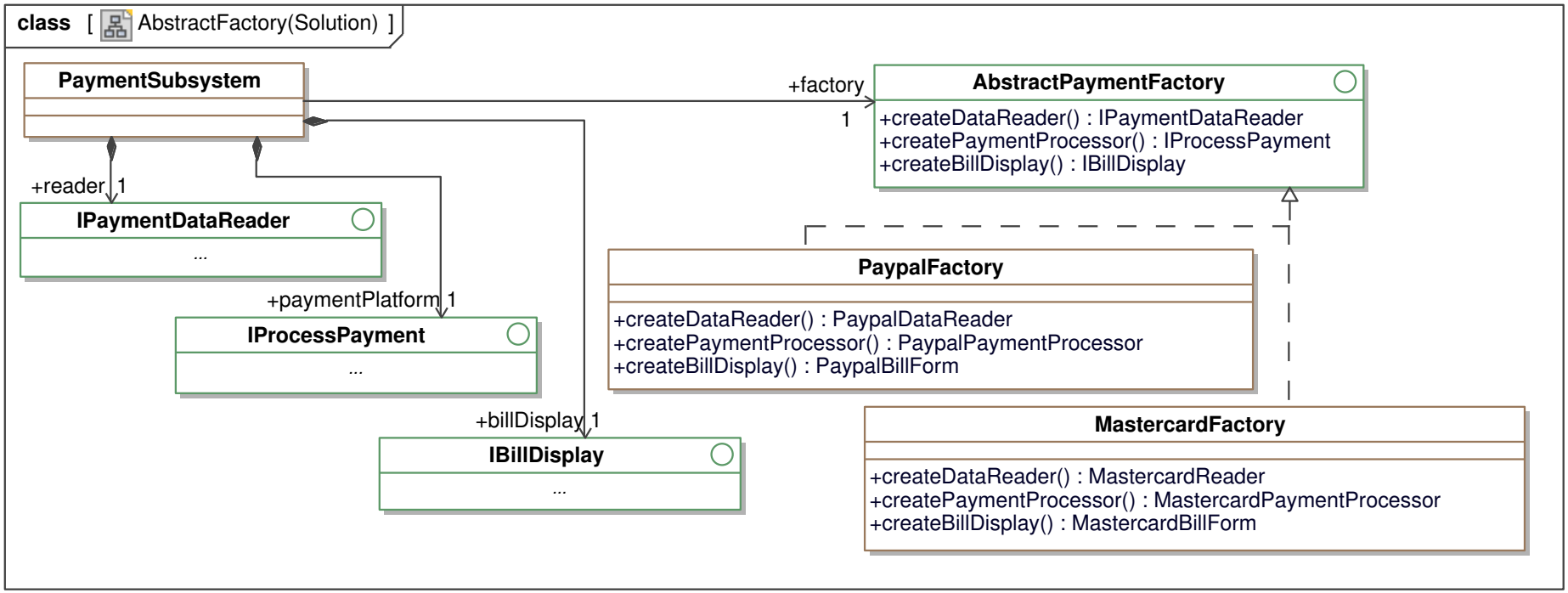


Patrón Factoría Abstracta

Aplicabilidad

1. Un sistema debe ser independiente de como sus productos son creados, compuestos y representados.
2. Un sistema debe ser configurado usando un conjunto coherente de clases concretas.
3. Deseamos crear una librería de productos, pero sólo queremos hacer públicas las interfaces, no las implementaciones.

Patrón Factoría Abstracta - Solución



Patrón Factoría Abstracta

Ventajas

1. Hace independientes las clases clientes de clases concretas particulares.
2. Hace más fácil cambiar la configuración actual de un producto, ya que sólo hay que cambiar la factoría concreta.
3. Favorece la consistencia entre productos.

Patrones GoF (Gang of Four)

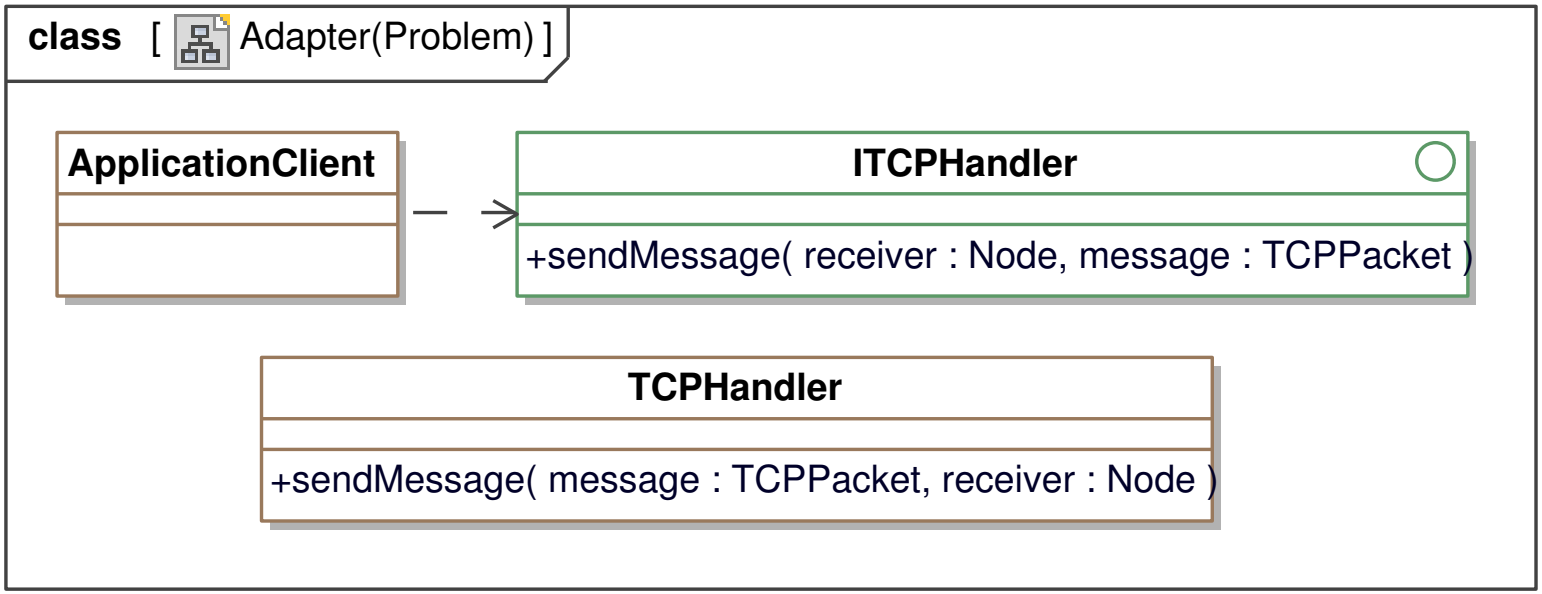
1. Singleton.
2. Factoría abstracta.
3. Adaptador.
4. Estrategia.
5. Estado.

Patrón Adaptador

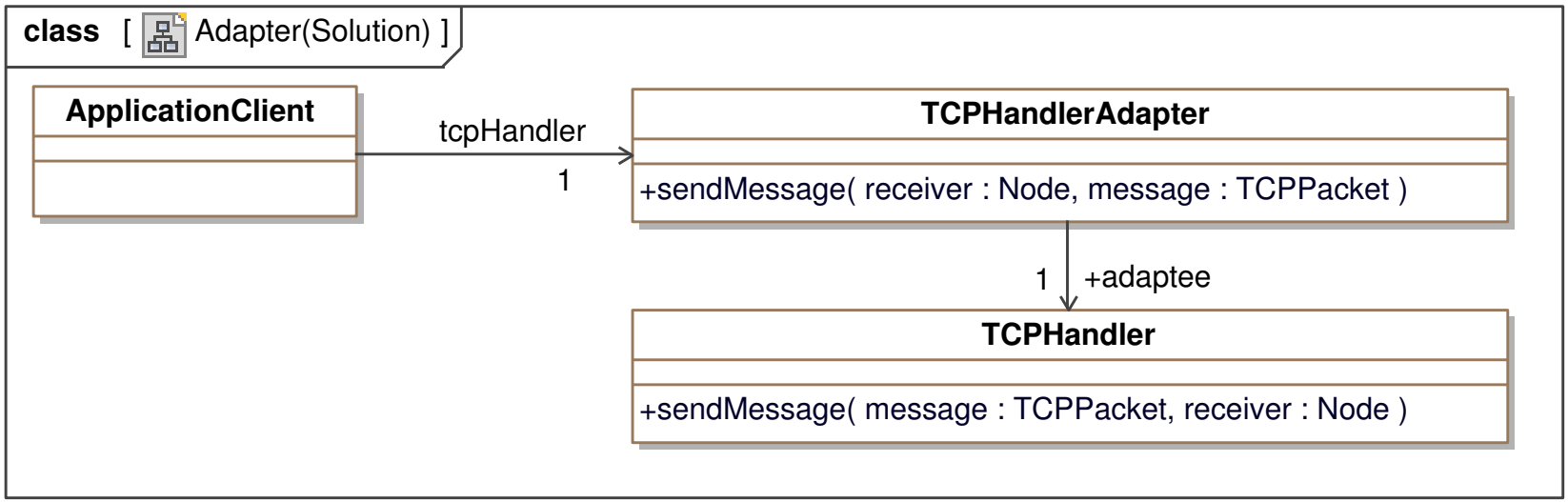
Problema

La interfaz de una clase no se corresponde con la interfaz esperada por los clientes.

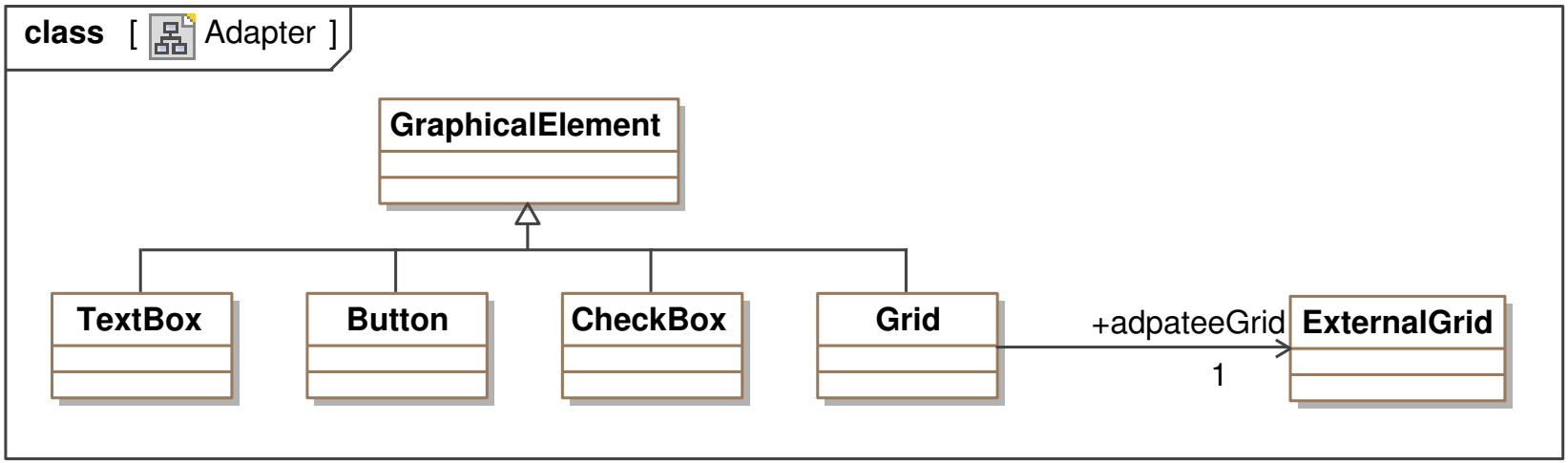
Patrón Adaptador



Patrón Adaptador



Patrón Adaptador



Patrones GoF (Gang of Four)

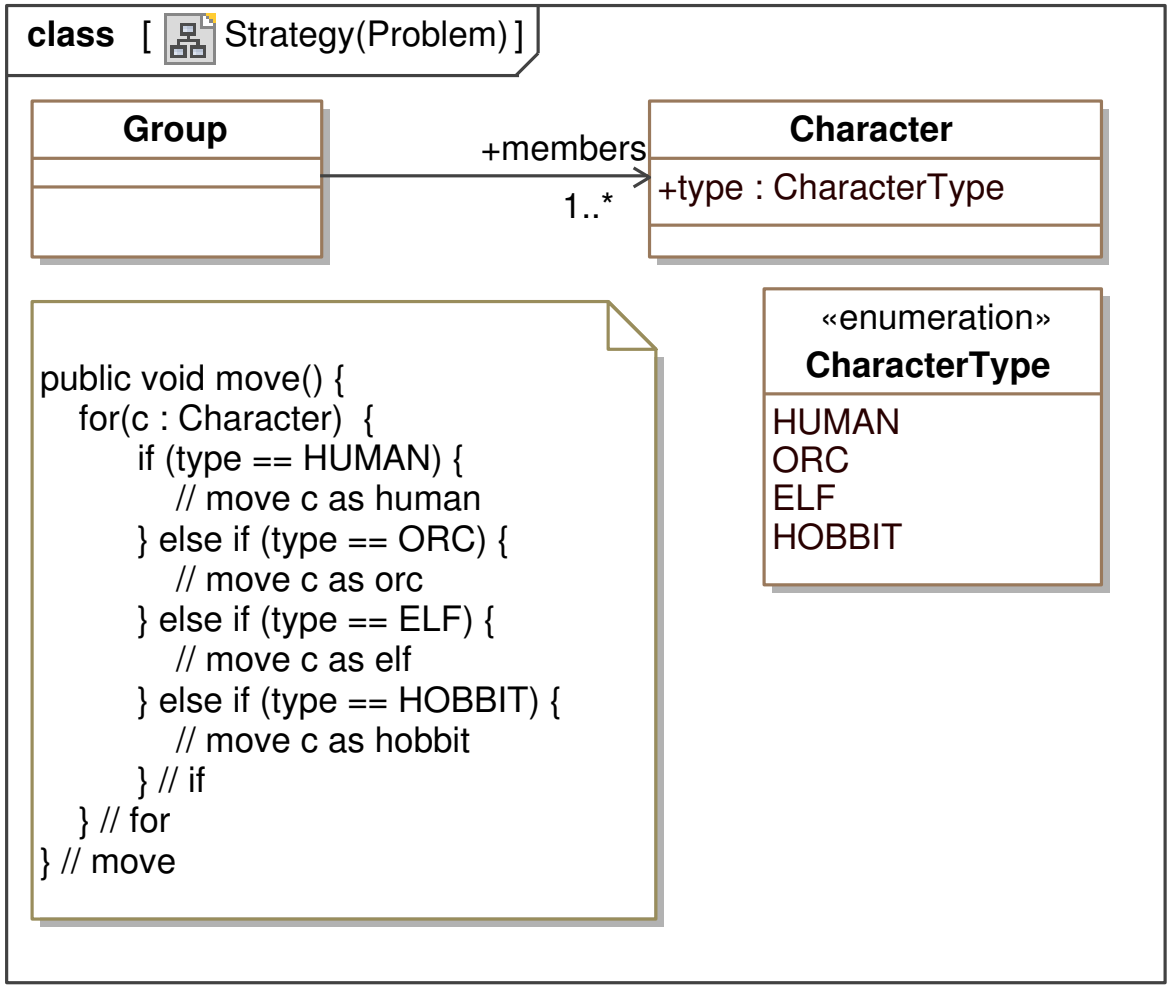
1. Singleton.
2. Factoría abstracta.
3. Adaptador.
4. Estrategia.
5. Estado.

Patrón Estrategia

Problema

La implementación de un método debe variar, pero queremos hacer a las clases clientes ignoren dicha cuestión.

Patrón Estrategia

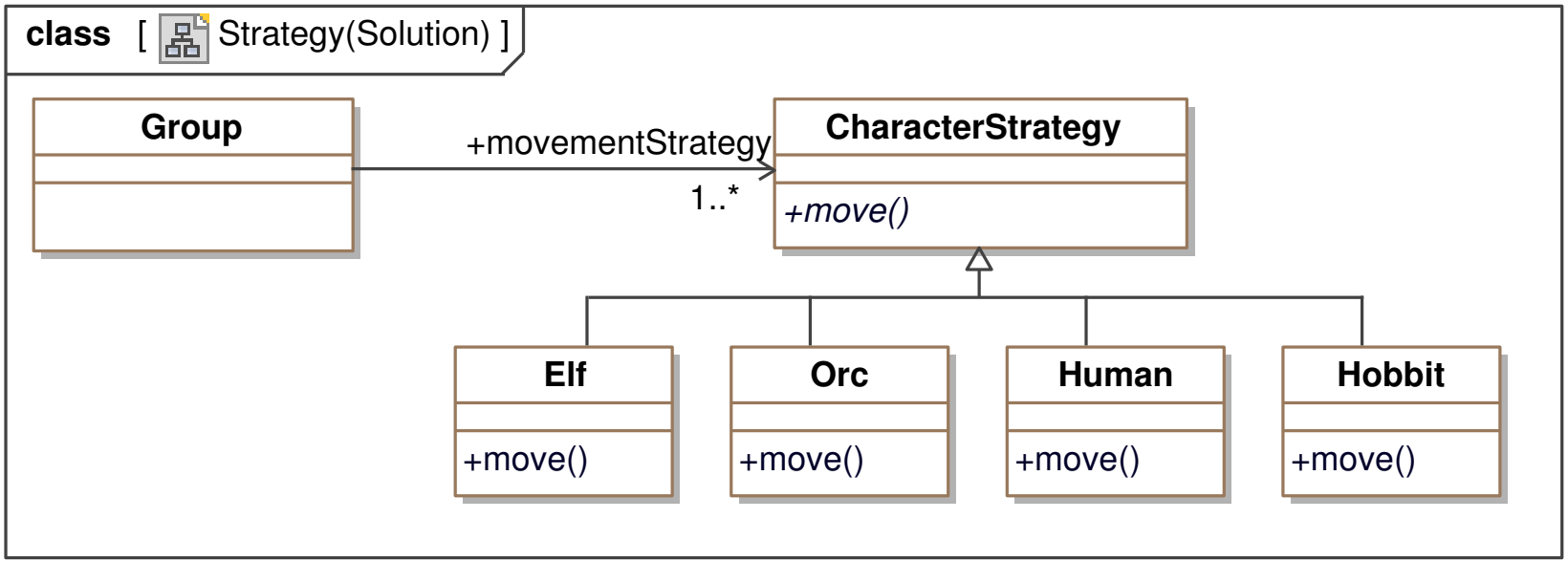


Patrón Estrategia

Aplicación

1. Varias clases sólo difieren en el comportamiento de una o dos responsabilidades
2. Es necesario implementar diversas variantes de un mismo algoritmo y seleccionar una variante concreta en tiempo de ejecución.

Patrón Estrategia



Patrones GoF (Gang of Four)

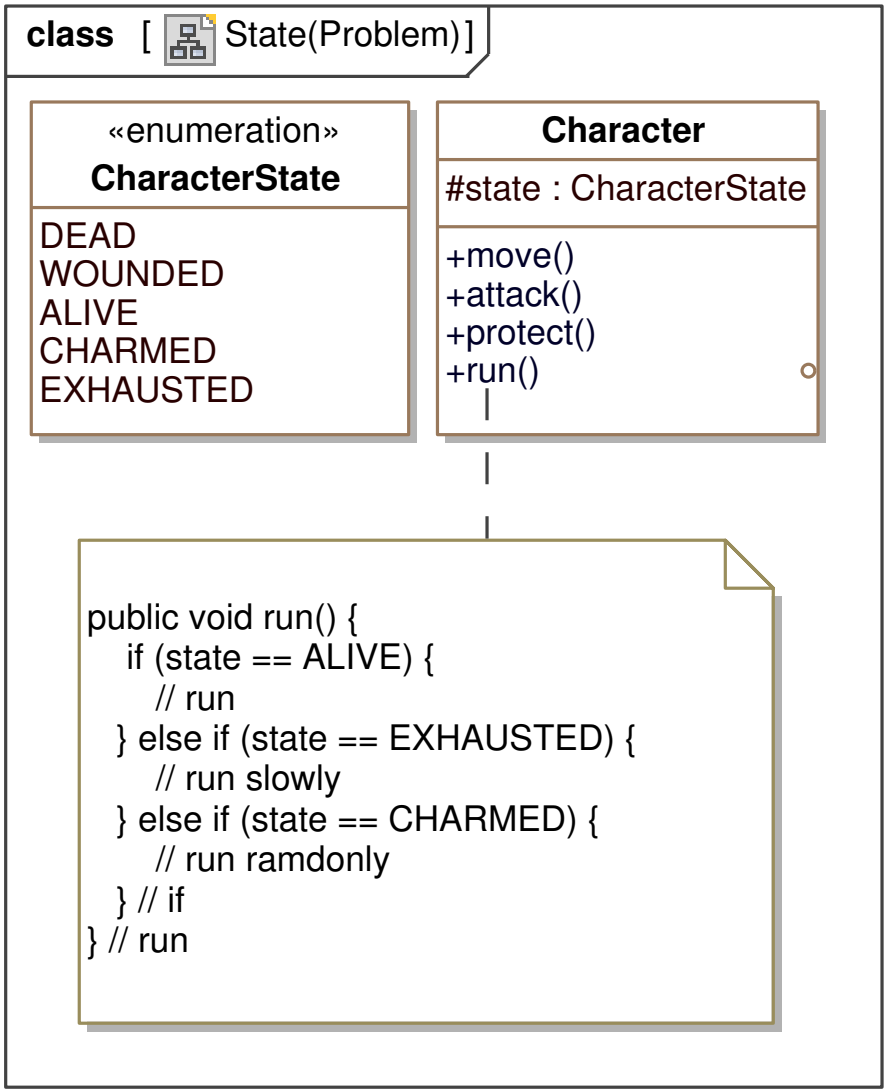
1. Singleton.
2. Factoría abstracta.
3. Adaptador (Wrapper).
4. Estrategia.
5. Estado.

Patrón Estado

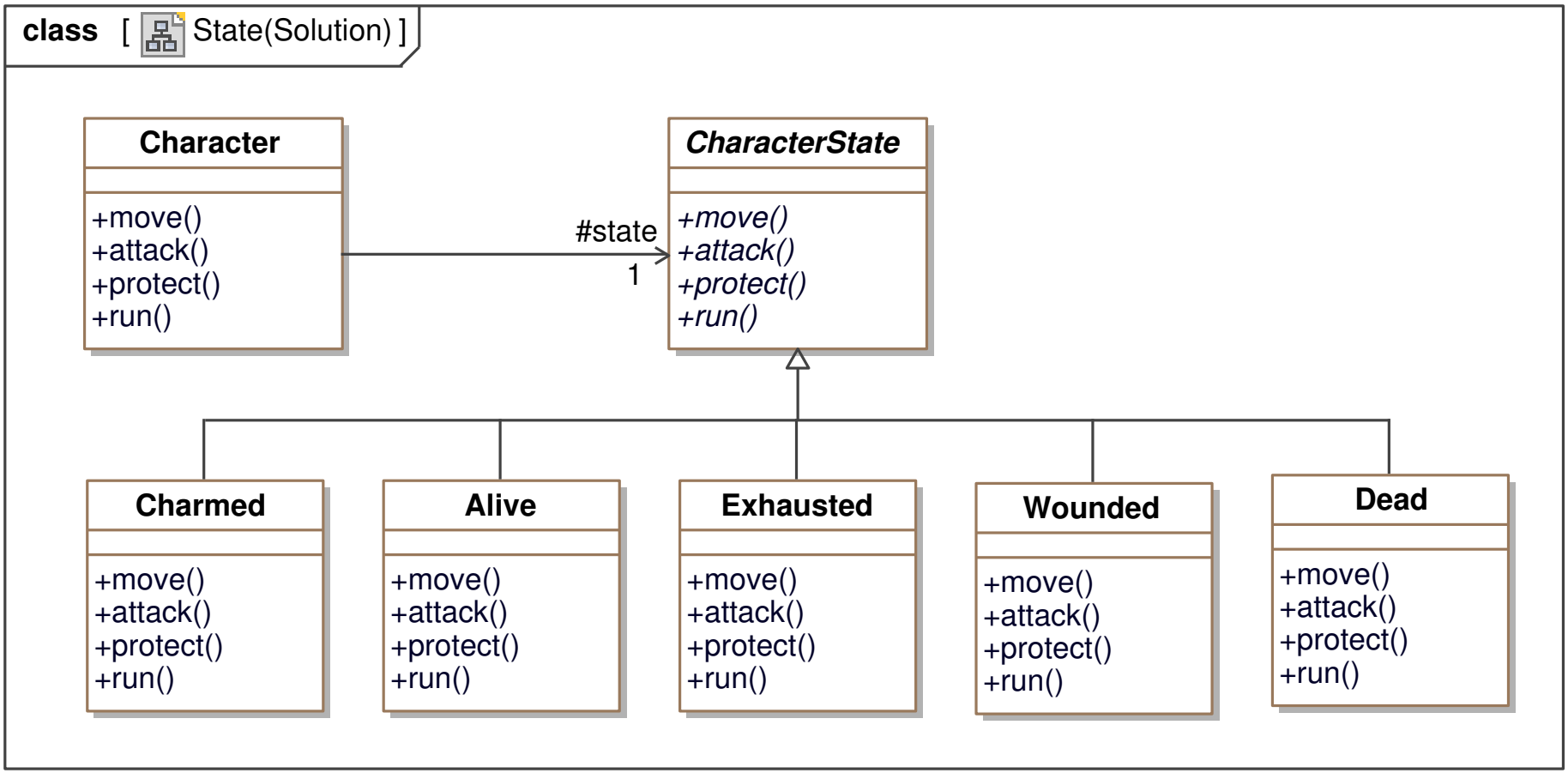
Problema

Un objeto necesita variar su comportamiento cuando cambia de estado.

Patrón Estado



Patrón Estado



Índice

Patrones de Diseño no GoF

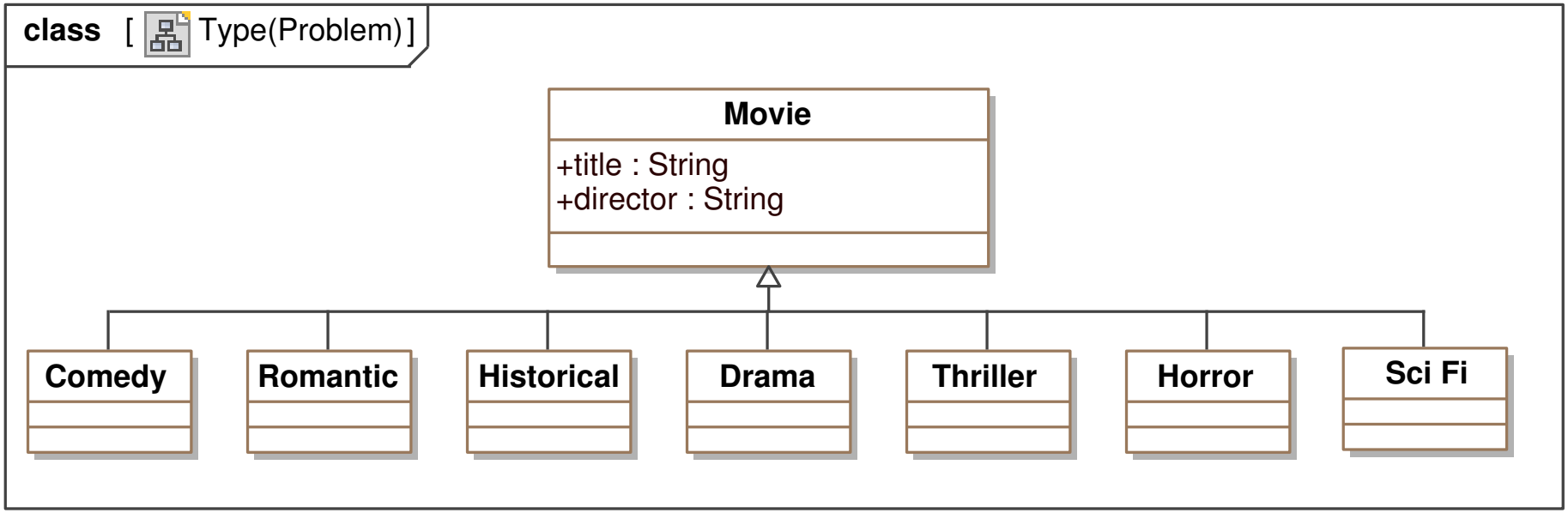
1. Patrón Tipo.
2. Patrón *Mixin*.

Patrón Tipo

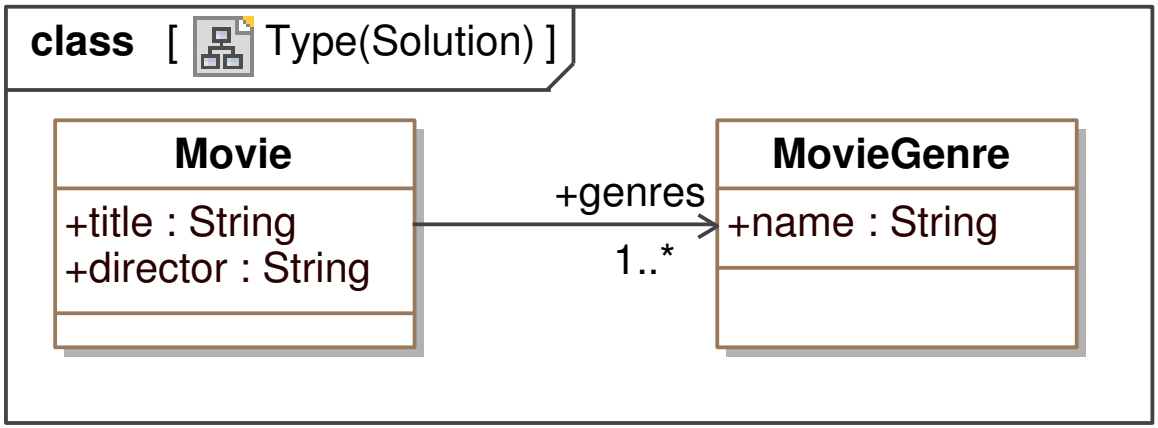
Aplicación

1. Permitir que un objeto pueda cambiar su(s) tipo(s) de forma dinámica en tiempo de ejecución.
2. Permitir que se puedan crear nuevos tipos de forma dinámica en tiempo de ejecución.

Patrón Tipo


class [Type(Problem)]
Movie
+title : String
+director : String
Comedy
Romantic
Historical
Drama
Thriller
Horror
Sci Fi

Patrón Tipo


class [Type(Solution)]
Movie
+title : String
+director : String
+genres
1..*
MovieGenre
+name : String

Patrones de Diseño no GoF

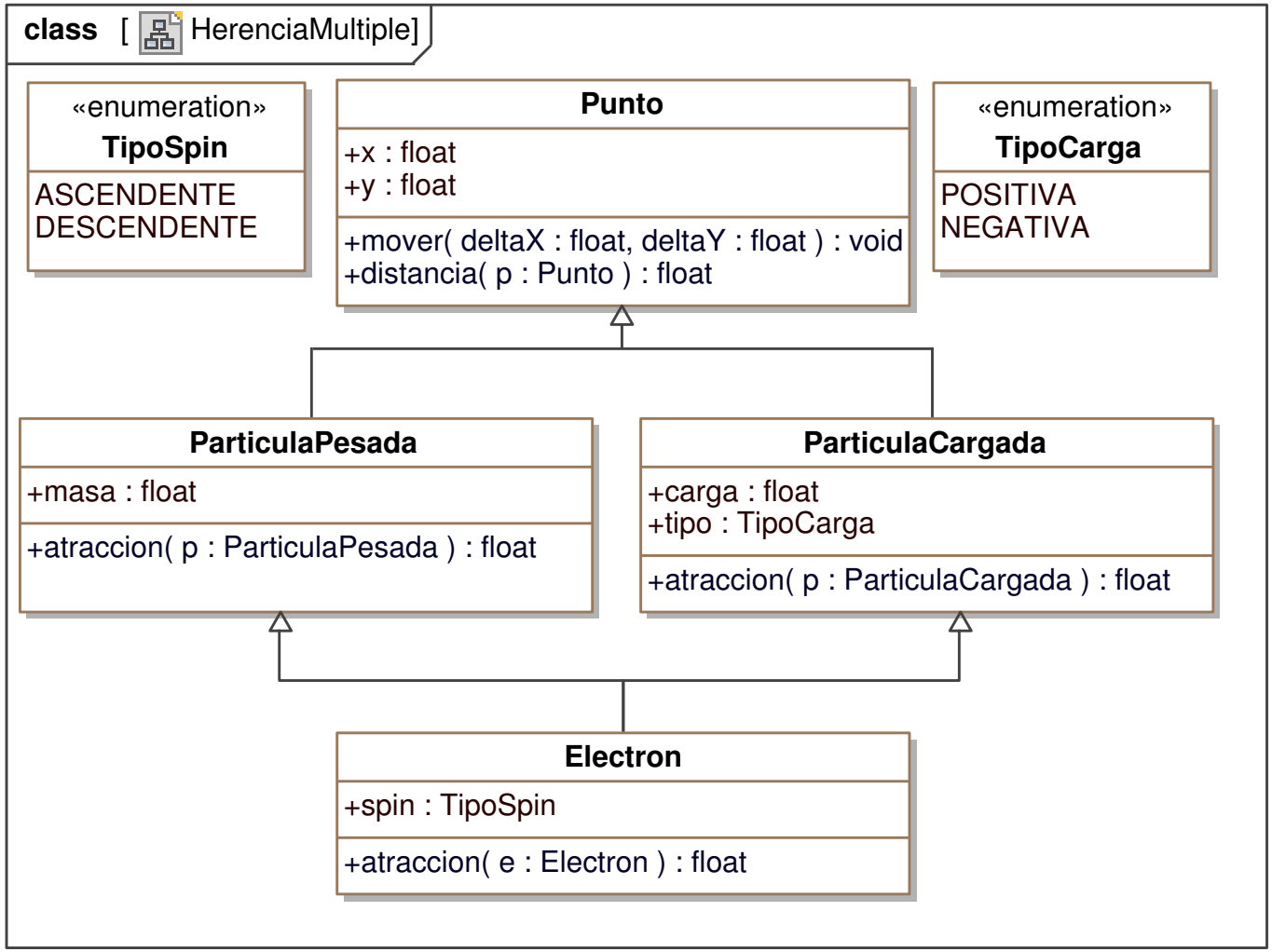
1. Patrón Tipo.
2. Patrón *Mixin*.

Patrón Mixin

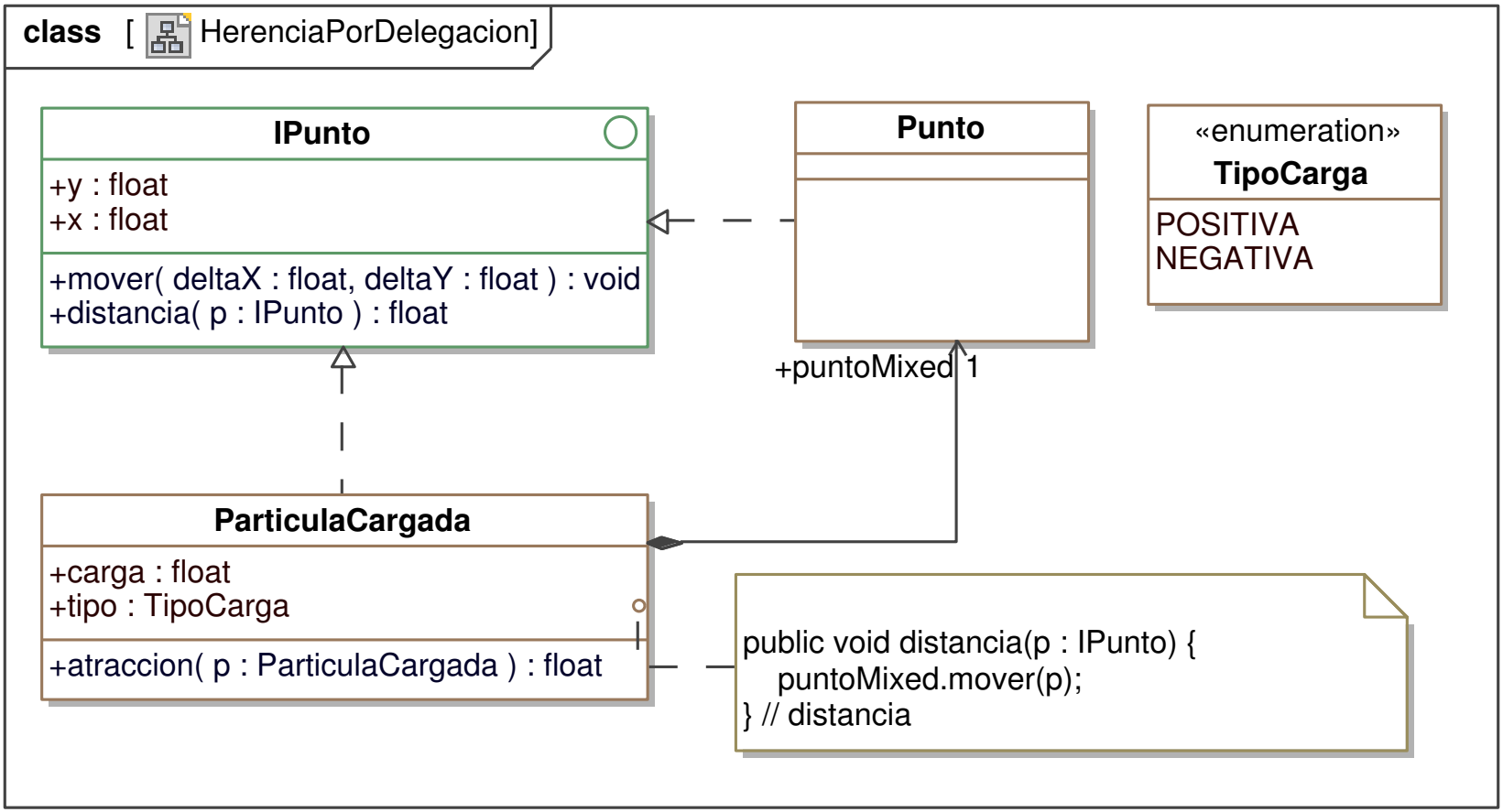
Aplicación

Permitir herencia múltiple en lenguajes que no permiten herencia múltiple de clases, pero si de interfaces.

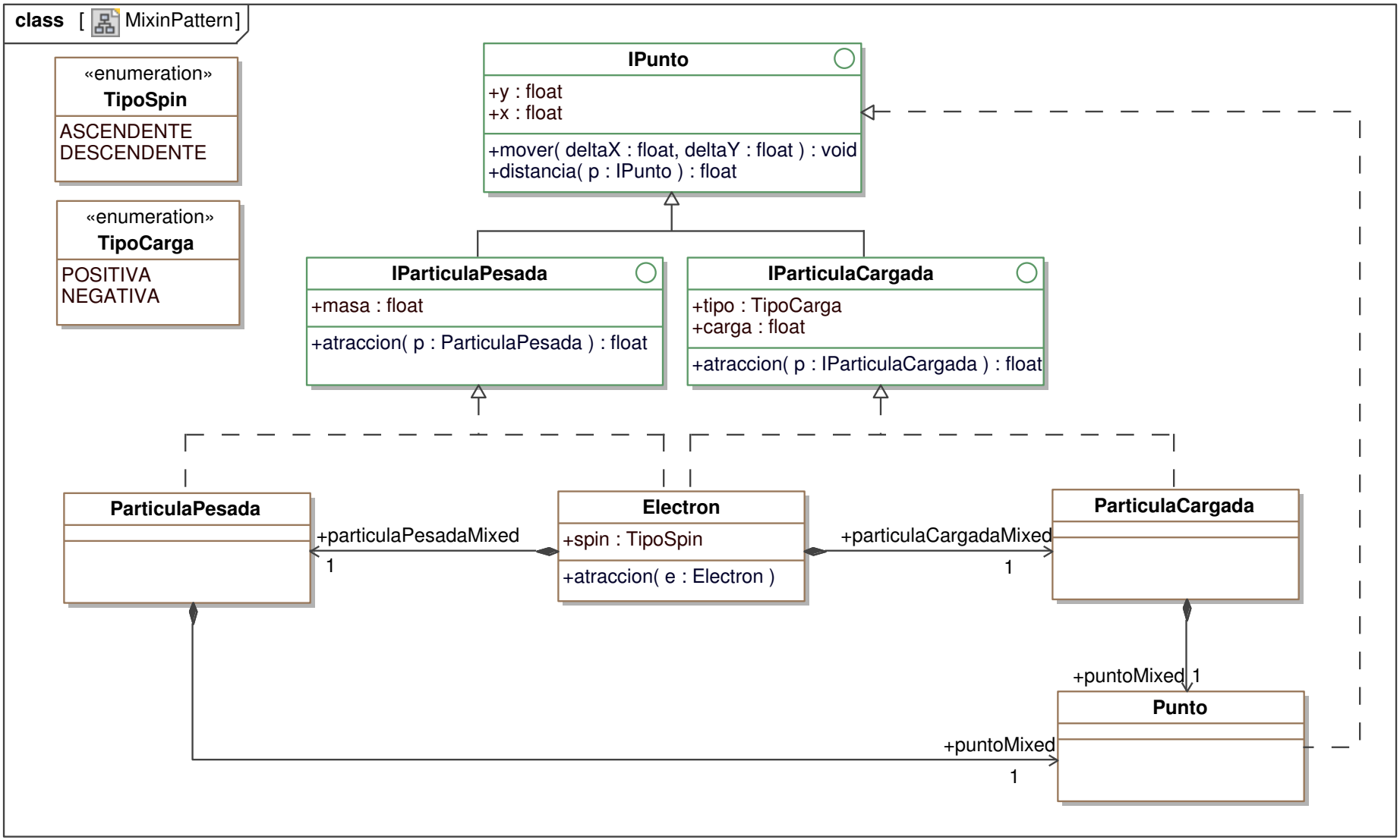
Patrón Mixin



Patrón Mixin



Patrón Mixin



Índice

Antipatrón de Diseño Software

Antipatrón de Diseño Software

Solución nefasta que sin embargo suele aplicarse recurrentemente a problemas típicos que aparecen durante el diseño de un producto software. (Ej. echar agua sobre aceite hirviendo (ver)).

Programación basada en Copiar y Pegar

Escenario

Un trozo de código cuasi-idéntico es copiado y pegado a lo largo de una clase o aplicación.

Problema

Una modificación en el trozo de código original implica la búsqueda y modificación de todos los trozos de código copiados.

Solución

Encapsulamiento y reutilización.

Singletonitis

Escenario

Un programador abusa del patrón *Singleton*, el cual usa principalmente para evitar tener que mantener referencias entre objetos o pasar parámetros.

Problemas

1. Todos los asociados con las variables globales.
2. No se pueden crear varias instancias de clases que no tienen porqué tener una instancia única.

Solución

No usar el patrón *singleton* cuando no tiene sentido usarlo.

Índice

Refactorizaciones

Refactorización

Proceso de cambio de un sistema software de forma que su comportamiento externo no se vea afectado pero que mejora su estructura interna.

Malos olores (bad smells)

Indicios sobre potenciales problemas en el código (ej. código replicado, mismo método en varias subclases).

Esquema de una Refactorización

1. Nombre
2. Síntomas
3. Causas
4. Refactorizaciones propuestas
5. Beneficios esperados
6. Efectos colaterales y contraindicaciones

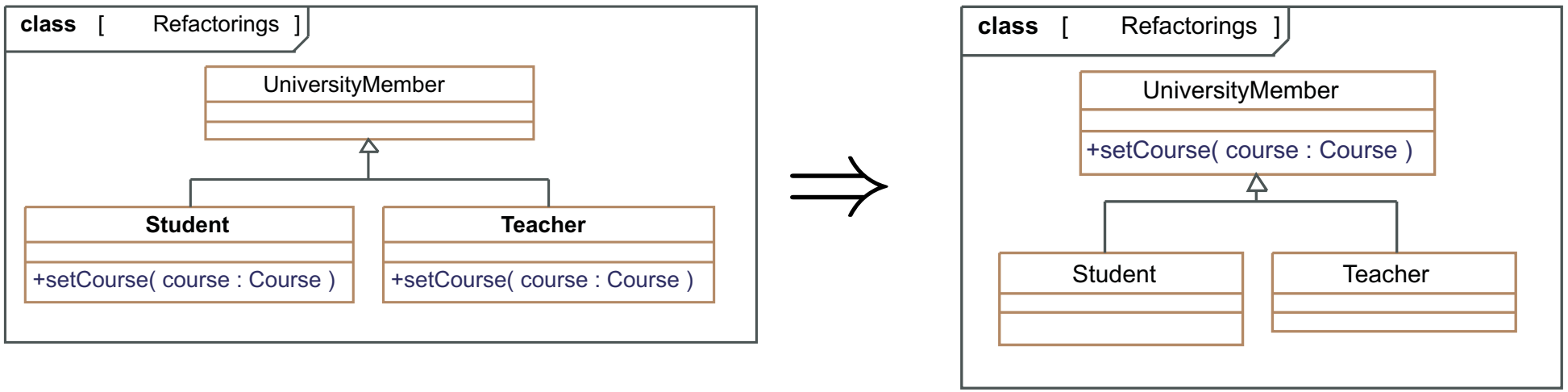
Refactorizaciones más Populares [Counsell et al., 2006]

1. Pull Up Method.
2. Move Method.
3. Introduce Parameter Object.
4. Move Field.
5. Rename Method/Field.
6. Replace Magic Number with Symbolic Constant.
7. Replace Type Code with State/Strategy.

Pull Up Method

Descripción

Existen métodos *cuasi*-idénticos en las subclases



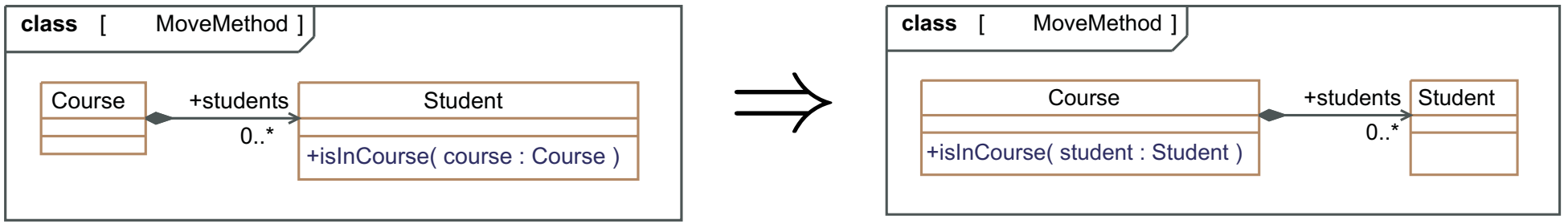
Refactorizaciones más Populares

1. Pull Up Method.
2. Move Method.
3. Introduce Parameter Object.
4. Move Field.
5. Rename Method/Field.
6. Replace Magic Number with Symbolic Constant.
7. Replace Type Code with State/Strategy.

Move Method

Descripción

Un método de una clase A es más utilizado en una clase B que en la clase A donde está definido.



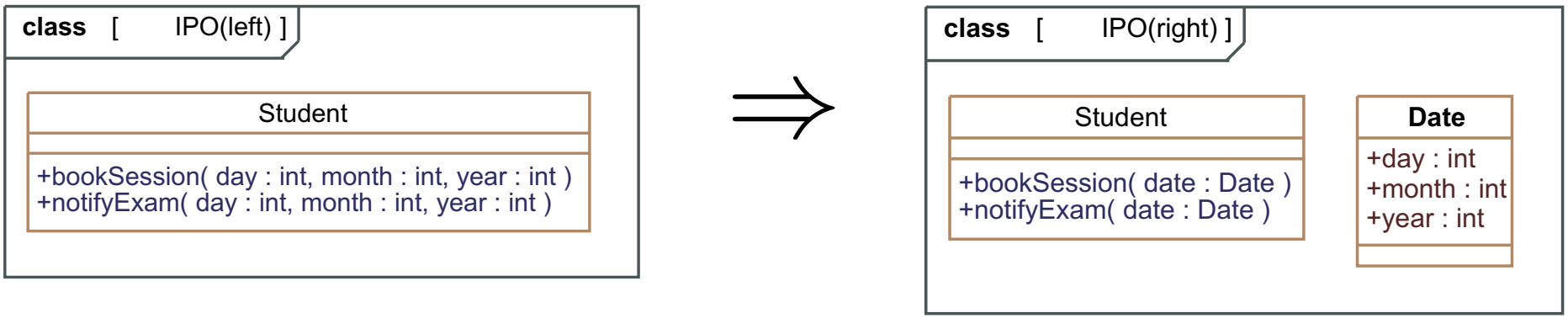
Refactorizaciones más Populares

1. Pull Up Method.
2. Move Method.
3. Introduce Parameter Object.
4. Move Field.
5. Rename Method/Field.
6. Replace Magic Number with Symbolic Constant.
7. Replace Type Code with State/Strategy.

Introduce Parameter Object

Descripción

Existen grupos de parámetros que están naturalmente relacionados



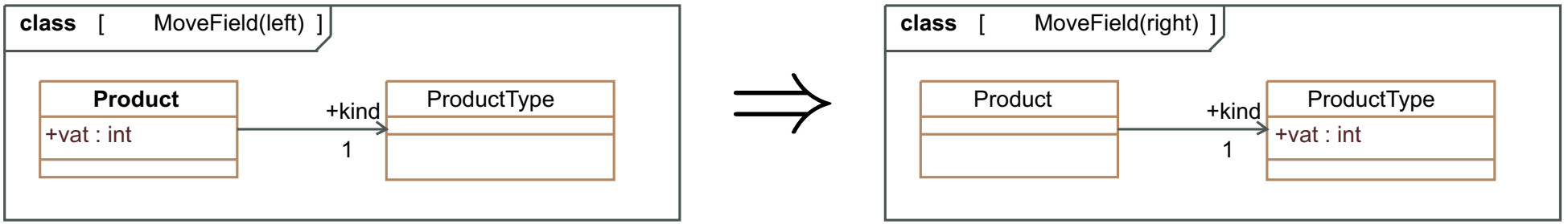
Refactorizaciones más Populares

1. Pull Up Method.
2. Move Method.
3. Introduce Parameter Object.
4. Move Field.
5. Rename Method/Field.
6. Replace Magic Number with Symbolic Constant.
7. Replace Type Code with State/Strategy.

Move Field

Descripción

Un atributo de una clase A es más utilizado en una clase B que en la clase A donde está definido.



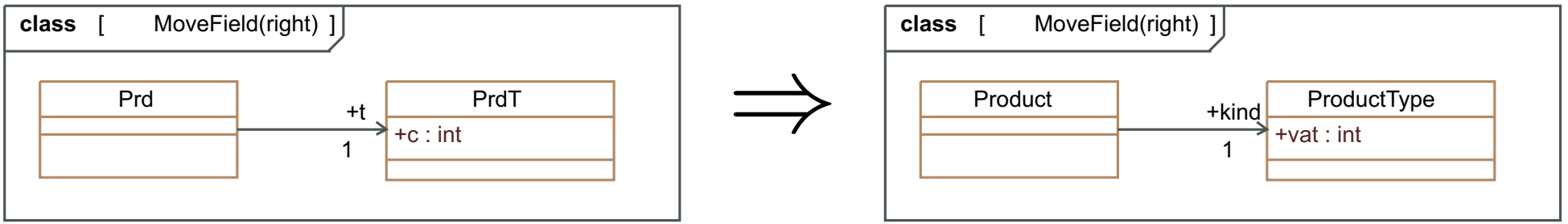
Refactorizaciones más Populares

1. Pull Up Method.
2. Move Method.
3. Introduce Parameter Object.
4. Move Field.
5. Rename Method/Field.
6. Replace Magic Number with Symbolic Constant.
7. Replace Type Code with State/Strategy

Rename Field/Method

Descripción

El nombre de un atributo o método no es significativo



Refactorizaciones más Populares

1. Pull Up Method.
2. Move Method.
3. Introduce Parameter Object.
4. Move Field.
5. Rename Method/Field.
6. Replace Magic Number with Symbolic Constant.
7. Replace Type Code with State/Strategy

Replace Magic Number with Symbolic Constant

Descripción

Tenemos una constante numérica con un significado bien definido

```
double calculateTotalCharge() {
   return this.totalAmount +
 (this.totalAmount*16.0/100.0);
}
```

refactors to

```
static final double NORMAL_VAT = 16.0;

double calculateTotalCharge() {
   return this.totalAmount +
 (this.totalAmount*NORMAL_VAT/100.0);
}
```

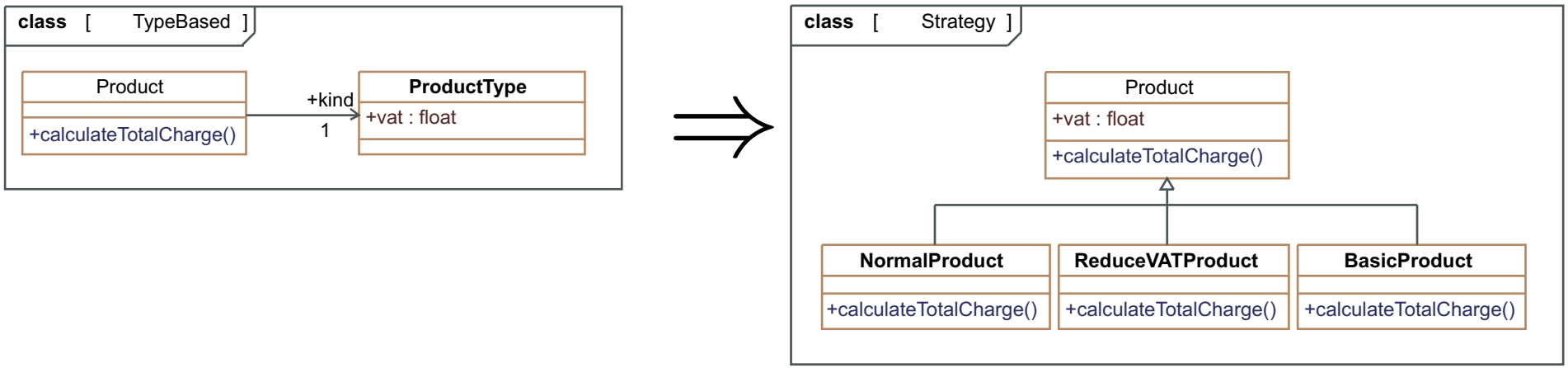
Refactorizaciones más Populares

1. Pull Up Method.
2. Move Method.
3. Introduce Parameter Object.
4. Move Field.
5. Rename Method/Field.
6. Replace Magic Number with Symbolic Constant.
7. Replace Type Code with State/Strategy.

Replace Type Code with State/Strategy

Descripción

Una clase tiene un atributo que indica tipo y que afecta al comportamiento de la clase



Índice

¿Qué tengo que saber de todo esto?

1. Comprender el concepto de patrón.
2. Saber aplicar el catálogo de patrones GoF.
3. Saber aplicar los patrones *tipo* y *mixin*.
4. Comprender el concepto de antipatrón.
5. Saber aplicar los antipatrones *programación basada en copiar y pegar* y *singletonitis*.
6. Comprender el concepto de refactorización.
7. Saber aplicar las refactorizaciones *pull up method*, *move method*, *add parameter*, *move field*, *rename method* y *rename field*.

Referencias

Counsell, S., Hassoun, Y., Loizou, G., and Najjar, R. (2006).
Common Refactorings, a Dependency Graph and Some Code Smells: an Empirical Study of Java OSS.
In Travassos, G. H., Maldonado, J. C., and Wohlin, C., editors, *Proc. of the Int. Symposium on Empirical Software Engineering (ISESE)*, pages 288–296, Rio de Janeiro (Brazil).