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

5 de marzo de 2021

Tema 1. Desarrollo utilizando patrones de diseño

Contenidos

Tema 1. Desarrollo utilizando patrones de diseño

1.1. Análisis y diseño basado en patrones

1.1.1. Origen e historia de los patrones software

Los patrones software fueron la adaptación al mundo de las Tecnologías de la Información y de la Comunicación (TICs) de los patrones arquitectónicos, que fueron definidos en 1966 por el arquitecto y teórico del diseño, Christopher Alexander 1.1 (estadounidense nacido en Austria) 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.

Figura 1.1: Christopher Alexander. Arquitecto que define el concepto de patrón arquitectónico.

En 1987 se adaptaron al desarrollo de software y se presentó la idea en un congreso (Beck and Cunningham, 1987).

El primer libro sobre patrones software se publicó en 1994 por la llamada "Gang of Four" (Gamma et al., 1994a), con una versión en CD (Gamma et al., 1994b) y una nueva edición en 1995 (Gamma et al., 1995). En él se explica cómo el concepto de patrón software es una adaptación directa del concepto de patrón usado por los arquitectos, que es definido como:

> 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).

Se trata de no "reinventar la rueda", tampoco en ingeniería del software:

> One thing expert designers know not to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, you'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.(Gamma et al., 1994a)

Un año antes de la publicación de este libro, y ya conocidas las ideas de Erich Gamma y su "banda", Kent Beck, junto con Grady Booch (uno de los creadores del lenguaje UML, junto con Rumbaugh y Jacobson) organizaron un retiro en las montañas del Colorado al que fueron expertos en desarrollo de software para intentar casar las ideas de patrón arquitectónico con la de objeto software, a la manera en que lo hicieron la GoF pero intentando que el patrón recogiera la idea original de creatividad de los patrones arquitectónicos de Christopher Alexander. Como se alojaron en la loma de una colina (1.2), al grupo que crearon le llamaron el "HillSide Group" (The HillSide Group).

En la actualidad se considera una ONG educativa y organiza congresos relacionados con los patrones software, como por ejemplo la European Conference on Pattern Languages of Programs (EuroPLoP) y mantienen catálogos de patrones y editan libros relacionados.

En 1996 se publicó otro libro de patrones software, esta vez más general, con el apoyo del "HillSide group". Si el primero era sólo sobre patrones de diseño (nivel medio), éste abarcaba desde los patrones de alto nivel o arquitectónicos hasta los llamados patrones a nivel de código (idioms) o patrones de bajo nivel(Buschmann et al., 1996). A veces al grupo de cinco autores que lo publicó se le ha llamado la "Gang of Five" por similitud con la GoF.

Figura 1.2: El lugar donde se creó el "HillSide Group", en las montañas de Colorado en 1993.

1.1.2. Conceptos generales y clasificación

Definición de patrón El patrón software se ha definido de la siguiente forma:

> 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).

La GoF, siendo los primeros en hablar de patrones, se enfocaron de forma específica en los patrones de diseño, haciendo un catálogo de ellos que veremos más adelante. Pero ellos mismos ya consideraban que existen otros tipos de patrones y daban algunos ejemplos de ellos:

> - patterns dealing with concurrency or distributed programming or real-time programming ...
> - application domain-specific patterns ...
> - how to build user interfaces,
> - how to write device drivers, or
> - how to use an object-oriented database.
>
> Each of these areas has its own patterns, and it would be worthwhile for someone to catalog those too. (Gamma et al., 1994a).

Clasificación general de los patrones Los patrones surgen desde la orientación a objetos y resuelven problemas a nivel de diseño orientado a objetos:

> 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. The design pattern identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether or not in can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our designs, a design pattern also provides sample ... code to illustrate an implementation. Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages ... (Appleton, 2000).

Pero enseguida se amplían a cualquier paradigma de programación, básicamente eliminando los términos "orientado a objetos" y cambiando los términos de clase por módulo o subsistema. Además se conciben para aportar soluciones en un espectro mucho más amplio en el nivel de abstracción. Se pasa de concebirse únicamente como soluciones de diseño (los llamados patrones de diseño, que están en un nivel medio de abstracción) a concebirse como soluciones en cualquier nivel desde el más abstracto o genérico (los patrones arquitectónicos, en el nivel arquitectónico) hasta el más específico (los patrones de código o expresiones ligüísticas, "idioms" en inglés) (Appleton, 2000; Buschmann et al., 1996):

- Architectural Patterns.- An architectural pattern expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- Design Patterns.- A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.
- Idioms.- An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Otra clasificación de los patrones los divide según la fase dentro del ciclo de vida de desarrollo del software, en patrones conceptuales, de diseño y de programación (Appleton, 2000):

- *Conceptual Patterns.- A conceptual pattern is a pattern whose form is described by means of terms and concepts from an application domain.*
- *Design Patterns.- A design pattern is a pattern whose form is described by means of software design constructs, for example objects, classes, inheritance, aggregation and use-relationship.*
- *Programming Patterns.- A programming pattern is a pattern whose form is described by means of programming language constructs.*

Clasificación de los patrones de diseño GoF subdivide los patrones de diseño en base a dos criterios. El primero, es el propósito, y según él, establecen tres tipos de patrones de diseño (Gamma et al., 1995) (pp 21 y 22):

- 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.

El segundo criterio es el ámbito de aplicación, y según él, hay dos tipos de patrones de diseño:

- De clase.- El patrón se aplica principalmente a clases.
- De objeto.- El patrón se aplica principalmente a objetos.

La Tabla 1.1 muestra ejemplos de patrones de diseño en base a estos dos criterios de clasificación.

Otra forma en la que los patrones de diseño son comprendidos, es mediante las semejanzas entre ellos. La GoF las representa usando un grafo dirigido, tal y como se muestra en la figura 1.3 (Gamma et al., 1994b) (p. 23).

1.1.3. Relación con el concepto de marco de trabajo (framework)

Por otro lado, hay una cierta relación entre el concepto de patrón de diseño y el concepto de marco de trabajo (framework), pero nunca deben ser confundidos:

- 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.

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

- Patrón software (de diseño, arquitectónico ...).- NO ESTÁ IMPLEMENTADO; es una guía, una "receta", una prescripción de desarrollo software, aplicable en cualquier dominio de aplicaciones, capaz de dar la misma solución a distintos problemas con una base similar, haciendo una abstracción de la parte común de los mismos que es crucial para llegar a una solución que simplifique la implementación y reusabilidad del código

1.1.4. Elementos de un patrón de diseño

La GoF identificó cuatro elementos esenciales en un patrón de diseño (Gamma et al., 1994a):

1. The pattern name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog.
2. The problem describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will

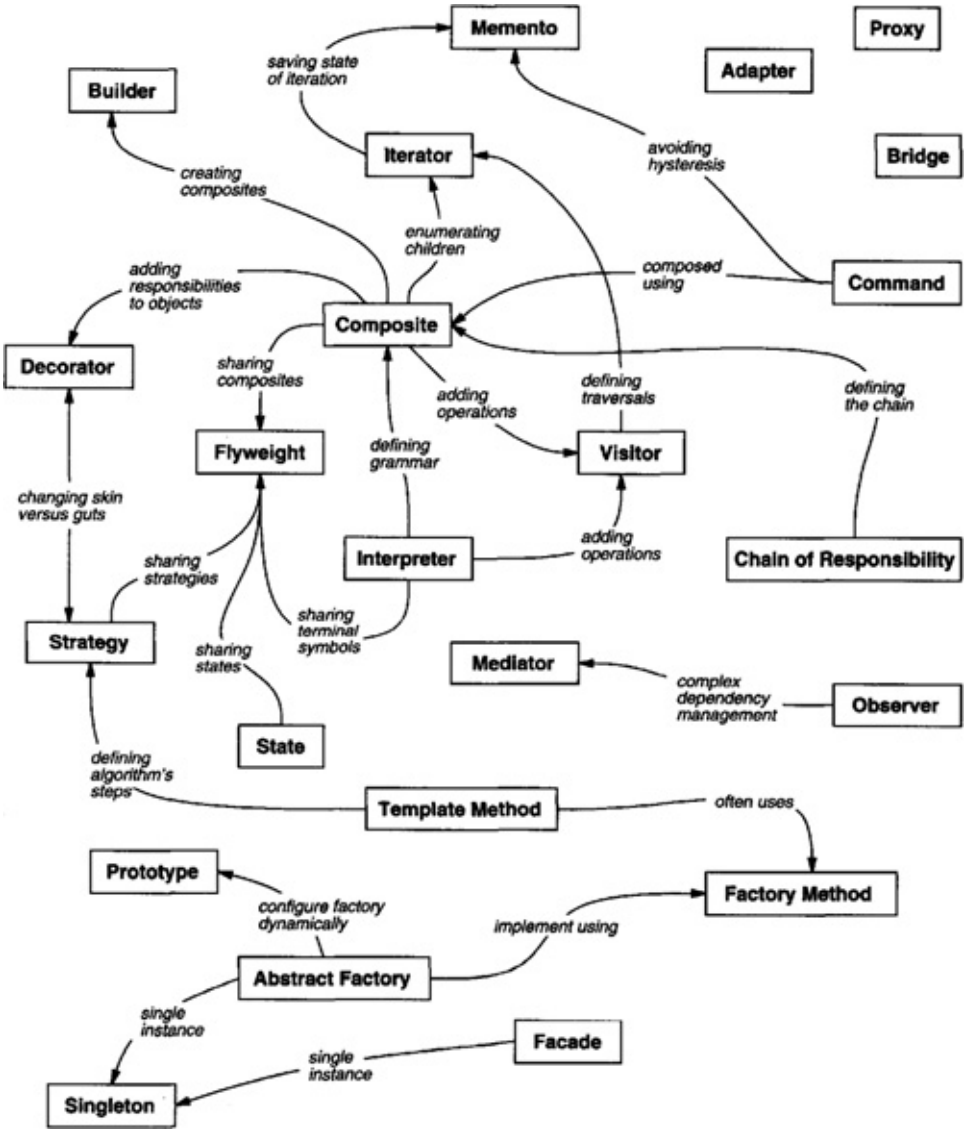

Figura 1.3: Relación entre los patrones de diseño. [Fuente: (Gamma et al., 1994b), p. 23]

include a list of conditions that must be met before it makes sense to apply the pattern.

3. The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the

pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

4. The consequences are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

En la actualidad se han identificado más elementos y se utiliza una plantilla (Tabla 1.2) con todos esos elementos cuando se proporciona un patrón (en negrita los cuatro elementos principales identificados por la GoF):

Plantilla	
Nombre	
Clasificación	arquitectónico/de diseño/de programación
Contexto	describe el entorno en el que se ubica el problema incluyendo el dominio de aplicación
Problema	una o dos frases que explican lo que se pretende resolver
Consecuencias	lista el sistema de fortalezas que afectan a la manera en que ha de resolverse el problema; incluye las limitaciones y restricciones que han de respetarse
Solución	proporciona una descripción detallada de la solución propuesta para el problema
Intención	describe el patrón y lo que hace
Anti-patrones	"soluciones" que no funcionan en el contexto o que son peores; suelen ser errores cometidos por principiantes
Patrones relacionados	referencias cruzadas relacionadas con los patrones de diseño
Referencias	reconocimientos a aquellos desarrolladores que desarrollaron o inspiraron el patrón que se propone
Estructura	Diagrama UML
participantes	descripción de los componentes (clases/objetos) que lo forman y su papel

Tabla 1.2: Plantilla utilizada para describir un patrón.

1.1.5. Un ejemplo práctico a partir de las clases Modelo-Vista-Controlador (MVC)

La GoF pone un ejemplo práctico de uso de patrones en las clases que se proporcionan en Smalltalk para construir interfaces gráficas de usuario (Graphical User Interfaces, GUIs) [1]. En otros lenguajes pueden implementarse GUIs usando el mismo diseño, o variantes. Así por ejemplo, en Java podemos usar más de una clase, agrupada en un paquete, para el modelo, otro para la vista y otro para el controlador. Pero la idea es siempre la misma, separar el objeto de la aplicación (el modelo) de la vista. En este diseño tripartito, se divide también la presentación visual (la vista) de la gestión de las respuestas del sistema a las entradas del usuario (controlador). En otros diseños, por ejemplo el que utiliza el paquete Java SWING de diseño de GUIs, controlador y vista se mantienen unidos en lo que se llama modelo de gestión de eventos (event handling modelling). En todo caso, siempre se trata de separar el modelo de su presentación e interacción con él, para aumentar su flexibilidad y reusabilidad. Así, podemos decidir poner diferentes GUIs, o hacer incluso una modalidad web o una app para el móvil, y el modelo quedaría siempre intacto. Android Studio, como ejemplo de entorno integrado para desarrollo de aplicaciones móviles, ha incorporado también el modelo tripartito MVC de diseño arquitectónico de aplicaciones gráficas. La GoF utiliza el diseño MVC para identificar e introducir los tres primeros patrones de su libro (Gamma et al., 1994a).

[1] Estas clases forman precisamente otro patrón, pero a nivel arquitectónico y por aquel entonces aún no se había indentificado como tal.

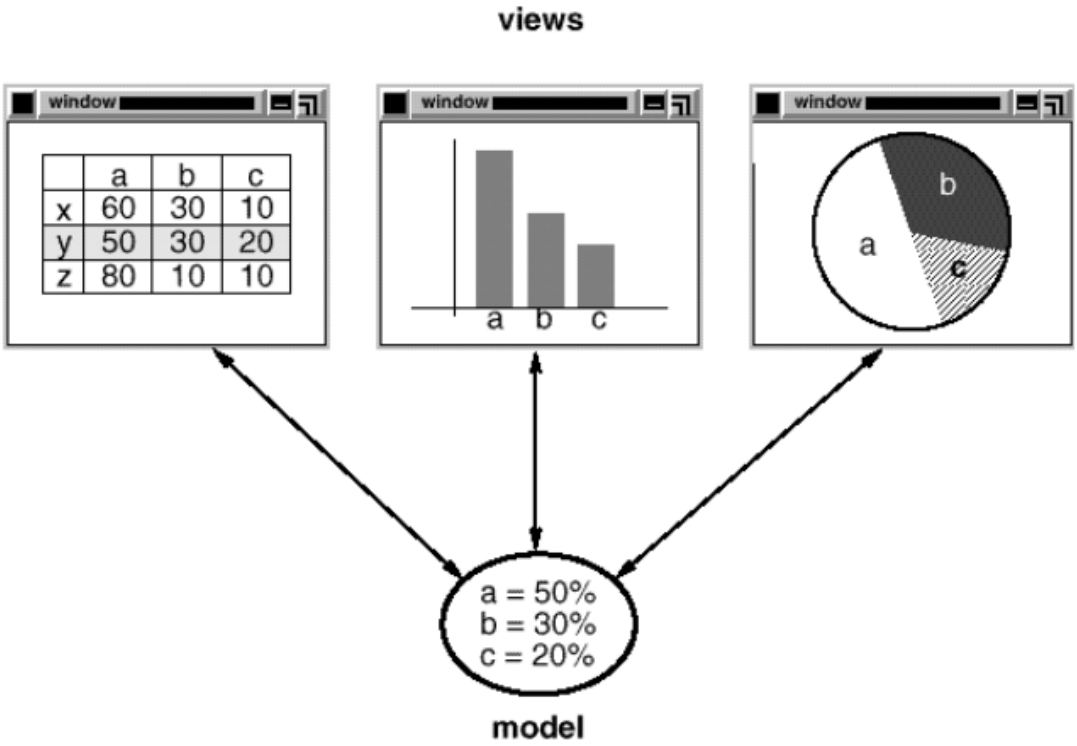

Figura 1.4: Ejemplo de modelo con tres vistas, en un diseño MVC. [Fuente: (Gamma et al., 1994b), p. 15]

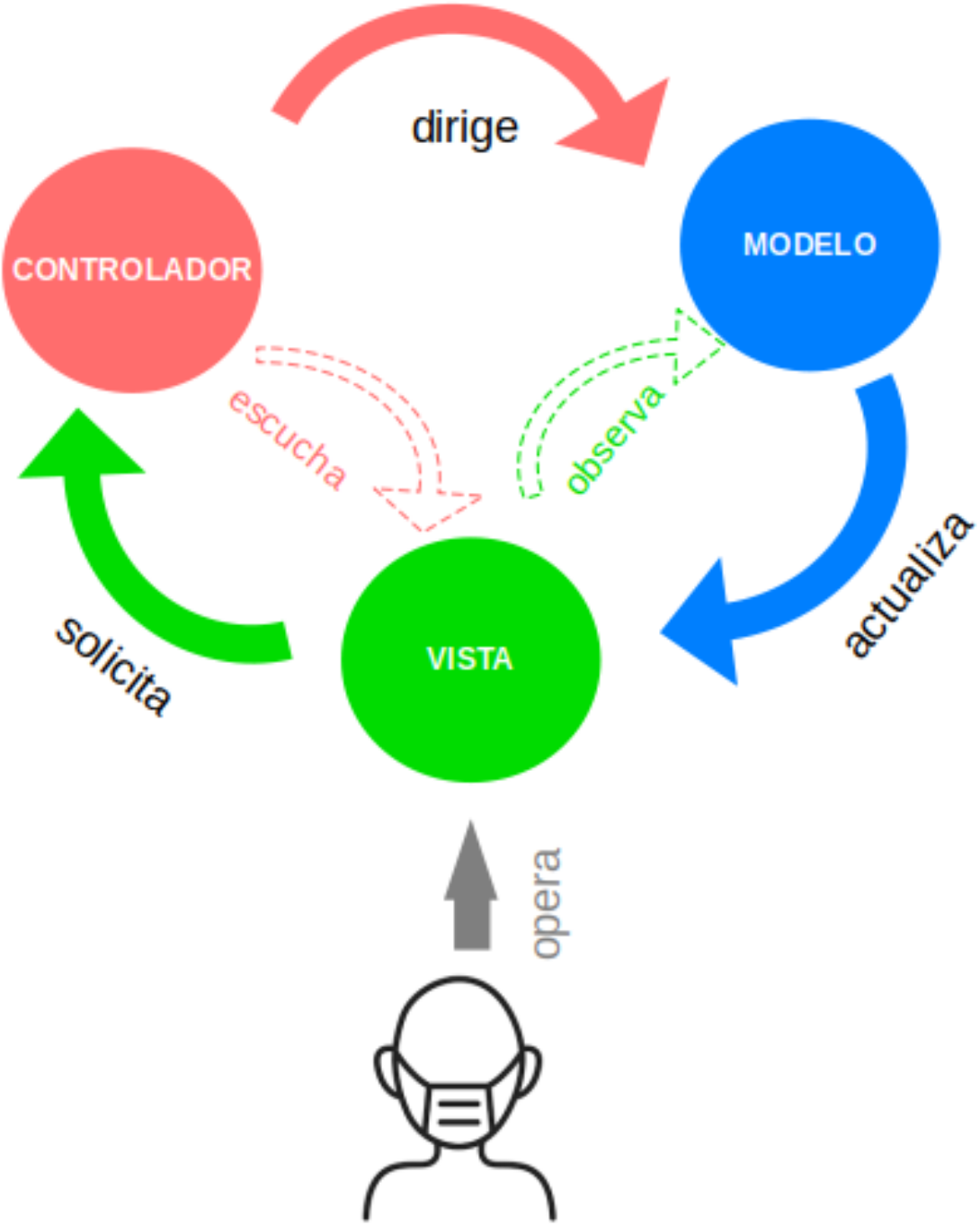

Figura 1.5: Ejemplo de diagrama relacional de la terna de clases en el diseño MVC.

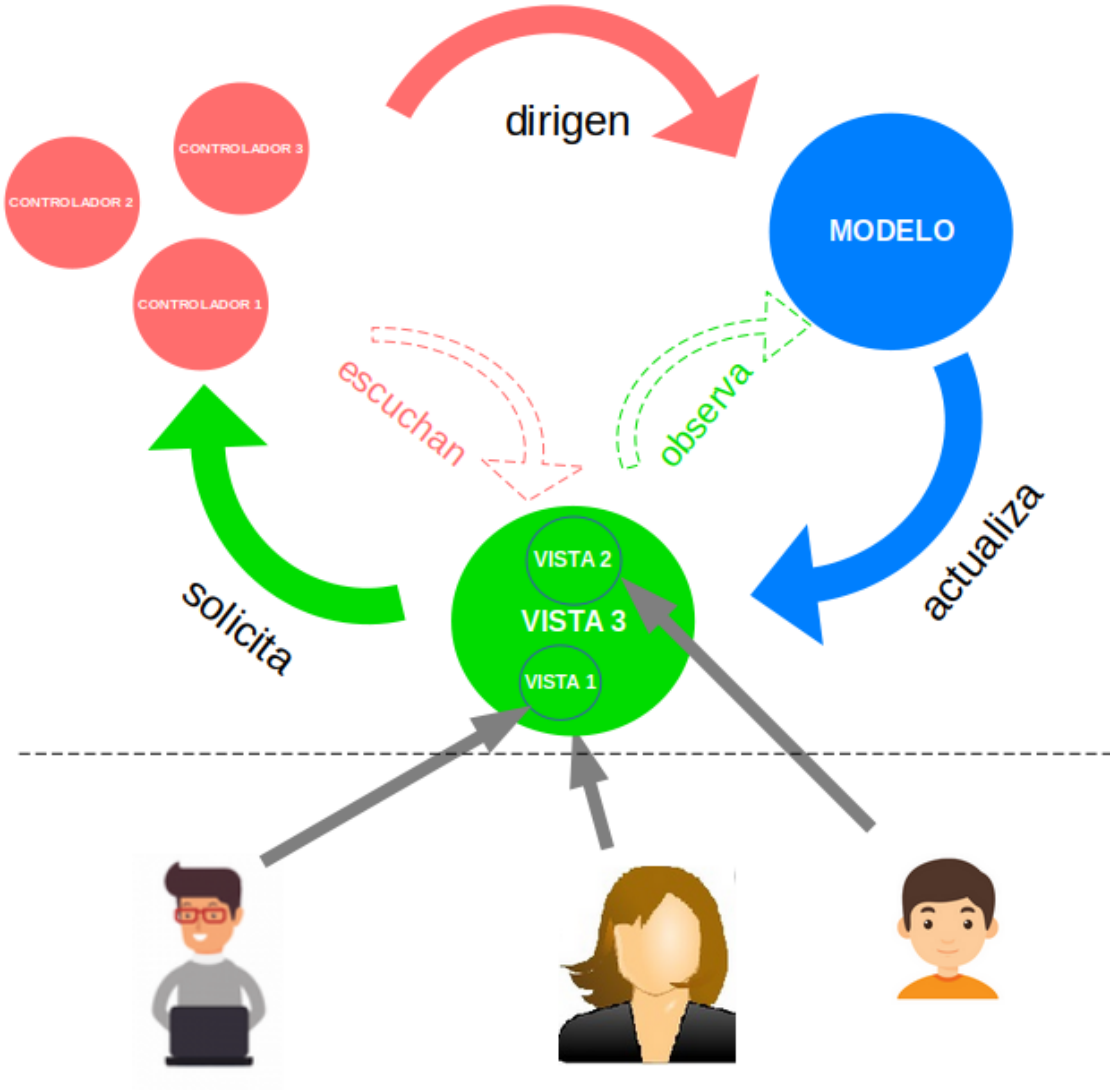

Figura 1.6: Ejemplo más complejo de diagrama relacional en el diseño MVC.

- *Observer*.- Se trata de establecer un protocolo de subscripción/notificación entre el modelo y la vista para desacoplar la vista del modelo, es decir, que la vista no se asocie (no navegue hacia) el modelo. Cada vez que el modelo cambia, notifica a todas las vistas que tiene suscritas, el cambio, y las vistas se modifican a sí mismas. En la Figura 1.6 una aparece un ejemplo de modelo con tres vistas (Gamma et al., 1994b). El patrón *observador* generaliza la idea para desacoplar cualquier tipo de objetos, y no solo los que representan la vista y el modelo en un diseño MVC.

- *Composite.*- Se trata de un patrón de diseño en el que un grupo de objetos son tratados como uno individual. Para ello, una clase definirá objetos simples, y otra clase definirá objetos complejos, como agregación o composición de los objetos simples o de otros compuestos. En el caso del diseño MVC para GUIs, una vista puede contener otras vistas (vistas anidadas), por ejemplo un panel de control de botones es una vista compuesta por vistas simples de botones. Por ejemplo, esto puede usarse para implementar un inspector de objetos (así se hace en Suby y en Smalltalk) y reusarse en la implementación de un depurador. El patrón *compuesto* generaliza esta idea de forma que se aplica cada vez que queramos tratar un grupo de objetos de la misma forma que a sus componentes individuales, de forma que se definen clases de objetos individuales y de compuestos de objetos como herederas de una misma 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.

Otros patrones de MVC son el Método Factoría para especificar la clase controlador y el Decorador, para añadir desplazamiento (scrolling) a una vista.

1.2. Cómo resolver problemas de diseño usando patrones de diseño

- Encontrando los objetos (clases) apropiados.- Los patrones de diseño llevan a usar clases que no forman parte del diagrama de clases de análisis porque no existen en la realidad, por ejemplo la clase Composite o la clase Strategy. La necesidad de usar de patrones no aparece en la fase de análisis ni al principio del diseño, sino en el momento en el que pensamos en diseñar un sistema flexible y reutilizable.

- Considerando distinta granularidad.- Hay objetos que representan (1) a todo un sistema o en todo caso son muy grandes y (2) suele haber una sola instancia de ellos y otros más pequeños, existiendo a veces (3) muchos objetos similares de muy poco contenido. Esto se translada a los patrones de diseño, siendo un ejemplo del primero el patrón *fachade*, uno del sengundo el patrón *singleton* y uno del tercero el patrónflyweight.

- Especificando la interfaz de un objeto.- Muchos patrones de diseño están relacionados con la ligadura dinámica de la orientación a objetos y el hecho de que los objetos con

una misma interfaz o una parte de la misma en común, se pueden intercambiar en la parte común. Por ejemplo, si comparten la misma signatura de un método (nombre del método, argumentos y valor de retorno), aunque lo implementen de forma distinta.

- Programando en función de las interfaces y no de las implementaciones. Se trata de usar las interfaces para disminuir el acoplamiento de un sistemas (dependencias entre subsistemas). En algunos lenguajes no existe esta diferencia (como Ruby o Smalltalk), porque al no ser tipados, la interfaz de un objeto viene especificada por la clase y por tanto la herencia de interfaces es la que se define de forma implícita al definir herencia de clases. Pero en otros lenguajes, como en Java o c++, que son tipados, sí que existen diferencias entre el tipo-s (estático-s) de la variable, que definen su interfaz y las clases que las implementan, de forma que un objeto puede ser de varios tipos (estáticos), es decir, puede implementar métodos declarados en varias interfaces. En Java además las interfaces se declaran de forma explícita y por tanto también la herencia entre las mismas. En c++ la herencia entre interfaces se lleva a cabo solo mediante clases abstractas y métodos virtuales (ligadura dinámica) puros (abstractos). Algunos patrones están basados en esta diferencia entre herencia de clases y de interfaces, como el patrón *composite* y el patrón *observer*. En todo caso, siempre podemos relacionar los objetos no por la implementación de los métodos sino por la interfaz (signatura) de estos métodos. Esto se consigue haciendo uso de clases abstractas (o el concepto explícito de interfaz en Java). Gracias a esto, los objetos clientes de estos objetos no tienen que saber nada sobre la implementación de los métodos a los que invocan, les basta saber que el método forma parte de la interfaz del objeto. Incluso tampoco tiene por qué conocer el tipo específico (tipo dinámico) de esos objetos. Los patrones creaciones tienen la función de permitir este desacoplamiento, con distintas propuestas para asociar una interfaz con la implementación concreta en el momento de instanciar un objeto.

- 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).- Incluso cuando se cumple la relación «es un» entre dos clases, no siempre es lo mejor el uso de la herencia. Hay una tendencia a abusar de la herencia que aumenta la dependencia del código, por ejemplo los cambios del código en clases superiores a menudo obligan a cambiar el de las subclases.
  - Usando la delegación como alternativa extrema de la composición que sustituya la herencia.- En algunos casos es más indicado delegar en otros, es decir, pasar a objetos de otras clases la responsabilidad que debe tener el objeto receptor, para aumentar la reusabilidad. Algunos patrones se basan en la delegación, como el patrón *strategy* y el patrón *visitor*.

- Usando tipos parametrizados como alternativa a la herencia.- Los genéricos o plantillas (templates) permite un tercer modo de reutilizar código en lenguajes tipados que permiten definir clases genéricas especificando como un parámetro el tipo de los objetos que usarán de forma que en tiempo de compilación se creen las clases ya concretas según el tipo del parámetro usado. Una aplicación de este método es por ejemplo el uso de contenedores parametrizados de modo que métodos como la ordenación de los componentes se definen de forma genérica.

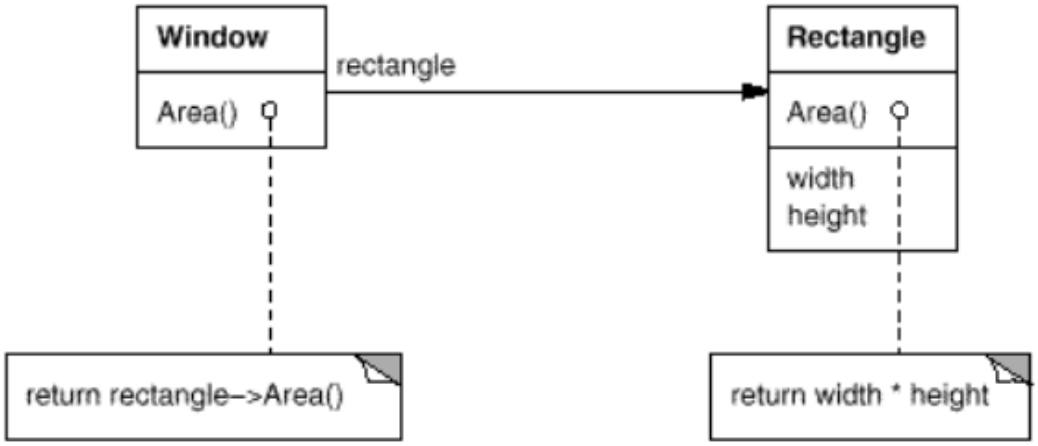

Figura 1.7: Ejemplo de uso de delegación (Gamma et al., 1994a, p.33) p. 33.

1.3. Estudio del catálogo GoF de patrones de diseño

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 1.11)
- *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 1.34)
- *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 1.17)
- *Builder* (Gamma et al., 1994b) pp. 110-120
- *Visitor* (Gamma et al., 1994b) pp. 366-381 (Figura 1.35)
- *Facade* (Gamma et al., 1994b) pp. 208-217
- *Filtro de intercepción* (Figura 1.39)

1.3.1. Patrones creacionales *Factoría Abstracta*, *Método Factoría*, *Prototipo* y *Builder*

GoF (Gamma et al., 1994a) también presentan un quinto patrón creacional, el patrón *Singleton*, que exige que una clase sólo se pueda instanciar una vez. Sin embargo este patrón es de un ámbito mucho más reducido que los otros, afecta a una sola clase y es aplicable con cualquier de los otros cuatro patrones creacionales. El resto de patrones creacionales están muy relacionados y es importante entender las características de cada uno para poder elegir el más adecuado a un problema concreto. Estos cuatro patrones son 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 de GoF (Gamma et al., 1994a), pp. 94-95 (ver Figuras **??** y 1.9).

Figura 1.8: Ejemplo de un laberinto según este juego. Las puertas cerradas no son accesibles según en la implementación. Es decir, al otro lado puede haber otra habitación (clase *Room*) o un muro (clase *Wall*). [Fuente: https://www.megapixl.com/rooms-and-doors-maze-game-illustration-40888326].

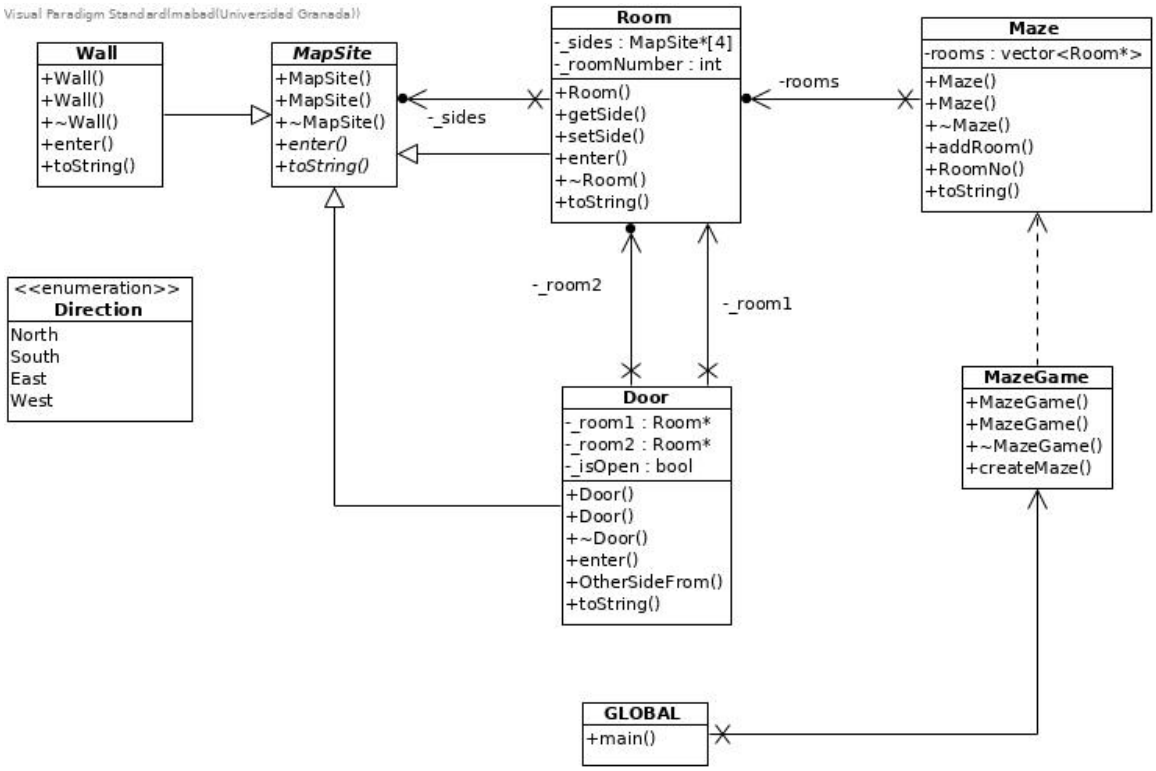

Figura 1.9: Diagrama de clases correspondiente al ejemplo del juego del laberinto de GoF (Gamma et al., 1994a).

El concepto de factoría en OO

Antes de presentar estos patrones, es necesario aclarar qué significa el concepto de factoría en OO. El concepto puede cambiar según el tipo de lenguaje –*basado en clases* o *basado en prototipos* (este último cuando no hay clases, sino que los objetos se crean por "delegación" a partir de otros)– o el lenguaje de programación específico. Sin embargo, veremos aquí la acepción más general del concepto, que algunos consideran como un patrón de código o de bajo nivel (idiom). Una factoría es simplemente un objeto con algún método (método factoría) para crear objetos (ver Figura 1.10). En lenguajes OO "puros" (donde todo es un objeto), tales como Ruby o Smalltalk, esta definición contempla por tanto a la más simple de las factorías, la que crea una instancia de la propia clase, pues es un método más. Sin embargo, en los lenguajes OO híbridos, como Java o C++, las factorías no pueden considerarse generalizaciones de los llamados "constructores", pues estos son métodos especiales que, además de seguir reglas sintácticas diferentes al resto de los métodos, no permiten polimorfismo, debiéndose explicitar la clase concreta que se quiere crear. Un

ejemplo de método factoría por delegación es el método de clonación (clone). El constructor de copia en Java o C++, no puede considerarse un método factoría. En el siguiente ejemplo puede verse que el método clone se invoca de la misma manera en una clase y su subclase:

```
//metodo factoria clone en clase Objeto:
Objeto clone (){
  return new Objeto(this);
   }
// metodo factoria clone en subclase SubObjeto de Objeto:
Objeto clone (){
  return new SubObjeto(this);
   }
// Copia de objetos con metodo factoria en una clase distinta
// 1. Creamos dos objetos de una clase y su subclase
Objeto unObjeto=new Objeto();
Objeto unSubObjeto=new SubObjeto();
// 2. Hacemos copias de ellos usando clone, un ejemplo de metodo factoria
   :
Objeto otroObjeto=unObjeto.clone();
Objeto otroSubObjeto=unSubObjeto.clone();
```

En el siguiente ejemplo puede verse cómo copiar objetos usando directamente el constructor de copia:

```
// Copia de objetos sin metodo factoria en una clase distinta
// 1. Creamos dos objetos de una clase y su subclase
Objeto unObjeto=new Objeto();
Objeto unSubObjeto=new SubObjeto();
// 2. Hacemos copias de ellos usando los constructores de copia:
Objeto otroObjeto=new Objeto(unObjeto);
Objeto otroSubObjeto=new SubObjeto(unSubObjeto);
```

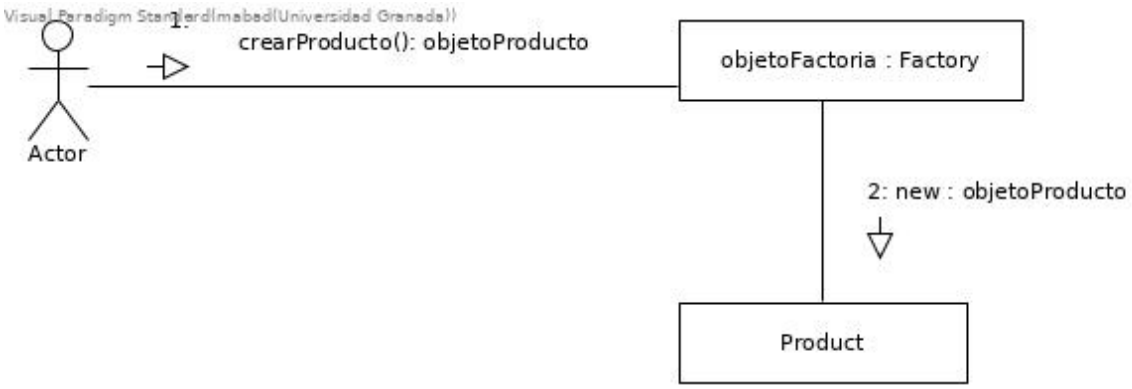

Figura 1.10: Diagrama de comunicación que muestra un ejemplo de creación de una instancia de la clase *Product* mediante el método factoría *crearProducto* de la clase *Factory*.

Patrón *Factoría Abstracta* (*kit*)

Recomendado cuando en una aplicación tenemos líneas, temáticas o familias "paralelas" "de productos" y se prevé que puedan añadirse nuevas líneas. Con este patrón se podrá elegir una familia de entre todas las definidas sin que cambie el código al cambiar de familia elegida, y además el cliente solo tiene que conocer la interfaz de acceso a cada producto (común a todas las líneas), pero no cómo se implementa la forma de crearlos o de operar con ellos.

Sin embargo este patrón no está recomendado si se piensan agregar nuevas clases a líneas ya creadas.

Por ejemplo, una línea de clases puede ser una librería gráfica (Swing, AWT ...) o un estilo de componentes gráficos o un tipo de escritorio (GNOME o KDE en el caso de linux) y las clases pueden ser los componentes gráficos ($Boton$, $Menu$, $Panel$, ...) o elementos del escritorio ($BarraTareas$, $AreaTrabajo$, $Ventana$, $Menu$, ...). que existirán para cada librería, estilo o escritorio. El patrón utiliza una interfaz ($AbstractFactory$ en la Figura 1.12) o una clase completamente abstracta (sin ningún método implementado en la propia clase). Sin embargo, también se podría usar una clase abstracta convencional, donde puede haber métodos de creación implementados en esa clase si la creación de los objetos de una clase no depende de la familia (por ejemplo, si un panel se crea de la misma forma para GNOME y para KDE). En la interfaz o clase abstracta se declara un método de creación para cada tipo de objetos ($crearProductoA$, $crearProductoB$ en la Figura 1.12). En nuestro ejemplo, serían los métodos $crearBoton$, $crearMenu$, $crearPanel$. Este patrón está menos recomendado si lo que vamos a cambiar o añadir son nuevas clases ($Frame$, Box, ... en el ejemplo) y no líneas de clases (como un nuevo escritorio, vg. Cinnamon, en el ejemplo).

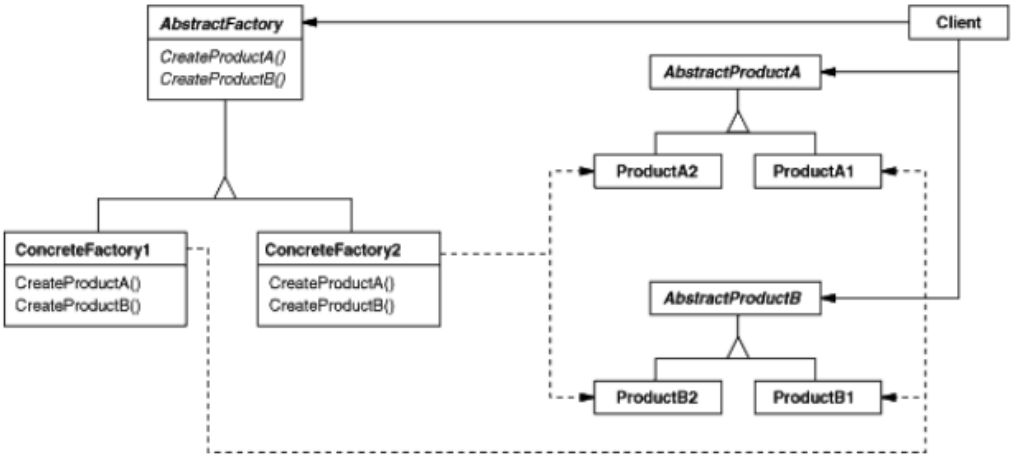

Figura 1.11: Estructura del patrón Factoría abstracta [Fuente: (Gamma et al., 1994b), p. 101]

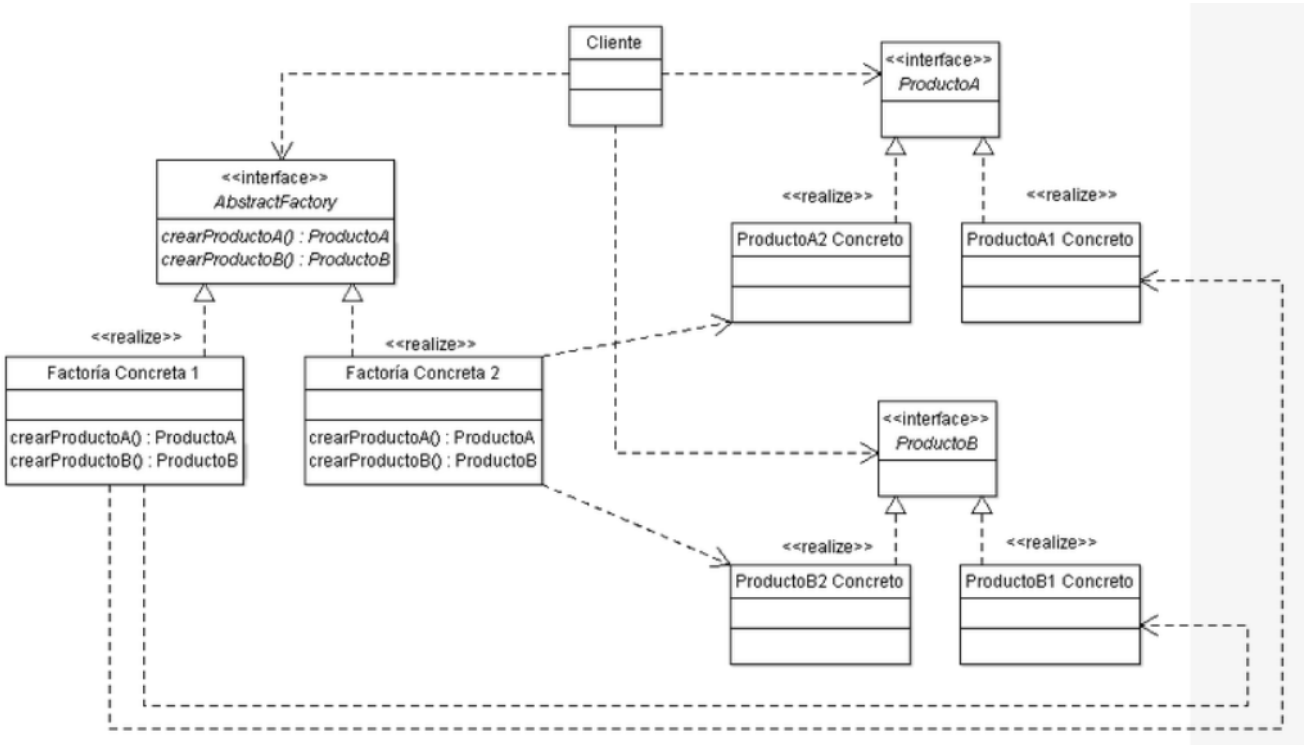

Figura 1.12: Otro diagrama de clases (más ampliado) del patrón *Factoría Abstracta*. [Fuente: Abstract Factory].

Hay dos formas en las que se pueden crear los objetos por las factorías abstractas, utilizando a su vez otros patrones creacionales: (1) patrón *método factoría*, que crea haciendo uso de métodos factoría, y (2) patrón *prototipo*, que crea mediante delegación (ver Figura 1.13).

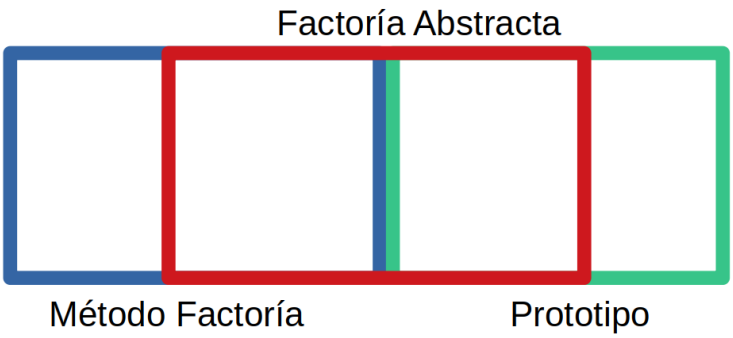

Figura 1.13: Relación de coexistencia entre los patrones *Factoría Abstracta*, *Método Factoría* y *Prototipo*.

En la Figura 1.14 puede verse el resultado de aplicar este patrón al juego del laberinto, usando métodos factoría.

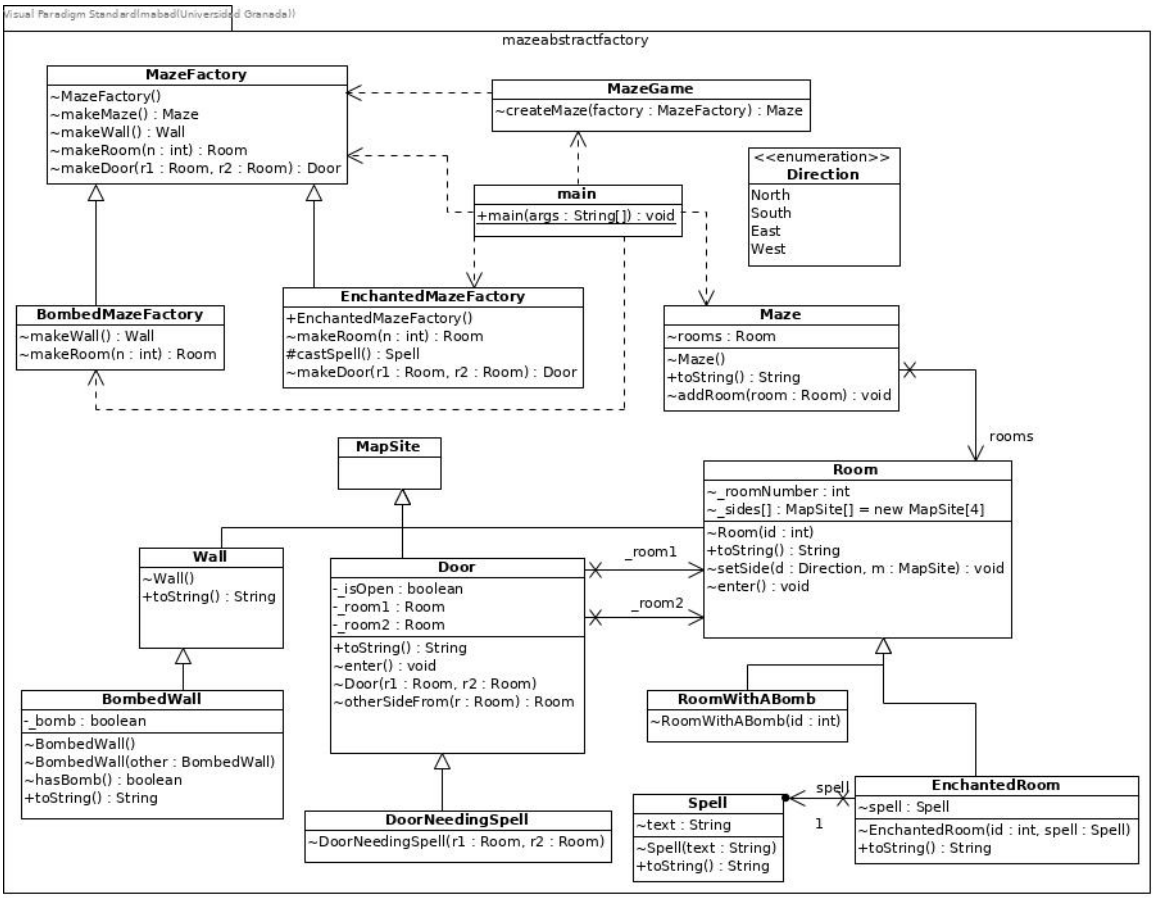

Figura 1.14: Diagrama de clases correspondiente a la aplicación del patrón *Abstract Factory* en el ejemplo del juego del laberinto de GoF (Gamma et al., 1994a). Este patrón se usa aquí junto con el patrón *Factory Method*.

Patrón *Método Factoría* (*Virtual constructor*)

Se trata de un patrón que define métodos factoría en clases que crearán y usarán objetos de una aplicación o marco de trabajo, en vez de llamar directamente a los constructores, para que pueda beneficiarse de la ligadura dinámica y se construya el objeto en la subclase adecuada. La redefinión de los métodos factoría en distintas subclases permite crear distintas variaciones de la aplicación, según la combinación de subclases concretas elegidas para crear los objetos a partir de ellas. Este patrón es utizado en la mayoría de las implementaciones del patrón *Factoría Abstracta*, aunque no es obligatorio (la alternativa es usar prototipos). En el patrón *Factoría Abstracta*, el patrón *Método Factoría* está implementado en la clase o interfaz AbstractFactory y sus subclases (ver Figuras 1.11 y 1.12).

Por otro lado, este patrón puede utilizarse sin utilizar 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.

Un caso extremo consiste en usar una única interfaz (signatura) del método factoría (una clase en la que se declara pero no se implementa), para todos los productos (ver como ejemplo la clase *AbstractCreator* en la Figura 1.15), redefiniéndose en subclases. Otra posibilidad, si hay varias clases en la aplicación sin relación de herencia entre ellas (varias clases Producto), es aplicar parametrización en el método factoría único para saber a qué clase debe pertenecer un objeto que se deba crear.

Un caso en el extremo contrario es permitir tantos métodos factoría como clases distintas coexistan en una instancia de la aplicación, agrupándolos todos en la misma clase (abstracta), junto con el método para crear la propia instancia de la aplicación. Esta clase es la que se llama generalmente clase "gestora" y con este patrón los métodos factoría se podrían redefinir en subclases que representen distintas variaciones de la aplicación.

Así, este patrón es útil cuando no sepamos las clases concretas de los objetos que vamos a crear, o puedan cambiar en el futuro, añadiéndose subclases, considerándose que todas heredan de una clase abstracta (clase *AbstractProducto* en la Figura 1.15) con un método de operación común a los posibles subtipos de productos (método *operacion* en Figura 1.15). Para cada nuevo producto (clases *ProductoTipo*, *ProductoOtroTipo* en la Figura 1.15) deberán crearse creadores o factorías concretas (*CreadorProductoTipo* y *CreadorOtroProducto* en la Figura 1.15)."

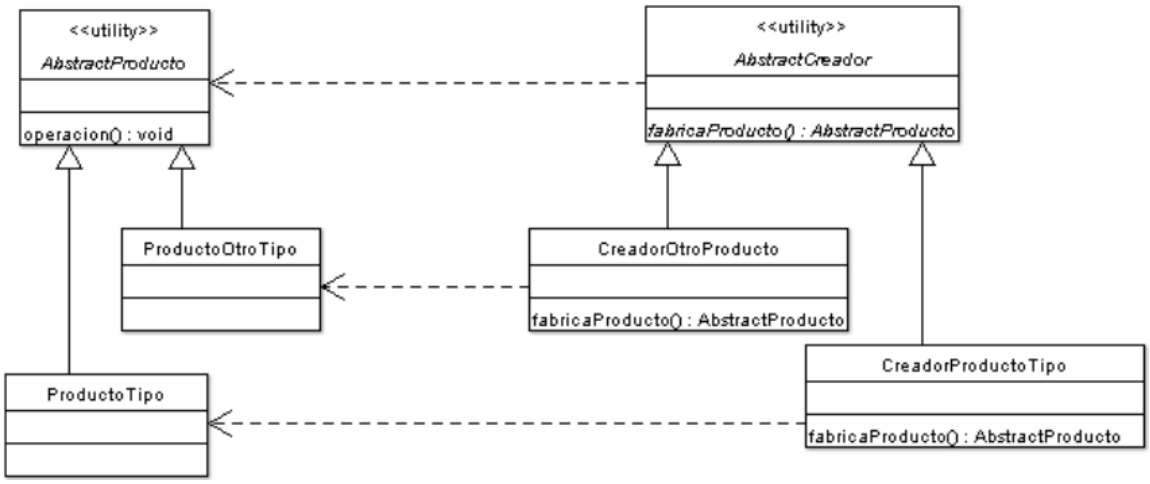

Figura 1.15: Diagrama de clases del patrón Método Factoría. [Fuente: Factory Method].

En la Figura 1.16 puede verse el resultado de aplicar este patrón al juego del laberinto.

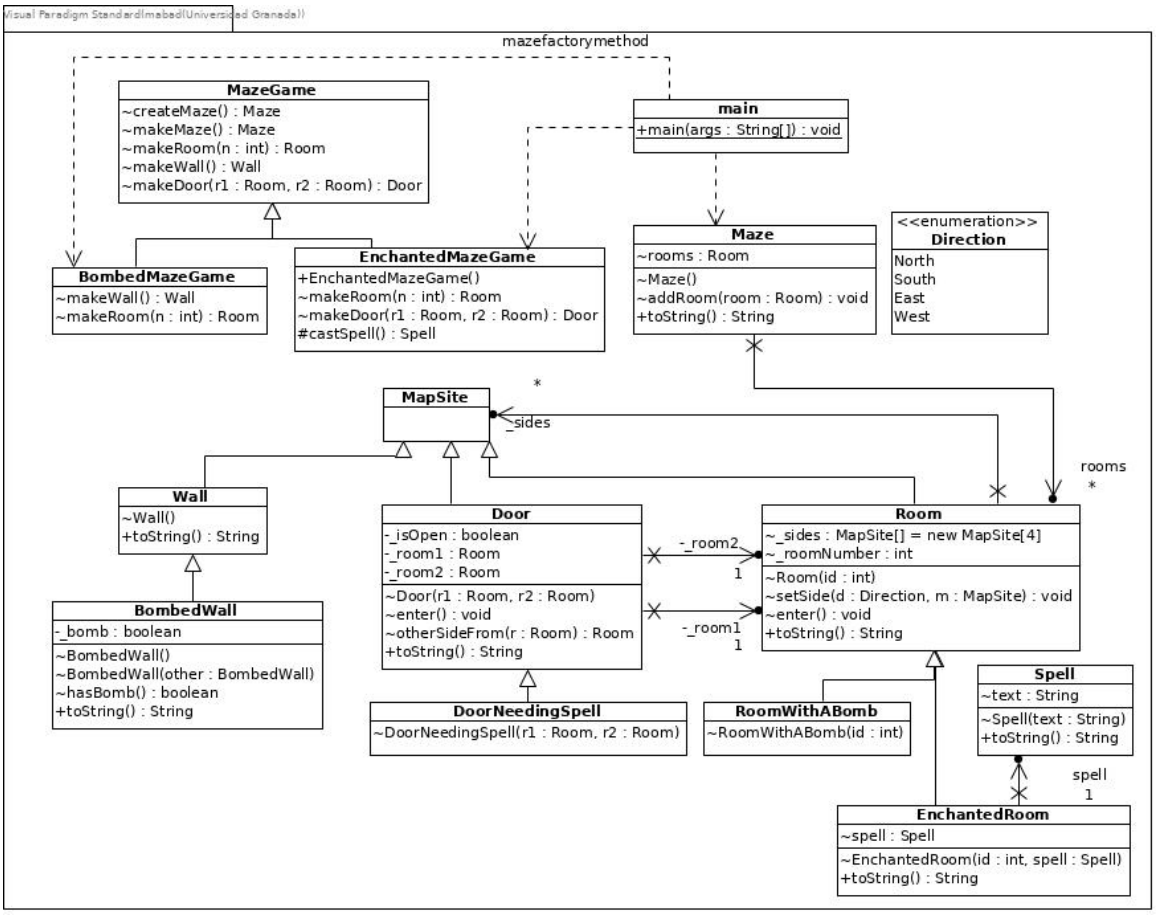

Figura 1.16: Diagrama de clases correspondiente a la aplicación del patrón "Factory Method" en el ejemplo del juego del laberinto de GoF (Gamma et al., 1994a).

Patrón *Prototipo*

Este patrón usa un prototipo para cada tipo de objeto que se vaya a usar en la aplicación, y crea uno nuevo por clonación a partir de su prototipo. Este método, considera que las clases de todos los objetos que se quieren utilizar en la aplicación heredan de la clase abstracta *Prototype*, y el cliente implementa un único método (método *operation* en la Figura 1.17) para crear objetos, pidiendo la clonación al prototipo del objeto que se desea.

Una forma alternativa al uso del patrón *Método Factoría* con el patrón *Factoría Abstracta* es la posibilidad de usar el patrón *Prototipo*. Usando prototipos no tenemos que crear la jerarquía de clases factoría concretas, paralela a la jerarquía de clases de productos concretos, sino que basta con una sola clase factoría concreta, que es la clase cliente del patrón

prototipo con un único método de creación, que pide la clonación al prototipo del objeto que se desea crear. La única clase factoría concreta heredará de la clase factoría abstracta que deberá tener un contenedor con los prototipos de todos los tipos de objetos que en la aplicación se deseen crear.

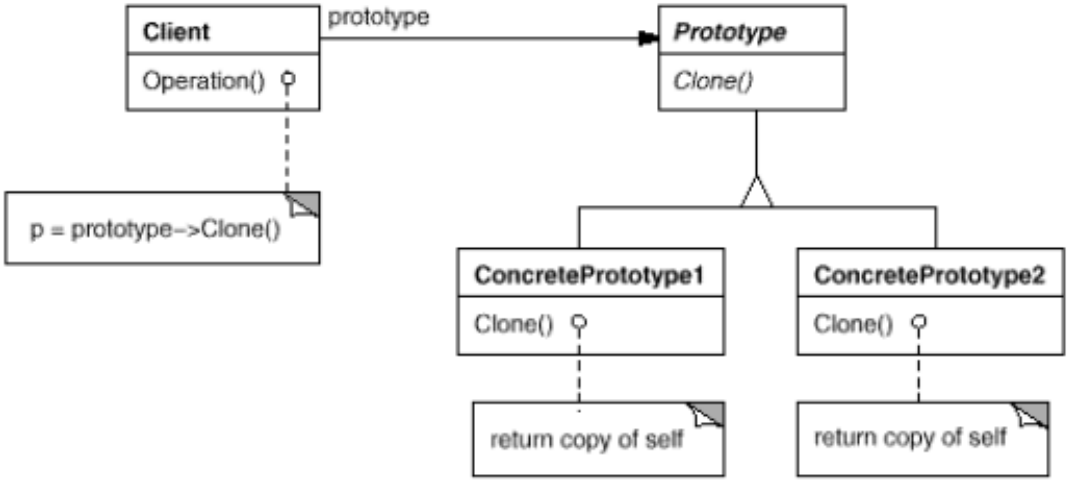

Figura 1.17: Estructura del patrón Prototype [Fuente: (Gamma et al., 1994b), p. 135]

En la Figura 1.18 puede verse el resultado de aplicar este patrón junto con el de *Factoría abstracta* al juego del laberinto.

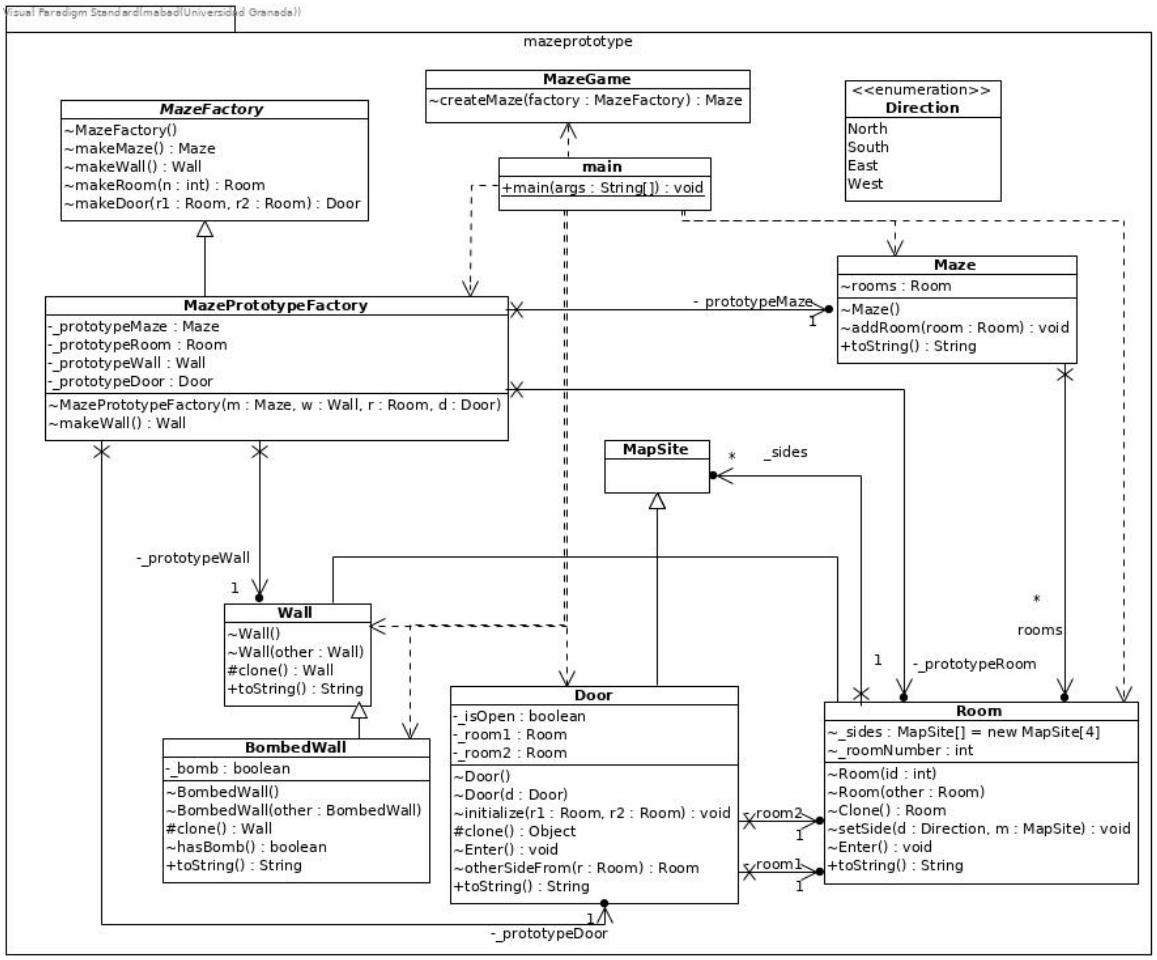

Figura 1.18: Diagrama de clases correspondiente a la aplicación del patrón *Prototype* junto con el de *Abstract Factory* en el ejemplo del juego del laberinto de GoF (Gamma et al., 1994a).

Patrón *Builder*

Este patrón se utiliza cuando queremos crear un objeto de cierta complejidad formado por componentes que puedan cambiar, así como ensamblarse y representarse de distintas formas, pero siendo común el algoritmo que se necesita para construir las partes y ensamblarlas obteniendo el objeto complejo.

Así, este algoritmo de creación es más bien lo que hace un director o guía de la construcción (un objeto de la clase Director), el cual llama a objetos constructores (builders

concretos) que saben cómo construir cada una de las partes y ensamblarlas. Cada objeto concreto builder, compone y ensambla de forma distinta, pero todos comparten los métodos, por eso el director puede utilizar una interfaz común (de declaración explícita para los lenguajes tipados), que gracias a la ligadura dinámica, en tiempo de ejecución se asociará a un código concreto. La Figura 1.19 muestra el diagrama de clases con la estructura del patrón y la Figura 1.20 un diagrama de secuencias con la forma en la que cooperan el *Director* y el *Builder* con el cliente.

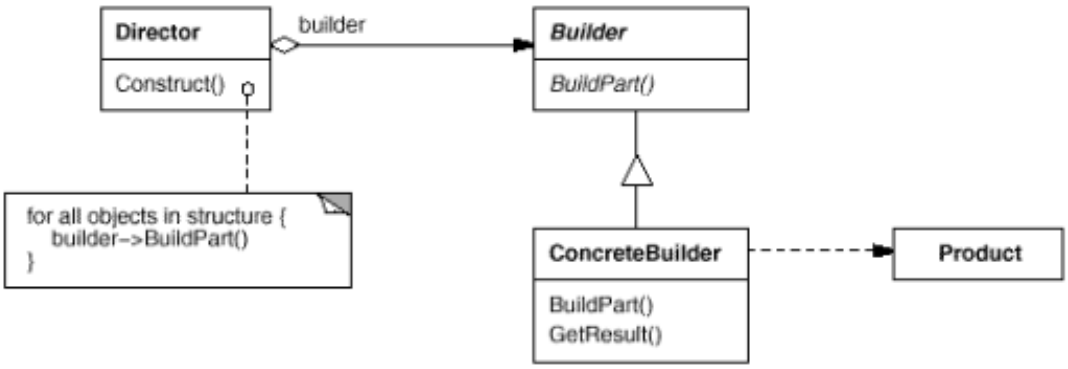

Figura 1.19: Estructura del patrón Builder [Fuente: (Gamma et al., 1994b), p. 112]

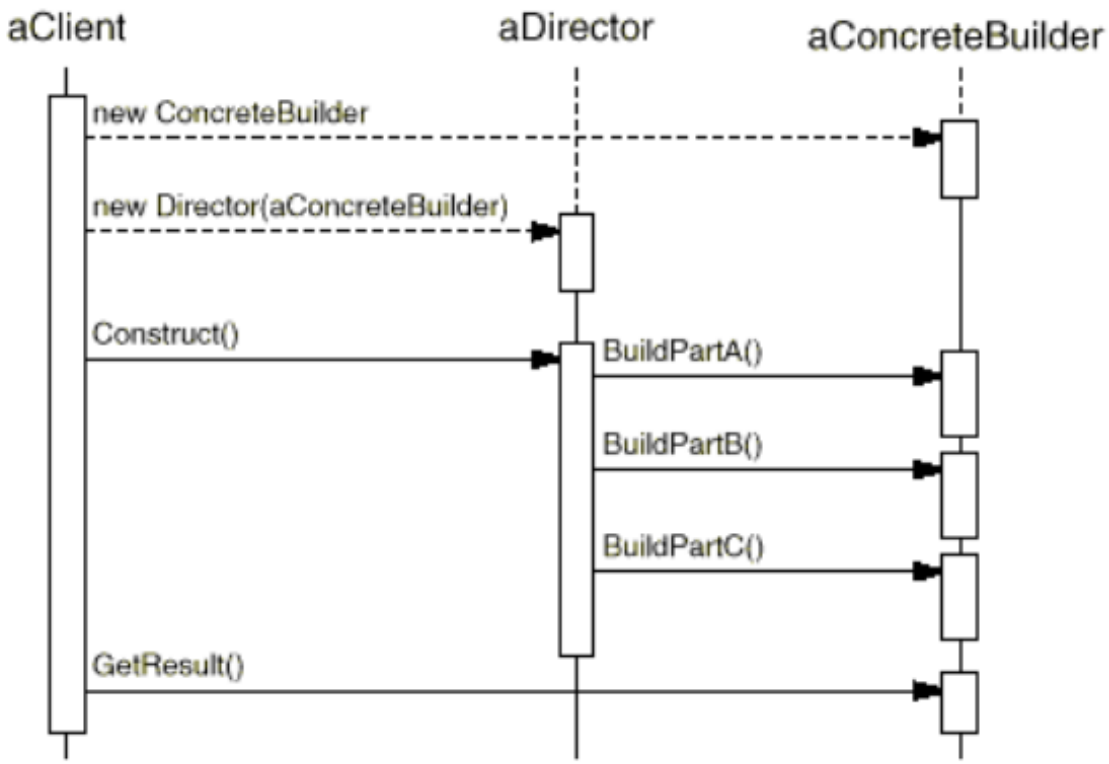

Figura 1.20: Estructura del patrón Builder [Fuente: (Gamma et al., 1994b), p. 113]

En la Figura 1.21 puede verse el resultado de aplicar este patrón al juego del laberinto.

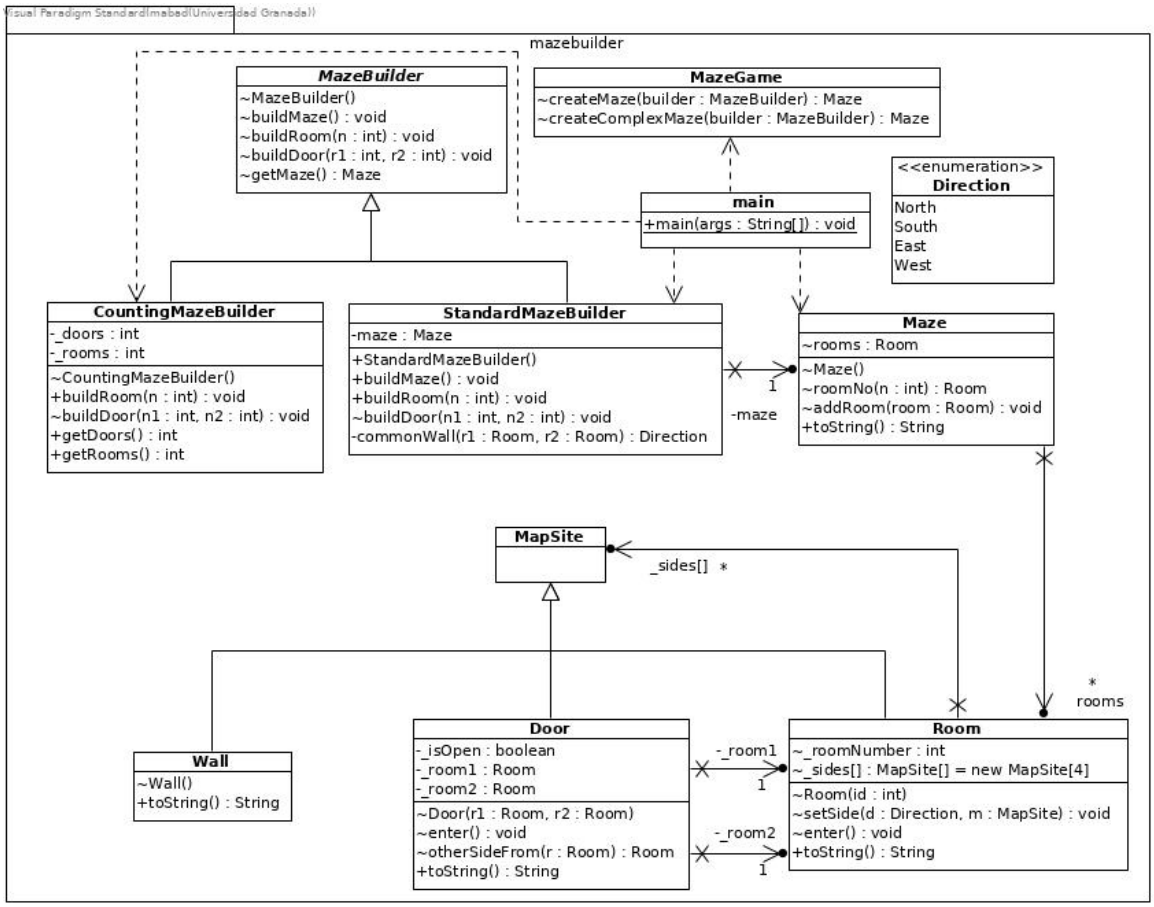

Figura 1.21: Diagrama de clases correspondiente a la aplicación del patrón "Builder" en el ejemplo del juego del laberinto de GoF (Gamma et al., 1994a).

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.

1.3.2. Patrones estructurales *Facade*, *Composite*, *Decorator* y *Adapter*

Patrón *Fachada*

Por este patrón se proporciona una interfaz única de un subsistema o componente de un sistema, para facilitar el uso del sistema mediante el bajo acoplamiento. Es un patrón muy importante para garantizar el bajo acomplamiento. El bajo acoplamiento y la alta cohesión, son **criterios de calidad** en el diseño de software orientado a objetos. Garantizando ambos, el software se hace más reusable, mantenible y fácil de probar. En la Figura 1.22 puede verse un esquema de un diagrama de clases antes y después de la aplicación del patrón y en la Figura 1.23 la estructura de un diagrama con el patrón ya aplicado.

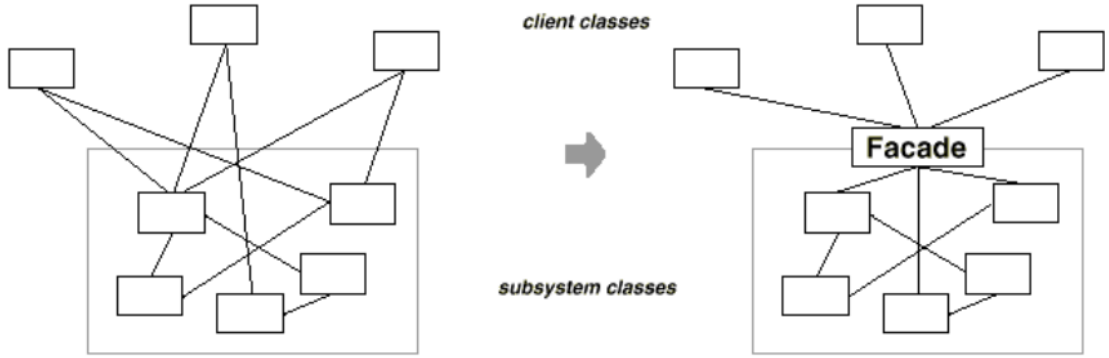

Figura 1.22: Un esquema de diagrama de clases antes y después de la aplicación del patrón Fachada [Fuente: (Gamma et al., 1994b), p. 208].

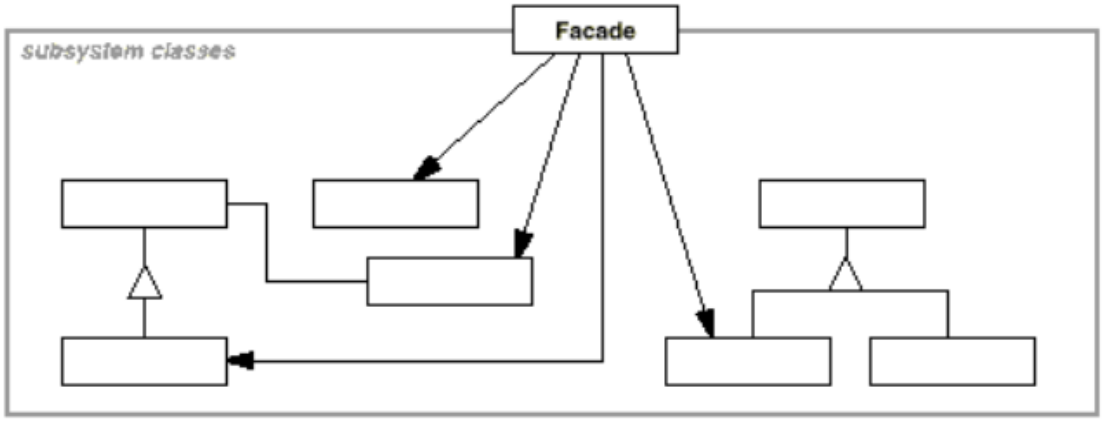

Figura 1.23: Estructura del patrón Fachada [Fuente: (Gamma et al., 1994b), p. 210].

Patrón *Composición*

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. Esto permite representar los objetos en forma de árbol. Es apropiado cuando el cliente o usuario de esos objetos no quiera distinguir si son compuestos o simples. La Figura 1.24 muestra la estructura del patrón, mientras que la Figura 1.25 muestra la estructura jerárquica de los objetos compuestos.

Un ejemplo concreto es el uso del patrón en un editor de dibujo, tal y como aparece en la Figura 1.26 con el diagrama de clases que representa la estructura del patrón y en la Figura 1.27 con una jerarquía posible de elementos gráficos según el patrón.

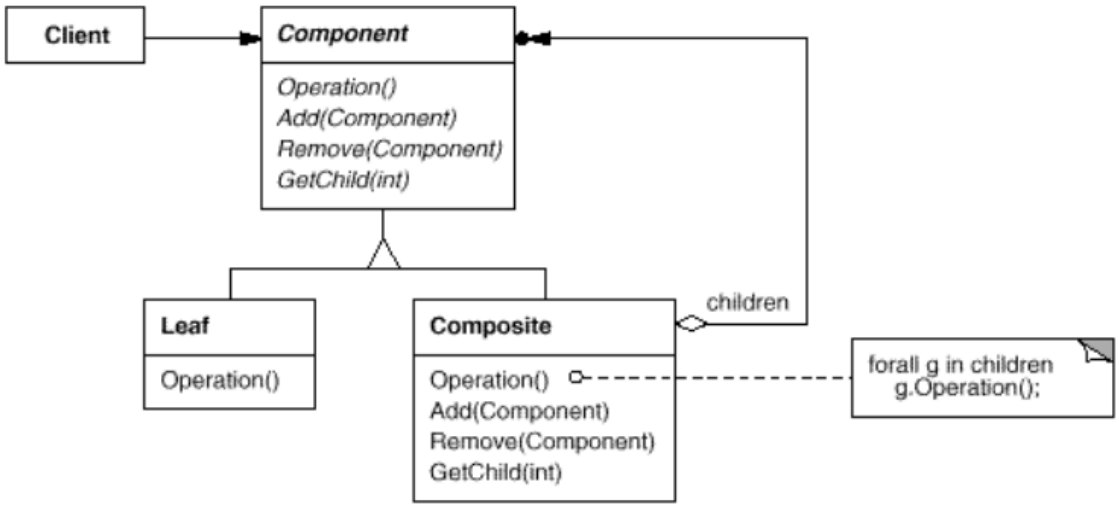

Figura 1.24: Estructura del patrón Composición [Fuente: (Gamma et al., 1994b), p. 185].

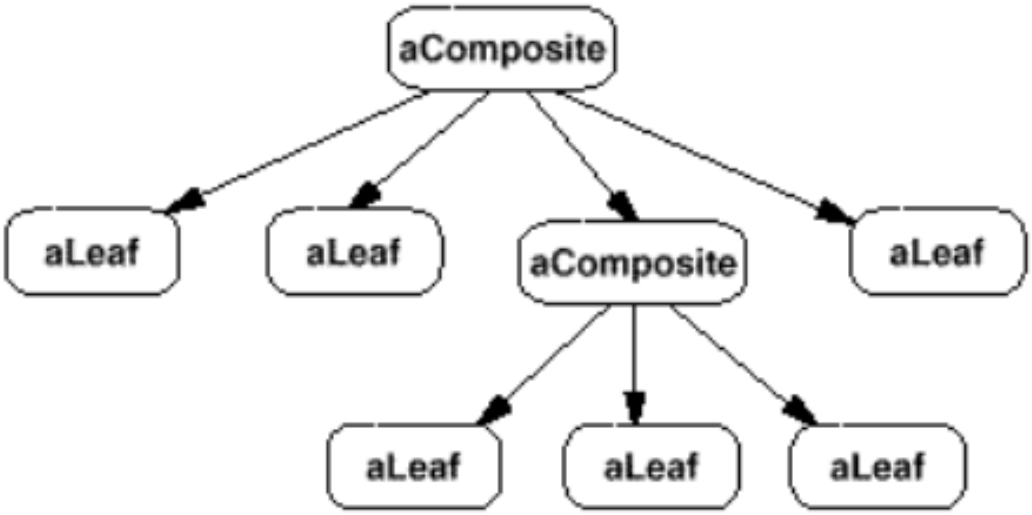

Figura 1.25: Un ejemplo de jerarquía de objetos donde se ve la relación entre objetos compuestos y simples para el patrón "Composición" [Fuente: (Gamma et al., 1994b), p. 185].

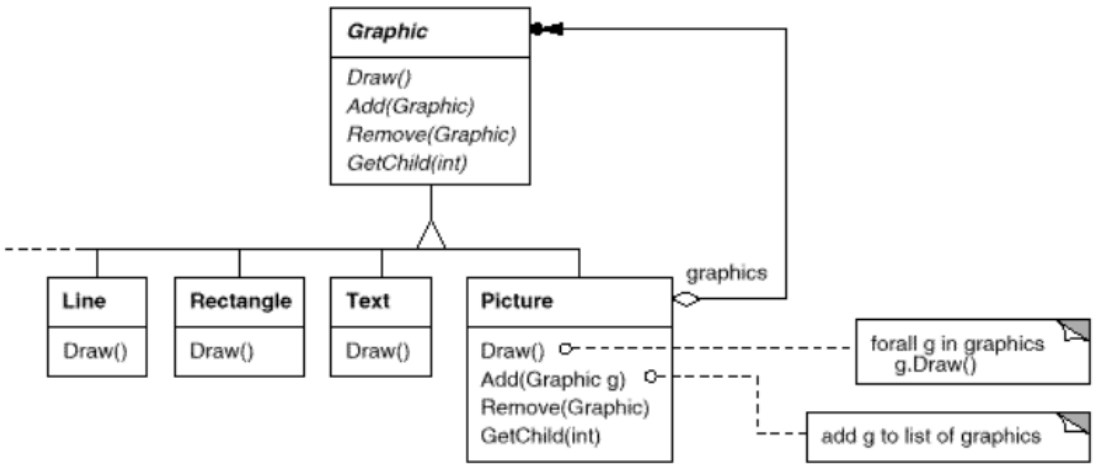

Figura 1.26: Estructura del patrón Composición en el ejemplo de componentes gráficos. [Fuente: (Gamma et al., 1994b, pg. 183)].

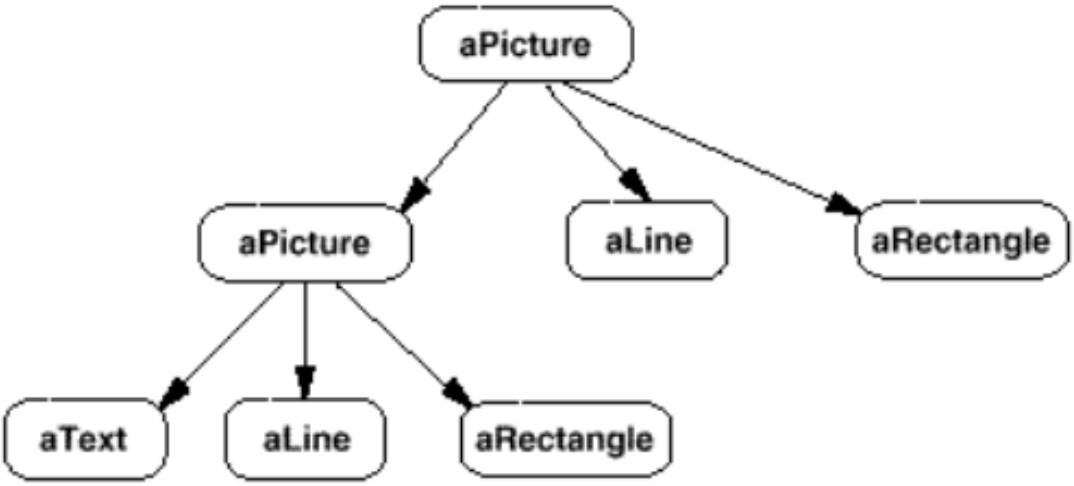

Figura 1.27: Una de jerarquía de objetos en el ejemplo de componentes gráficos [Fuente: (Gamma et al., 1994b, pg. 184)].

Patrón *Decorador* **(*Wrapper*)**

También conocido como *Envoltorio* (del inglés, wrapper). Su función es la de proporcionar funcionalidad adicional a un objeto de forma dinámica. Esto supone una interesante alternativa a la herencia como forma de extender funcionalidad que puede añadir flexibilidad. Se aplica si queremos dotar de funcionalidad distinta a sólo algunos objetos, especialmente si ésta varía, o cuando no podemos extender las clases. La alternativa si se pudieran extender las clases, podrían ser diversos criterios de subclasificación, lo que llevaría a un número demasiado elevado de subclases de las que, a su vez, podrían heredar nuevas subclases. La Figura 1.28 muestra la estructura del patrón.

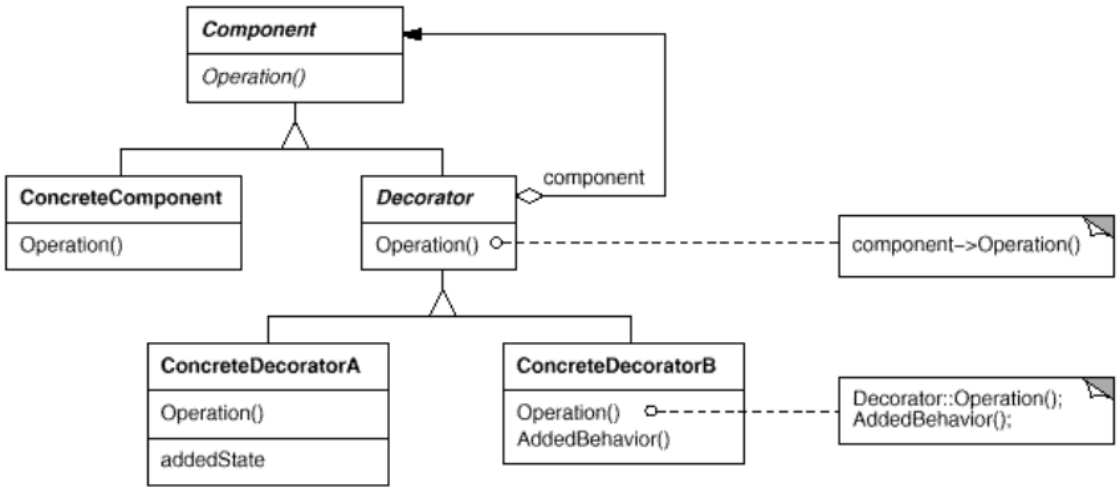

Figura 1.28: Estructura del patrón decorator [Fuente: (Gamma et al., 1994b), p. 199].

Como se ve en la figura, un objeto de la clase *Decorator* reenvía los mensajes al objeto de la clase *Component* que está decorando. Además puede tener sus métodos propios que ejecutar antes o después de hacer el reenvío.

El mayor problema que presenta este patrón es que 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.

La Figura 1.29 muestra la estructura del patrón aplicada a un ejemplo concreto de un cuadro de texto (clase $TextView$) que se "decora" con una ventana que permite hacer (clase $ScrollDecorator$) y con un borde (clase $BorderDecorator$). La aplicación y el resultado puede verse de forma gráfica en la Figura 1.30.

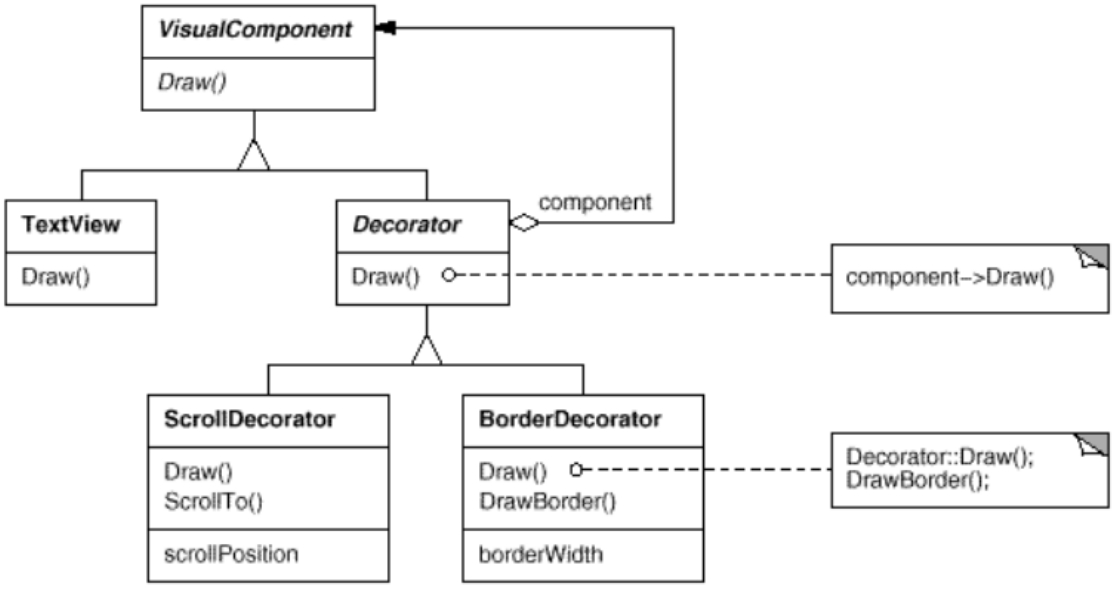

Figura 1.29: Ejemplo de estructura del patrón decorator aplicada a la presentación gráfica de un texto [Fuente: (Gamma et al., 1994b), p. 198].

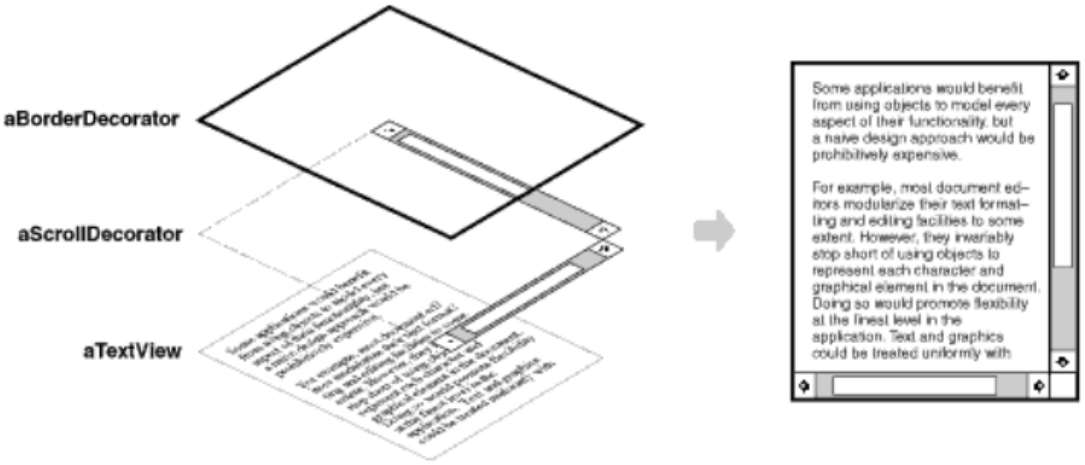

Figura 1.30: Aspecto de un cuadro de texto y sus decoradores [Fuente: (Gamma et al., 1994b, 197)].

La estructura del patrón es similar a la del patrón "Composición", y puede parecer que este patrón es una versión degenerada del patrón "Composición" con un sólo componente.

Pero esto no es cierto, pues tienen funciones diferentes. El patrón "Decorador" está hecho para añadir funcionalidad a los objetos de forma dinámica mientras que el patrón "Composición" pretende unificar la interfaz de los objetos compuestos con sus componentes para que el cliente de los mismos pueda tratarlos de la misma forma.

Sin embargo, a veces pueden coexistir cuando se busque cubrir ambos objetivos, haciéndolo del siguiente modo:

> «There will be an abstract class with some subclasses that are composites, some that are decorators, and some that implement the fundamental buildingblocks of the system. In this case, both composites and decorators will have a common interface. From the point of view of the Decorator pattern, a composite is a *ConcreteComponent*. From the point of view of the Composite pattern, a decorator is a *Leaf*. Of course, they don't have to be used together and, as we have seen, their intents are quite different.» (Gamma et al., 1994b, pág. 247)

Patrón *Adaptador*

Este patrón convierte la interfaz de una clase en otra que se adapte a lo que el cliente esperaba para que pueda así usar esa clase.

Este patrón se suele aplicar en una fase de diseño avanzada, en la que ya hemos terminado el diseño de clases que usan otras, considerando que tienen una interfaz concreta y hemos observado que la misma funcionalidad están implementada en otro lugar, quizás en una librería o en otro módulo o paquete y queremos hacerlas reusables, de tal modo que no debamos/podamos cambiar nuestro código ni tampoco el de las clases a usar. El adaptador convertirá la interfaz de las clases externas para que puedan ser usadas por el código que ya hemos implementado.

Pero también puede aplicarse 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 hay razones para no usar en las clases que se desarrollan la misma interfaz de las clases que se reutilizarán, quizás porque en el futuro se reutilizarán otras con una interfaz distinta.

Hay dos versiones del patrón, de clase (ver Figura 1.31) y de objeto (ver Figura 1.32).

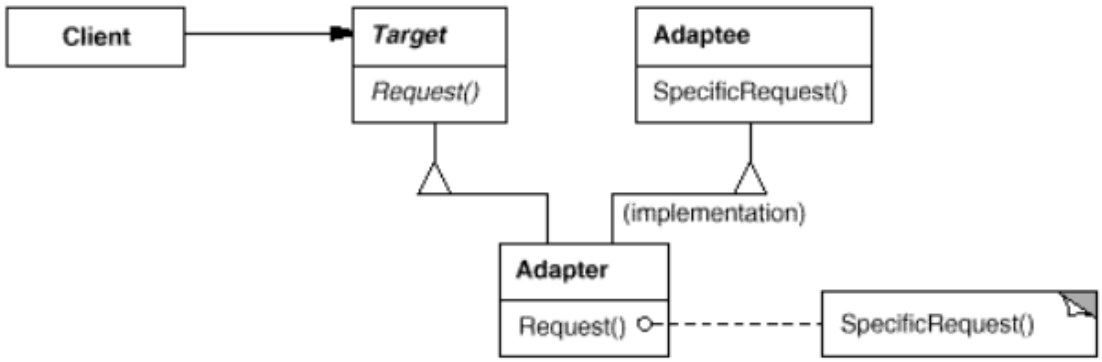

Figura 1.31: Estructura del patrón adaptador, en un ámbito de clase [Fuente: (Gamma et al., 1994b, pg. 159)].

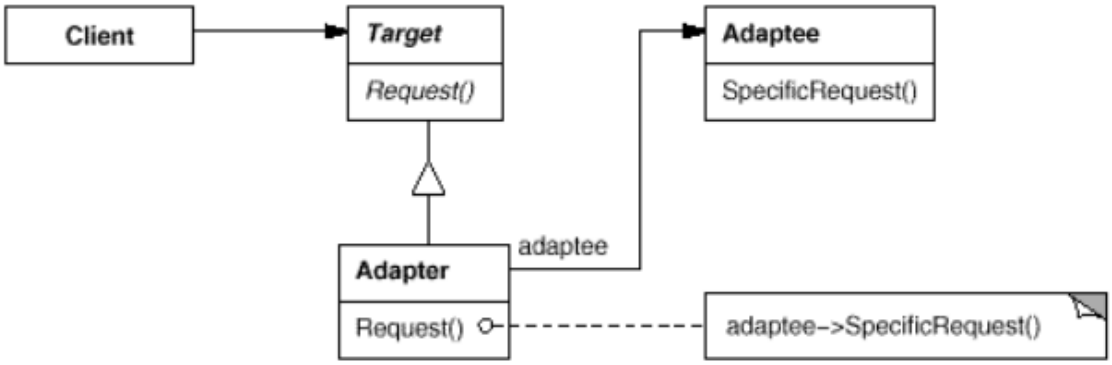

Figura 1.32: Estructura del patrón adaptador en un ámbito de objeto [Fuente: (Gamma et al., 1994b, pg. 159)].

La versión del ámbito de objeto está más indicada si queremos usar una gran variedad de subclases ya existentes. Con este patrón, se proporcionará una interfaz única adaptando la interfaz de la clase padre. La versión del ámbito de clase requiere del uso de herencia múltiple.

En la Figura 1.33 se muestra la estructura del patrón "Adapter" aplicado a un editor de dibujo. El software externo tiene un componente de texto pero con un nombre de clase y métodos distintos a los esperados.

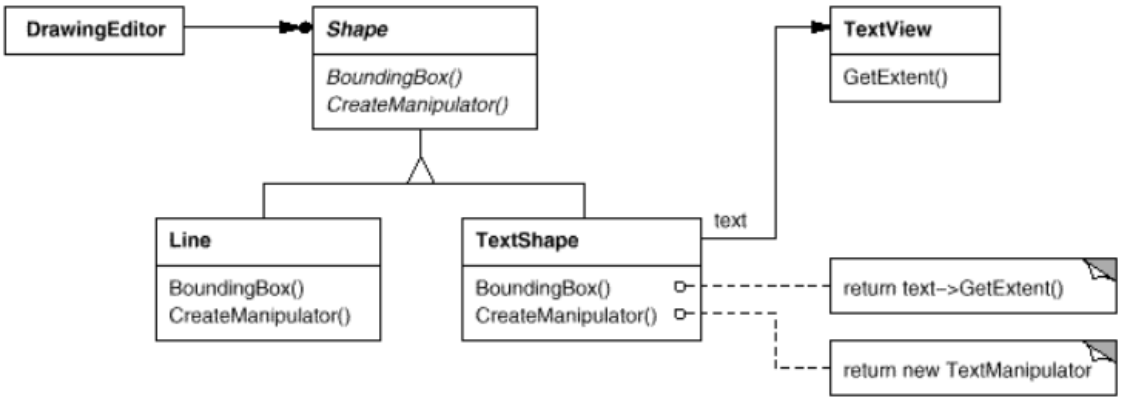

Figura 1.33: Estructura del patrón "Adapter" aplicado a un editor de dibujo [Fuente: (Gamma et al., 1994b, pg. 158)].

1.3.3. Patrones conductuales *Observer*, *Visitor*, *Strategy*, *TemplateMethod* e *InterceptingFilter*

Patrón *Observador*

Este patrón también se conoce con el nombre de *Dependientes* o *Publicar – Subscribir*. La idea es que, cuando un objeto tiene varios objetos que dependen de él, la consistencia entre ellos se garantice sin que sea a costa de aumentar el acoplamiento. Así, cada cambio en el estado del primer objeto –llamado también "sujeto observable"– es notificado, no a cada uno de ellos directamente, sino a modo de publicación común a todos los objetos en una lista de suscriptores (todos los objetos dependientes, los observadores), de forma que todos ellos recibirán simultáneamente la notificación por estar subscritos. Se pueden subscribir cualquier número de observadores a la lista de subscripción del sujeto observable.

En este patrón se define una interfaz para los observables que declara métodos para añadir o quitar suscriptores y para notificarles los cambios (clase abstracta *Subject* en la Figura 1.34 y otra interfaz para los observadores que declara un método de actualización cada vez que se les notifica un cambio (clase abstracta *Observer* en la Figura 1.34).

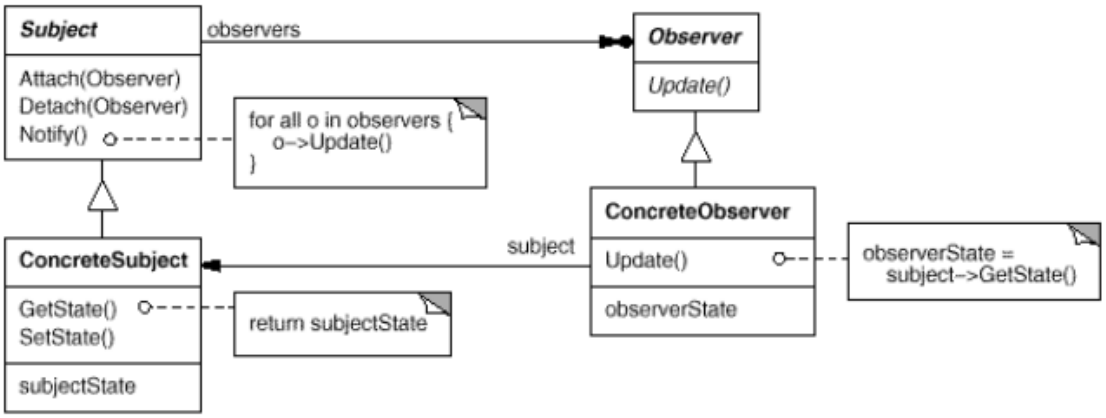

Figura 1.34: Estructura del patrón Observador [Fuente: (Gamma et al., 1994b), p. 328]

Este patrón se suele aplicar para desacoplar la funcionalidad de un sistema de su presentación al usuario. De este modo, varias interfaces distintas de usuario se pueden implementar sin tener que tocar la funcionalidad de la aplicación. Es un patrón que se aplica a menudo junto con el diseño arquitectónico MVC, que es considerado también un patrón, pero a nivel arquitectónico.

Patrón *Visitante*

Este patrón busca 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. Este patrón debe utilizarse cuando se quieren llevar a cabo muchas operaciones dispares sobre objetos de una estructura de objetos sin tener que incluir dichas operaciones en las clases. Dado que este patrón separa un algoritmo de la estructura de un objeto, es ampliamente utilizado en intérpretes, compiladores y procesadores de lenguajes, en general. Se debe utilizar este patrón si se quiere realizar un cierto número de operaciones, que no están relacionadas entre sí, sobre instancias de un conjunto de clases, y no se quiere "contaminar" a dichas clases.

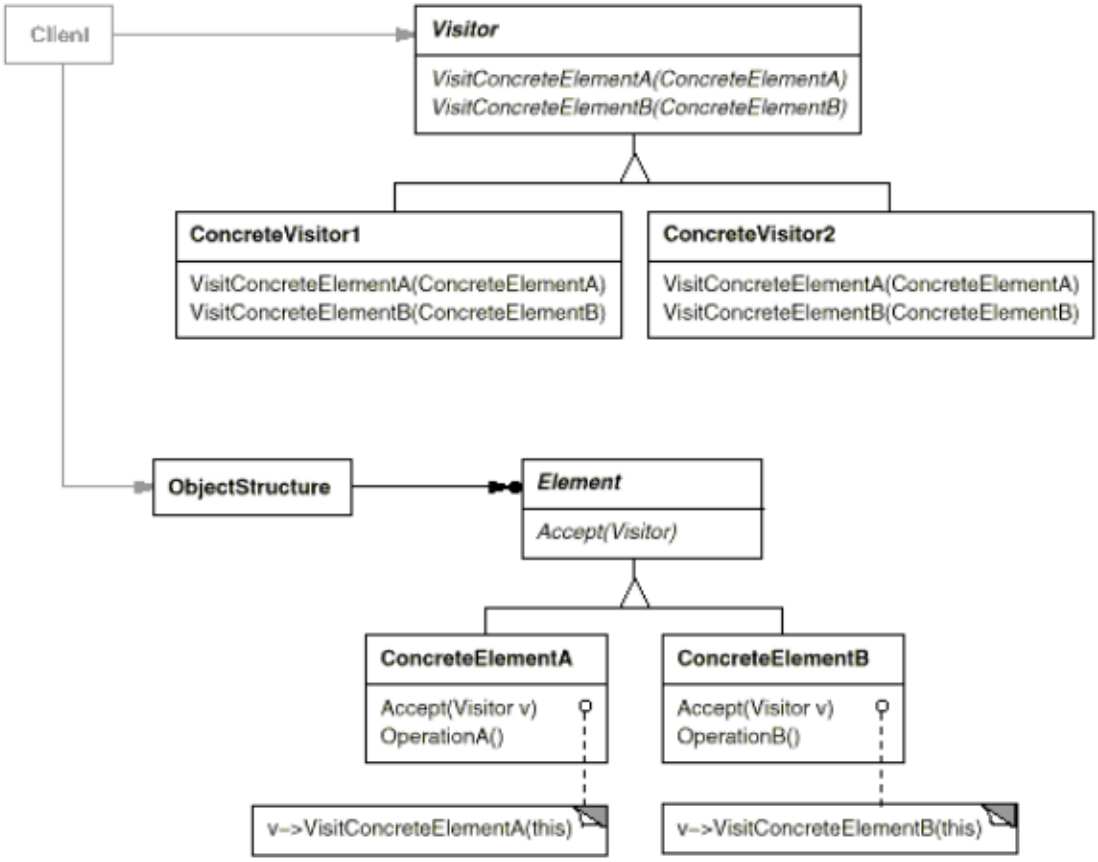

Figura 1.35: Estructura del patrón "Visitante" [Fuente: (Gamma et al., 1994b, pg. 369)]

Patrón *Estrategia*

Este patrón 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.

Se aplica cuando se prevé que la lista de algoritmos alternativos pueda ampliarse, y/o cuando son muchos y no siempre se usan todos. Así, en vez de que formen parte de una clase Cliente, se proporcionan como clases aparte accesibles por parte del Cliente a través de una interfaz común a todos. La Figura 1.36 muestra la estructura del patrón. La Figura 1.37 muestra la estructura cuando se ha aplicado a un ejemplo de algoritmos para partir el texto por líneas en un programa que muestra el aspecto final del texto (visor de textos). La clase *Composition* tiene la responsabilidad de mantener y actualizar los saltos de línea que se muestran en el visor.

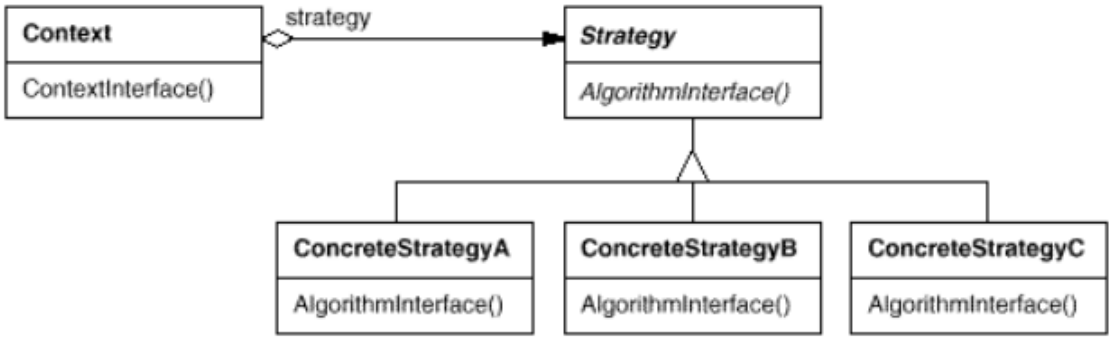

Figura 1.36: Estructura del patrón "Estrategia" [Fuente: (Gamma et al., 1994b, pg. 351)]

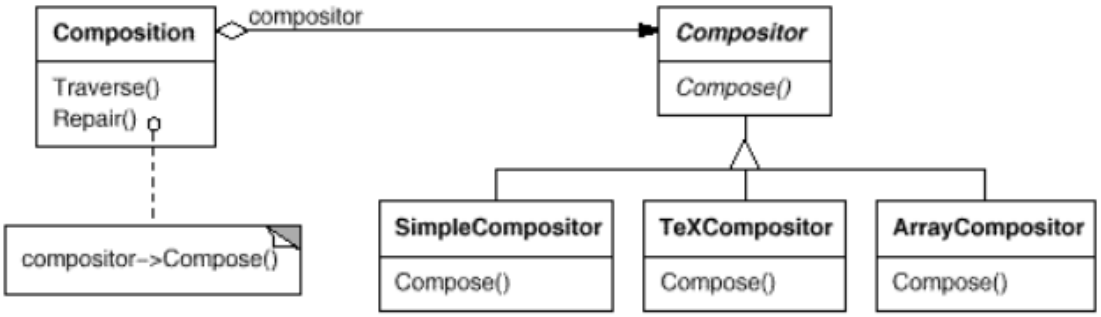

Figura 1.37: Estructura del patrón "Estrategia" en un ejemplo de visualización de textos [Fuente: (Gamma et al., 1994b, pg. 349)]

Patrón *Método Plantilla*

Un método plantilla es un método de una clase abstracta[2] que define una secuencia de pasos (métodos) que son usados por un algoritmo de la clase, y deja la implementación de cada uno de los métodos concretos a las subclases.

Es una técnica básica muy importante de reutilización de código. Permite además poder controlar qué métodos que forman parte del algoritmo se podrán sobreescribir en las subclases –los llamados métodos gancho (del inglés hook)– y cuáles deben sobreescribirse forzosamente, declarando estos últimos como abstractos en la clase.

La Figura 1.38 muestra un diagrama de clases con la estructura de este patrón. Los métodos plantilla pueden llamar a distintos tipos de operaciones (Gamma et al., 1994b, pg. 363)]:

[2]Forzosamente será parcialmente abstracta pues la clase debe al menos implementar el método plantilla.

- 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

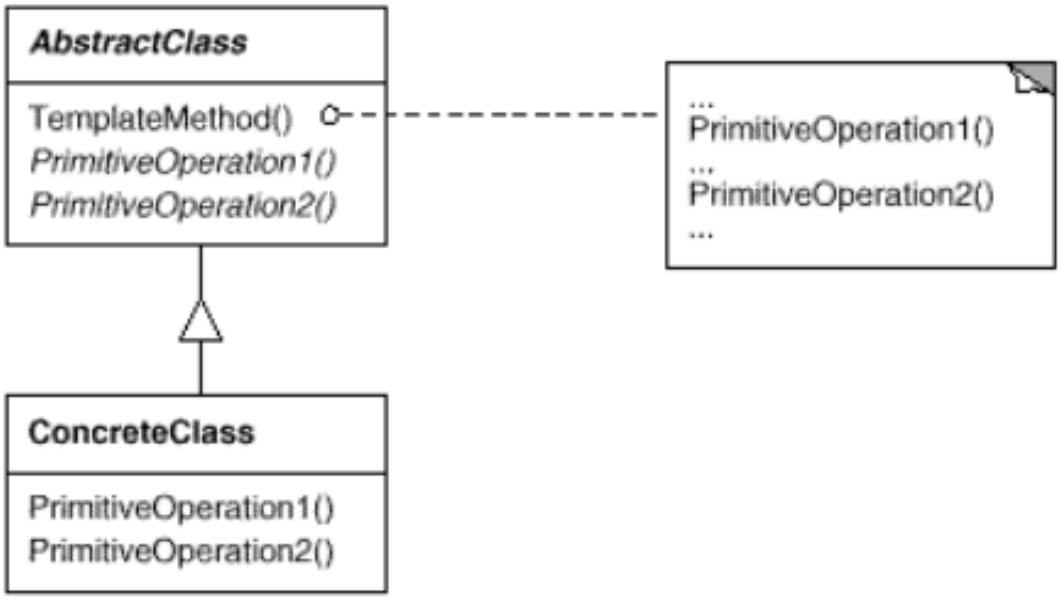

Figura 1.38: Estructura del patrón "Método Plantilla"[Fuente: (Gamma et al., 1994b, pg. 362)]

El uso de métodos gancho 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). Una extensión normal de un método *operation* de una *ParentClass* en una *DerivedClass* se implementaría de la siguiente forma en C++:

```
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*

Este patrón (ver Figura 1.39) está basado en (1) el patrón Interceptor, que permite añadir servicios de forma transparente que puedan ser iniciados de forma automática y en (2) el patrón Filtros de encauzamiento, que encadena los servicios de forma que la salida de uno es el argumento de entrada del siguiente. En concreto, el patrón Filtros de intercepcion permite añadir un componente de filtrado antes de la petición de un servicio (pre-procesamiento) o después su respuesta (post-procesamiento). 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[3]. 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.

Como puede verse en la Figura 1.39, este patrón tiene un $GestorDeFiltros$ ($FilterManager$), una $CadenaDeFiltros$ ($FilterChain$) y varios objetos $Filtro$, que son los componentes de procesamiento que interceptan la petición de la tarea principal de la clase Objetivo ($Target$) que solicita la clase $Cliente$. La $CadenaDeFiltros$ proporciona varios filtros al $GestorDeFiltros$ y los ejecuta en el orden en que fueron introducidos en la aplicación. El $GestorDeFiltros$ 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).

[3]Cf. https://www.oracle.com/technetwork/java/interceptingfilter-142169.html

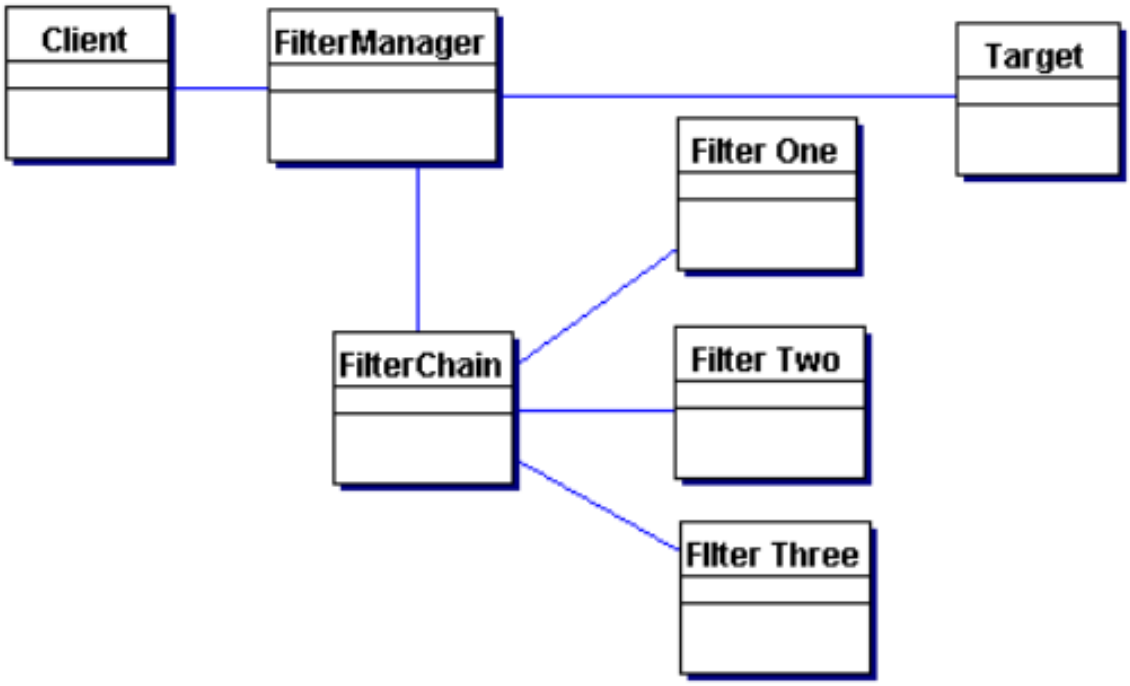

Figura 1.39: Diagrama de clases correspondiente al patrón interceptor de filtros. [Fuente: Patrón Filtros de intercepción.]

A continuación se explican las entidades de modelado necesarias para programar el 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*).

Bibliografía

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.