Tema 1. Desarrollo utilizando patrones de diseño

Desarrollo de Software
Curso 2020-2021
3º Grado Ingeniería Informática

Dto. Lenguajes y Sistemas Informáticos
ETSIIT
Universidad de Granada

26 de febrero de 2021

Tema 1. Desarrollo utilizando patrones de diseño

1.1 Análisis y diseño basado en patrones
- Origen e historia de los patrones software
- Conceptos generales y clasificación
- Relación con el concepto de marco de trabajo (framework)
- Elementos de un patrón de diseño
- Un ejemplo práctico a partir de las clases Modelo-Vista-Controlador (MVC)
- Cómo resolver problemas de diseño usando patrones de diseño

1.2 Estudio del catálogo GoF de patrones de diseño
- Patrones creacionales "Factoría Abstracta", "Método Factoría", "Prototipo" y "Builder"
- Patrones estructurales "Facade", "Composite", "Decorator" y "Adapter"
- Patrones conductuales "Observer", "Visitor", ' "Strategy", "Template Method" e "Intercepting Filter"

1.3 Resolución de ejercicios: casos prácticos

- Los patrones software fueron la adaptación al mundo de las TICs de los patrones arquitectónicos
- Patrones arquitectónicos: definidos por Christopher Alexander (1966) como la identificación de ideas de diseño arquitectónico mediante descripciones arquetípicas y reusables

- Sus teorías sobre la naturaleza del diseño centrado en el hombre han repercutido en otros campos como en la sociología y en la ingeniería informática

- 1987: Llevada la idea a un congreso (Beck and Cunningham, 1987)
- 1994: Primer libro ("Gang of Four") (Gamma et al., 1994a)
  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 (Appleton, 2000).
- 1993: Se retiran a la nieve para pensar, Kent Beck y Grady Booch organizan:
  - Patrón arquitectónico y objeto software (creatividad)
  - Crean el "HillSide Group" (The HillSide Group).
  - 1996: Nuevo libro, patrones en todos lo niveles de abstracción/etapas (Buschmann et al., 1996)

- 1996: Nuevo libro, patrones en todos lo niveles de abstracción/etapas (Buschmann et al., 1996)

Figura 1: El lugar donde se creó el "HillSide Group", en las montañas de Colorado en 1993.

Fueron definidos por GoF como:

> *A pattern involves a general description of a recurring solution to a recurring problem replete with various goals and constraints. But a pattern does more than just identify a solution, it also explains why the solution is needed! (Appleton, 2000).*

- Los patrones surgen desde la orientación a objetos y resuelven problemas a nivel de diseño orientado a objetos
- Enseguida se amplían:
  - A cualquier paradigma de programación
  - Aportando soluciones en un espectro mucho más amplio en el nivel de abstracción
    - Patrones arquitectónicos (tema 2)
    - Patrones de diseño
    - Patrones de código (idioms)

- Otra clasificación: según la fase dentro del ciclo de vida de desarrollo del software:
  - Patrones conceptuales
  - Patrones de diseño
  - Patrones de programación

Clasificación de los patrones de diseño (Gamma et al., 1995) (pp 21 y 22).

- Según el el propósito:
  - Creacionales.- Relacionados con el proceso de creación de objetos
  - Estructurales o estáticos.- Relacionados con los componentes que forman las clases y los objetos
  - Conductuales o dinámicos.- Relacionados con la forma en la que los objetos y las clases interactúan entre sí y se reparten las responsabilidades
- Según el ámbito de aplicación:
  - De clase.- El patrón se aplica principalmente a clases
  - De objeto.- El patrón se aplica principalmente a objetos

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
Scope	**Object**	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Tabla 1: Espacio de los patrones de diseño [Fuente: (Gamma et al., 1994b), pp.21-22].

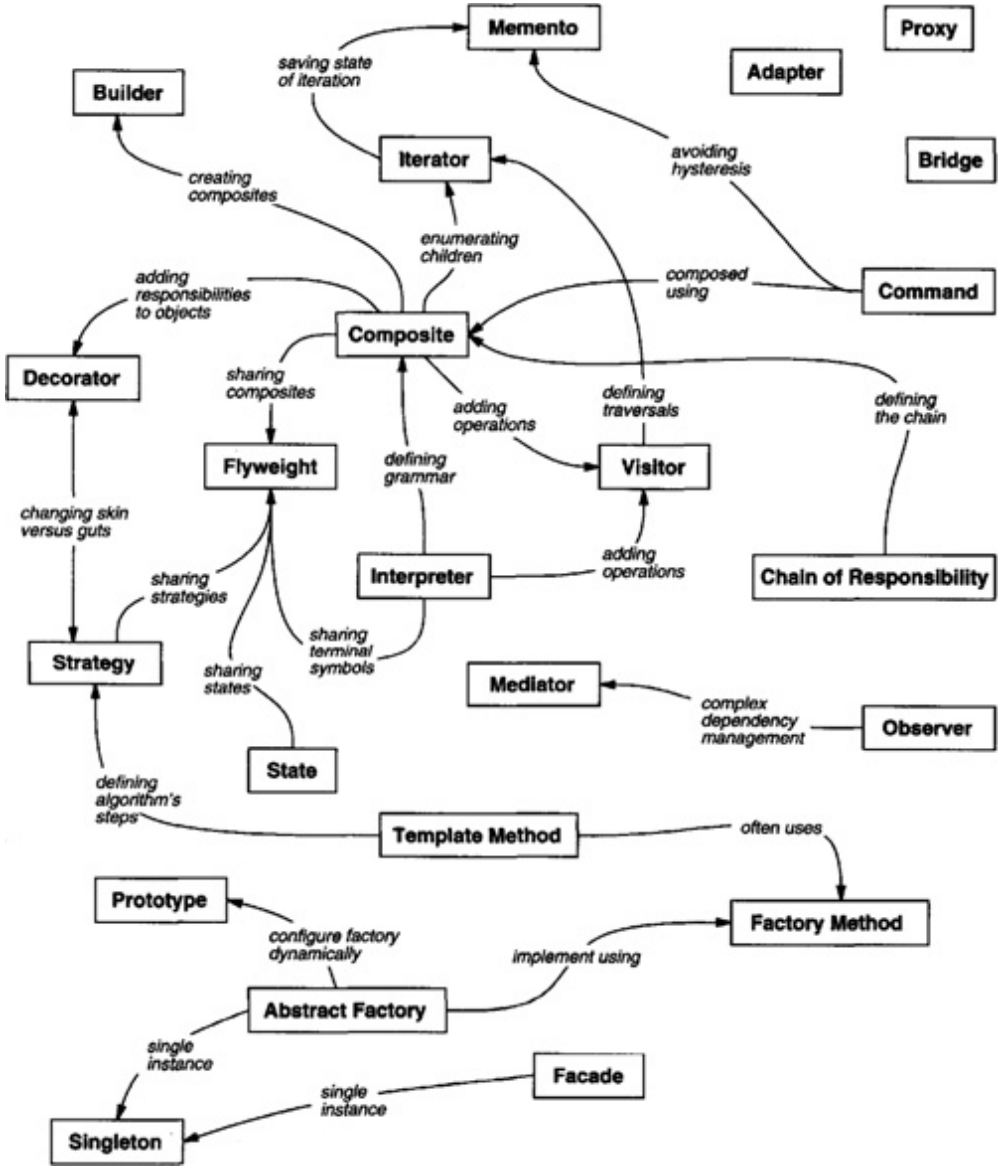


Figura 2: Relación entre los patrones de diseño. [Fuente: (Gamma et al., 1994b), p. 23]

1. Análisis y diseño basado en patrones
Conceptos generales y clasificación
Clasificación según la relación entre patrones (Gamma et al., 1994b) (p. 23)

- Marco de trabajo.- Un conjunto amplio de funcionalidad software ya implementada que es útil en un dominio de aplicación específico, tal como un sistema de gestión de bases de datos, un tipo de aplicaciones web, etc.
- Patrón software (de diseño, arquitectónico ...).- NO ESTÁ IMPLEMENTADO; "receta" aplicable en cualquier dominio de aplicaciones, capaz de dar la misma solución a distintos problemas con una base similar

Plantilla	
Nombre	
Clasificación	arquitectónico/de diseño/de programación
Contexto	entorno en el que se ubica el problema
Problema	qué se pretende resolver
Consecuencias	fortalezas, limitaciones y restricciones
Solución	descripción detallada de la solución
Intención	describe el patrón y lo que hace
Anti-patrones	"soluciones" que no funcionan
Patrones relacionados	referencias cruzadas
Referencias	reconocimientos a desarrolladores
Estructura	diagrama UML
Participantes	descripción de los componentes

- Idea: separar el objeto de la aplicación (el modelo) de la vista, para aumentar su flexibilidad y reusabilidad.
- La GoF utiliza el diseño MVC para identificar e introducir los tres primeros patrones de su libro (Gamma et al., 1994a).
- MVC hoy se considera un patrón arquitectónico.
- Patrones que incluye:
  - Observer.- En MVC hay un protocolo de subscripción/notificación modelo-vista que permite desacoplar la vista del modelo. El patrón *observador* generaliza esta idea para proponer el desacoplamiento entre dos objetos distintos cualesquiera.
  - Composite.- un grupo de objetos son tratados como uno individual (una vista puede contener otras vistas). El patrón *composite* generaliza esta idea para proponer una igual tratamiento de objetos simples y de compuestos de objetos, mediante la definición de clases de objetos simples y de objetos compuestos que heredan de una clase común.
  - Strategy.- La vista en el diseño MVC permite tener distintas formas de responder a las operaciones del usuario, es decir distintos algoritmos de control, representados por objetos controladores, que heredan todos de una misma clase para poder intercambiarlos incluso en tiempo de ejecución. El patrón *estrategia* generaliza esta idea de forma que se usen objetos para representar algoritmos, que hereden de una clase común, pudiendo cambiarse el algoritmo a usar en cada momento mediante el uso de otro objeto.
  - Factory method.- para especificar la clase Controlador.
  - Decorator.- para añadir desplazamiento (scrolling) a una vista.

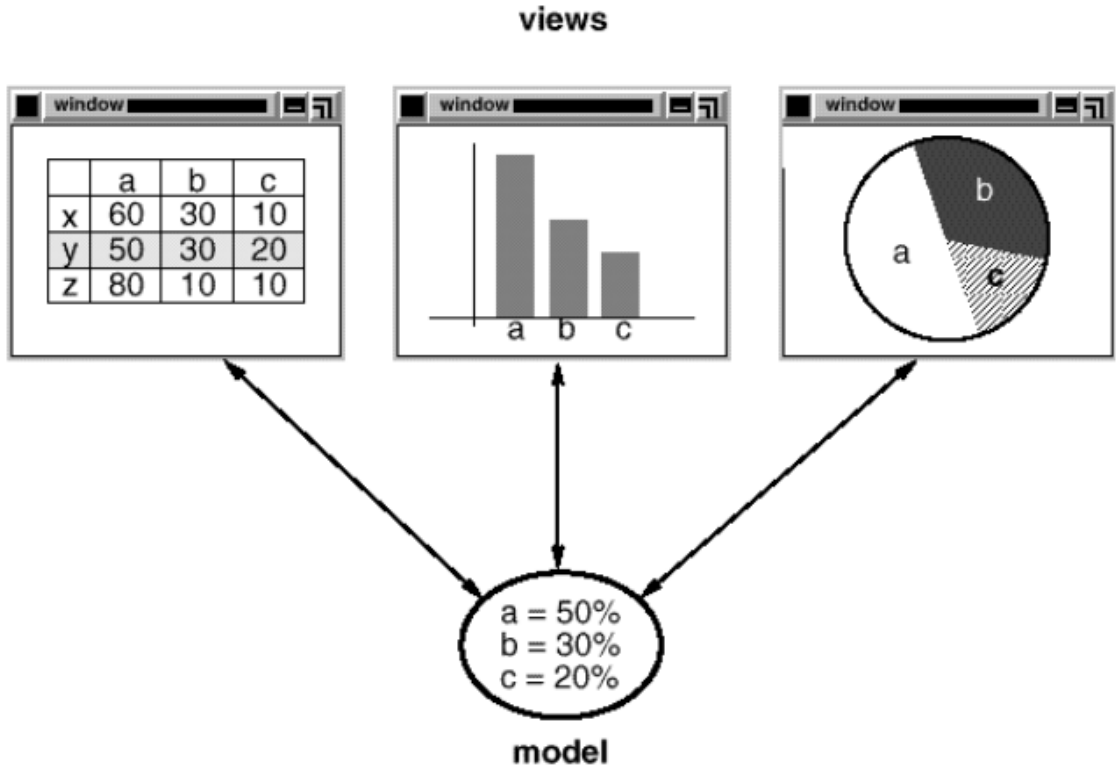


Figura 3: Ejemplo de modelo con tres vistas, en un diseño MVC. [Fuente: (Gamma et al., 1994b), p. 15]

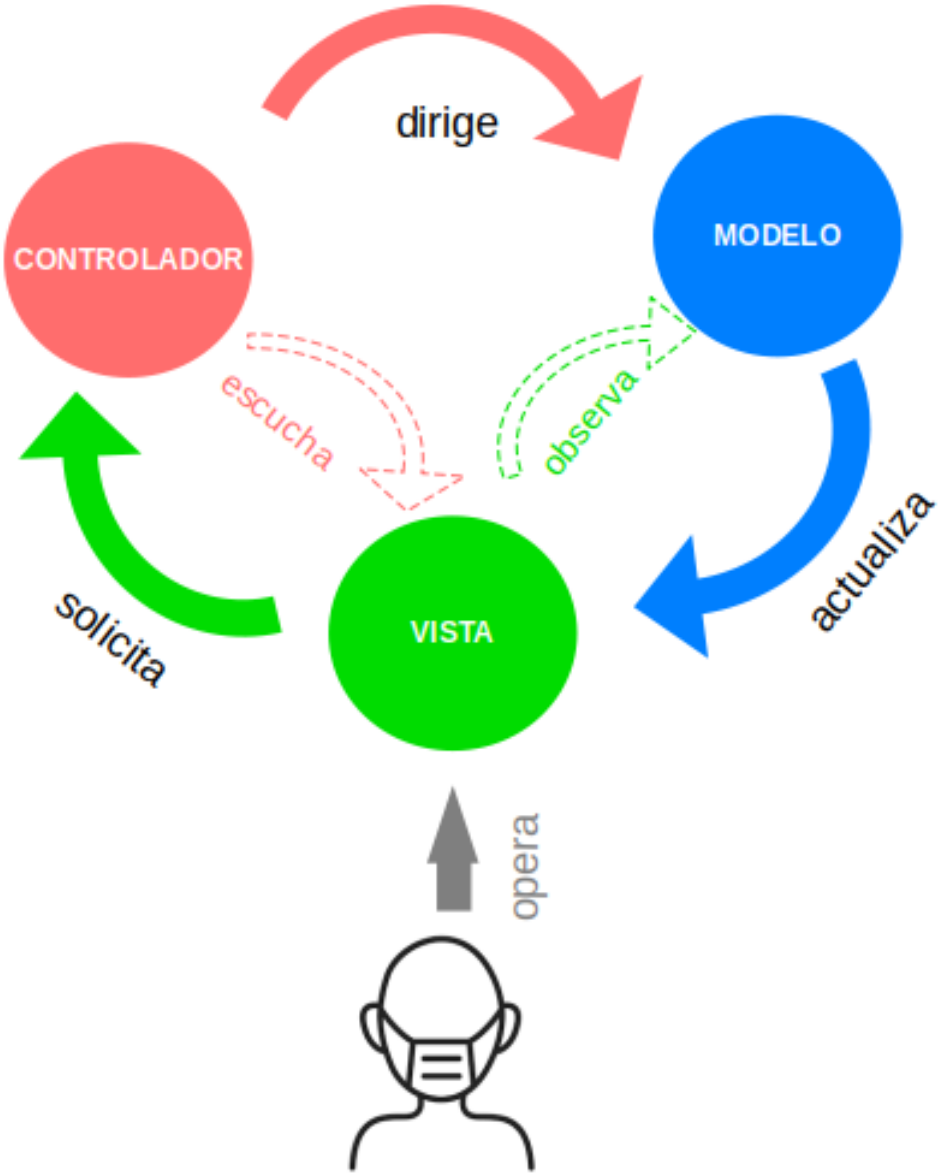


Figura 4: Ejemplo de diagrama relacional de la terna de clases en el diseño MVC.

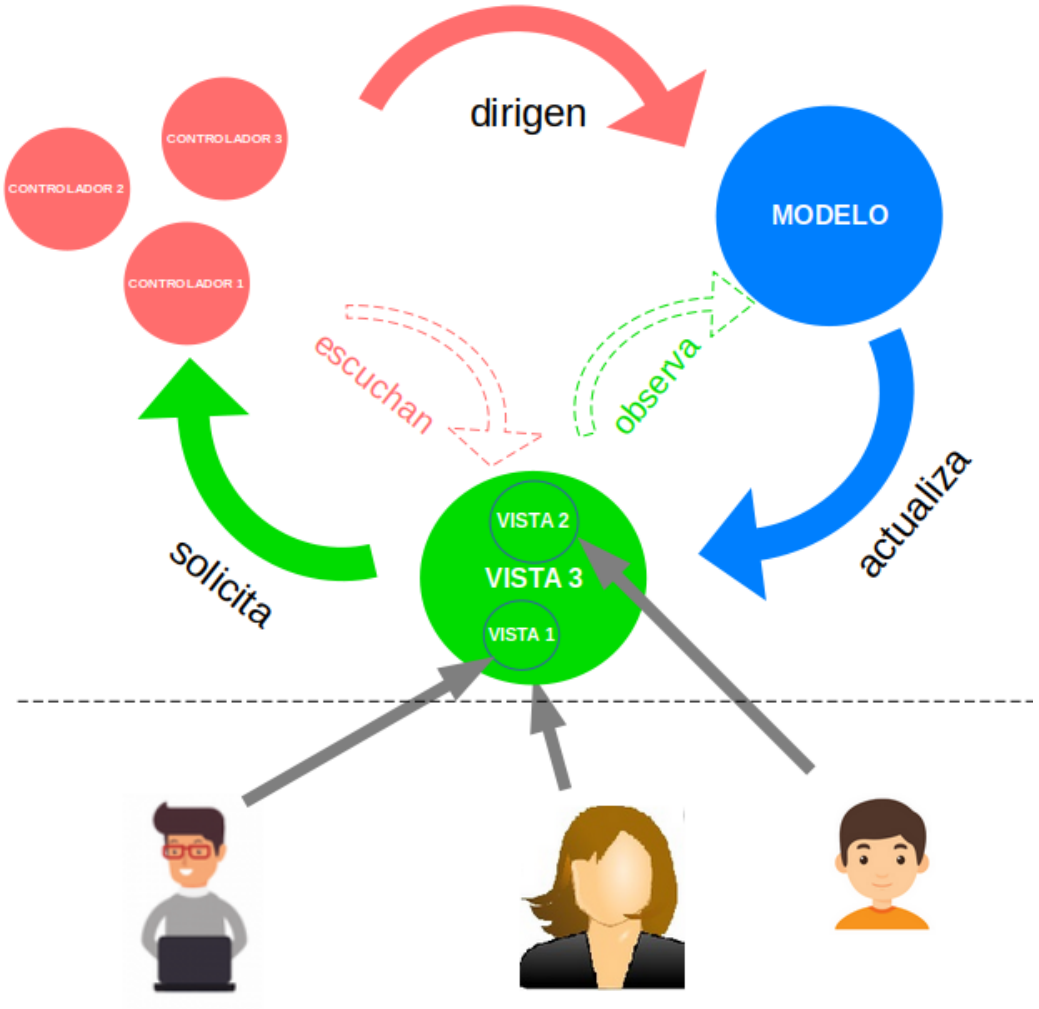


Figura 5: Ejemplo más complejo de diagrama relacional en el diseño MVC.

- Encontrando los objetos (clases) apropiados
- Considerando distinta granularidad
- Especificando la interfaz de un objeto
- Programando en función de las interfaces y no de las implementaciones
- Sacando el máximo partido de los distintos mecanismos de reusabilidad de código
  - Favoreciendo la composición (diseño de caja negra) sobre la herencia (diseño de caja blanca)
  - Usando la delegación como alternativa extrema de la composición que sustituya la herencia
  - Usando tipos parametrizados como alternativa a la herencia

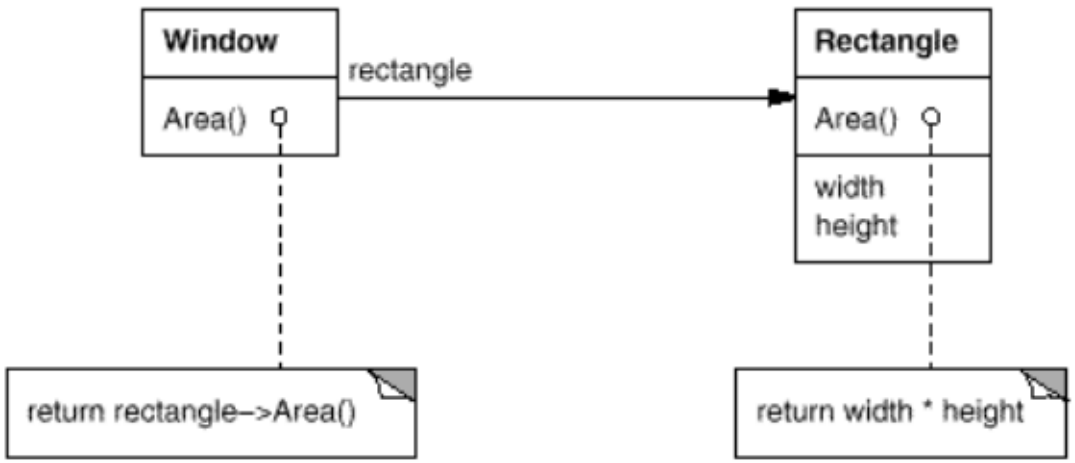


Figura 6: Ejemplo de uso de delegación (Gamma et al., 1994a, p.33) p. 33.

GoF (Gamma et al., 1994a) recomiendan empezar con el estudio de los siguientes patrones de diseño:

- Abstract Factory (Gamma et al., 1994b) pp. 99-109 (Figura 23)
- Adapter (Gamma et al., 1994b) pp. 157-170
- Composite (Gamma et al., 1994b) pp. 183-195
- Decorator (Gamma et al., 1994b) pp. 196-207
- Factory Method (Gamma et al., 1994b) pp. 121-132
- Observer (Gamma et al., 1994b) pp. 326-337 (Figura 49)
- Strategy (Gamma et al., 1994b) pp. 349-359
- Template method (Gamma et al., 1994b) pp. 360-365

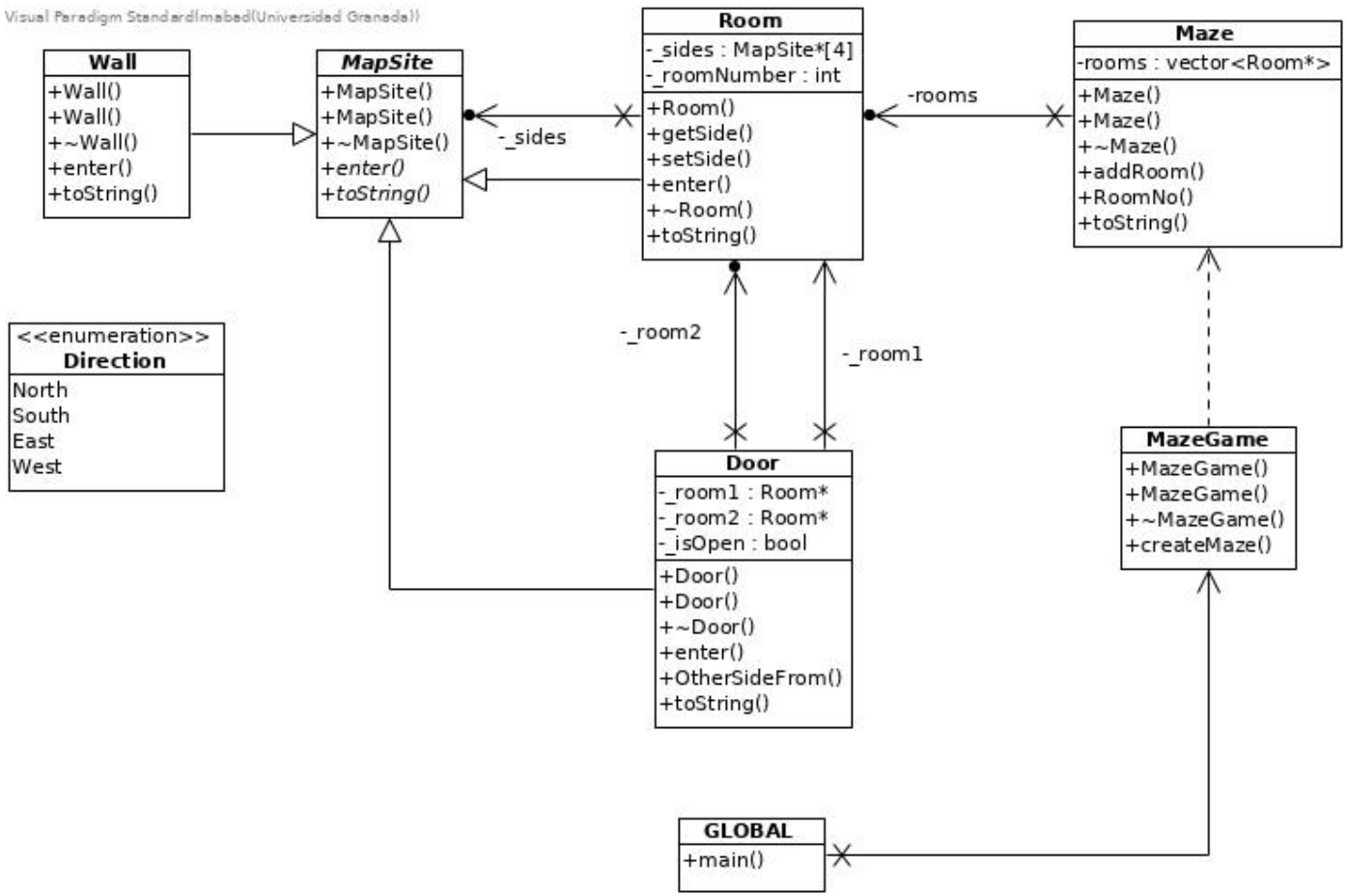
Veremos además los siguientes otros patrones:

- Prototype (Gamma et al., 1994b) pp. 133-143 (Figura 29)
- Builder (Gamma et al., 1994b) pp. 110-120
- Visitor (Gamma et al., 1994b) pp. 366-381 (Figura 51)
- Facade (Gamma et al., 1994b) pp. 208-217
- Filtro de intercepción (Figura 60)

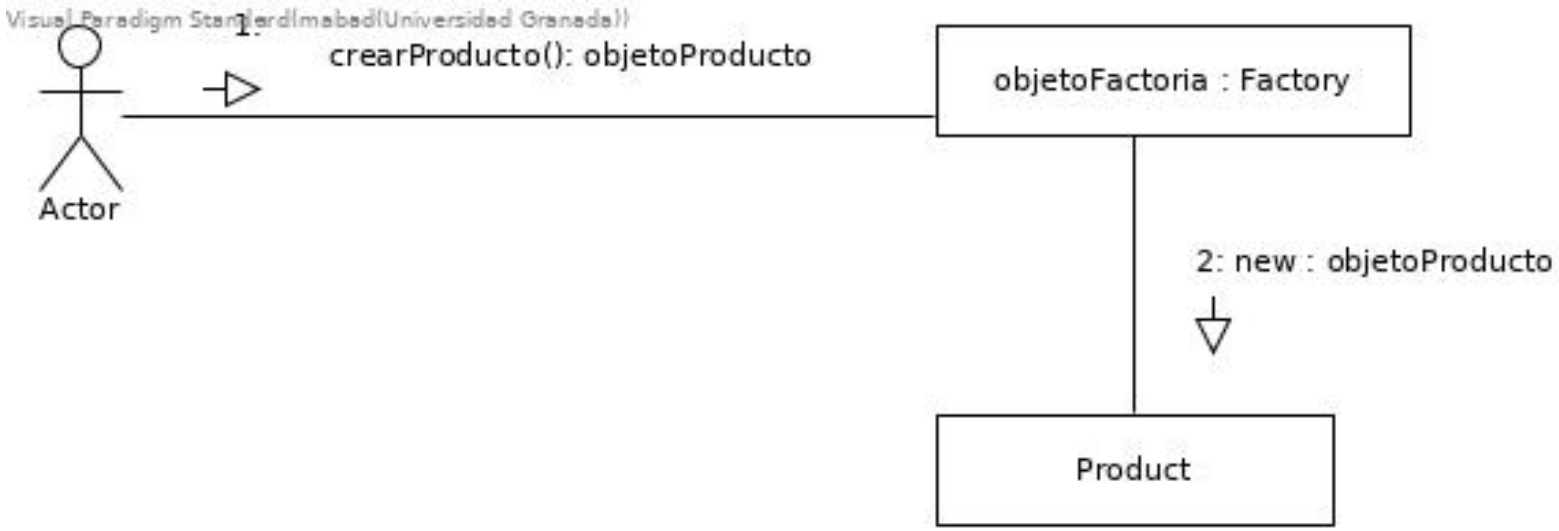
Patrones creacionales "Factoría Abstracta", "Método Factoría", "Prototipo" y "Builder"

Usados cuando necesitamos crear objetos dentro de un marco de trabajo o una librería software pero desconocemos la clase de estos objetos ya que depende de la aplicación concreta.

Usaremos como ejemplo comparativo el juego del laberinto:

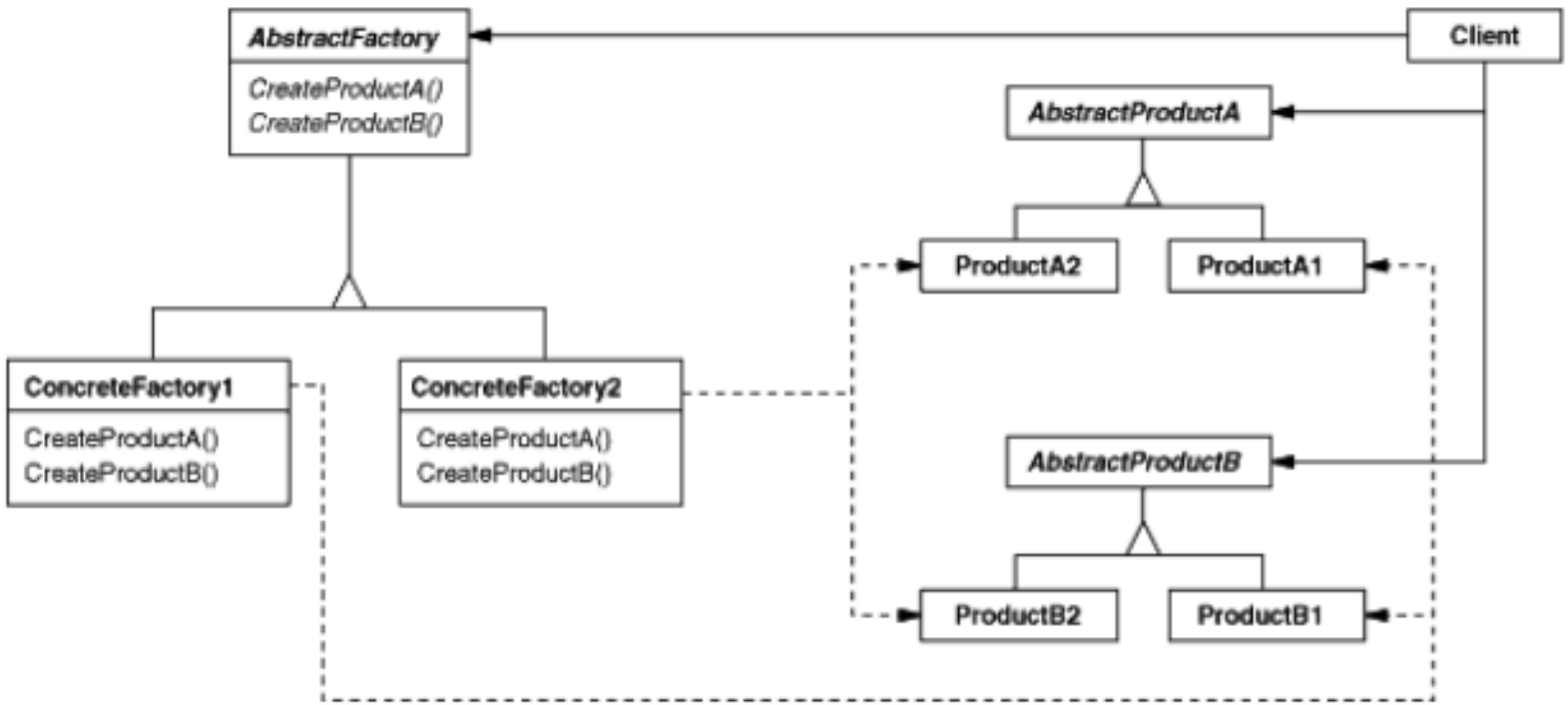


El concepto de factoría en OO

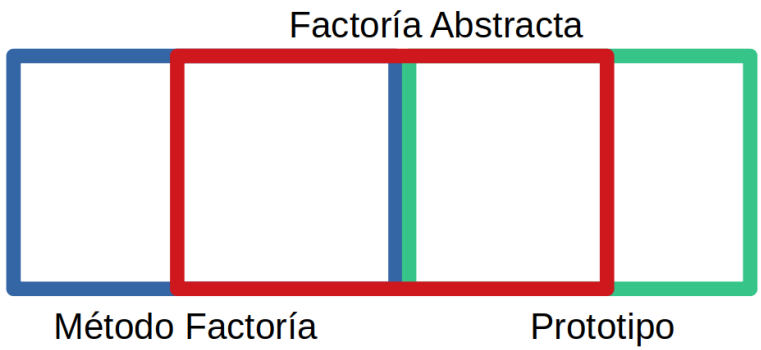


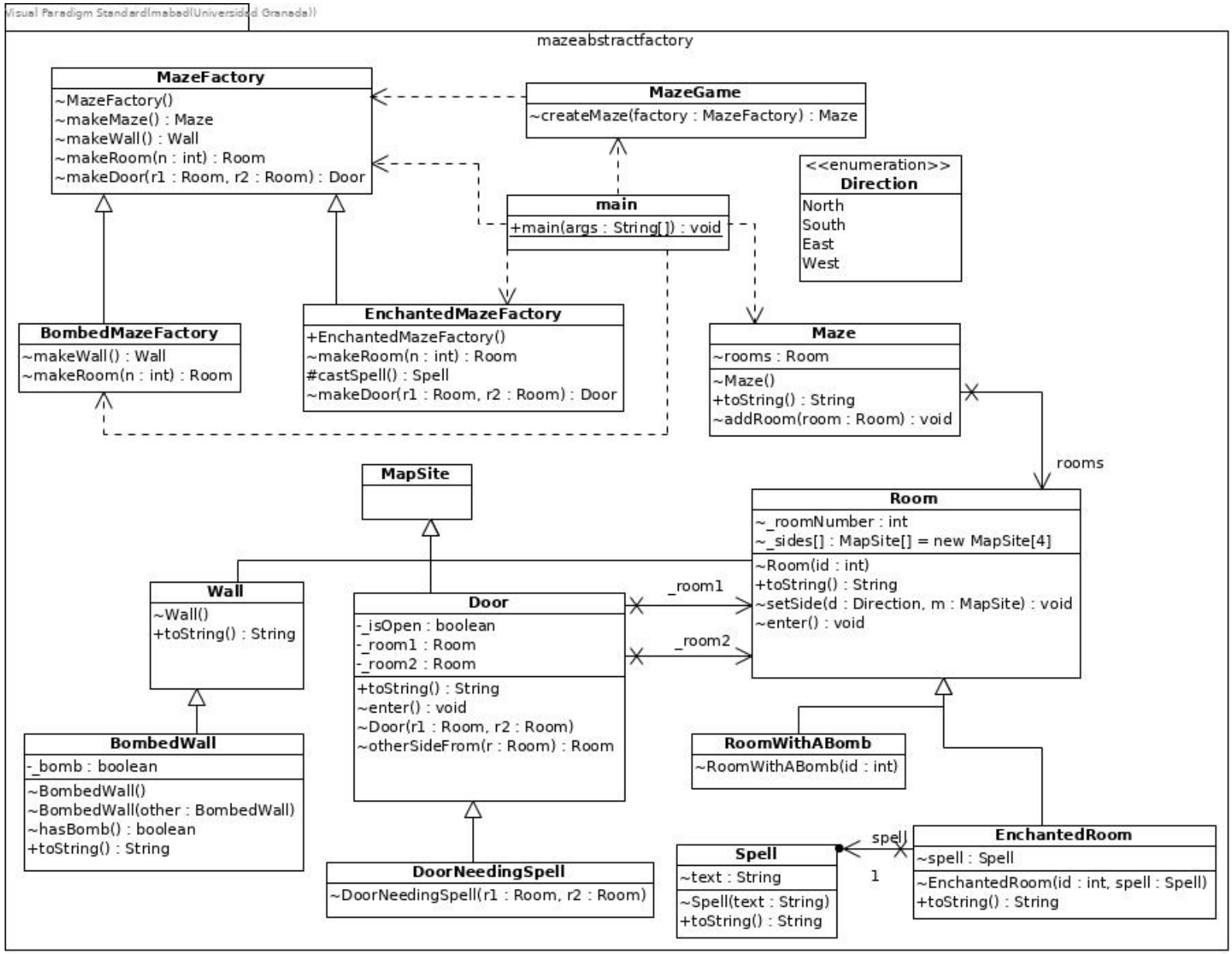
Patrón “Factoría Abstracta”

Recomendado cuando en una aplicación tenemos líneas, temáticas o familias “paralelas” de clases y se prevé que puedan añadirse nuevas líneas, es decir las mismas clases pero con alguna variación.



Hay dos formas en las que se pueden crear los objetos por las factorías, mediante métodos factoría o mediante delegación (prototipos), las cuales están reflejadas en otros dos patrones: “Método Factoría” y “Prototipo”, respectivamente.

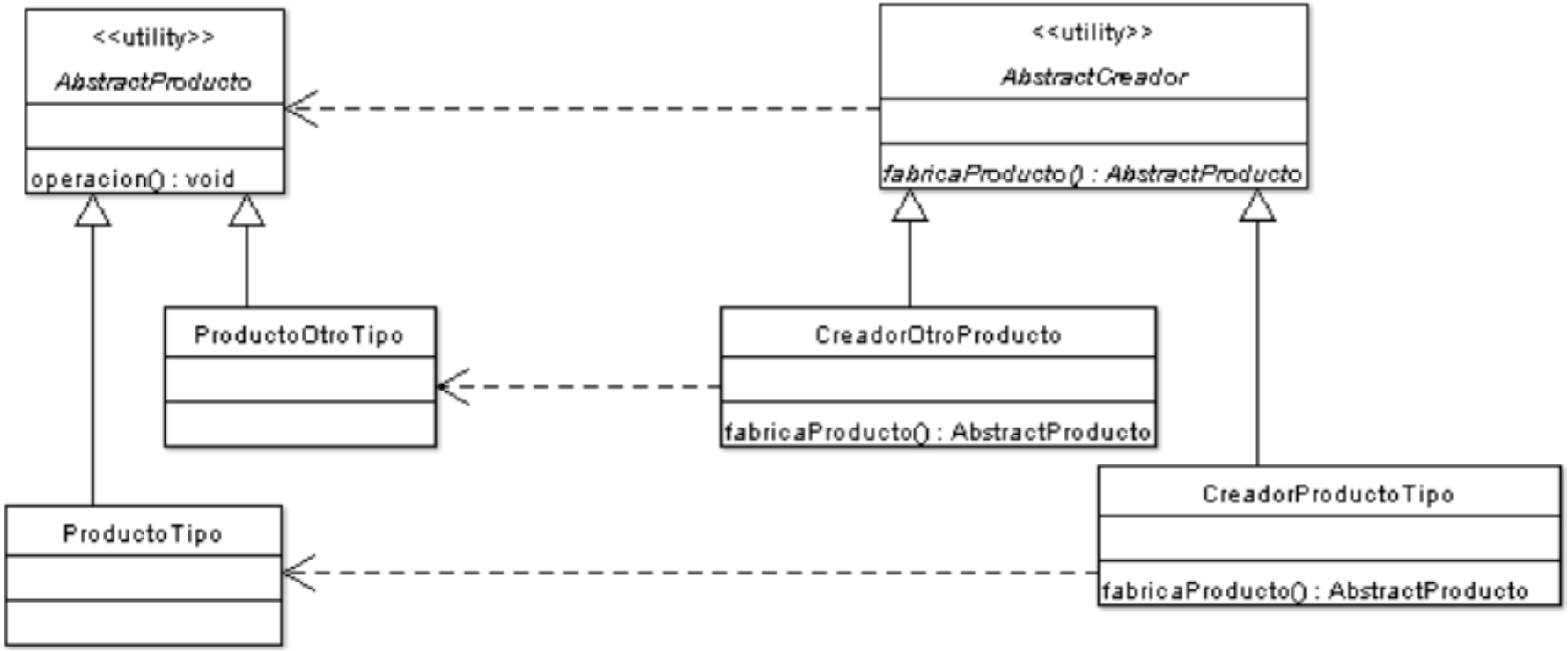



Visual Paradigm Standard(mabad(Universidad Granada))
mazeabstractfactory
MazeFactory
~MazeFactory()
~makeMaze() : Maze
~makeWall() : Wall
~makeRoom(n : int) : Room
~makeDoor(r1 : Room, r2 : Room) : Door
MazeGame
~createMaze(factory : MazeFactory) : Maze
main
+main(args : String[]) : void
<<enumeration>>
Direction
North
South
East
West
BombedMazeFactory
~makeWall() : Wall
~makeRoom(n : int) : Room
EnchantedMazeFactory
+EnchantedMazeFactory()
~makeRoom(n : int) : Room
#castSpell() : Spell
~makeDoor(r1 : Room, r2 : Room) : Door
Maze
~rooms : Room
~Maze()
+toString() : String
~addRoom(room : Room) : void
rooms
MapSite
Room
~_roomNumber : int
~_sides[] : MapSite[] = new MapSite[4]
~Room(id : int)
+toString() : String
~setSide(d : Direction, m : MapSite) : void
~enter() : void
Wall
~Wall()
+toString() : String
Door
-_isOpen : boolean
-_room1 : Room
-_room2 : Room
+toString() : String
~enter() : void
~Door(r1 : Room, r2 : Room)
~otherSideFrom(r : Room) : Room
_room1
_room2
BombedWall
-_bomb : boolean
~BombedWall()
~BombedWall(other : BombedWall)
~hasBomb() : boolean
+toString() : String
RoomWithABomb
~RoomWithABomb(id : int)
DoorNeedingSpell
~DoorNeedingSpell(r1 : Room, r2 : Room)
Spell
~text : String
~Spell(text : String)
+toString() : String
spell
1
EnchantedRoom
~spell : Spell
~EnchantedRoom(id : int, spell : Spell)
+toString() : String

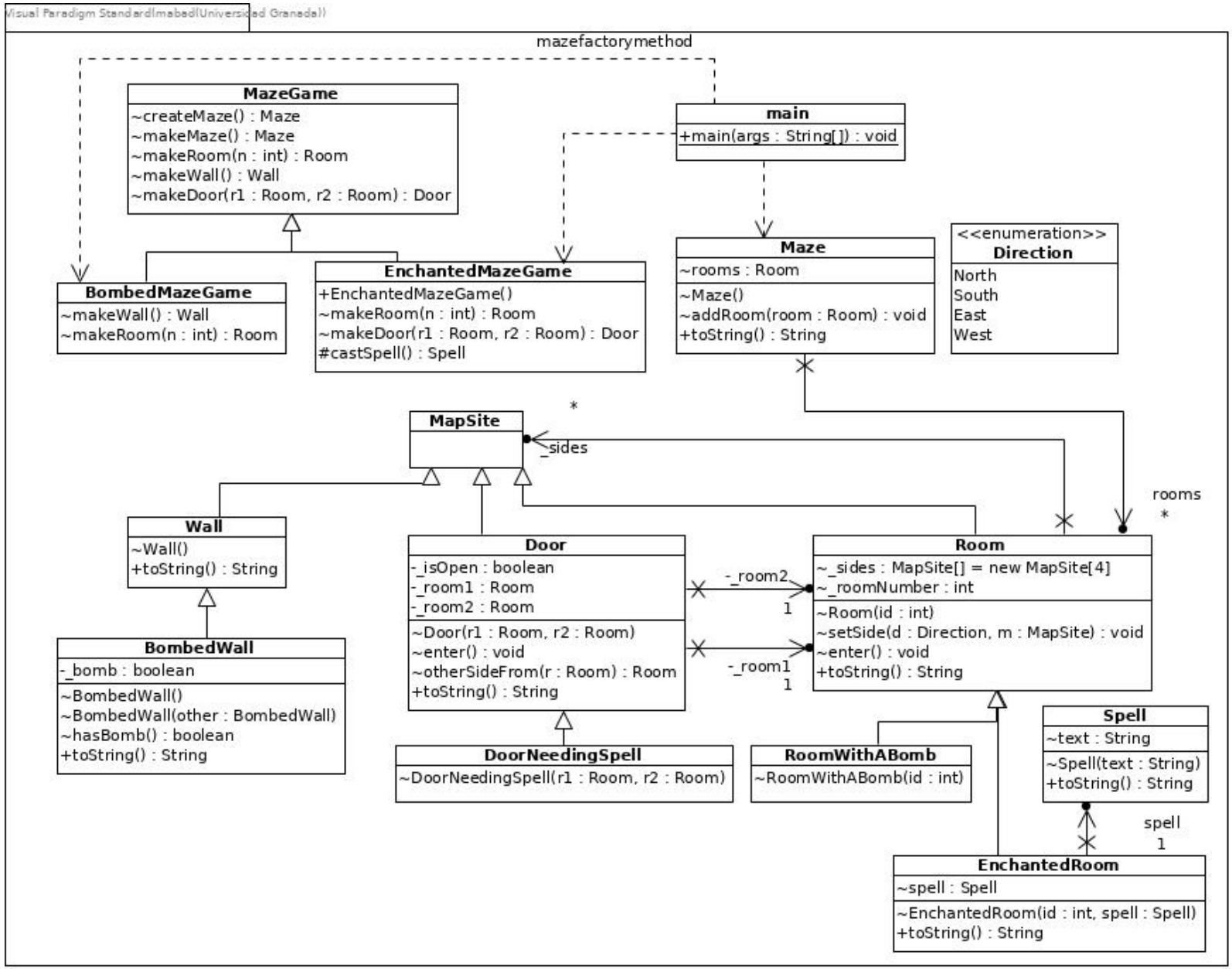
Patrón “Método Factoría”

- Usa métodos factoría en clases (parcialmente) abstractas y los redefine en distintas subclases para crear los objetos de una aplicación o marco de trabajo.
- La redefinición de los métodos factoría en subclases permite poder crear distintas variaciones de una aplicación o marco de trabajo según la combinación de subclases concretas de la misma elegidas.
- Usado sin el patrón “Factoría Abstracta”: cuando no declaramos clases factoría específicas para crear todos los objetos de una línea sino que los métodos factoría pueden agruparse con total flexibilidad.
- Usos extremos
  - Un única interfaz (signatura) del método factoría para crear todos los objetos.
  - Tantos métodos factoría como clases coexistan en una instancia de la aplicación, estando todos en una sola clase que es también el gestor de la aplicación (creará también una instancia de la misma) y que se podrá extender.
- Uso: cuando no sepamos los tipos (clases) de objetos que vamos a crear, o puedan cambiar (extenderse) en el futuro.

Patrón “Método Factoría”. Una sólo interfaz (signatura) del método factoría

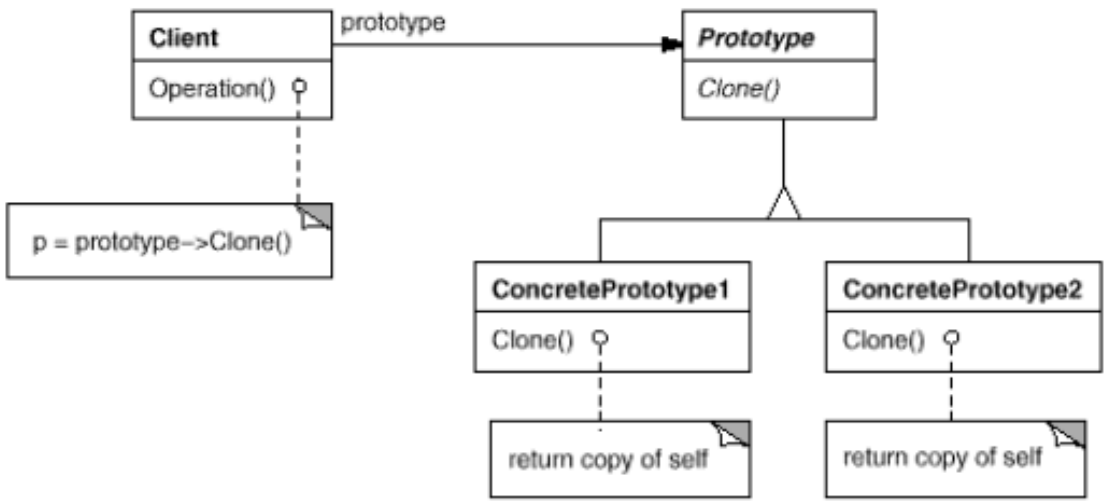


Patrón “Método Factoría” Ejemplo laberinto



Patrón “Prototipo”

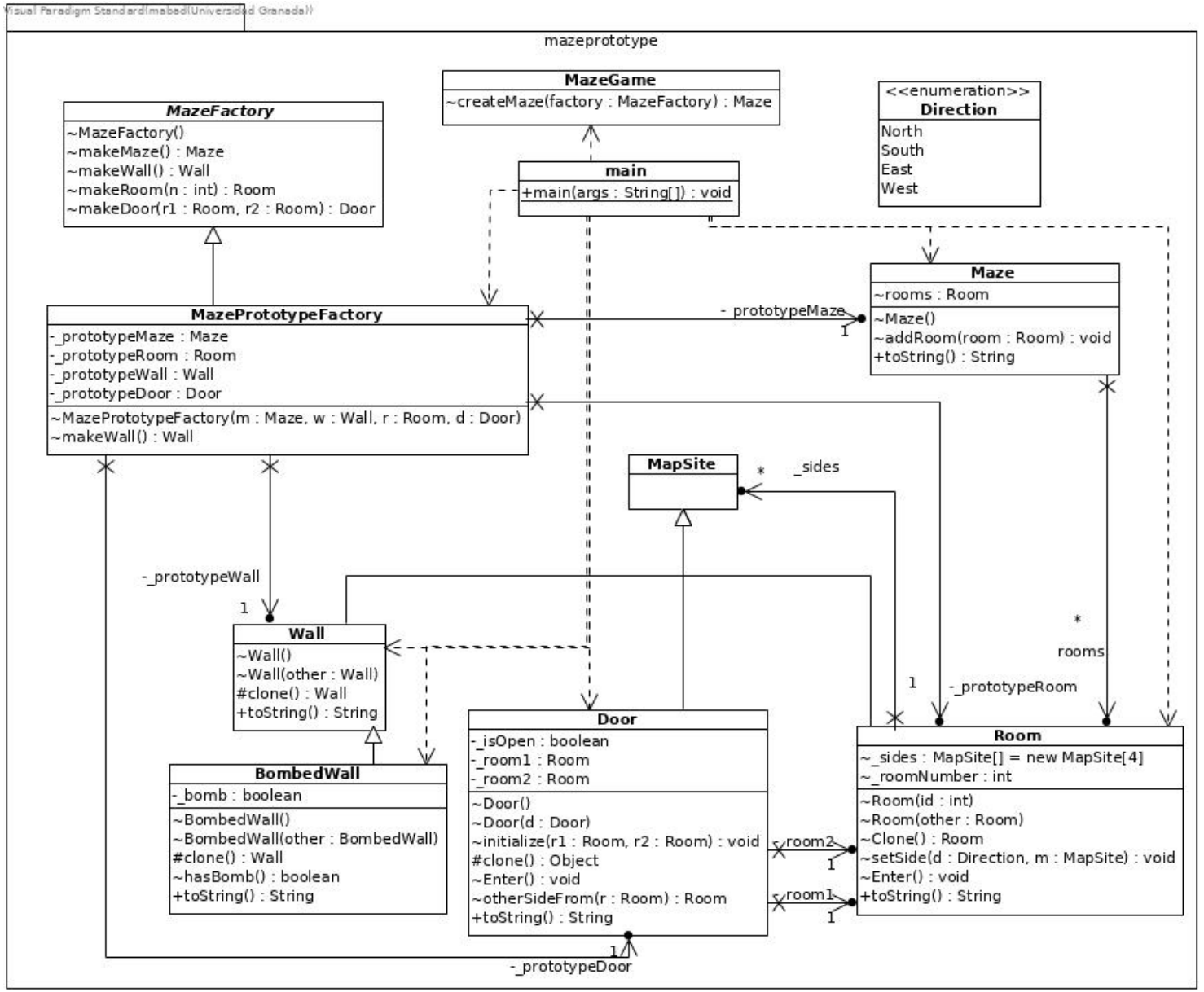
- Cuando se usan métodos factoría que crean por delegación, i.e. usando prototipos o métodos de clonación
- Flexibilidad: No tenemos que crear la jerarquía de clases factoría sino que basta con una sola clase factoría que tenga un sólo método para crear cada objeto dentro de una jerarquía (método “operation” en la Figura 29), que considera que las clases de todos los objetos que se quieren reutilizar en la aplicación heredan todas de la clase “Prototype”



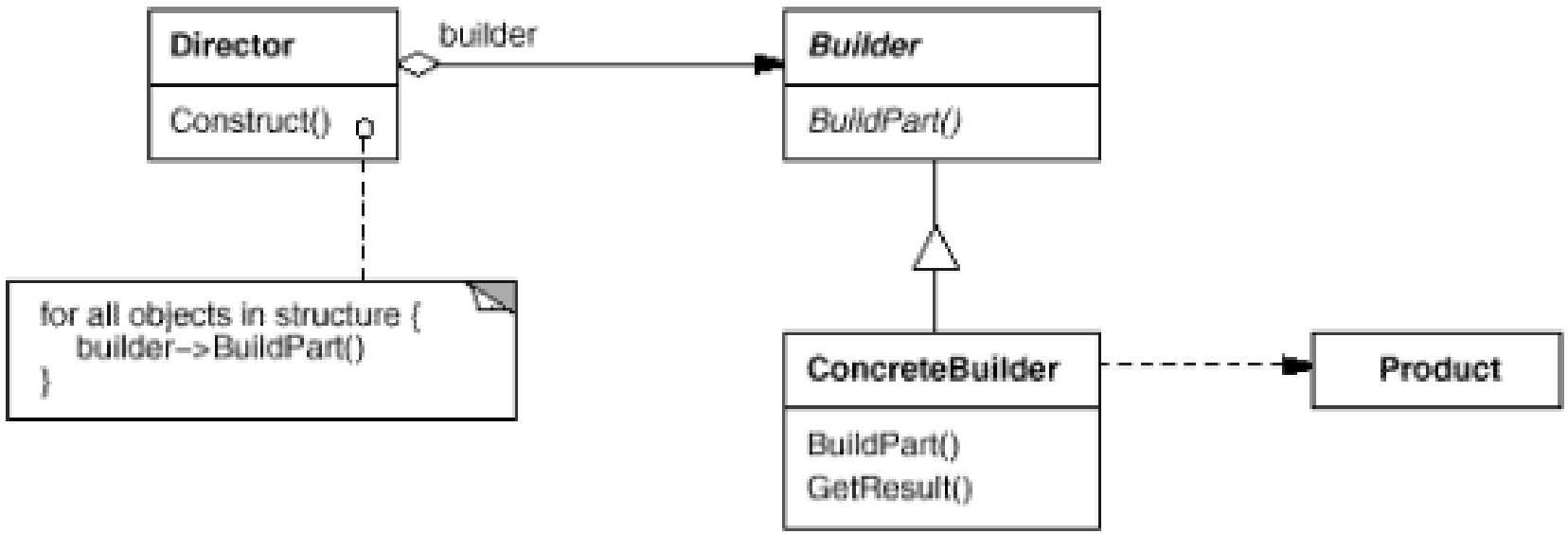
2. Estudio del catálogo GoF de patrones de diseño

Patrones creacionales "Factoría Abstracta", "Método Factoría", "Prototipo" y "Builder"

Patrón "Prototipo" Ejemplo laberinto



Patrón “Builder”



Los cuatro patrones estructurales más básicos y aconsejados como prioritarios por GoF.

CRITERIO DE CALIDAD: Bajo acoplamiento

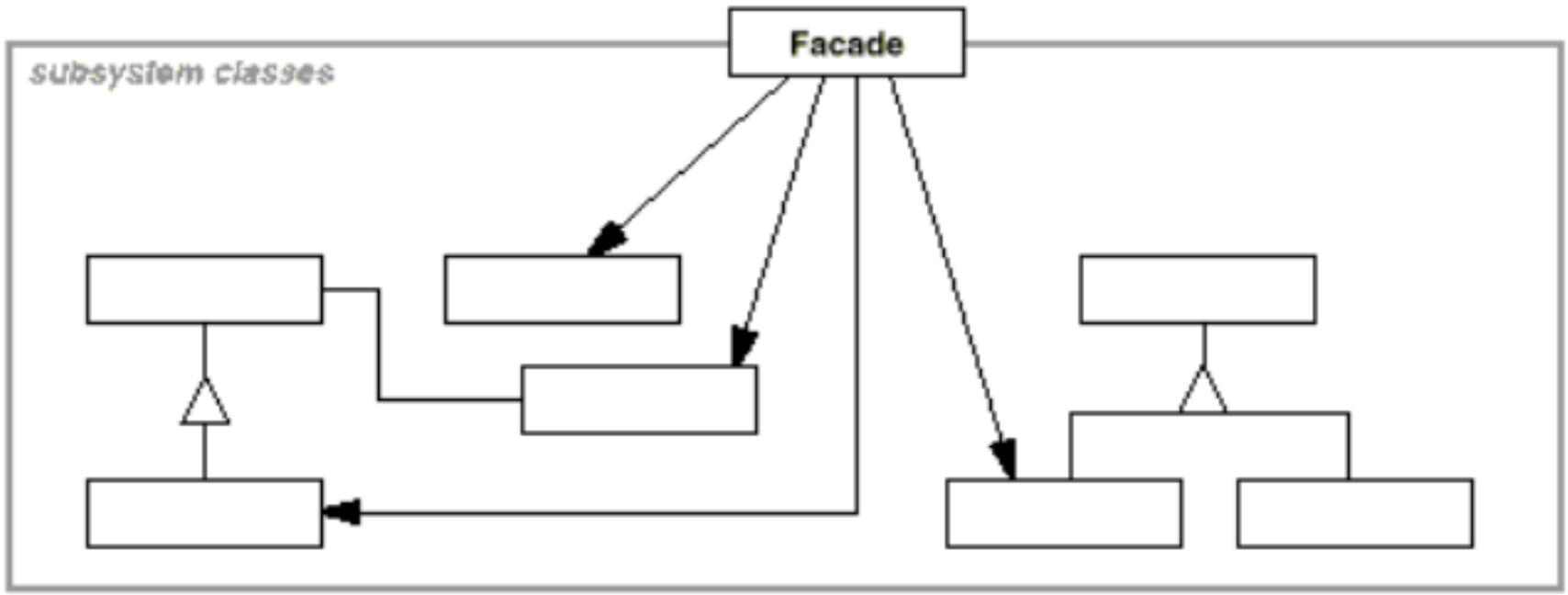
El bajo acoplamiento en diseño OO se refiere a que haya pocas dependencias entre clases pertenecientes a subsistemas distintos.

CRITERIO DE CALIDAD: Alta cohesión

La alta cohesión en diseño OO se refiere a que existan muchas dependencias –alta conectividad– entre clases pertenecientes a un mismo subsistema.

Patrón “Fachada”

Proporciona una interfaz única de un subsistema, reduciendo acoplamiento



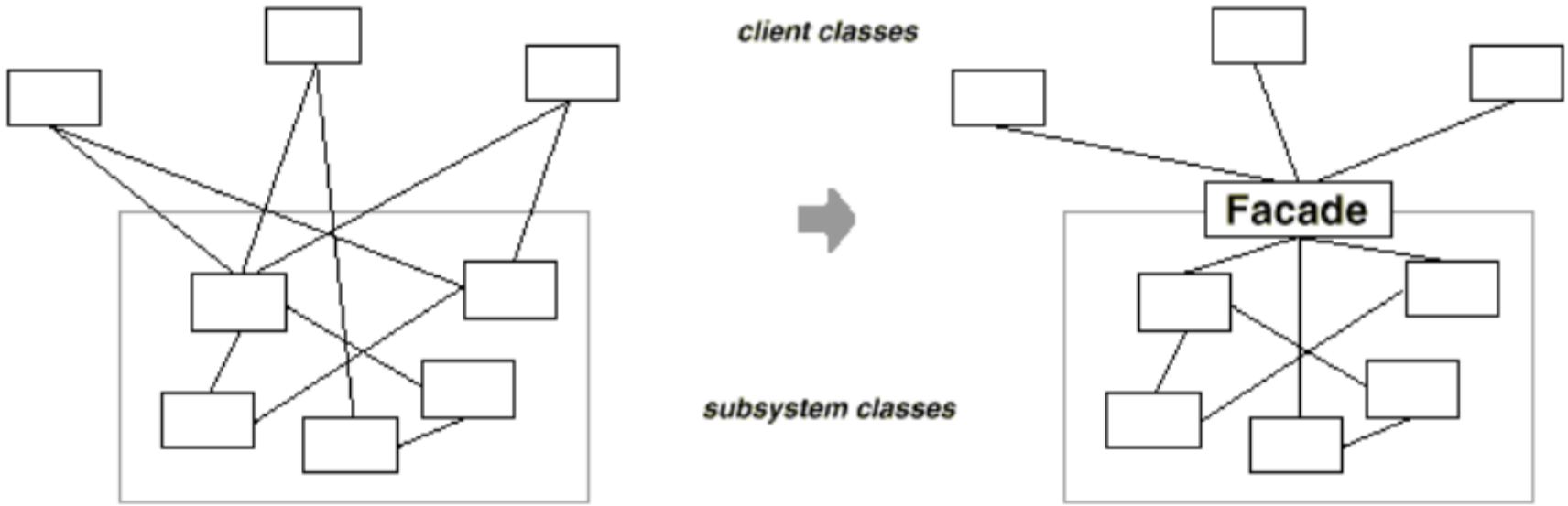


Figura 7: Un esquema de diagrama de clases antes y después de la aplicación del patrón Fachada [Fuente: (Gamma et al., 1994b, pg. 208)].

Patrón “Composición”

Qué hace: Permite tratar a objetos compuestos y simples de la misma forma, mediante una interfaz común a todos, que define un objeto compuesto de forma recursiva.

Cuándo usarlo: cuando el cliente o usuario de esos objetos no quiera distinguir si son compuestos o simples.

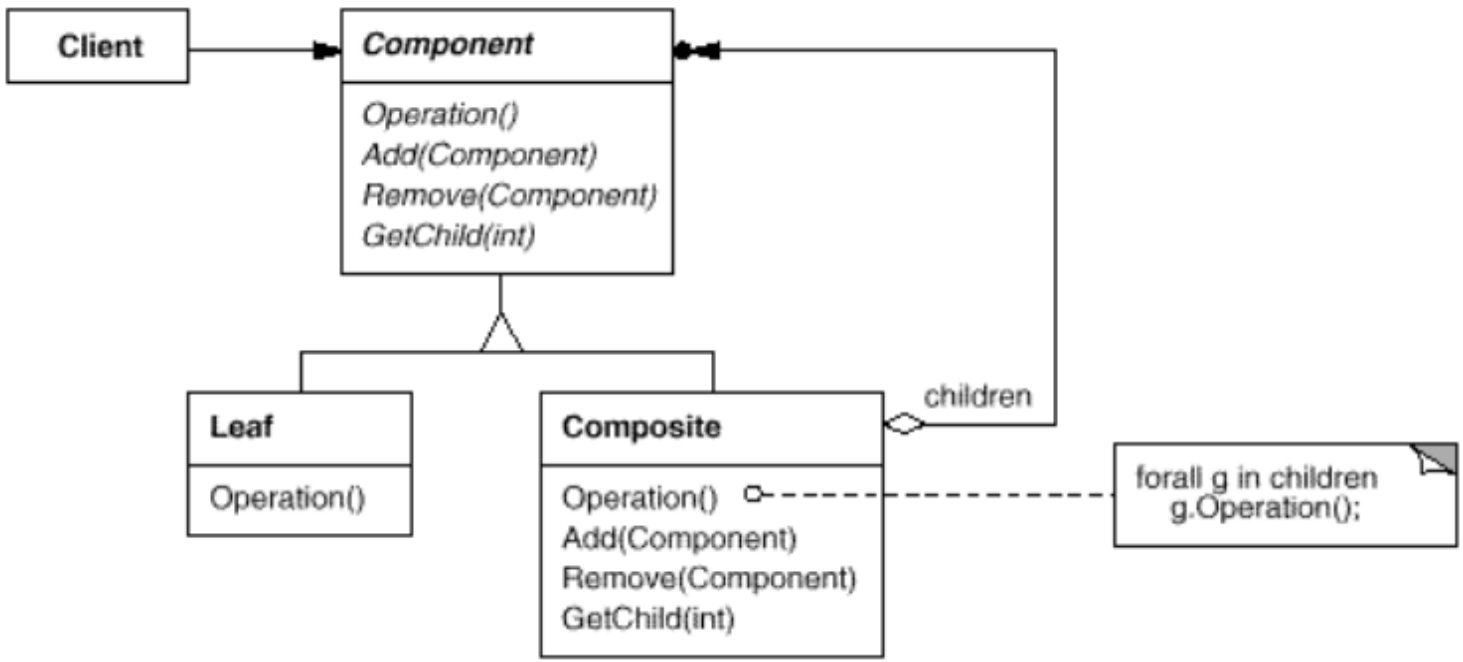


Figura 8: Estructura del patrón Composición [Fuente: (Gamma et al., 1994b), p. 185].

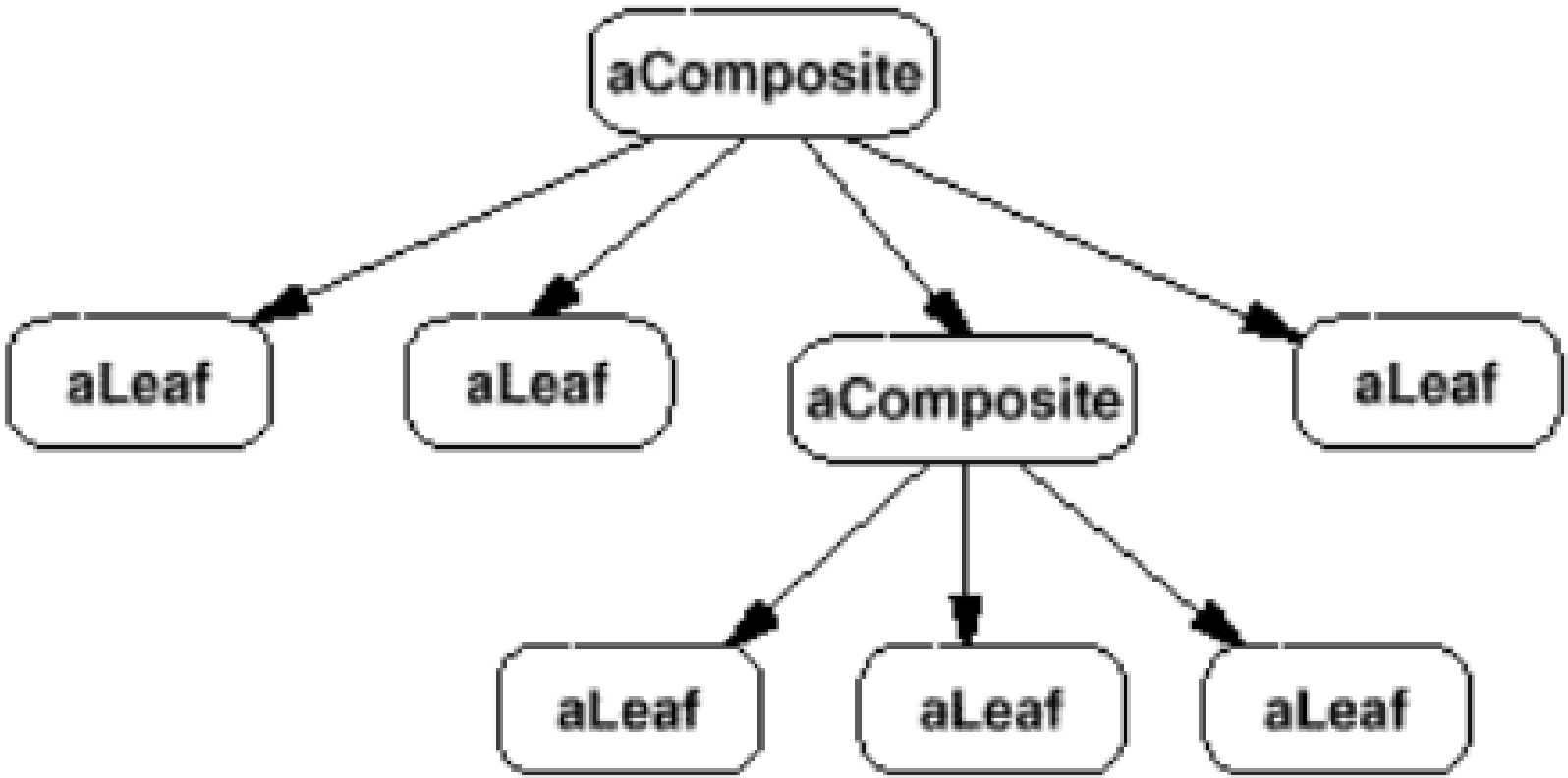


Figura 9: Un ejemplo de jerarquía de objetos donde se ve la relación entre objetos compuestos y simples [Fuente: (Gamma et al., 1994b), p. 185].

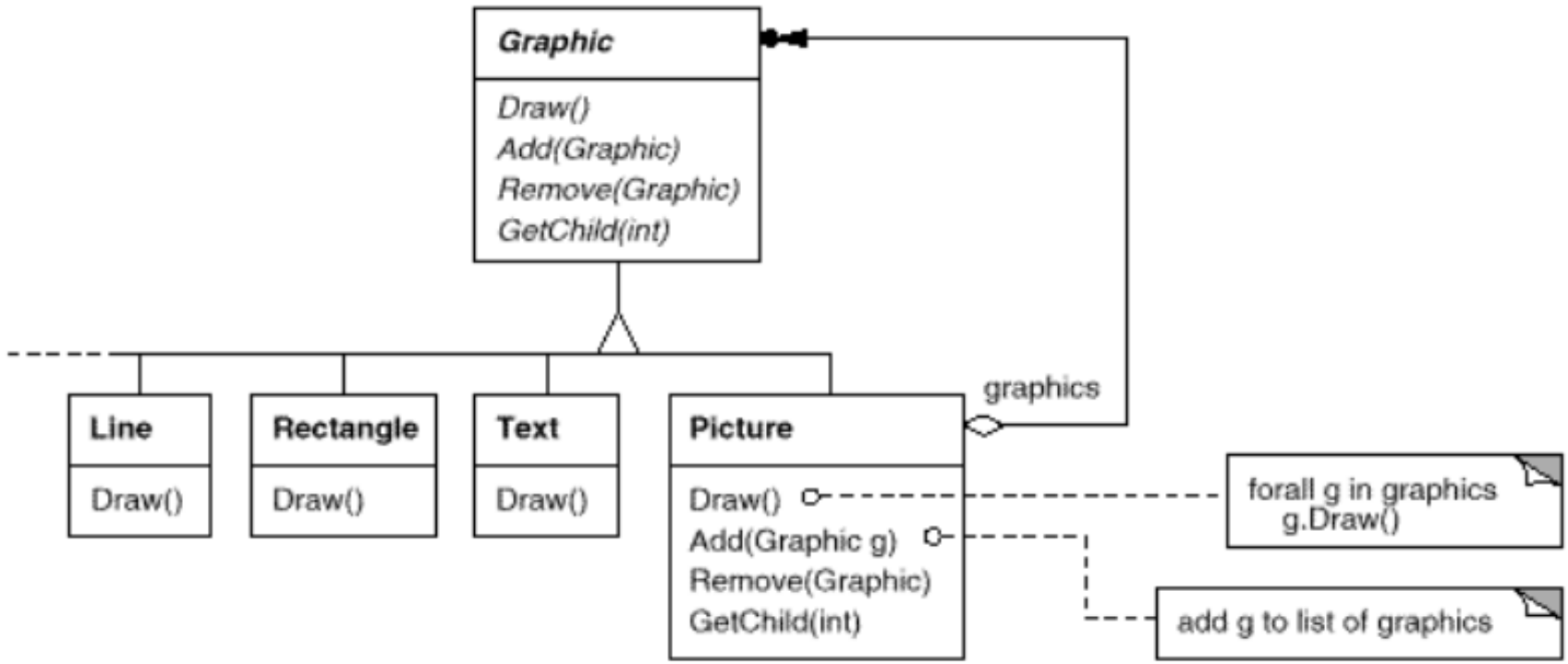


Figura 10: Estructura del patrón Composición en el ejemplo de componentes gráficos. [Fuente: (Gamma et al., 1994b), p. 183].

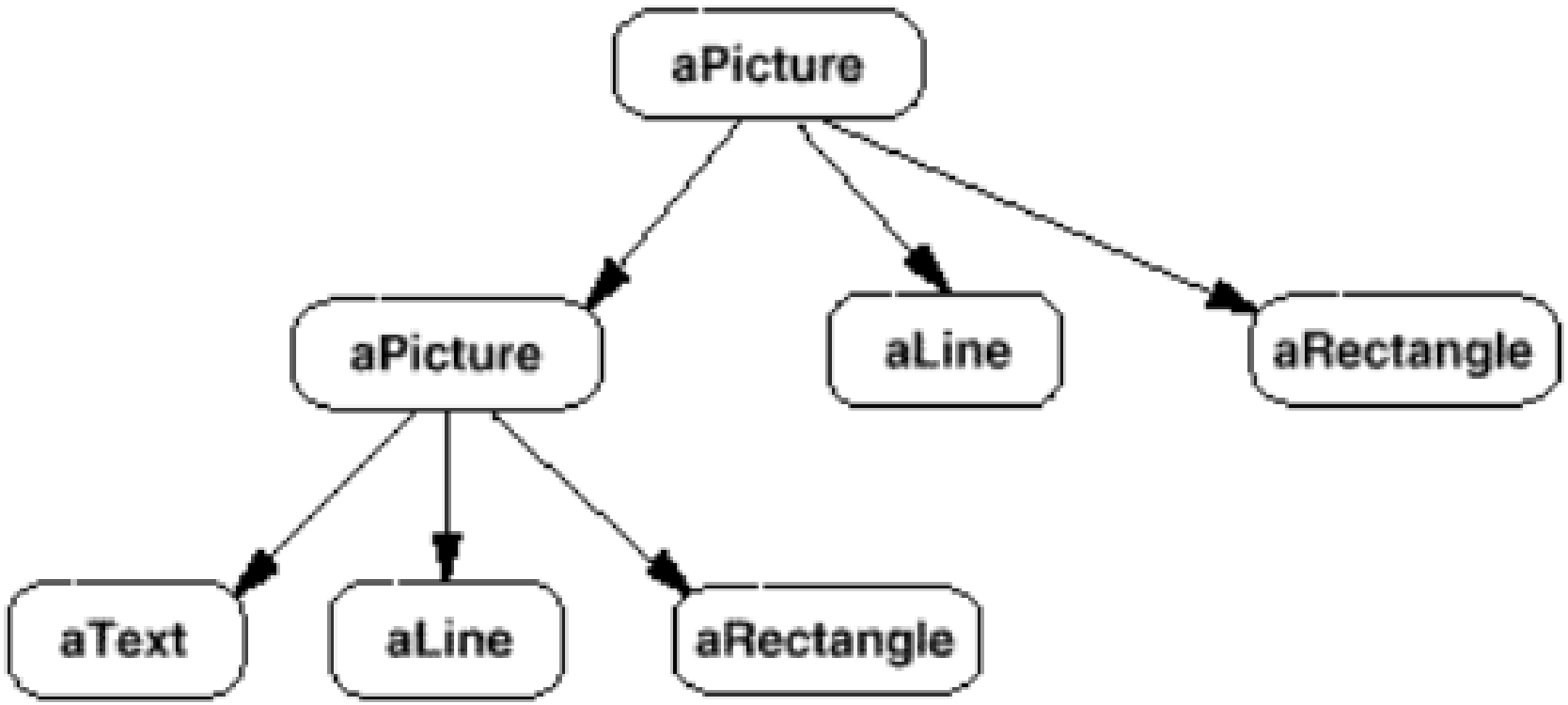


Figura 11: Una de jerarquía de objetos en el ejemplo de componentes gráficos [Fuente: (Gamma et al., 1994b), p. 184].

Qué hace: Proporciona funcionalidad adicional a un objeto de forma dinámica.
Cuándo usarlo: Si queremos dotar de funcionalidad distinta a sólo algunos objetos, especialmente si ésta varía, o cuando no podemos extender las clases.
Ventajas: Flexibilidad Inconvenientes: los sistemas creados con él son más difíciles de depurar y mantener además de ser también más difíciles de aprender a ser usados por otros.

Patrón “Decorador” (o “Envoltorio” –“Wrapper”–)

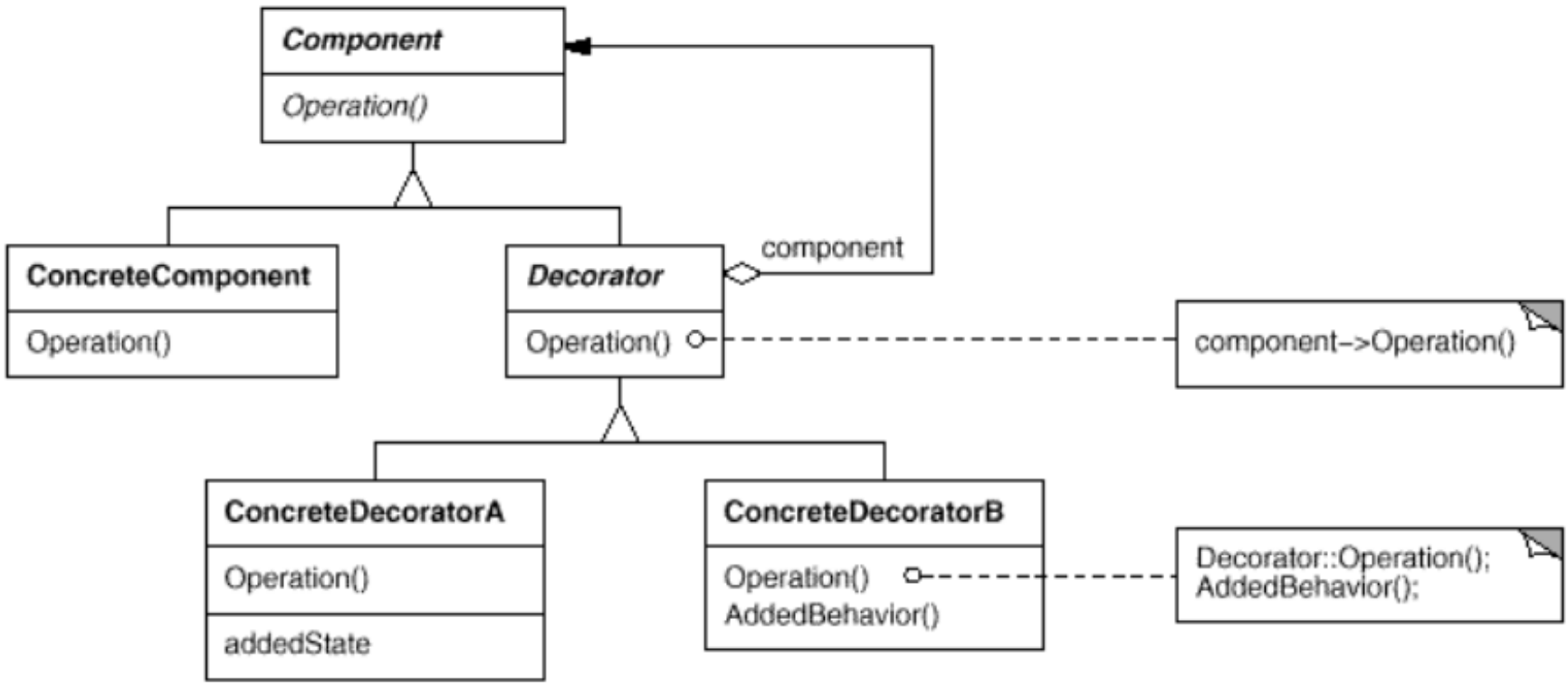


Figura 12: Estructura del patrón decorator [Fuente: (Gamma et al., 1994b), p. 199].

Patrón “Decorador” (o “Envoltorio” –“Wrapper”–)

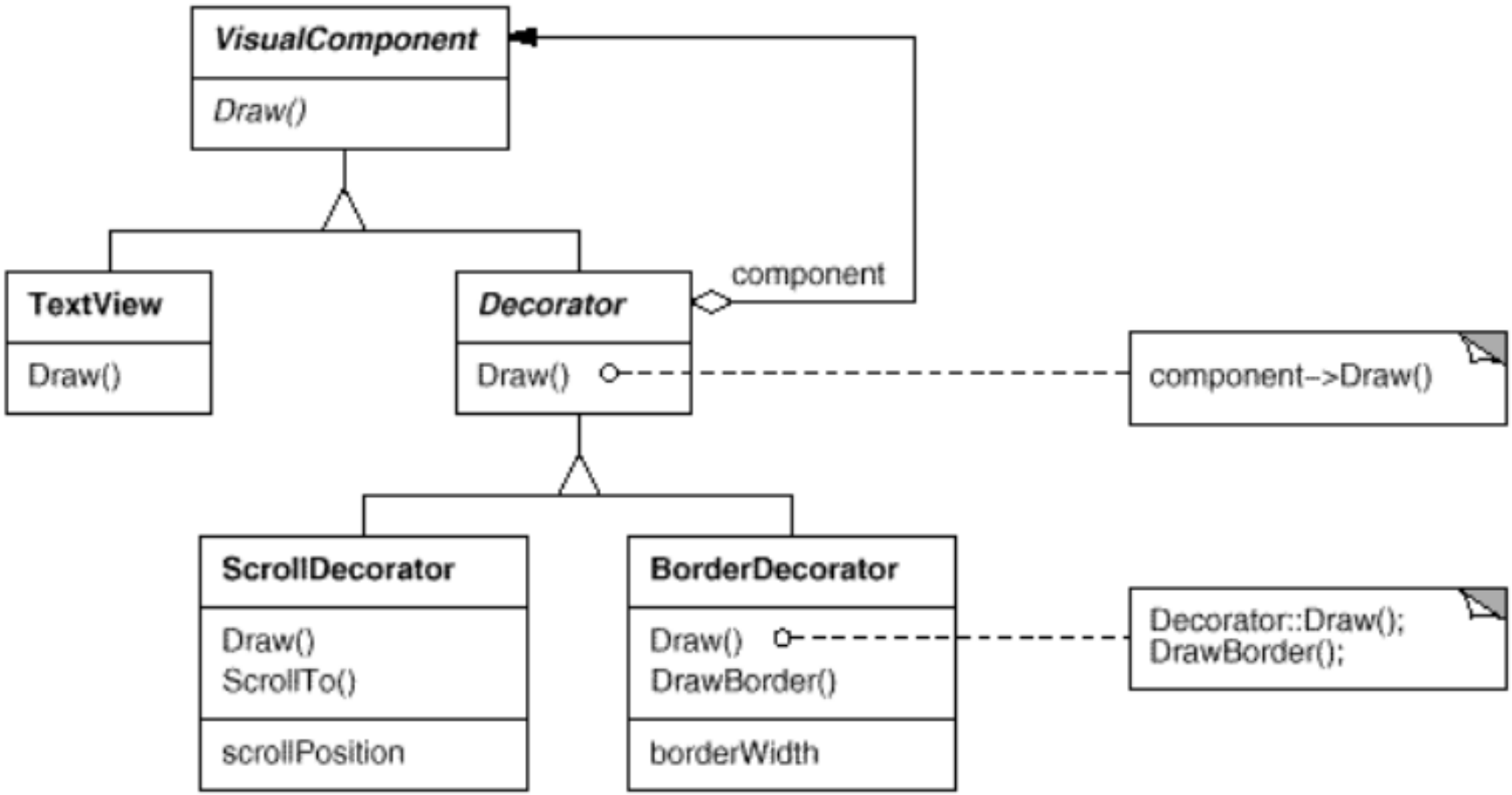


Figura 13: Ejemplo de estructura del patrón decorator aplicada a la presentación gráfica de un texto [Fuente: (Gamma et al., 1994b), p. 198].

Patrón “Decorador” (o “Envoltorio” –“Wrapper”–)

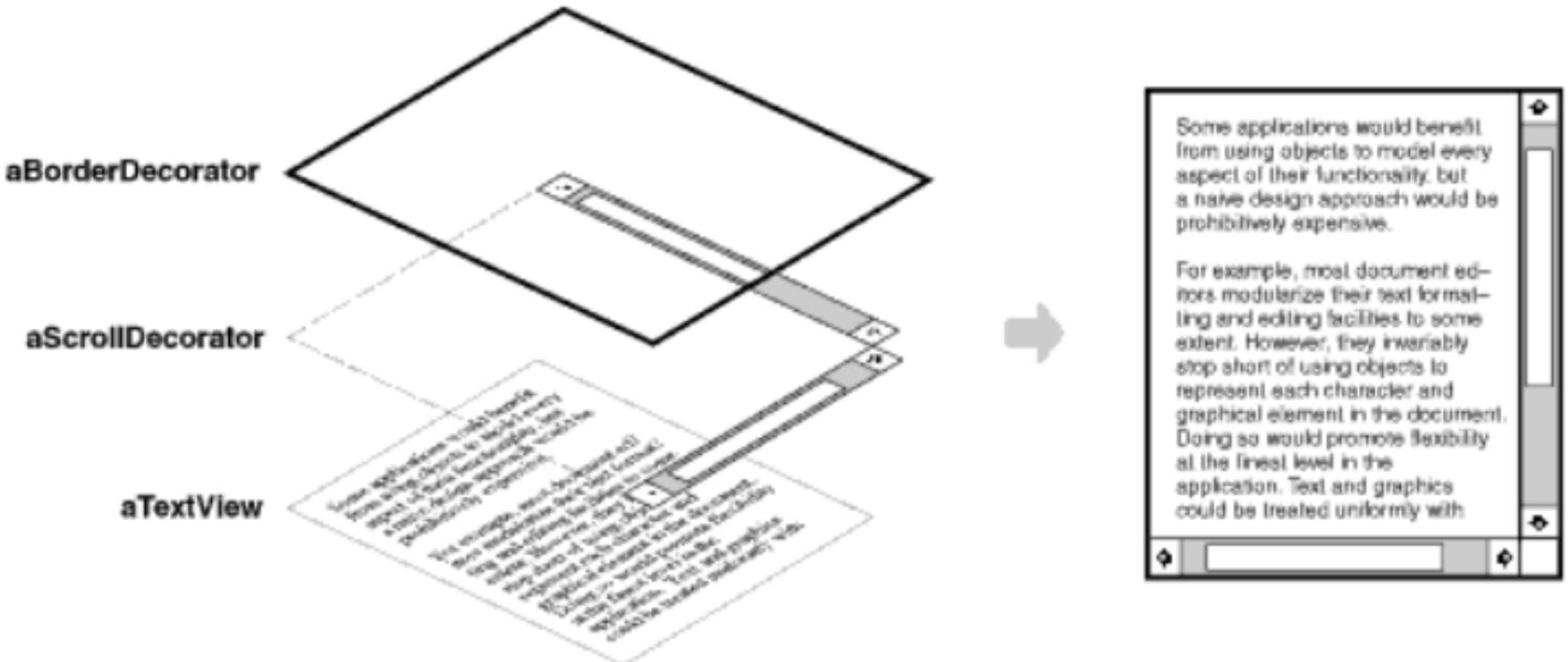


Figura 14: Aspecto de un cuadro de texto y sus decoradores [Fuente: (Gamma et al., 1994b, 197)].

Relación con el patrón “Composición”:

- Estructura parecida
- Funciones diferentes:
  - “Decorador”: añade funcionalidad a los objetos de forma dinámica
  - “Composición”: unifica la interfaz de los objetos compuestos con sus componentes para que el cliente de los mismos pueda tratarlos de la misma forma

Pueden coexistir

Qué hace: Convierte la interfaz de una clase en otra que se adapte a lo que el cliente esperaba para que pueda así usar esa clase. Cuándo usarlo:

- En una fase de diseño avanzada, con clases/módulos terminados, con dependencias de otro módulo sin hacer aún. Podemos reusar un módulo externo pero no debemos/podemos cambiar nuestro código ni tampoco el de las clases a usar
- En las primeras etapas del diseño, cuando se prevé que se pueda querer usar código externo para proveer parte de la funcionalidad del software que se está desarrollando, y en el futuro se puede cambiar el código externo elegido para ser reutilizado

Versiones:

- Patrón “Adaptador” en el ámbito de clases
- Patrón “Adaptador” en el ámbito de objetos

Patrón “Adaptador”

Ámbito de clase

Requiere del uso de herencia múltiple

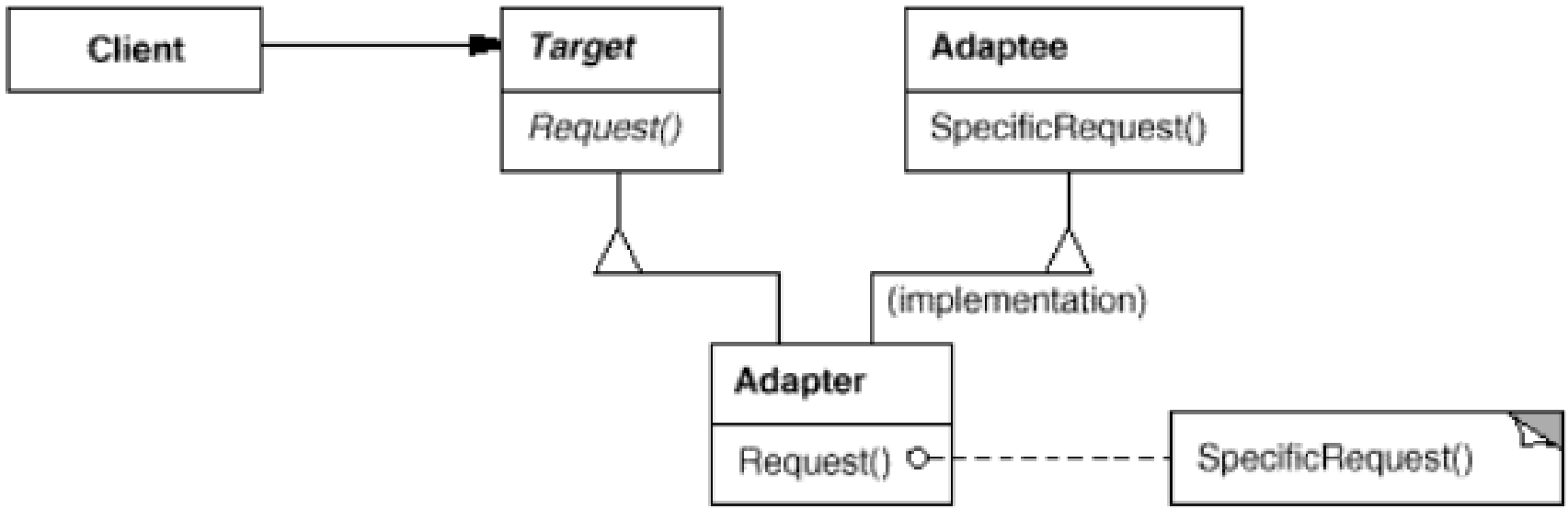


Figura 15: Estructura del patrón adaptador, en un ámbito de clase [Fuente: (Gamma et al., 1994b, pg. 159)].

Más indicada si queremos usar una gran variedad de subclases ya existentes

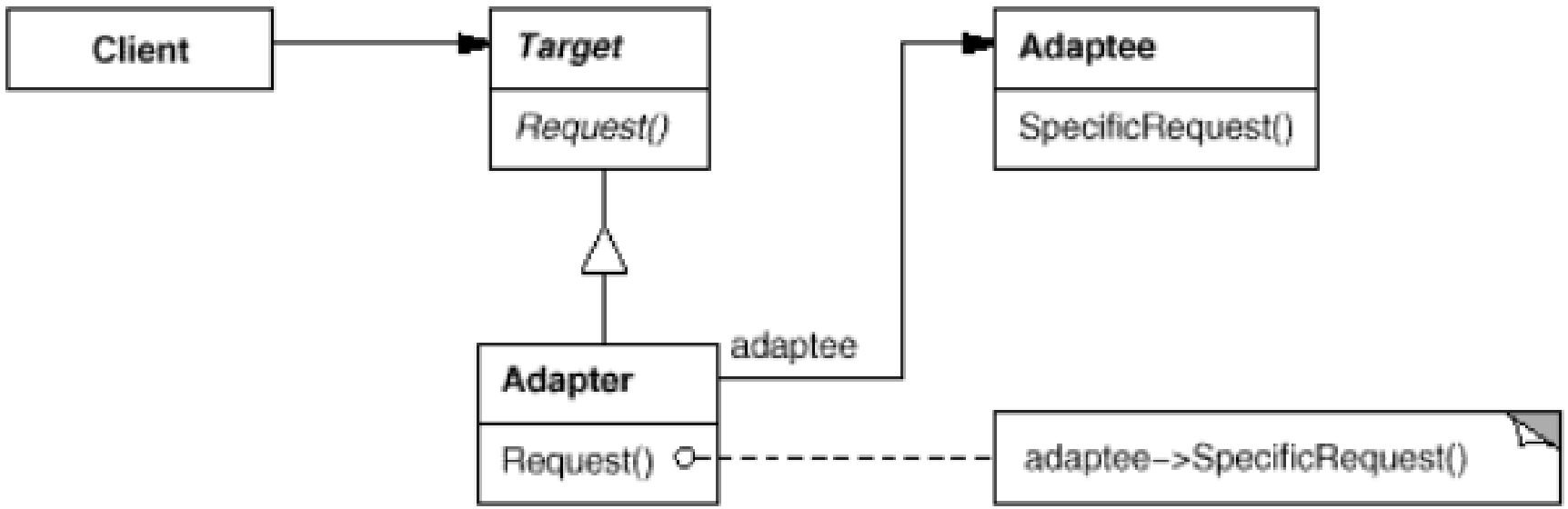


Figura 16: Estructura del patrón adaptador en un ámbito de objeto [Fuente: (Gamma et al., 1994b, pg. 159)].

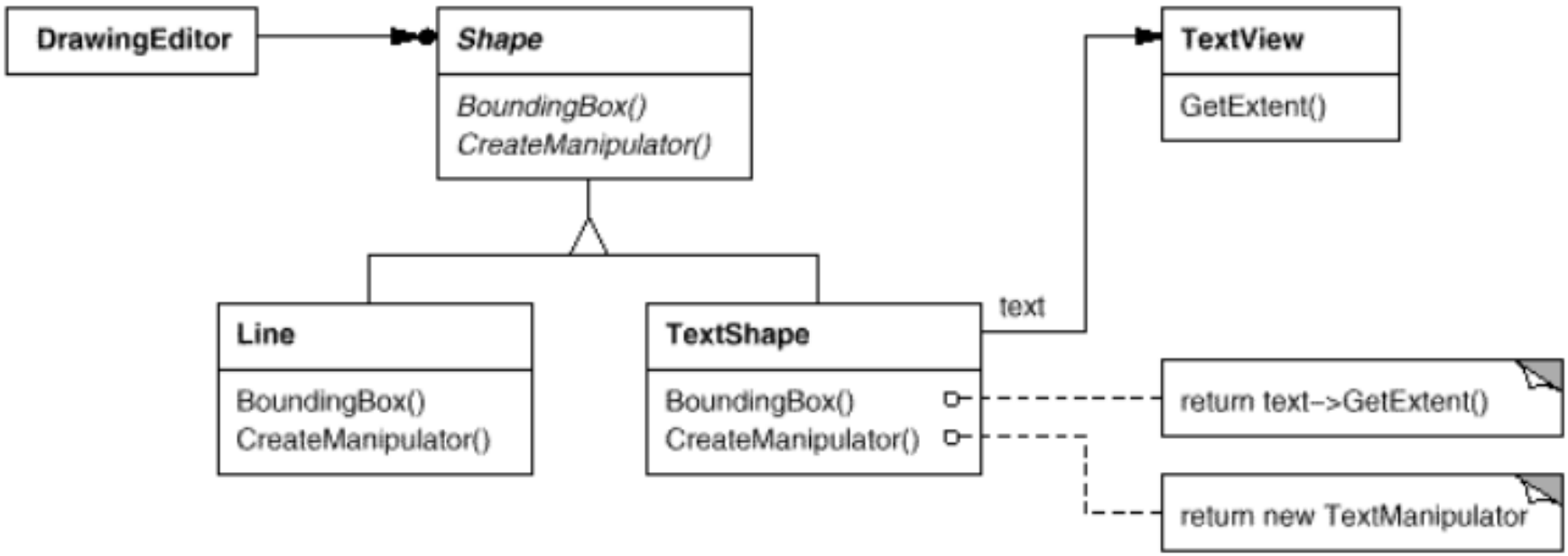


Figura 17: Estructura del patrón “Adapter” aplicado a un editor de dibujo [Fuente: (Gamma et al., 1994b, pg. 158)].

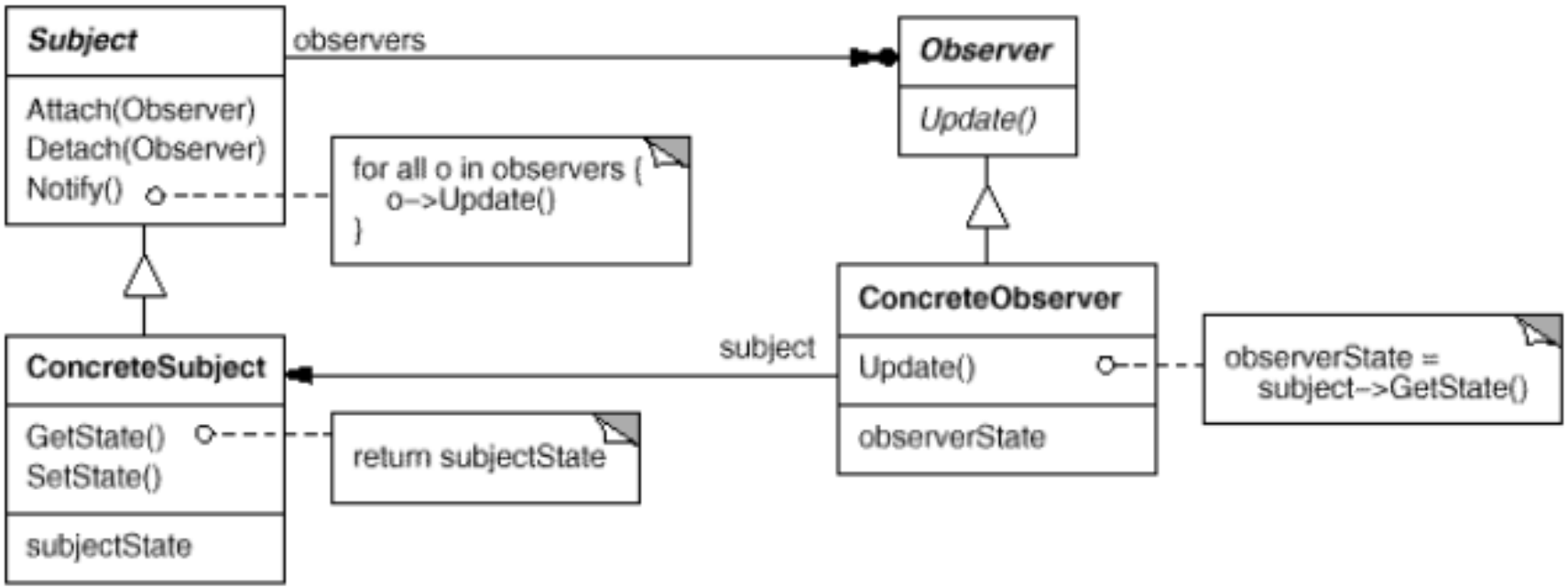
Patrón “Observador”

- Otros nombres “Dependientes” o “Publicar-Subscribir”
- Cuando un objeto (“sujeto observable”) tiene varios objetos que dependen de él, garantiza la consistencia entre ellos manteniendo un bajo acoplamiento
- Funcionamiento:
  - No hay comunicación directa entre el sujeto observable y sus observadores
  - Cada cambio en el estado del “sujeto observable” es notificado mediante publicación de modo que todos los “observadores”, objetos subscritos, reciben esa notificación en cuanto es publicada
  - Se pueden subscribir cualquier número de observadores a la lista de subscripción del sujeto observable.
- Suele usarse con el diseño MVC (patrón arquitectónico)

2. Estudio del catálogo GoF de patrones de diseño

Patrones conductuales "Observer", "Visitor", ' "Strategy", "Template Method" e "Intercepting Filter"

Patrón "Observador"

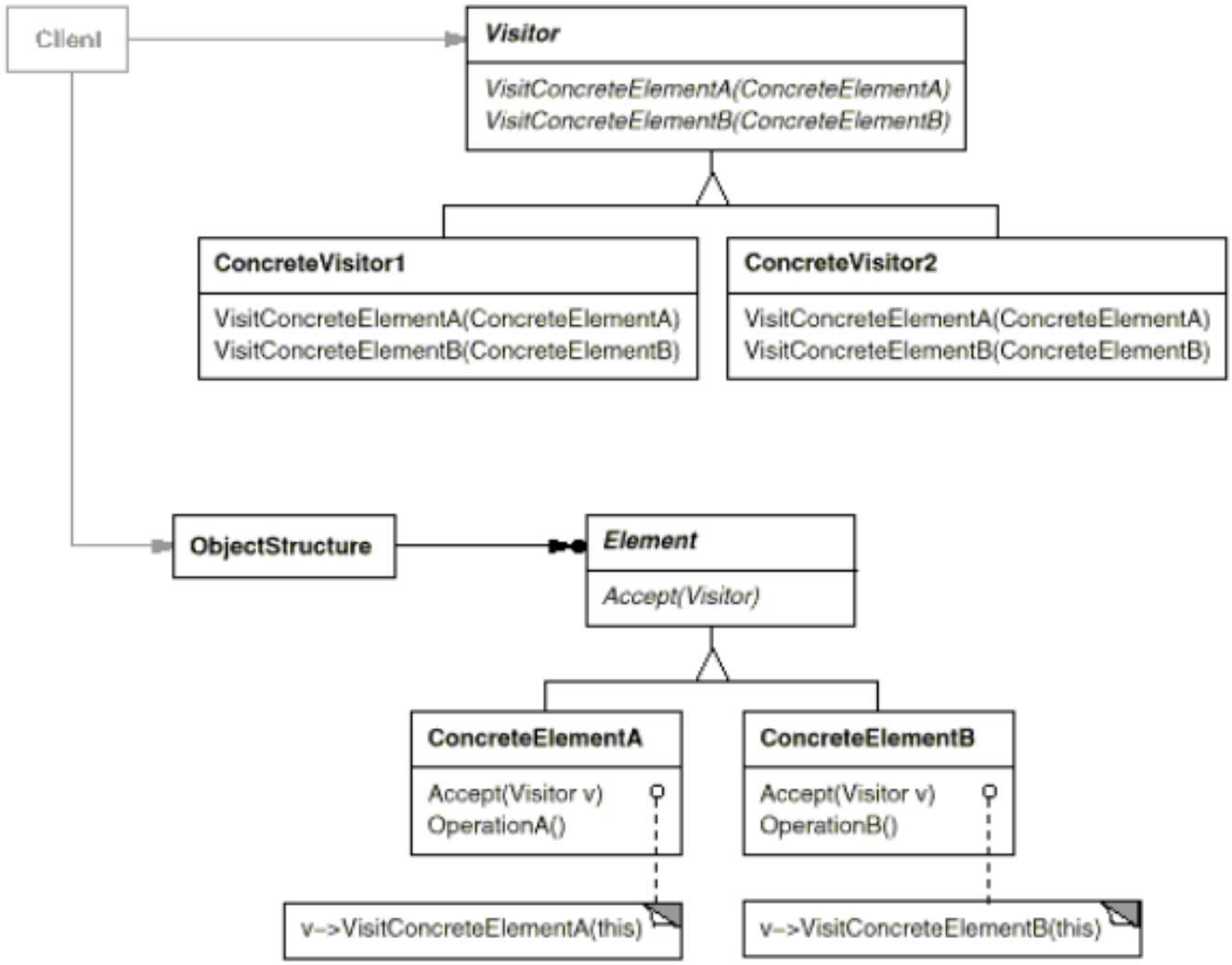


- Separar un algoritmo de la estructura de un objeto
- La operación se implementa de forma que no se modifique el código de las clases que forman la estructura del objeto
- Usado si se quiere realizar un cierto número de operaciones, que no están relacionadas entre sí, sobre instancias de un conjunto de clases

2. Estudio del catálogo GoF de patrones de diseño

Patrones conductuales "Observer", "Visitor", ' "Strategy", "Template Method" e "Intercepting Filter"

Patrón "Visitante"



Qué hace: Define una familia de algoritmos, con la misma interfaz y objetivo, de forma que el cliente puede intercambiar el algoritmo que usa en cada momento Cuándo usarlo:

- Cuando se prevé que la lista de algoritmos alternativos pueda ampliarse, y/o
- cuando son muchos y no siempre se usan todos.

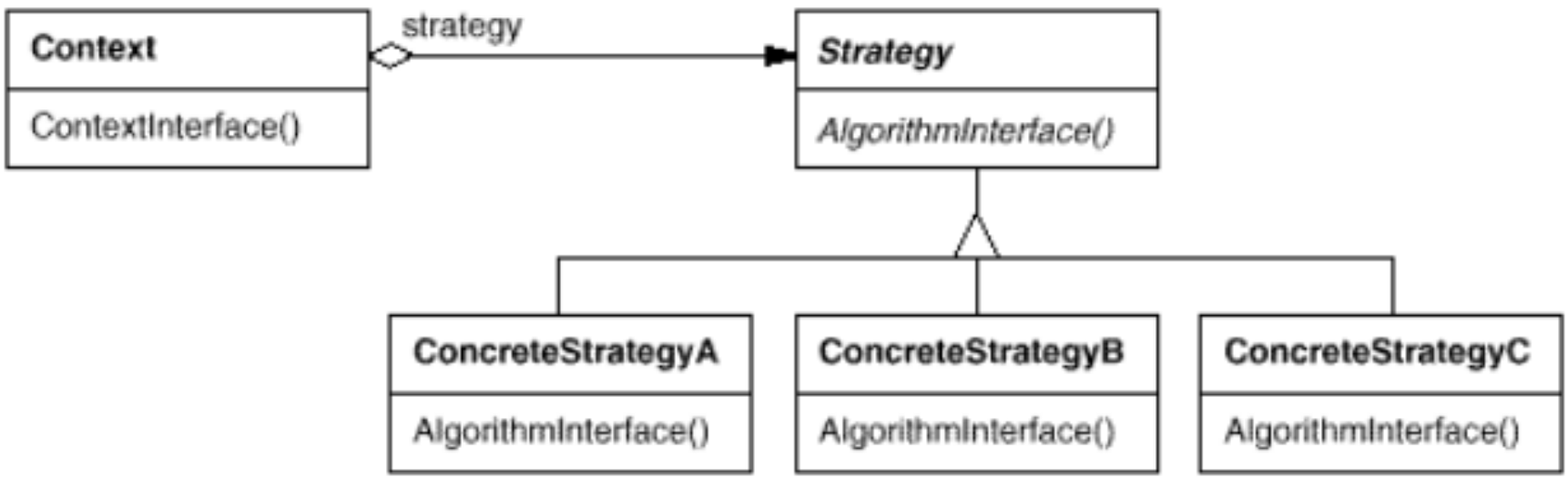


Figura 18: Estructura del patrón “Estrategia” [Fuente: (Gamma et al., 1994b, pg. 351)]

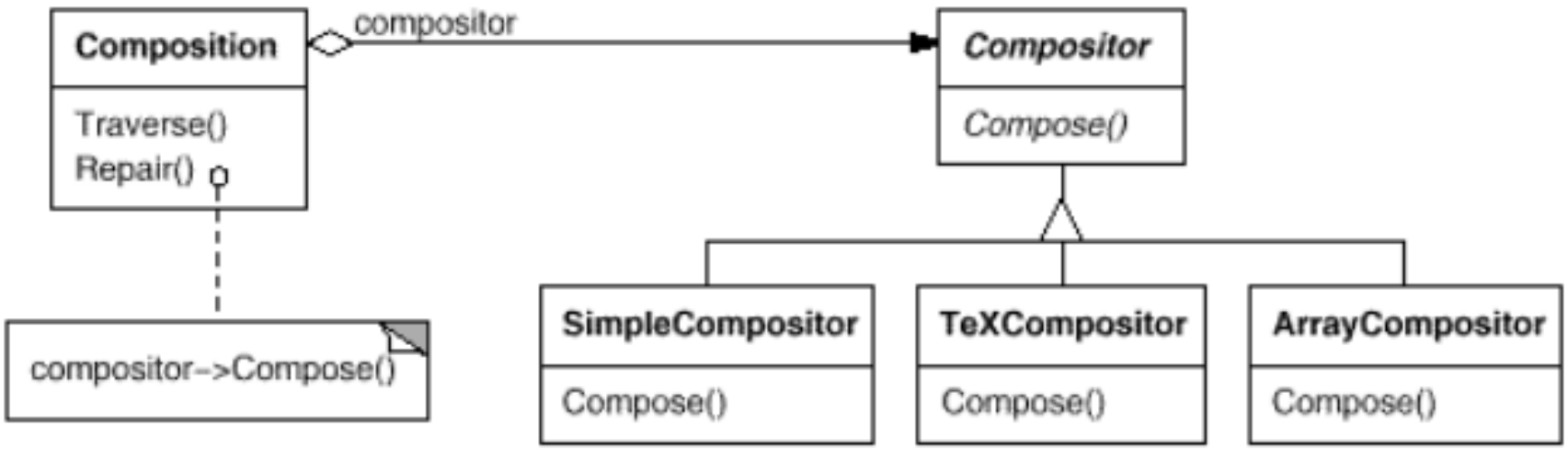


Figura 19: Estructura del patrón “Estrategia” en un ejemplo de visualización de textos [Fuente: (Gamma et al., 1994b, pg. 349)]

Patrón "Método Plantilla"

Qué hace: Técnica básica muy importante de reutilización de código mediante inclusión de un "método plantilla", método concreto de una clase abstracta, con las siguientes características:

- Implementa una secuencia de pasos (métodos) usados por un algoritmo de la clase
- Deja la implementación de cada uno de los métodos concretos a las subclases

Los métodos plantilla pueden llamar a distintos tipos de operaciones (Gamma et al., 1994b, pg. 363)]:

- Métodos concretos de las subclases *ConcreteClass*
- Métodos implementados en la clase *AbstractClass*
- Métodos abstractos (declarados pero no implementados) en la clase *AbstractClass*)
- Métodos factoría
- Métodos gancho: aquéllos implementados en la clase abstracta (aunque a menudo no hacen nada) que pueden ser sobreescritos en las subclases

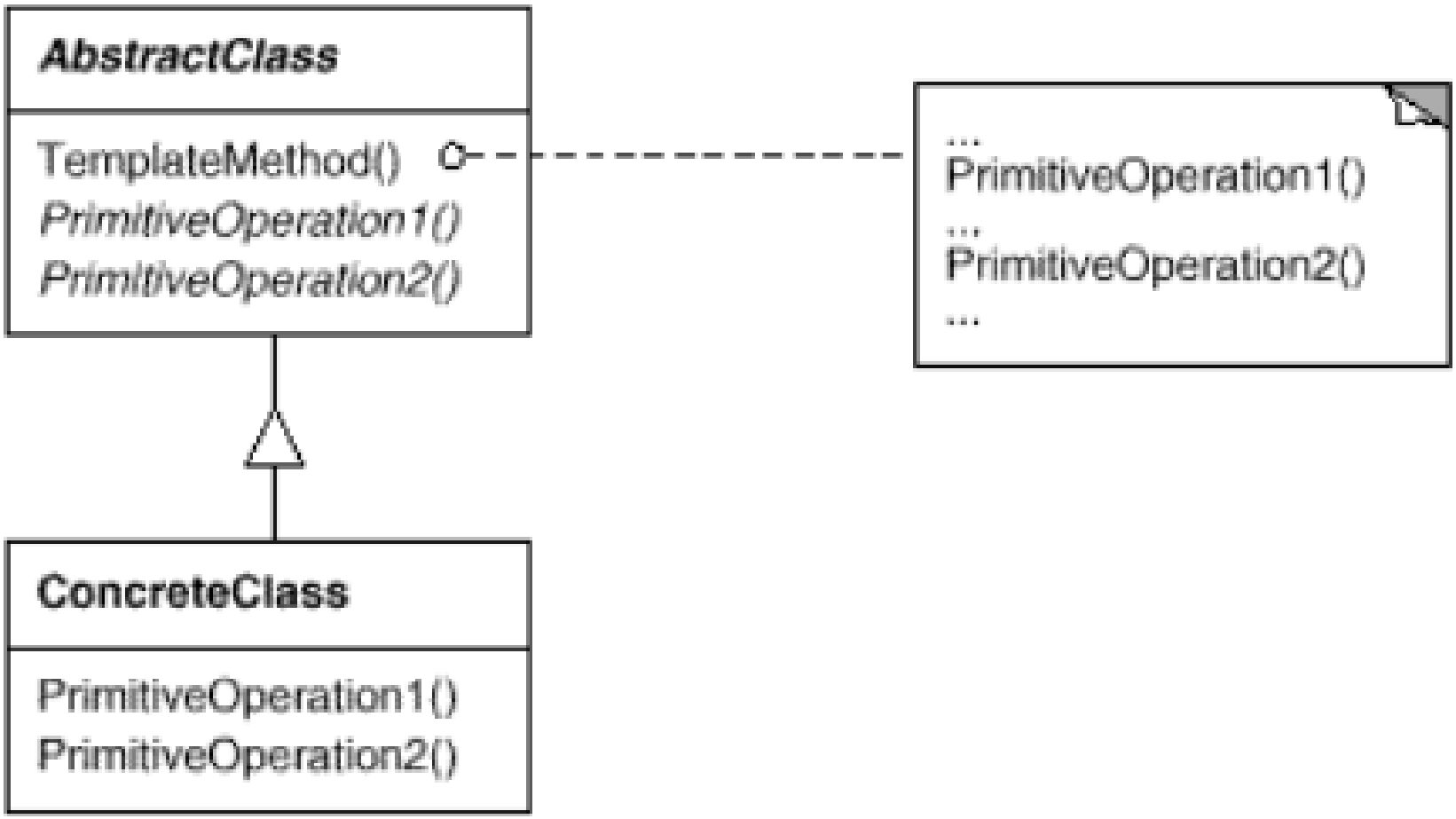


Figura 20: Estructura del patrón “Método Plantilla” [Fuente: (Gamma et al., 1994b, pg. 362)]

Evita que los métodos de las subclases que extienden a los de la clase olviden llamar al método que extienden (Gamma et al., 1994b, pg. 363)
Extensión normal:

```
void DerivedClass::operation () {
// DerivedClass extended behavior
// ...
ParentClass::operation();
}
```

Usando un método gancho *hookOperation* que se llama desde el método *operation* de la clase padre, no hay que llamar al método *operation* al desde la subclase:

```
void ParentClass::operation () {
// ParentClass behavior
hookOperation();
}
```

hookOperation no hace nada en la clase padre:

```
void ParentClass::hookOperation () { }
```

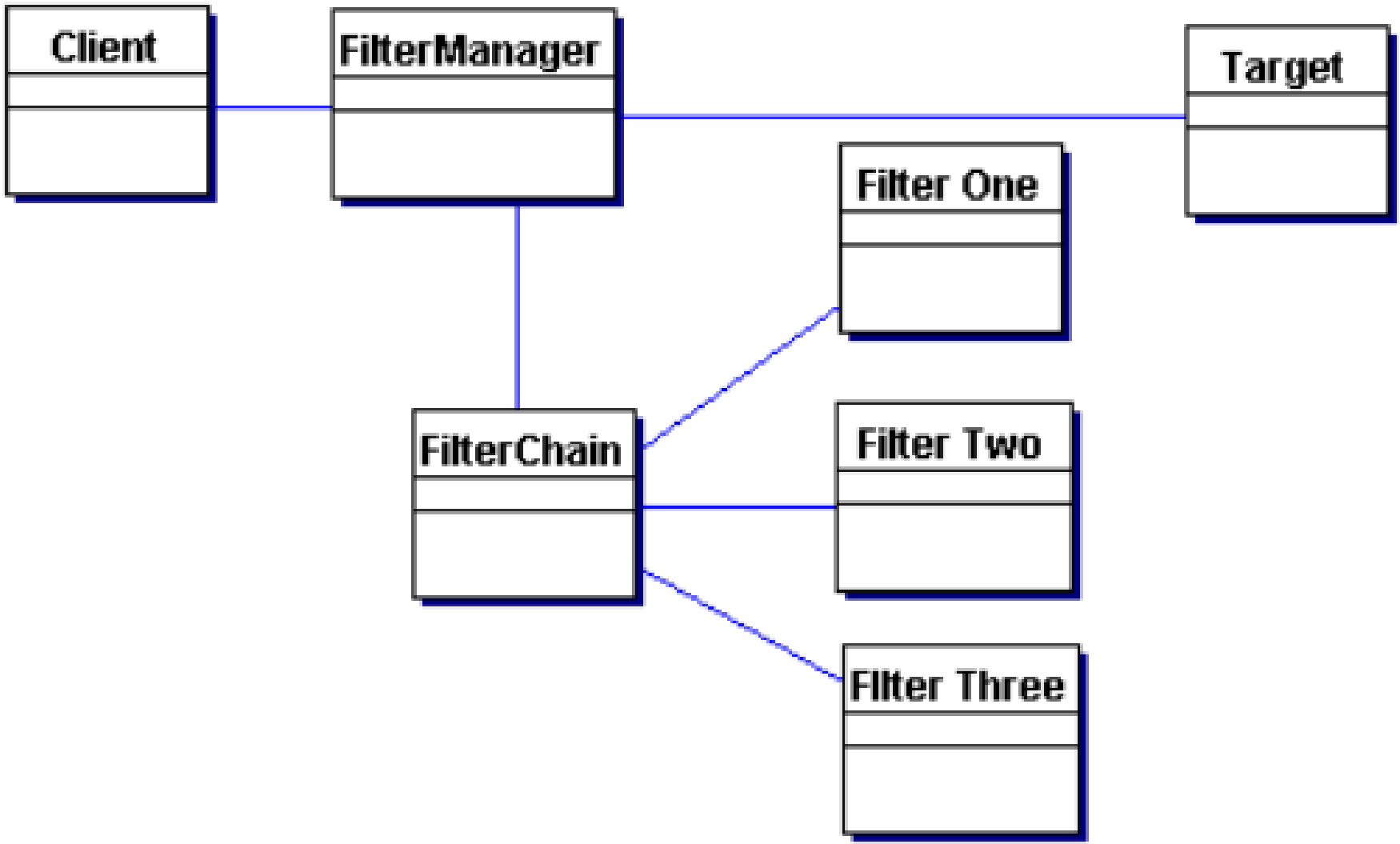
Las subclases lo extienden:

```
void DerivedClass::hookOperation () {
// derived class extension
// ...
}
```

Patrón “Filtros de Intercepción”

- Une en uno dos patrones de diseño:
  - Patrón Interceptor: añade servicios de forma transparente que puedan ser iniciados de forma automática
  - Patrón Filtros de encauzamiento: encadena servicios de forma que la salida de uno sea el argumento de entrada del siguiente. Puede añadir un componente de filtrado:
    - antes de la petición de un servicio (pre-procesamiento)
    - después de su respuesta (post-procesamiento)
- Los filtros deben poderse añadir y quitar sin afectar al resto del código
- Aunque los filtros generalmente implican que la petición del servicio se cancele si no se pasa el filtro, a veces pueden simplemente añadir funcionalidad dejando siempre pasar al servicio principal
- Los filtros se programan y se utilizan cuando se produce la petición y antes de pasar tal petición a la aplicación “objetivo” que tiene que procesarla.Por ejemplo, los filtros pueden realizar la autenticación/autorización/conexión(login) o trazar la petición antes de pasarla a los objetos gestores que van a procesarla

Patrón “Filtros de Intercepción”



- *Objetivo* (target): Es el objeto que será interceptado por los filtros
- *Filtro*: Interfaz (clase abstracta) que declara el método *ejecutar* que todo filtro deberá implementar. Los filtros que implementan la interfaz se aplicarán antes de que el objetivo (objeto de la clase *Objetivo*) ejecute sus tareas propias (método *ejecutar*)
- *Cliente*: Es el objeto que envía la petición a la instancia de *Objetivo*, pero no directamente, sino a través de un gestor de filtros (*GestorFiltros*) que envía a su vez la petición a un objeto de la clase *CadenaFiltros*
- *CadenaFiltros*:Tiene una lista con los filtros a aplicar, ejecutándose en el orden en que son introducidos en la aplicación. Tras ejecutar esos filtros, se ejecuta la tarea propia del objetivo (método *ejecutar* de la clase *Objetivo*), todo dentro del método *ejecutar* de *CadenaFiltros*
- *GestorFiltros*: Se encarga de gestionar los filtros: crea la cadena de filtros y tiene métodos para añadir filtros concretos y que se ejecute la petición por los filtros y el “objetivo” (método *peticionFiltros*)

Tareas a realizar: Partiendo de los casos prácticos presentados por los miembros del grupo pequeño[1]

1. Ejercicio en grupo pequeño:
   - 1.1 Identificad patrones de diseño que podrían aplicarse. Deben excluirse: Factoría Abstracta, Método Factoría, Prototipo y Visitante
   - 1.2 Seleccionad uno de ellos y dibujad un diagrama de clases adaptado a la parte de la aplicación en donde se aplique el patrón
   - 1.3 Repartid la posible puesta en común del ejercicio entre los cuatro componentes del grupo de la siguiente forma:
     - 1.3.1 Explicación del caso práctico: requisitos funcionales
     - 1.3.2 Explicación del caso práctico: requisitos no funcionales (interfaz de usuario, lenguaje de programación, sistema operativo, memoria, tiempo de respuesta, tipos de usuarios, etc.)
     - 1.3.3 Problema concreto que se quiere resolver y patrones de diseño considerados
     - 1.3.4 Diagrama de clases y explicación de la aplicación del patrón de diseño elegido
2. Puesta en común en grupo grande:
   - Elegido un grupo, expondrá cada miembro en secuencia la parte que le han asignado sus compañeros
   - Limitación del tiempo de exposición: 2 minutos máximo por persona; 8 minutos máximo para todo el grupo

[1] Sobre la formación de los grupos pequeños: (1) Formad grupos pequeños uniendo dos grupos pequeños de prácticas; (2) enumeradlos; y (3) asignad los componentes del grupo en PRADO.

Referencias

Brad Appleton. Patterns and software: Essential concepts and terminology, 2000. URL http://www.bradapp.com/docs/patterns-intro.html.

Kent Beck and Ward Cunningham. Using pattern languages for object oriented programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1987. URL http://c2.com/doc/oopsla87.html.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. ISBN 0471958697. URL https://learning.oreilly.com/library/view/pattern-oriented-software-architecture/9781118725269/.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1994a. ISBN 0201633612. URL https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software- CD*. Addison-Wesley Longman Publishing Co., Inc., USA, 1994b. ISBN 0201633612.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995. ISBN 0201633612.