Tema 14 – Patrones de diseño

Ingeniería del Software

Rubén Fuentes Fernández
Dep. Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense Madrid

Trabajando con Antonio Navarro, Juan Pavón y Pablo Gervás

Contenidos

- Introducción
  - Patrones de diseño
- Patrones simples
  - Iterador, Método plantilla, Fachada
- Algunos patrones más complejos
  - Método factoría, Factoría abstracta, *Singleton*, *Flyweight*
- Operaciones sobre estructuras complejas
  - Composición, Intérprete, Visitante
- Patrones adicionales
  - Estrategia, Decorador, Constructor, *Proxy*, Adaptador, Puente, Mediador, Observador, Cadena de responsabilidades, Recuerdo, Orden, Prototipo, Estado

Patrón de diseño

- Christopher Alexander [Alexander et al., 1977]
  - *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"*
- GoF [Gamma et al., 2008] (primera edición de 1994)
  - *"A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design"*

Visiones de patrón de diseño

- Una solución estándar a un problema de programación
- Una técnica para hacer código más flexible mediante la aplicación de ciertos criterios
- Un diseño o estructura de implementación que logra un propósito particular
- Un vocabulario de programación de alto nivel
- Un atajo para describir ciertos aspectos de la organización de programas
- Relaciones entre componentes de programa
- La forma de un modelo o diagrama de objetos

Los 23+1 clásicos

Nivel	Creación	Estructura	Comportamiento
Clase	• Método factoría	• Adaptador (clase)	• Intérprete • Método plantilla
Objeto	• Factoría abstracta • Constructor • Prototipo • *Singleton*	• Adaptador (objeto) • Puente • Composición • Decorador • Fachada • *Flyweight* • *Proxy*	• Cadena de responsabilidades • Orden • Iterador • Mediador • Recuerdo • Observador • Estado • Estrategia • Visitante

Selección de patrones

- Anticiparse a las situaciones que puedan obligar a modificar el diseño.
- Decidir qué cambios puede ser deseable hacer en el futuro sin necesidad de modificar el diseño.
- Flexibilizar la fuente de cambios de las últimas iteraciones.

Fuentes de cambio y patrones (1/2)

- Creación explícita de objetos
  - Factoría abstracta, método factoría y prototipo
- Dependencia de operaciones concretas
  - Cadena de responsabilidades y comando
- Dependencia de la plataforma
  - Factoría abstracta y puente
- Dependencia de la representación o la implementación de los objetos
  - Factoría abstracta, puente, recuerdo y *proxy*

Fuentes de cambio y patrones (2/2)

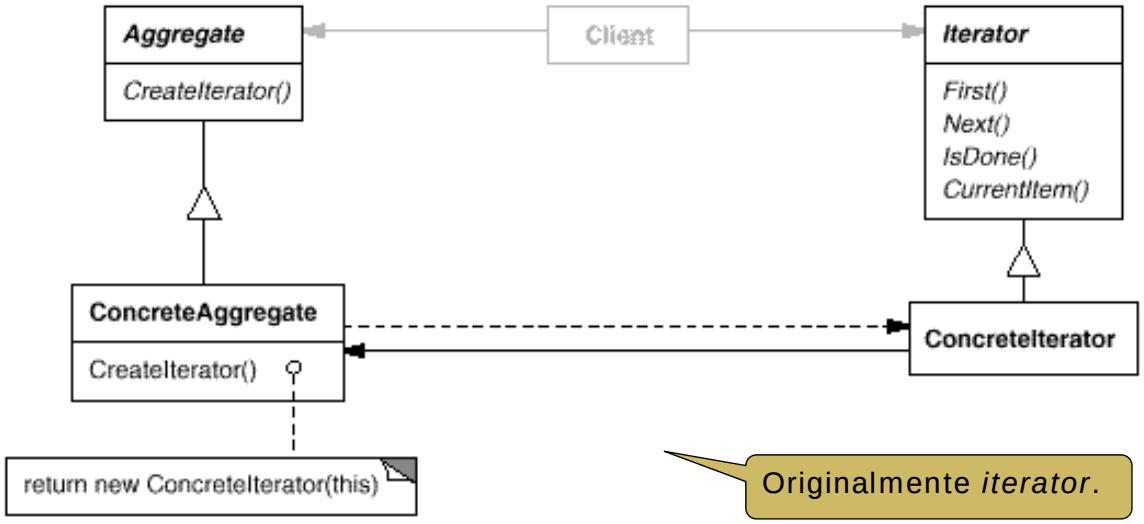
- Dependencia de los algoritmos
  - Puente, iterador, estrategia, método plantilla y visitante
- Acoplamiento
  - Factoría abstracta, puente, cadena de responsabilidades, comando, fachada, mediador y observador
- Extensión de la funcionalidad mediante subclases
  - Puente, cadena de responsabilidades, composición, decorador, observador y estrategia
- Imposibilidad de modificar las clases existentes
  - Adaptador, decorador y visitante

PATRONES SIMPLES

Patrones simples

ITERADOR

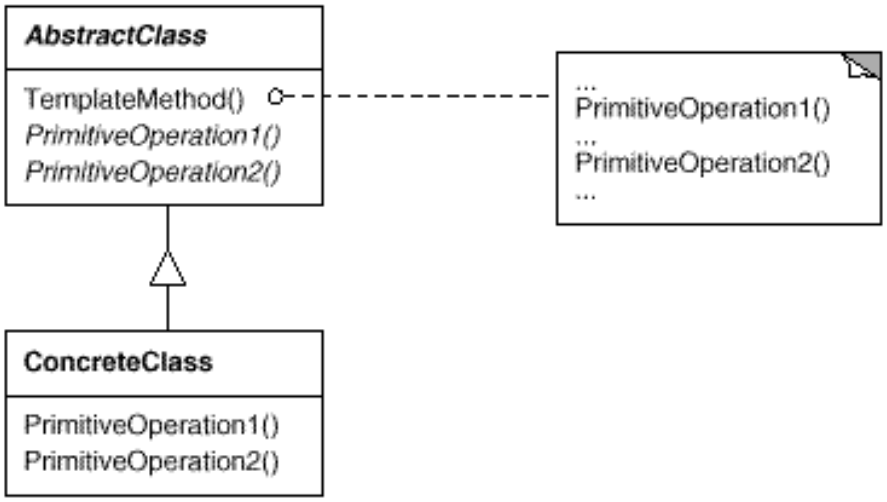
Iterador



Patrones simples

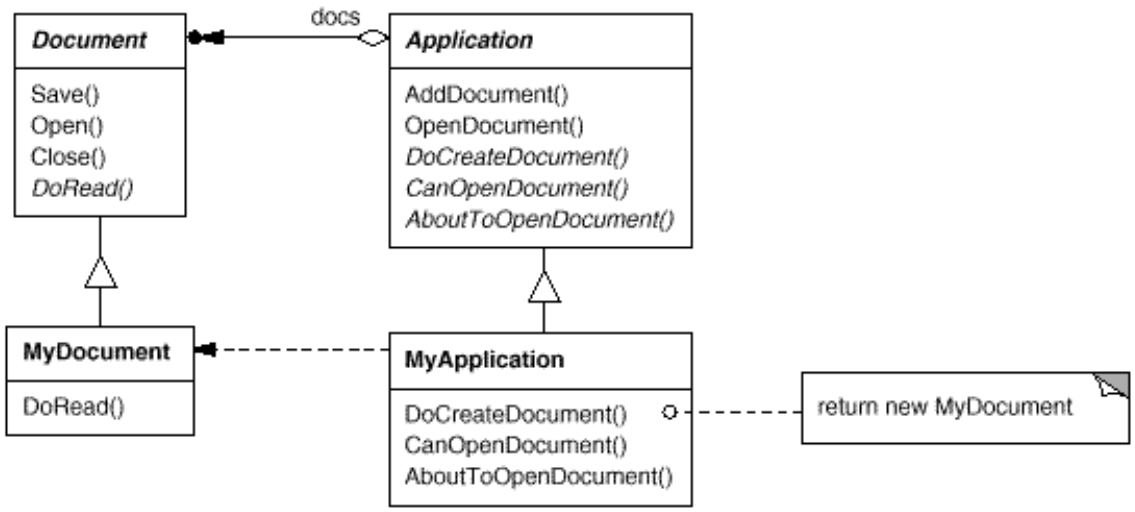
MÉTODO PLANTILLA

Método plantilla

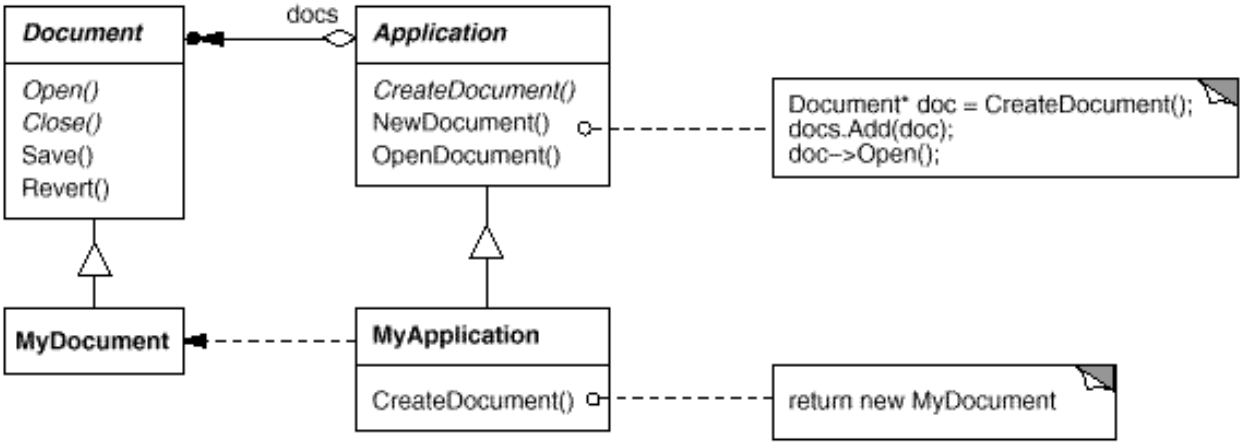


Originalmente *template method*.

Ejemplo: edición de documentos



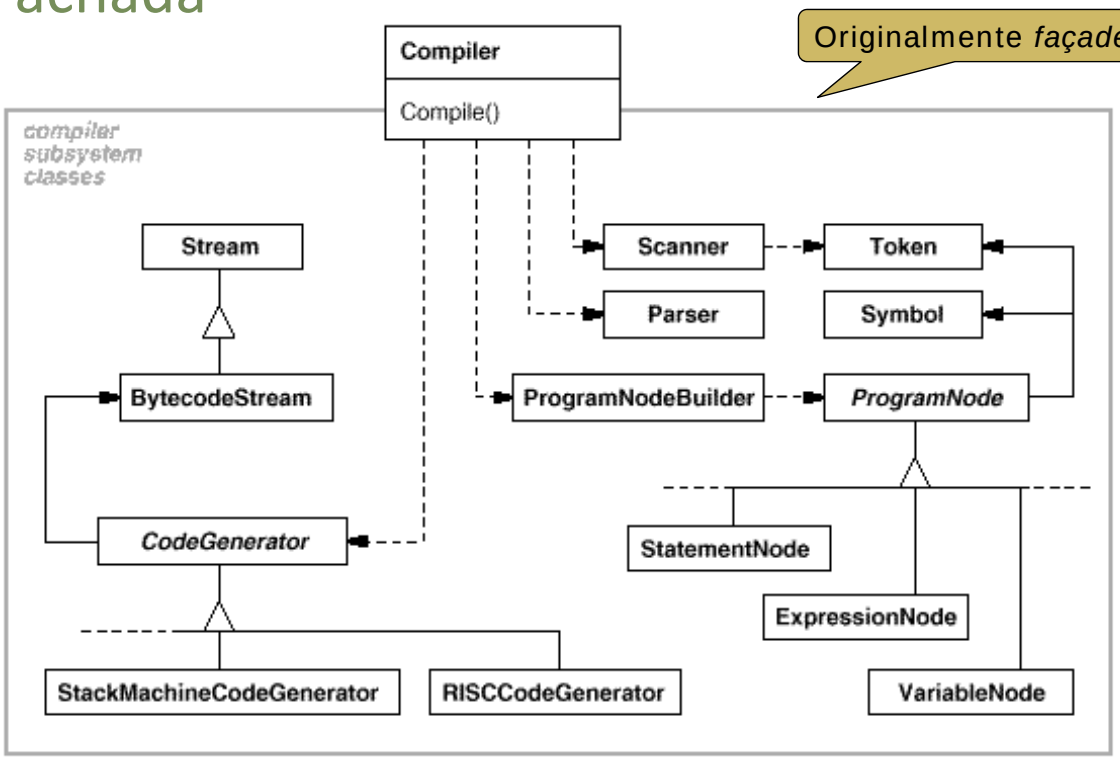
Ejemplo: creación de documentos



Patrones simples

FACHADA

Fachada



ALGUNOS PATRONES MÁS COMPLEJOS

Algunos patrones más complejos

MÉTODO FACTORÍA

Ejemplo: carreras de bicicletas

```
class Race {

  Race createRace() {
    Frame frame1 = new Frame();
    Wheel frontWheel1 = new Wheel();
    Wheel rearWheel1 = new Wheel();
    Bicycle bike1 = new Bicycle(frame1, frontWheel1, rearWheel1);
    Frame frame2 = new Frame();
    Wheel frontWheel2 = new Wheel();
    Wheel rearWheel2 = new Wheel();
    Bicycle bike2 = new Bicycle(frame2, frontWheel2, rearWheel2);
    ...
  }

}
```


Cómo particularizar

```
// French race
class TourDeFrance extends Race {

  Race createRace() {
    Frame frame1 = new RacingFrame();
    Wheel frontWheel1 = new Wheel700c();
    Wheel rearWheel1 = new Wheel700c();
    Bicycle bike1 = new Bicycle(frame1, frontWheel1, rearWheel1);
    Frame frame2 = new RacingFrame();
    Wheel frontWheel2 = new Wheel700c();
    Wheel rearWheel2 = new Wheel700c();
    Bicycle bike2 = new Bicycle(frame2, frontWheel2, rearWheel2);
    ...
  }

  ...
}
```

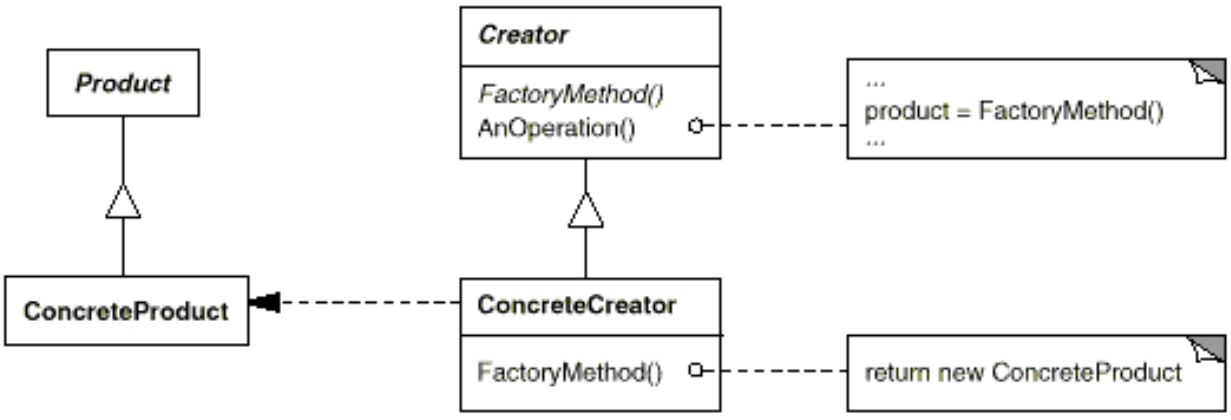

Una solución mejor

```
class Race {

  Frame createFrame() { return new Frame(); }
  Wheel createWheel() { return new Wheel(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new Bicycle(frame, front, rear);
  }
  // return a complete bicycle without needing any arguments
  Bicycle completeBicycle() {
    Frame frame = createFrame();
    Wheel frontWheel = createWheel();
    Wheel rearWheel = createWheel();
    return createBicycle(frame, frontWheel, rearWheel);
  }

  Race createRace() {
    Bicycle bike1 = completeBicycle();
    Bicycle bike2 = completeBicycle();
    ...
  }
}
```


...que permite reutilizar trozos

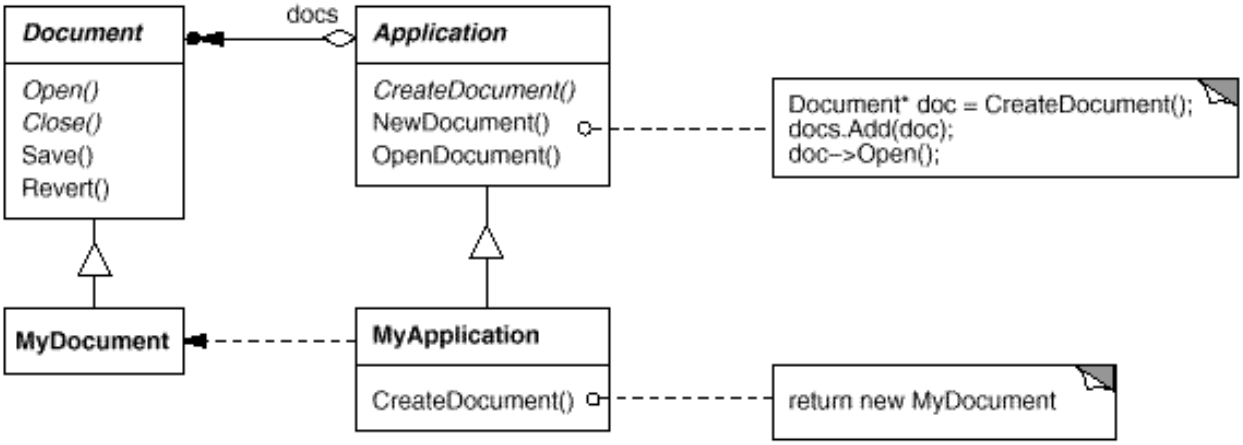
```
// French race
class TourDeFrance extends Race {

  Frame createFrame() { return new RacingFrame(); }
  Wheel createWheel() { return new Wheel700c(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }

}

class Cyclocross extends Race {

  Frame createFrame() { return new MountainFrame(); }
  Wheel createWheel() { return new Wheel26inch(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }

}
```


Método factoría



Originalmente *factory method*.

Ejemplo: creación de documentos



Algunos patrones más complejos

FACTORÍA ABSTRACTA

Encapsular métodos factoría

```
class BicycleFactory {
  Frame createFrame() { return new Frame(); }
  Wheel createWheel() { return new Wheel(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new Bicycle(frame, front, rear);
  }

  // return a complete bicycle without needing any arguments
  Bicycle completeBicycle() {
    Frame frame = createFrame();
    Wheel frontWheel = createWheel();
    Wheel rearWheel = createWheel();
    return createBicycle(frame, frontWheel, rearWheel);
  }
}
```


...que se pueden instanciar

```
class RacingBicycleFactory {
  Frame createFrame() { return new RacingFrame(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }
}

class MountainBicycleFactory {
  Frame createFrame() { return new MountainFrame(); }
  Wheel createWheel() { return new Wheel26inch(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }
}
```

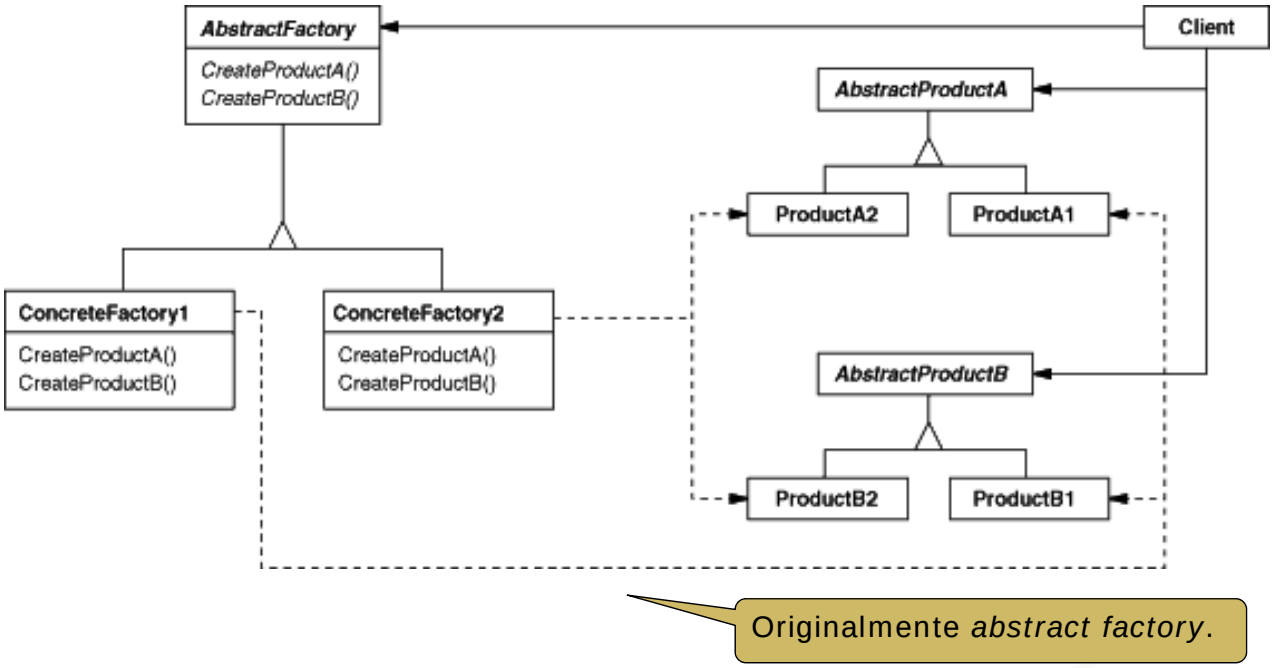

… y reutilizar

```
class Race {

  BicycleFactory bfactory;

  // constructor
  Race() {
    bfactory = new BicycleFactory();
  }

  Race createRace() {
    Bicycle bike1 = bfactory.completeBicycle();
    Bicycle bike2 = bfactory.completeBicycle();
    ...
  }

}

class TourDeFrance extends Race {
  // constructor
  TourDeFrance() {
    bfactory = new RacingBicycleFactory();
  }
}
```

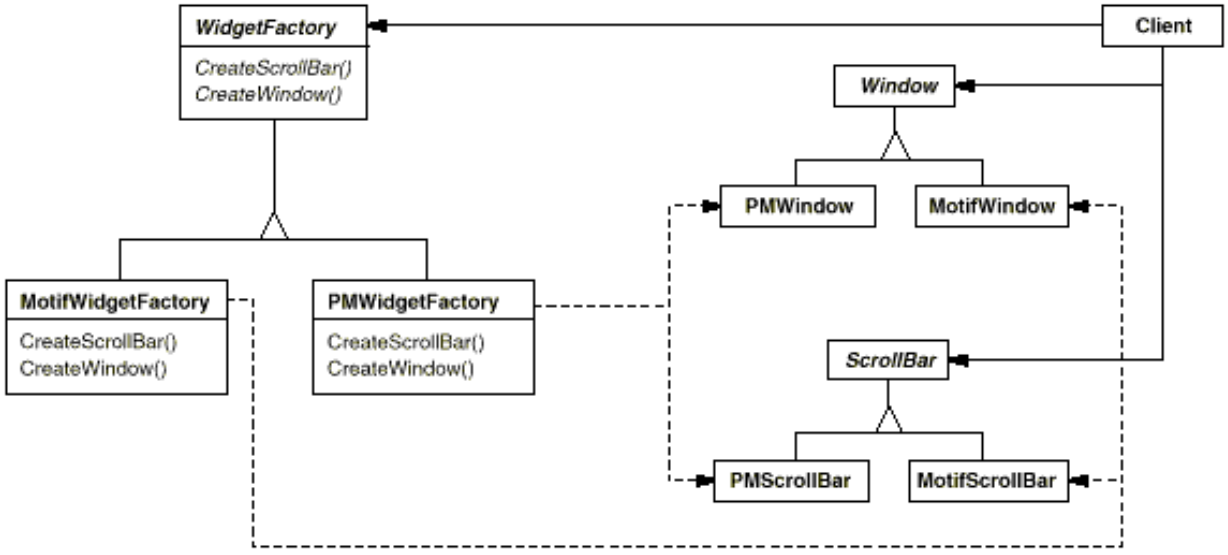

… y hacer más flexibles

```
class Race {

  BicycleFactory bfactory;

  // constructor
  Race(BicycleFactory bfactory) {
    this.bfactory = bfactory;
  }

  Race createRace() {
    Bicycle bike1 = bfactory.completeBicycle();
    Bicycle bike2 = bfactory.completeBicycle();
    ...
  }

}

class TourDeFrance extends Race {
  // constructor
  TourDeFrance(BicycleFactory bfactory) {
    this.bfactory = bfactory;
  }
}
```


Factoría abstracta



Ejemplo: sistema de ventanas



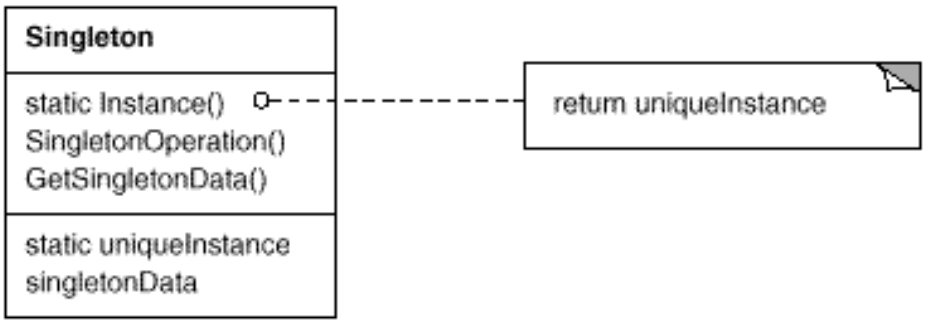
Algunos patrones más complejos

SINGLETON

Cuando una copia basta

```
class Gym {
  private static Gym theGym;
  // constructor
  private Gym() { ... }
  // factory method
  public static getGym() {
    if (theGym == null) {
      theGym = new Gym();
    }
    return theGym;
  }

  ...
}
```

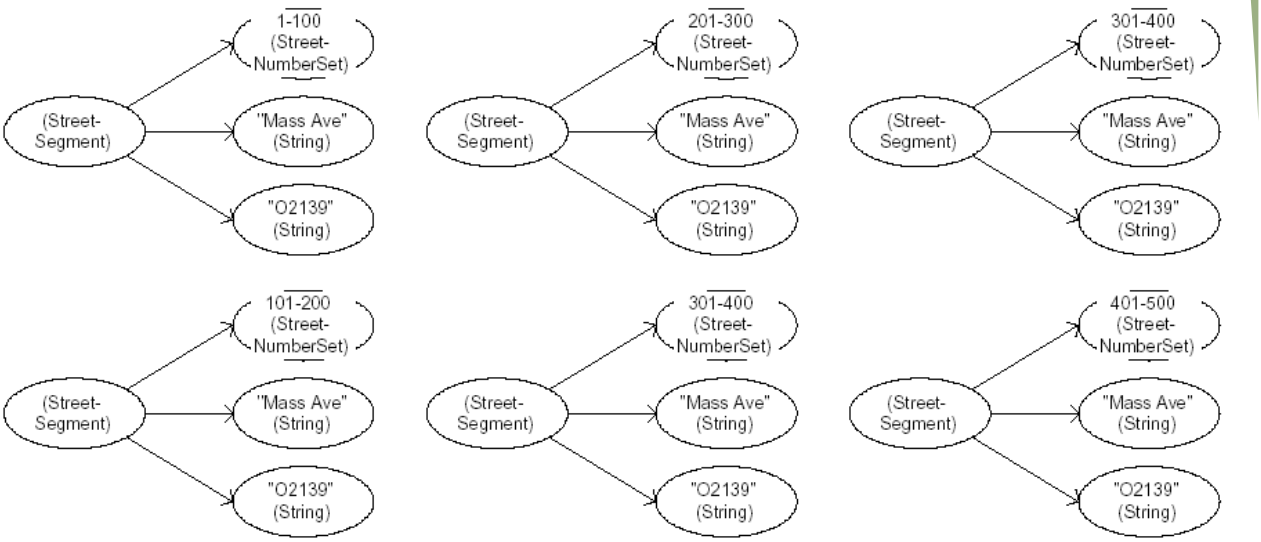

Singleton



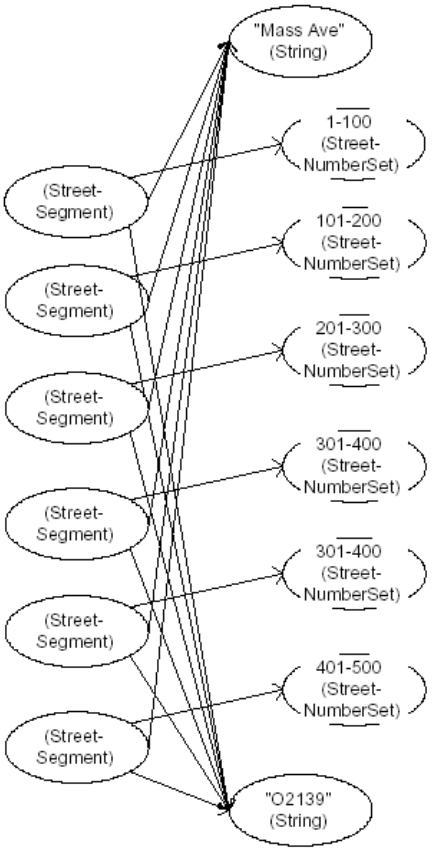
Algunos patrones más complejos

FLYWEIGHT

Ejemplo: representando una calle



Una representación más eficiente



El código para conseguirlo

```
HashMap segnames = new HashMap();

canonicalName(String n) {
  if (segnames.containsKey(n)) {
    return segnames.get(n);
  } else {
    segnames.put(n, n);
    return n;
  }
}
```


Ejemplo: ruedas de bicicleta

```
class Wheel {
  ...
  FullSpoke[] spokes;
  ...
}

// We'll name a trimmed-down version "Spoke", so call this "FullSpoke"
class FullSpoke {
  int length;
  int diameter;
  boolean tapered;
  Metal material;
  float weight;
  float threading;
  boolean crimped;
  int location;     // which rim and hub holes this is installed in
}
```


Ahorrando espacio

Se separa el estado en:

intrínseco...

```
class Spoke {
  int length;
  int diameter;
  boolean tapered;
  Metal material;
  float weight;
  float threading;
  boolean crimped;
}
```

y extrínseco
("location" viene dado
por el índice en el array)

```
class Wheel {
  ...
  Spoke[] spokes;
  ...
}
```


Pero las operaciones...

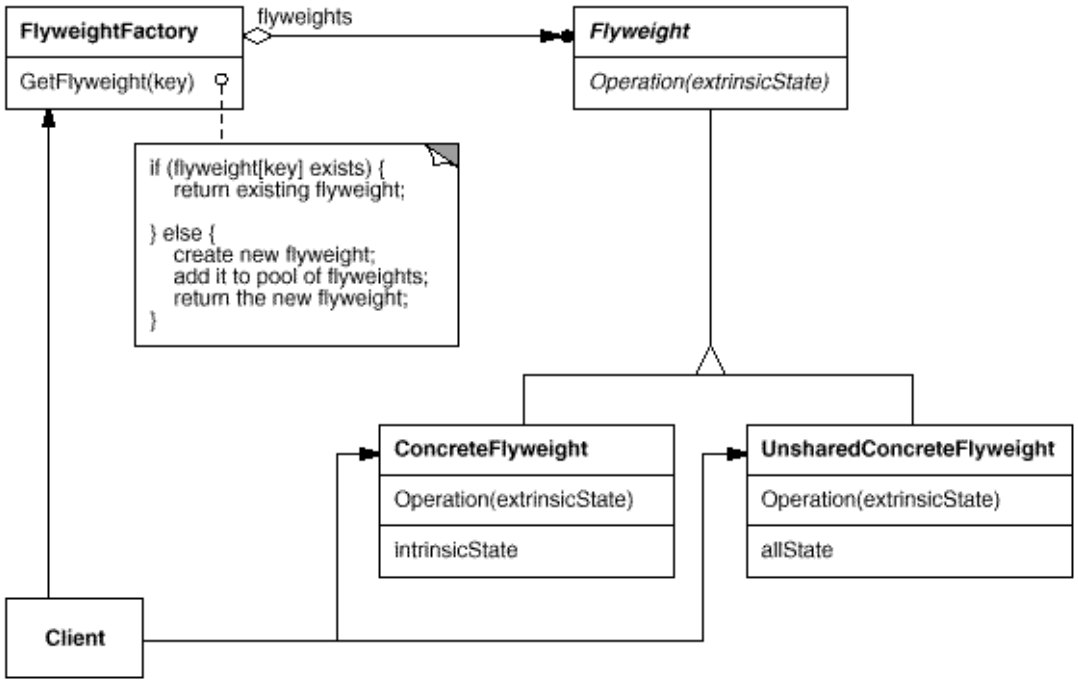
...que antes no necesitaban el estado extrínseco...

```
class FullSpoke {
  // tension the spoke by turning the nipple the specified number of turns
  void tighten(int turns) {
    ... location ...
  }
}
```

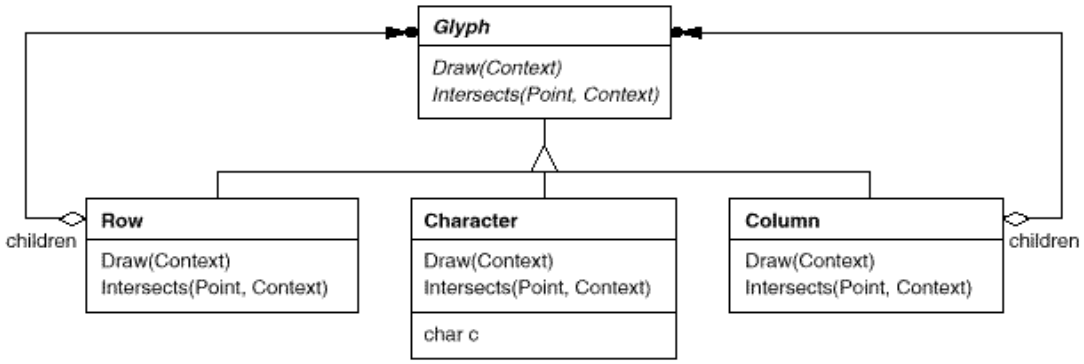
...ahora necesitan recibirlo como parámetro.

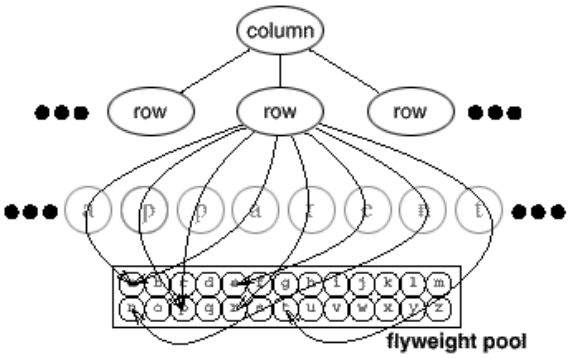
```
class Spoke {
  void tighten(int turns, int location) {
    ... location ...
  }
}
```

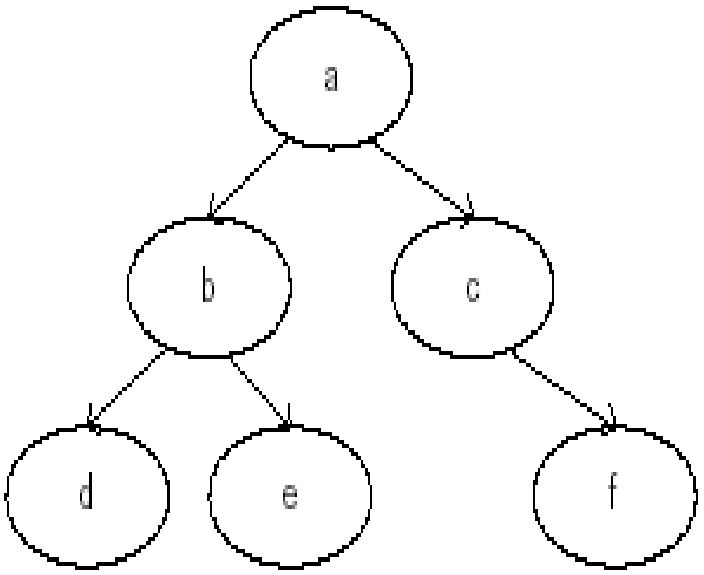

Flyweight



Ejemplo: editor de texto





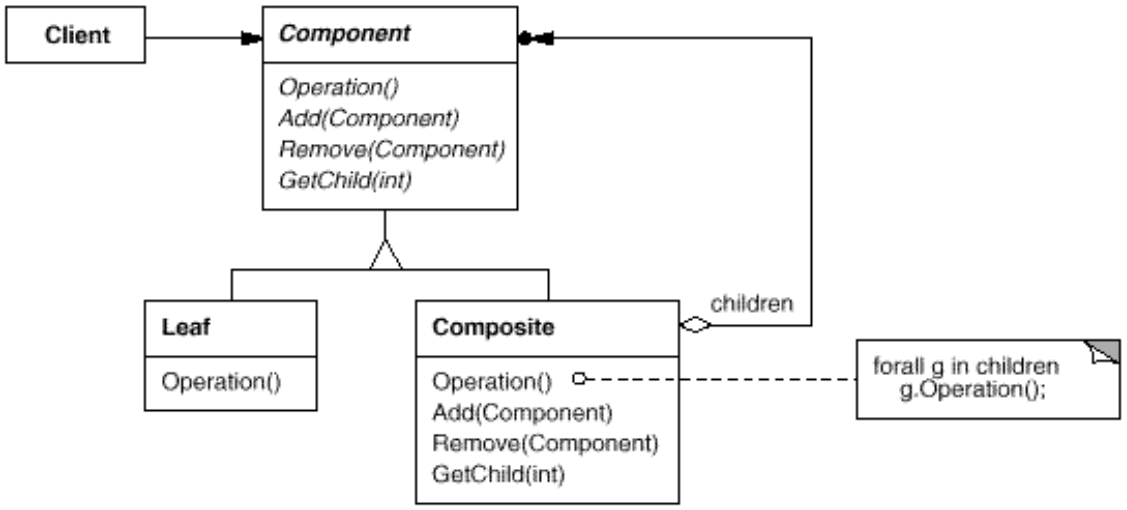


OPERACIONES SOBRE ESTRUCTURAS

Operaciones sobre estructuras

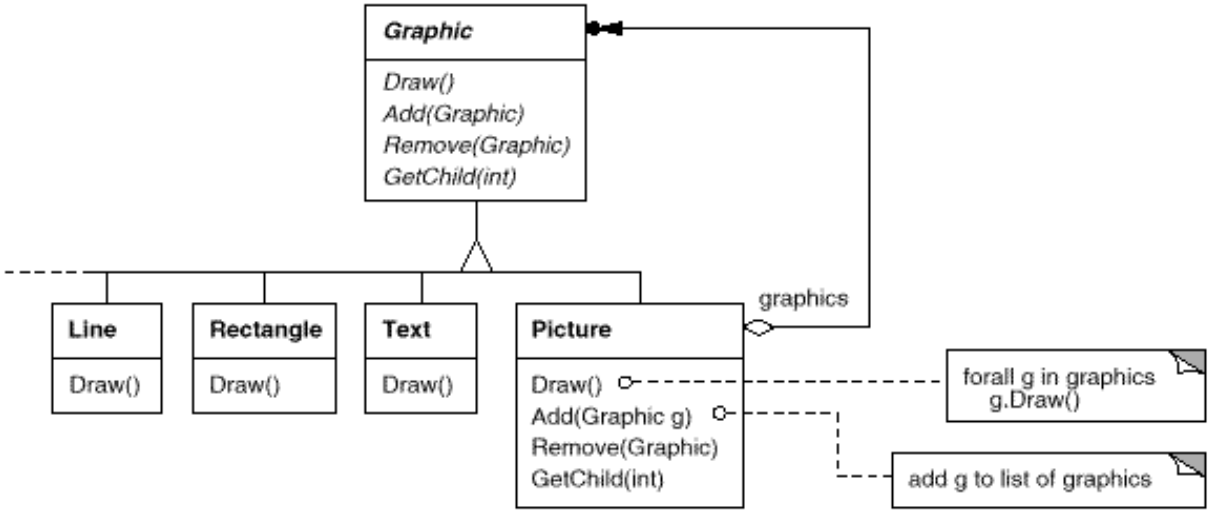
COMPOSICIÓN

Composición



Originalmente *composite*.

Ejemplo: dibujo



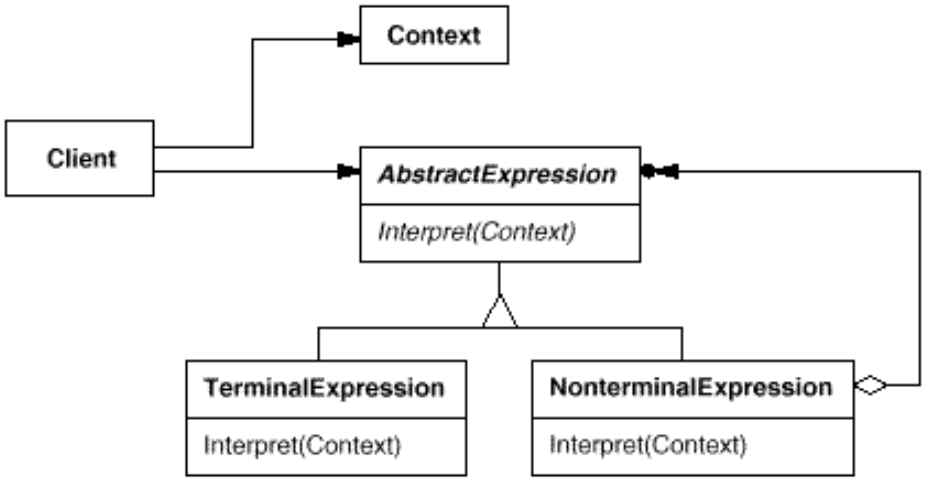
Operaciones sobre estructuras

INTÉRPRETE

Agrupa operaciones de un tipo de objeto

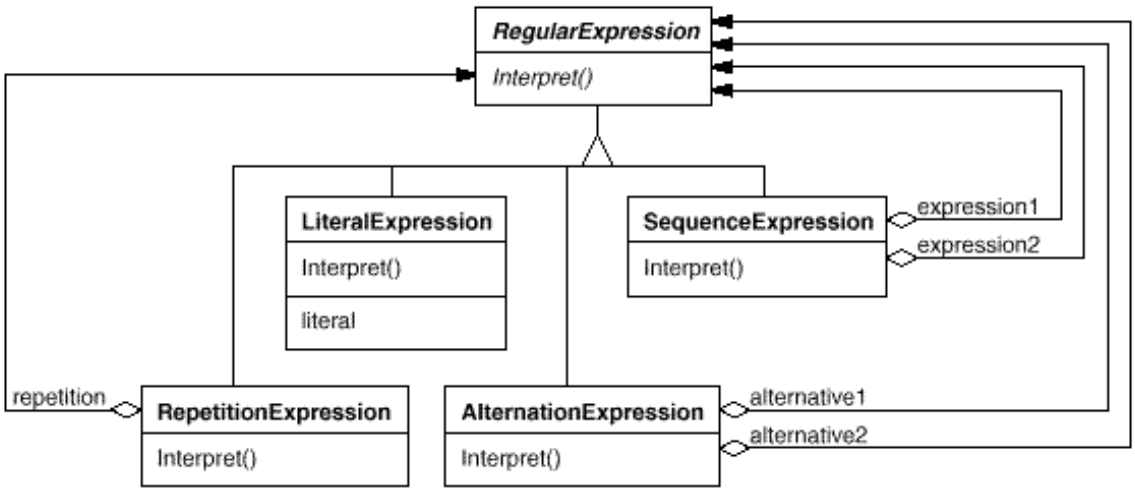
```
class Expression {
  ...
  Type typecheck();
  String prettyPrint();
}

...

class AssignOp extends Expression {
  ...
  Type typecheck() { ... }
  String prettyPrint() { ... }
}

class CondExpr extends Expression {
  ...
  Type typecheck() { ... }
  String prettyPrint() { ... }
}
```


Intérprete



Originalmente *interpreter*.

Ejemplo: compilador



Operaciones sobre estructuras

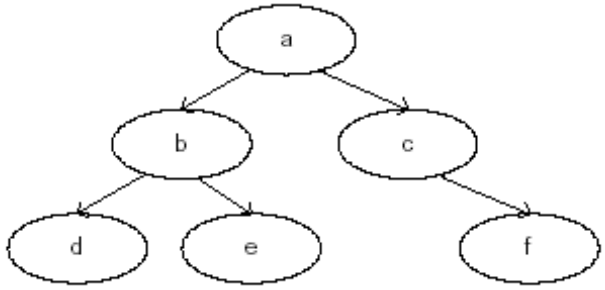
VISITANTE

Desacoplar operación y estructura

```
class Node {
  ...
  void accept(Visitor v) {
    for each child of this node {
      child.accept(v);
    }
    v.visit(this);
  }
}

class Visitor {
  ...
  void visit(Node n) {
    perform work on n
  }
}
```

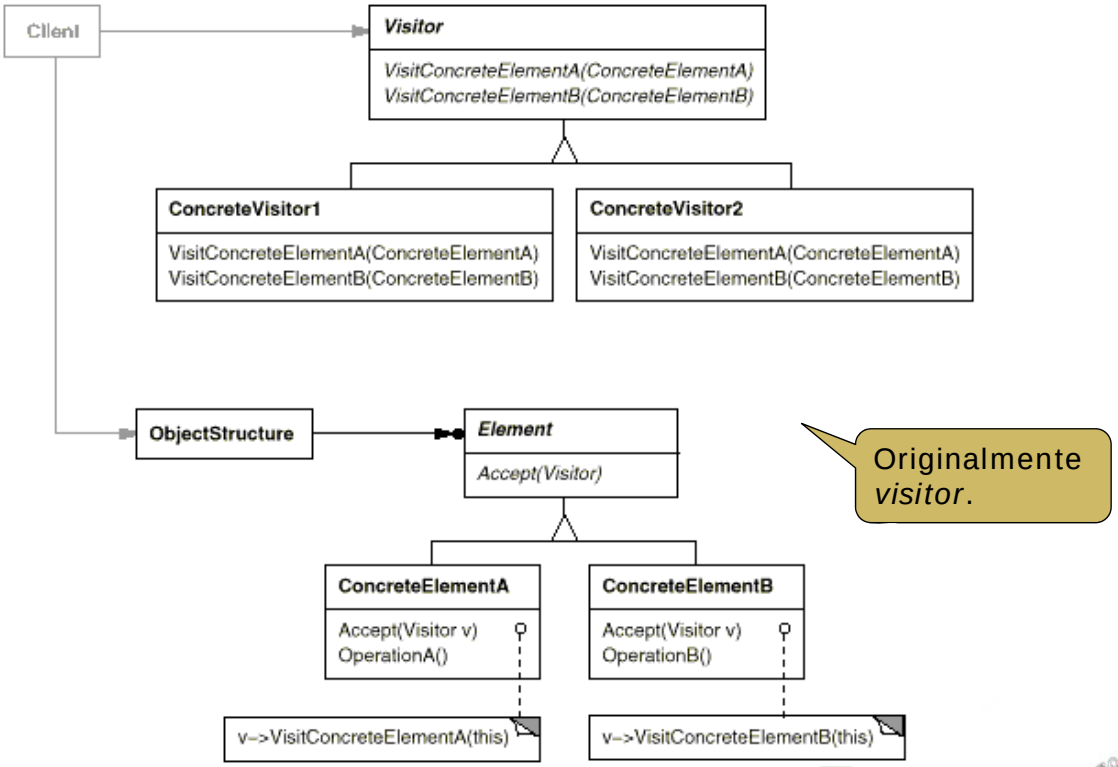

Visitar una estructura compuesta

a.accept(v) for some visitor v

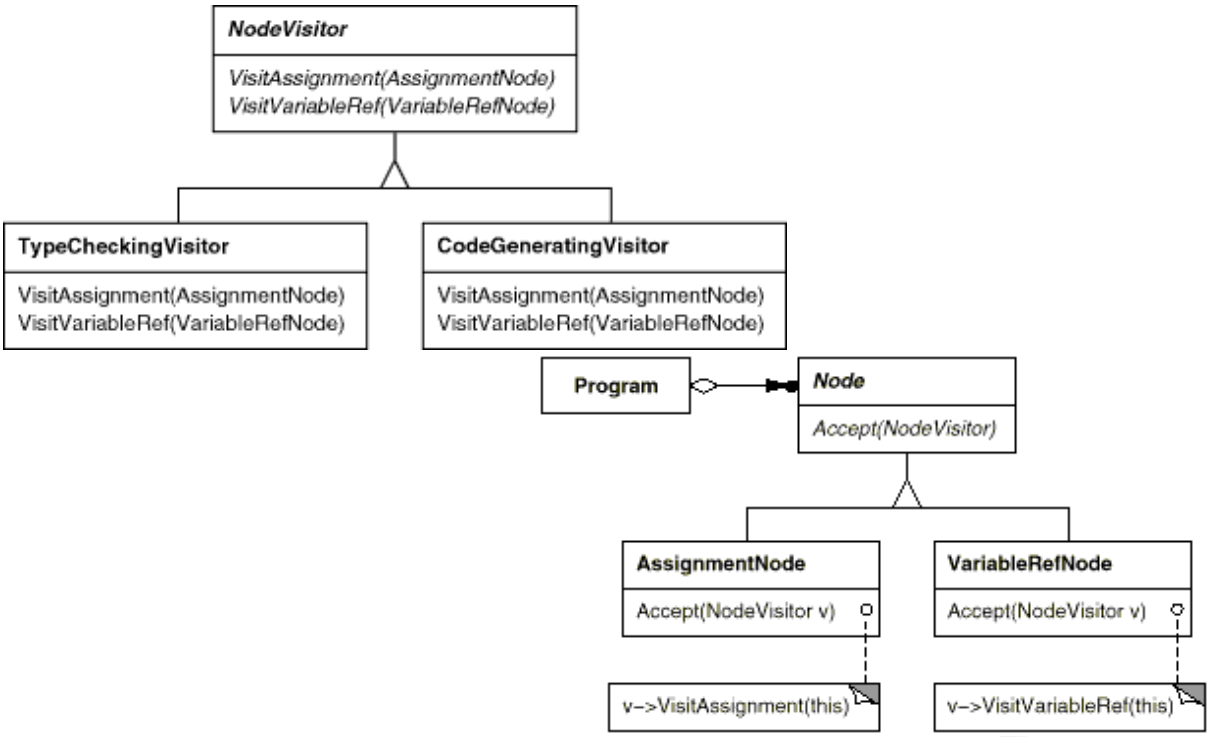



```
a.accept(v)
  b.accept(v)
    d.accept(v)
      v.visit(d)
    e.accept(v)
      v.visit(e)
    v.visit(b)
  c.accept(v)
    f.accept(v)
      v.visit(f)
    v.visit(c)
  v.visit(a)
```


Visitante



Ejemplo: compilador

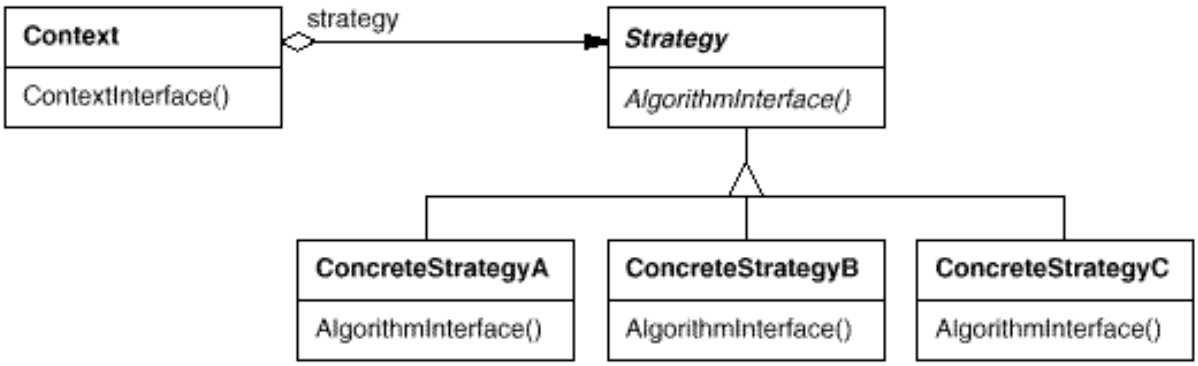


PATRONES ADICIONALES

Patrones adicionales

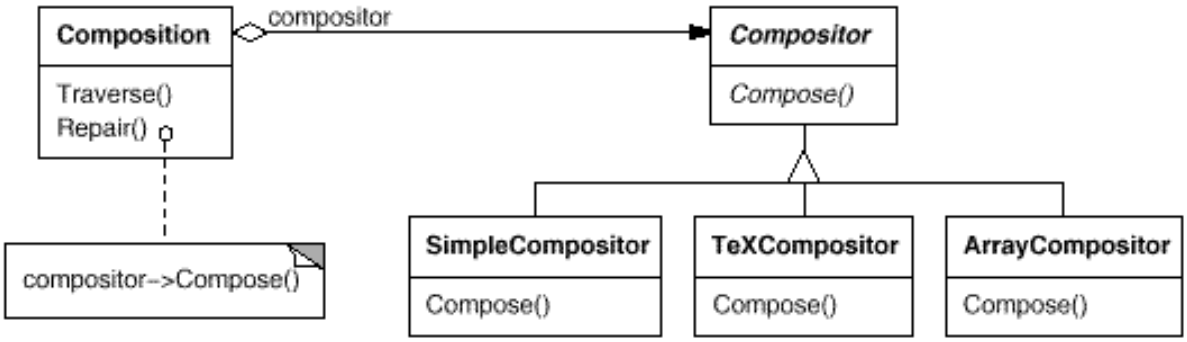
ESTRATEGIA

Estrategia



Originalmente *strategy*.

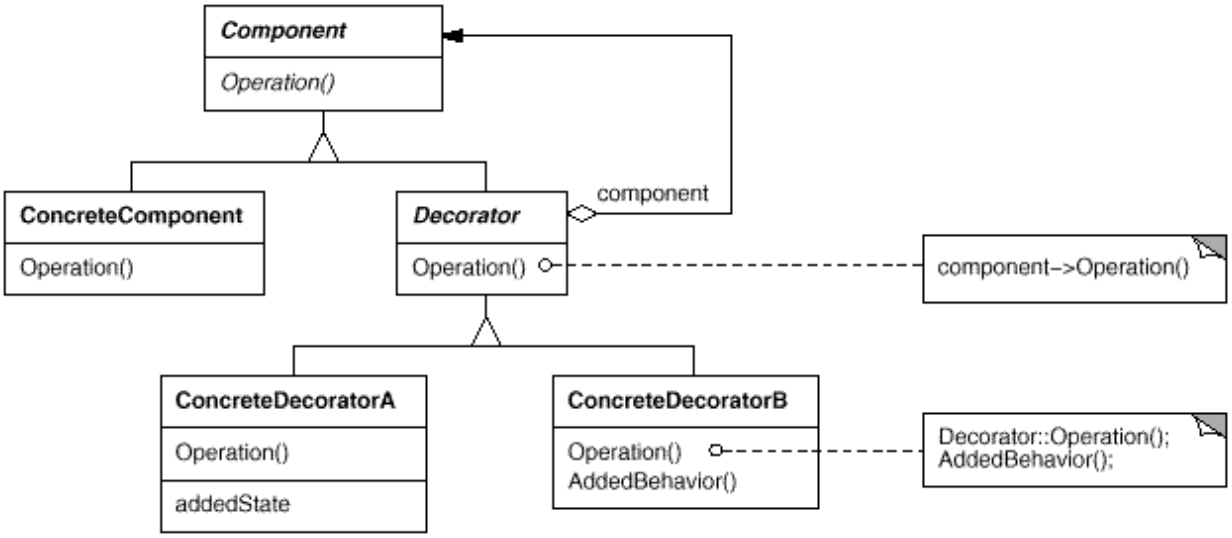
Ejemplo: editor y representación de estructuras



Patrones adicionales

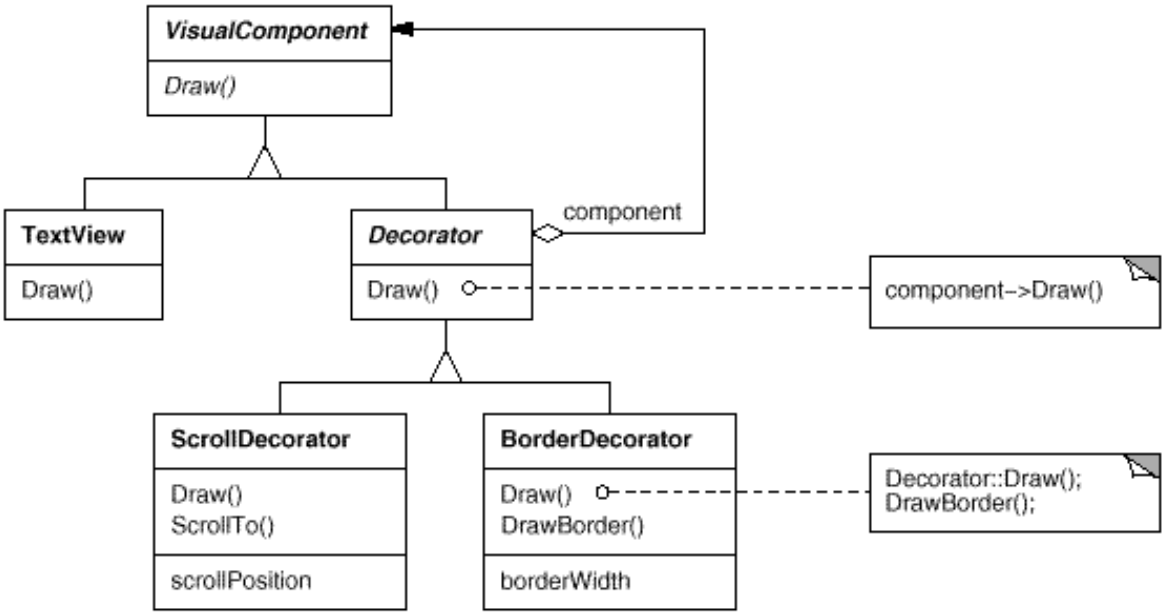
DECORADOR

Decorador



Originalmente *decorator*.

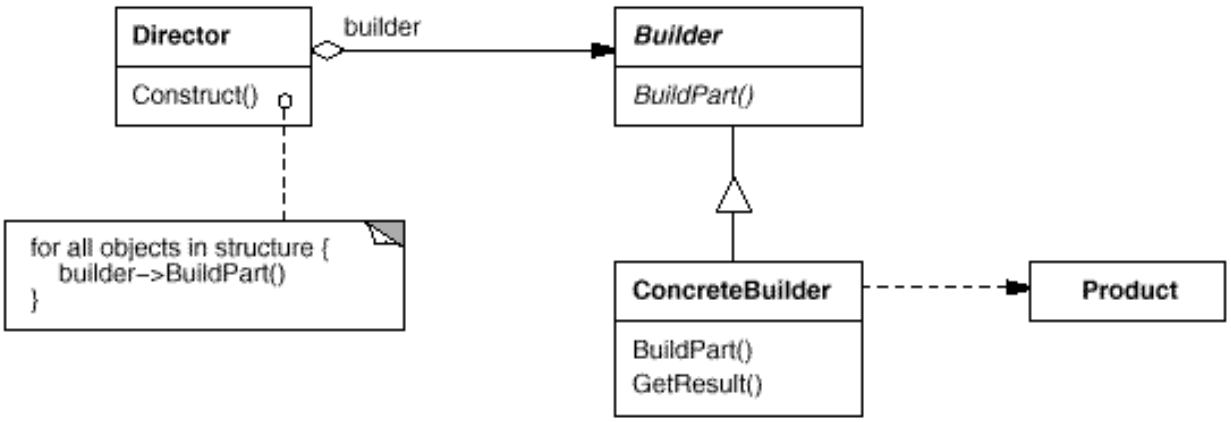
Ejemplo: componentes gráficos



Patrones adicionales

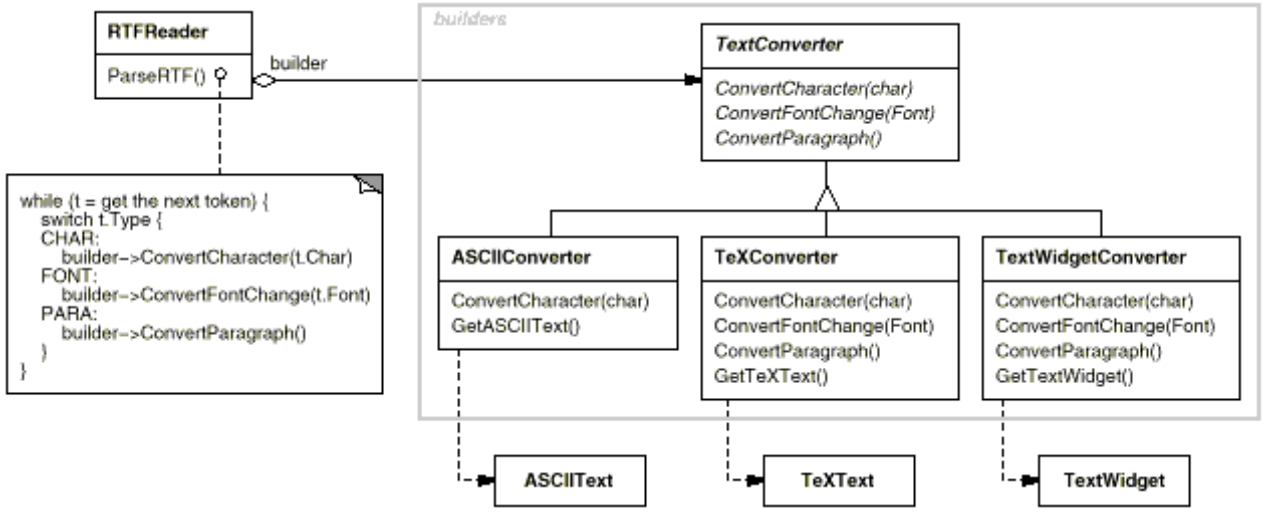
CONSTRUCTOR

Constructor



Originalmente *builder*.

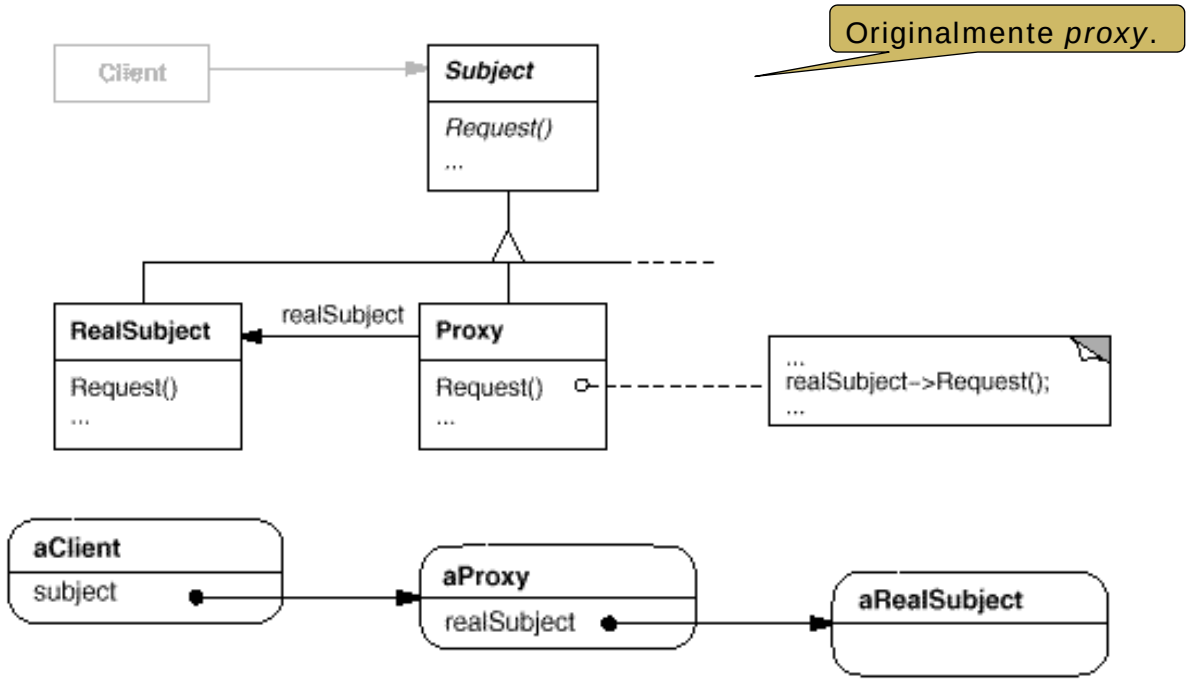
Ejemplo: conversor de formatos



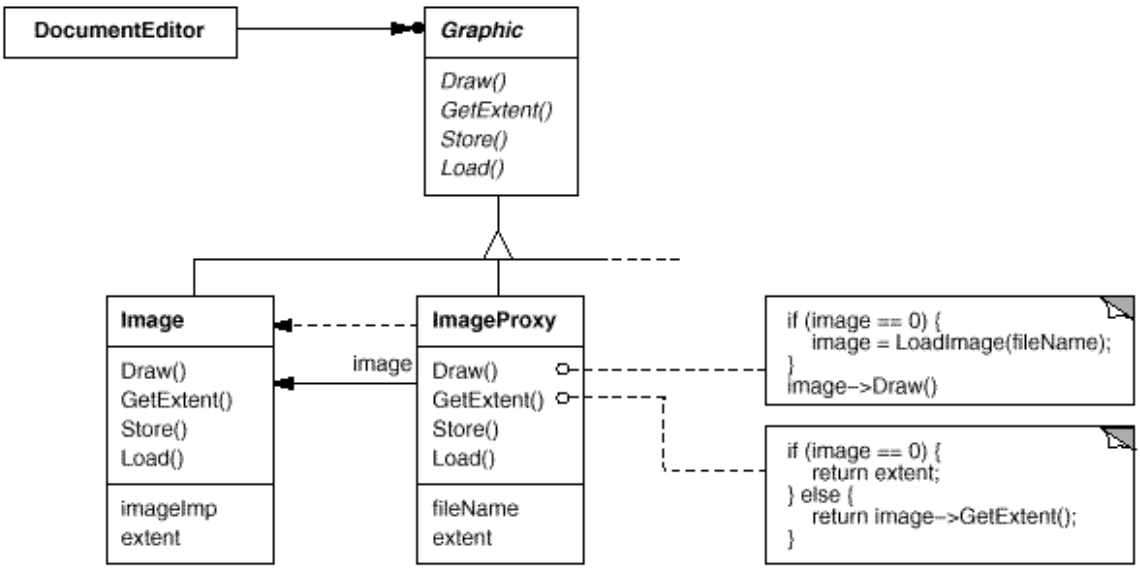
Patrones adicionales

PROXY

Proxy



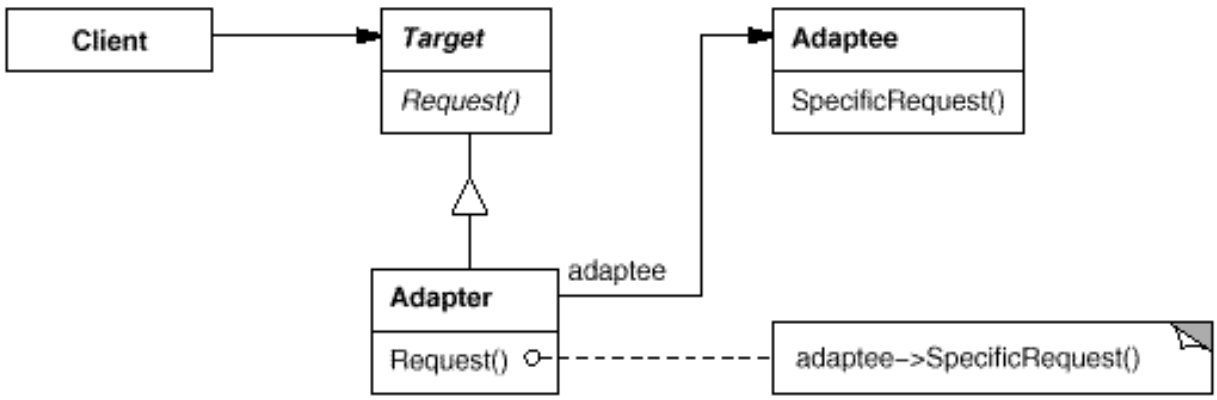
Ejemplo: editor con gráficos



Patrones adicionales

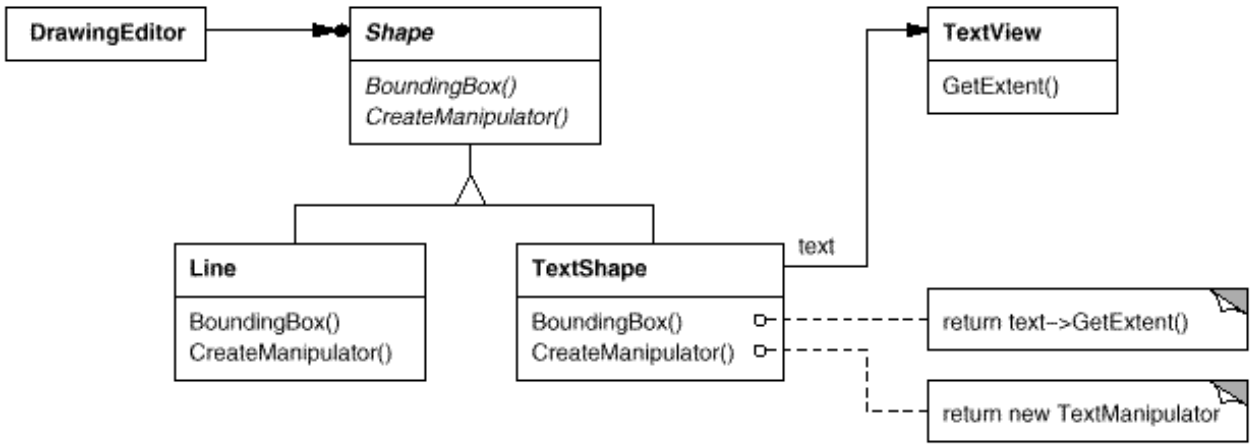
ADAPTADOR

Adaptador



Originalmente *adapter*. Esta es la versión para objetos pero también se puede hacer para clases.

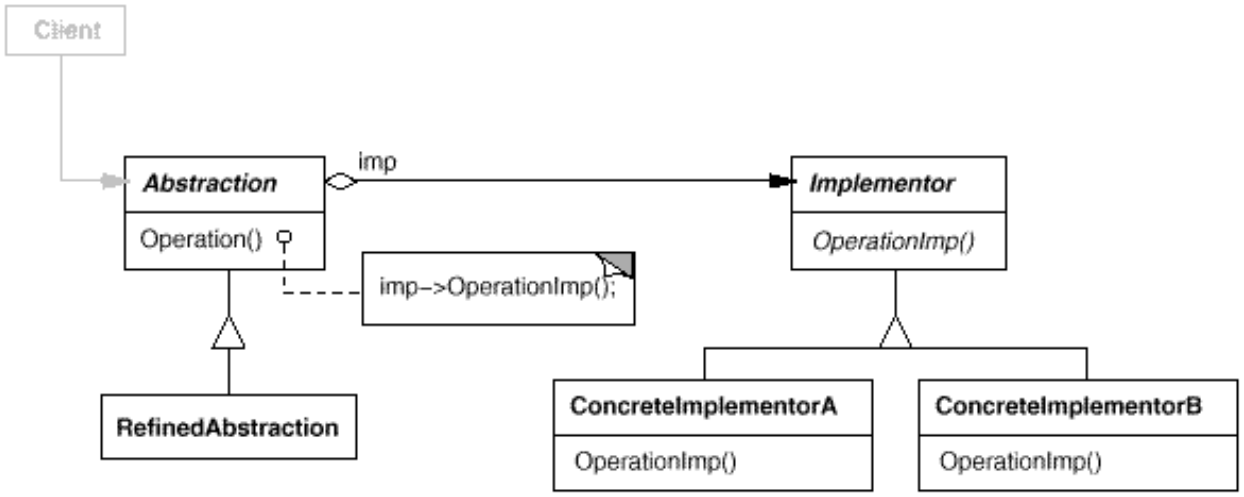
Ejemplo: gestión de texto



Patrones adicionales

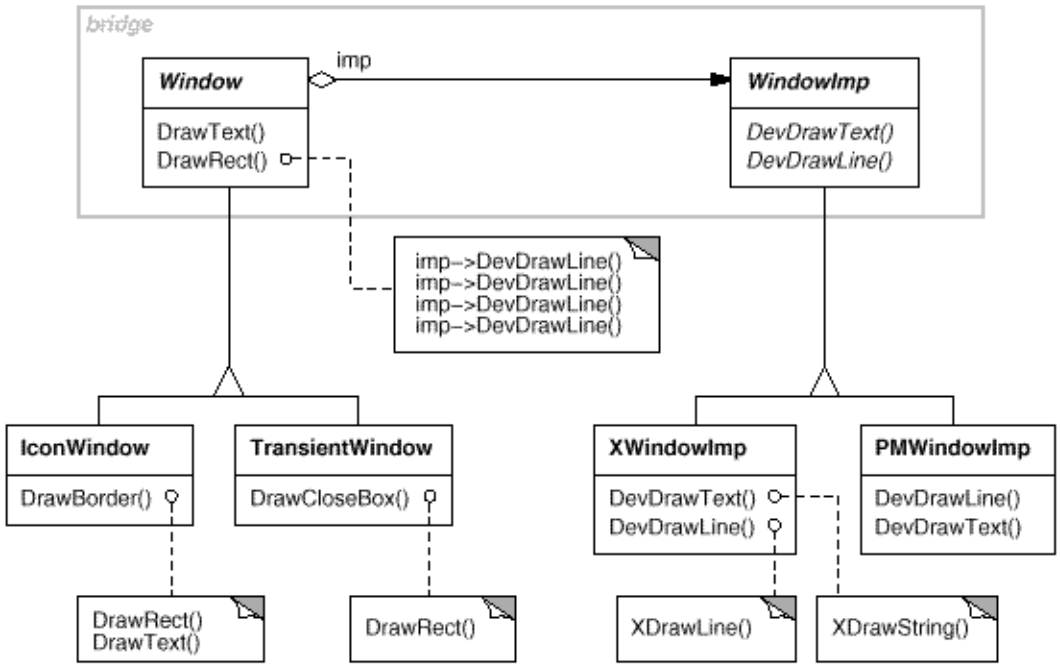
PUENTE

Puente



Originalmente *bridge*.

Ejemplo: sistema de ventanas



Patrones adicionales

MEDIADOR

Mediador

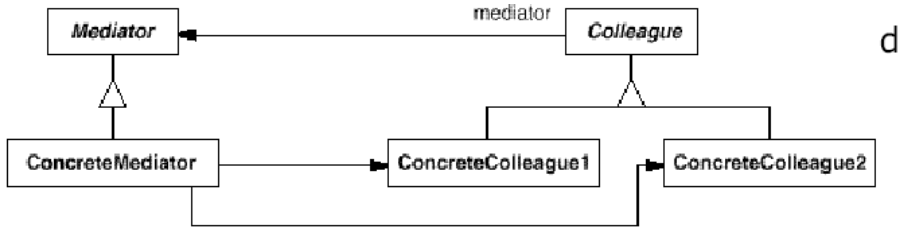


diagrama de clases

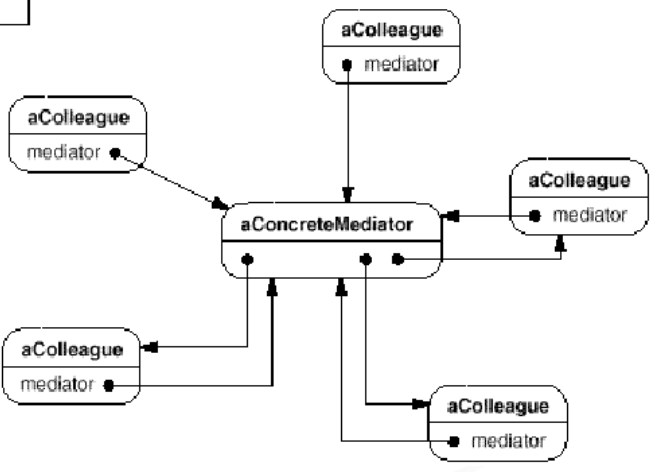
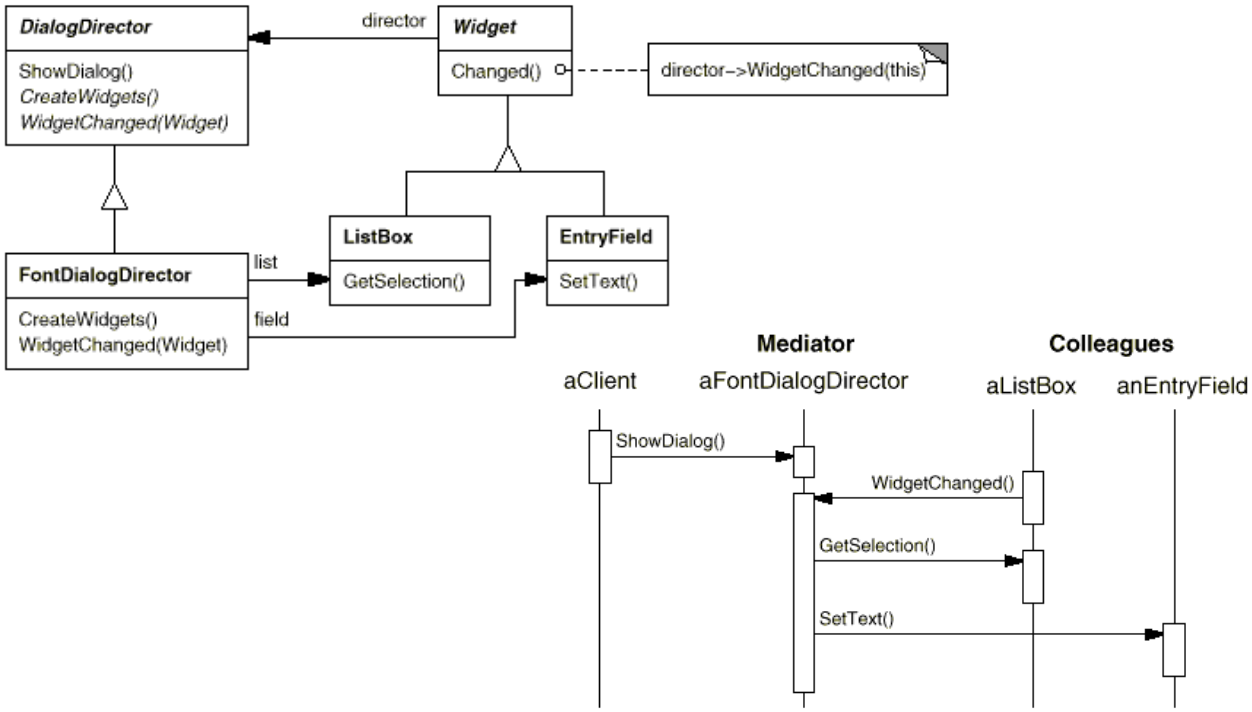


diagrama de objetos

Originalmente *mediator*.

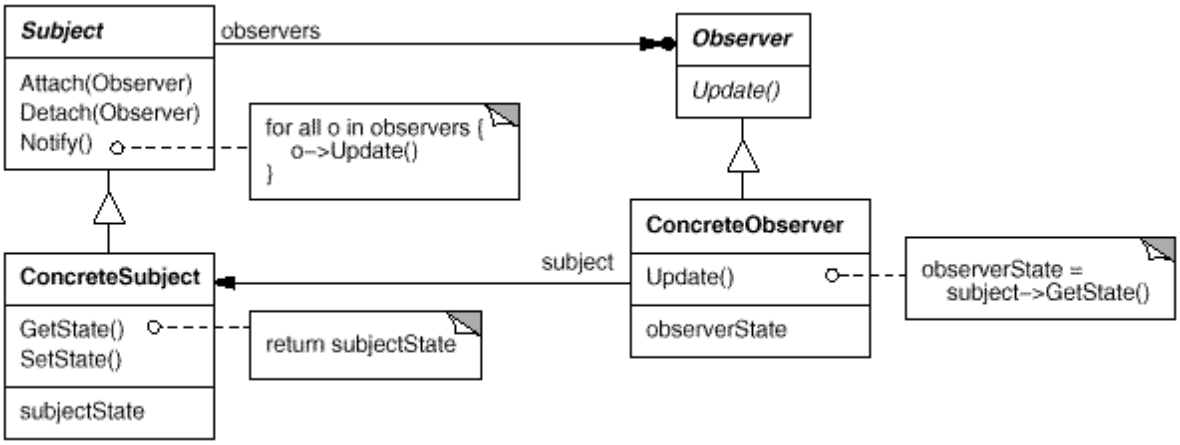
Ejemplo: gestión de textos



Patrones adicionales

OBSERVADOR

Observador

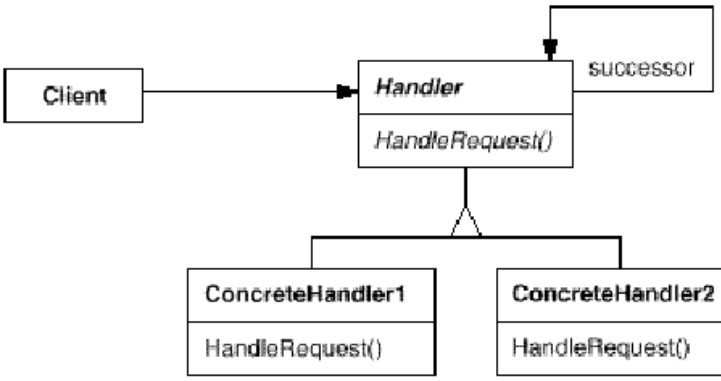


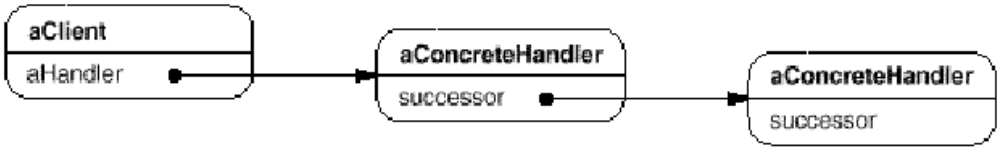
Originalmente *observer*.

Patrones adicionales

CADENA DE RESPONSABILIDADES

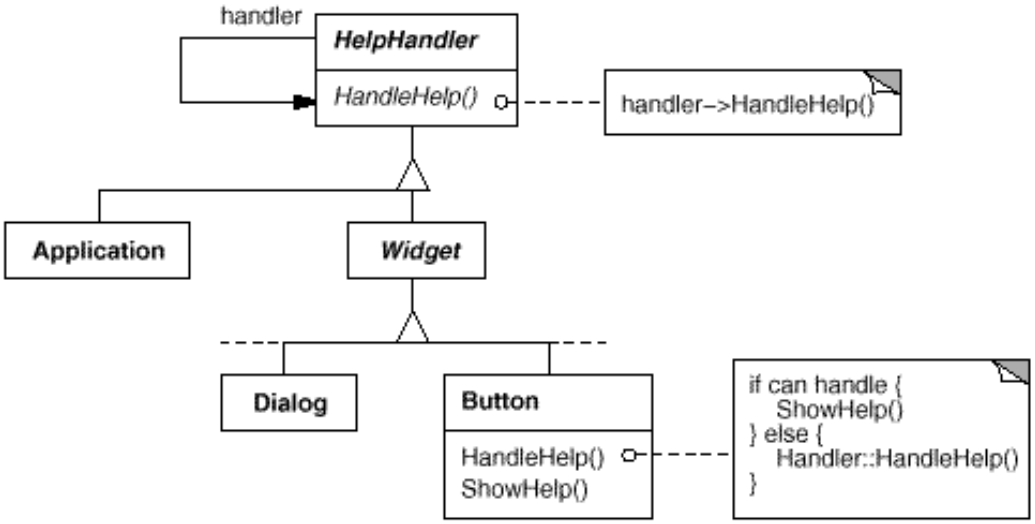
Cadena de responsabilidades





Originalmente *chain of responsibility*.

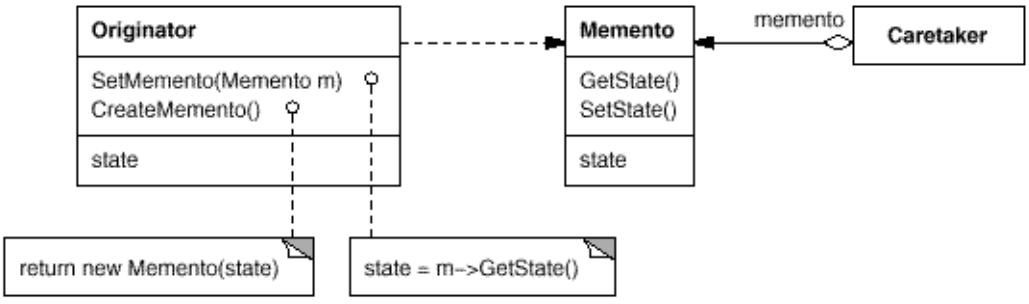
Ejemplo: control de eventos

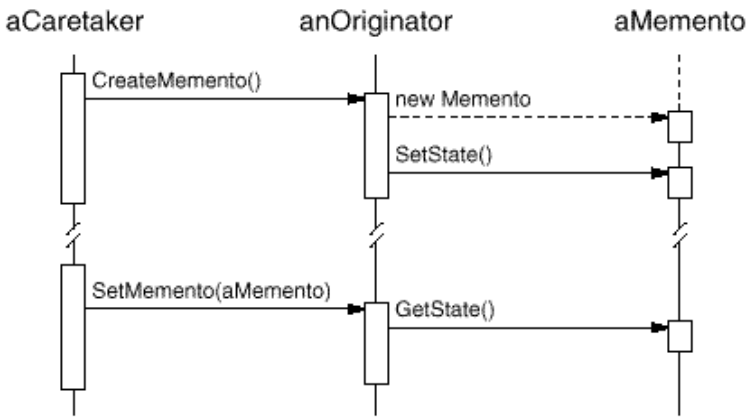


Patrones adicionales

RECUERDO

Recuerdo



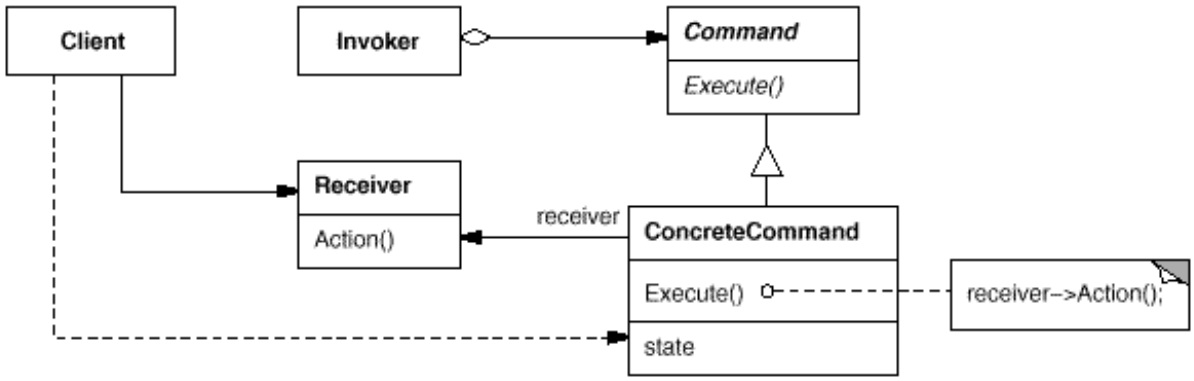


Originalmente *memento*.

Patrones adicionales

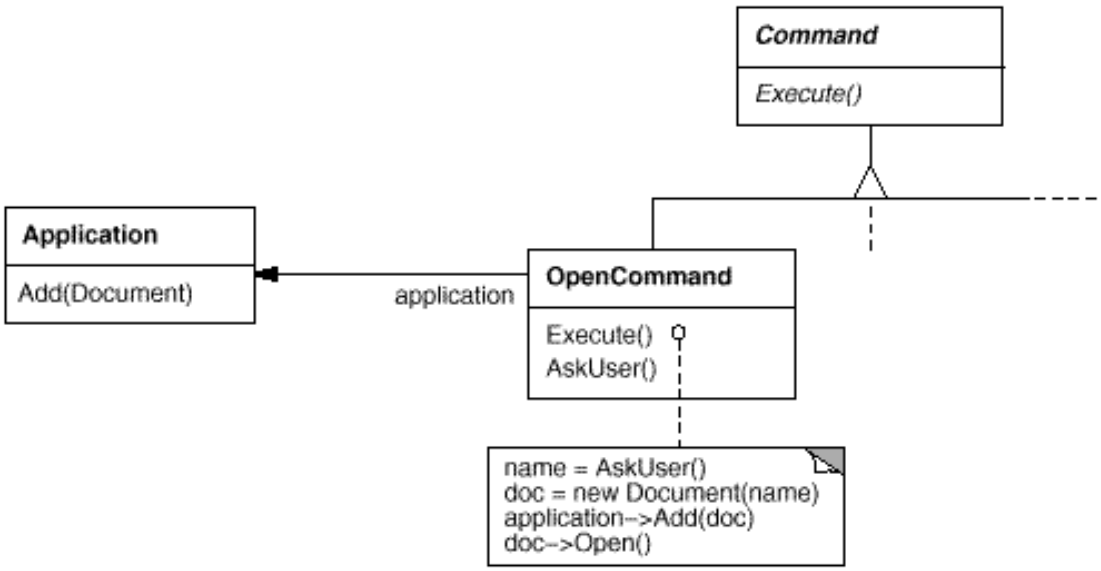
ORDEN

Orden



Originalmente *command*.

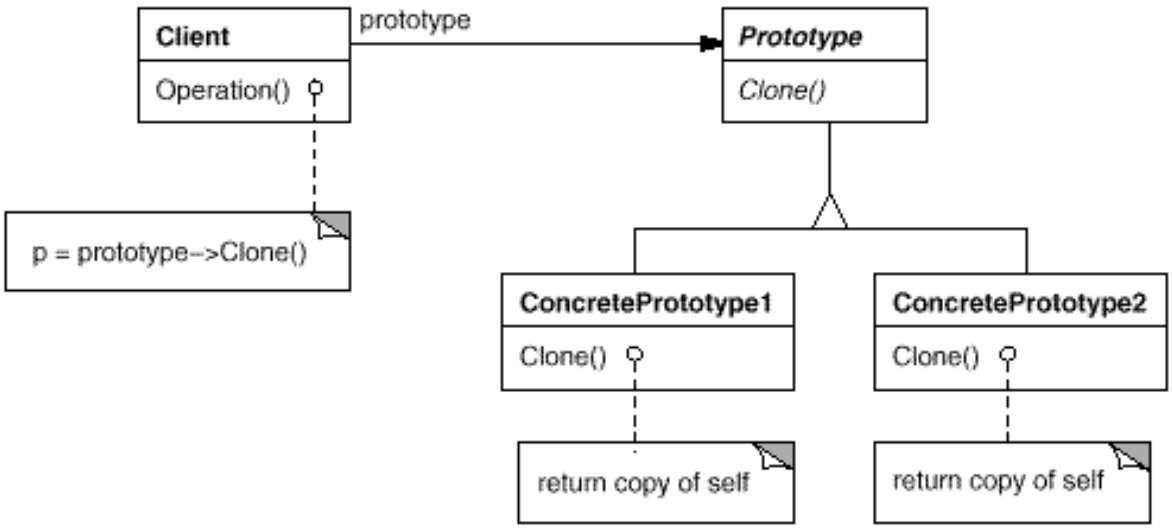
Ejemplo: opción de menú



Patrones adicionales

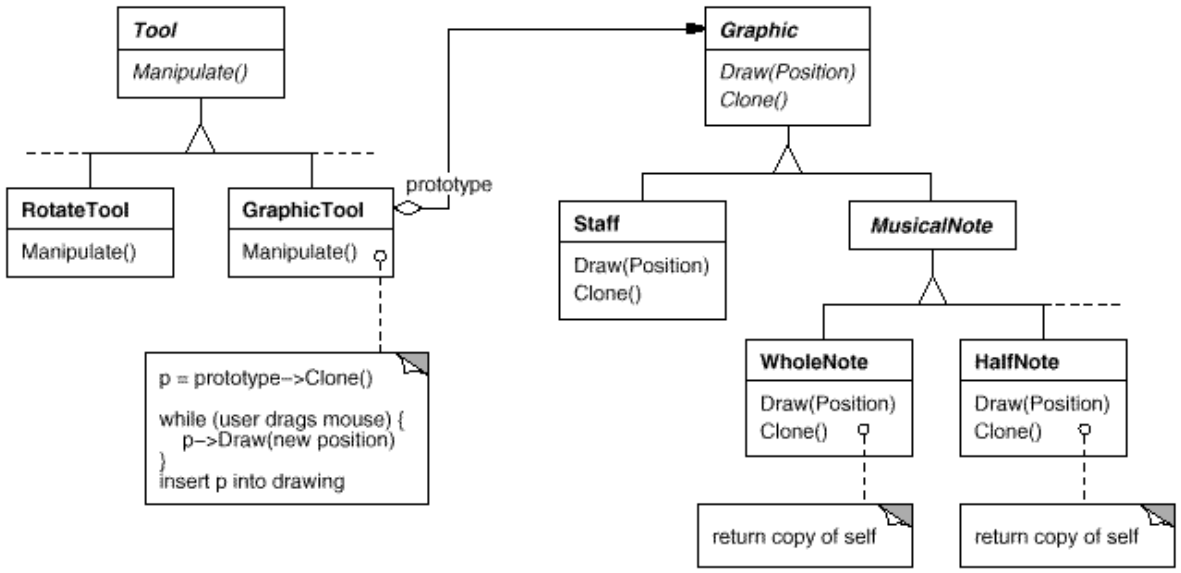
PROTOTIPO

Prototipo



Originalmente *prototype*.

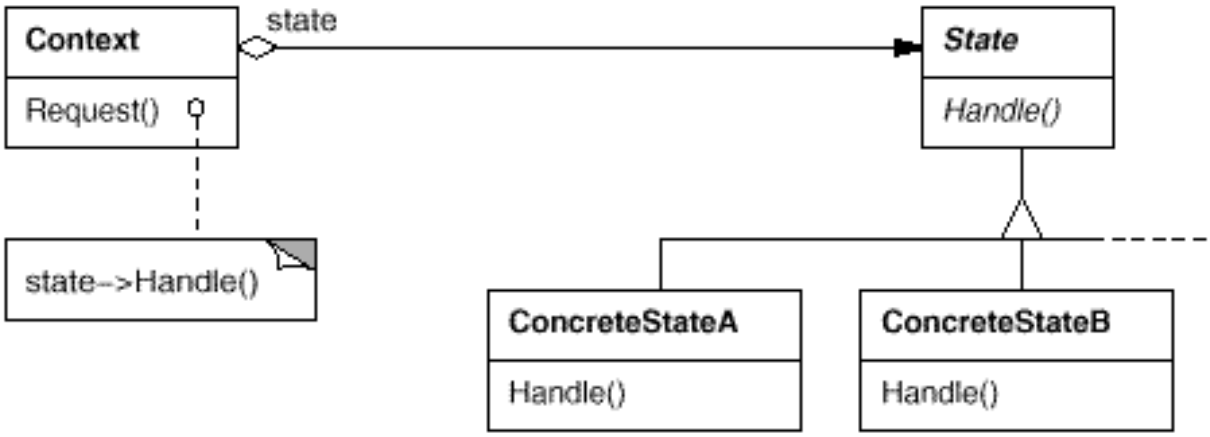
Ejemplo: dibujo



Patrones adicionales

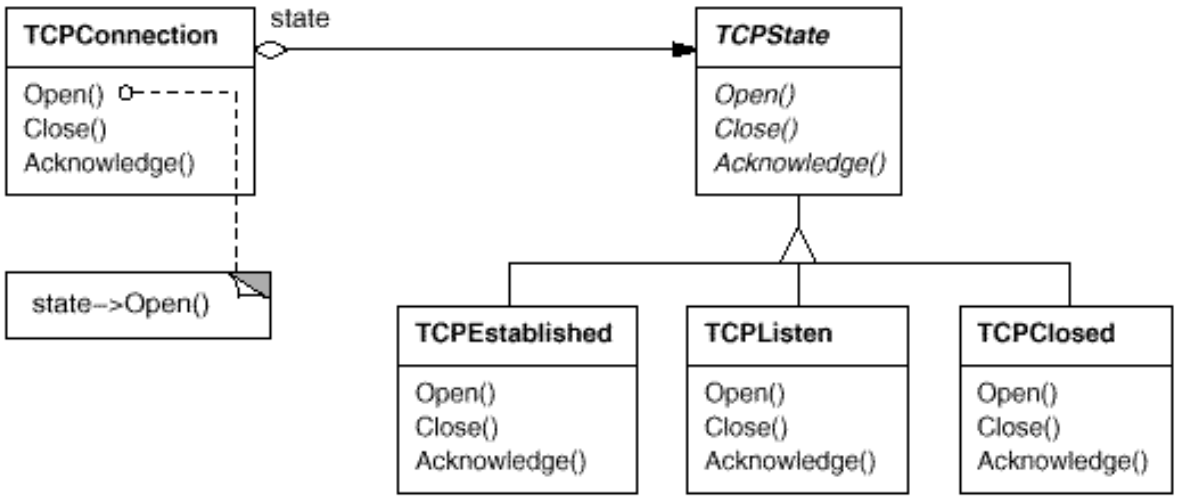
ESTADO

Estado



Originalmente *state*.

Ejemplo: conexiones de red



CONCLUSIONES

Conclusiones

- Los patrones de diseño son soluciones probadas a problemas recurrentes de diseño de software.
- Generalmente inciden en la flexibilidad del diseño y la facilidad para su modificación.
- Existen grupos centrados en diferentes problemáticas particulares:
  - Creación explícita de objetos
  - Dependencia de operaciones concretas
  - Dependencia de la plataforma
  - Dependencia de la representación o la implementación de los objetos
  - Dependencia de los algoritmos
  - Acoplamiento
  - Extensión de la funcionalidad mediante subclases
  - Imposibilidad de modificar las clases existentes

Glosario

- GoF = *Gang of Four*

Referencias

- C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel: A Pattern Language. Oxford University Press, 1977.
- E. Gamma, R. Helm, R. Johnson, J. Vlissides: Patrones de Diseño – Elementos de software orientado a objetos reutilizables. Addison-Wesley, 2008.
- A. Shalloway, J.R. Trott: Design Patterns Explained – A New Perspective on Object-Oriented Design. Addison-Wesley, 1998.
- J. Vlissides: Pattern Hatching – Design Patterns Applied. Addison-Wesley, 2002.