Tema 1. Desarrollo utilizando patrones de diseño y arquitectónicos

Desarrollo de Software
Curso 2022-2023
3º - Grado Ingeniería Informática

Dpto. Lenguajes y Sistemas Informáticos
ETSIIT
Universidad de Granada

3 de marzo de 2023

Tema 1. Desarrollo utilizando patrones de diseño y arquitectónicos

Contenidos

Desarrollo utilizando patrones de diseño y arquitectónicos

En este tema estudiaremos el concepto de patrón software y sus tipos (sección 1.1) y la forma de resolver problemas de diseño (sección 1.2 para pasar después a estudiar con detalle patrones de diseño (sección 1.3) y, por último, patrones o estilos arquitectónicos (sección 1.4).

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,

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

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 solo 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.

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).

Otra definición es:

> A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate (Buschmann et al., 1996).

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 C++ and (sometimes) Smalltalk 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 La 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).

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	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)].

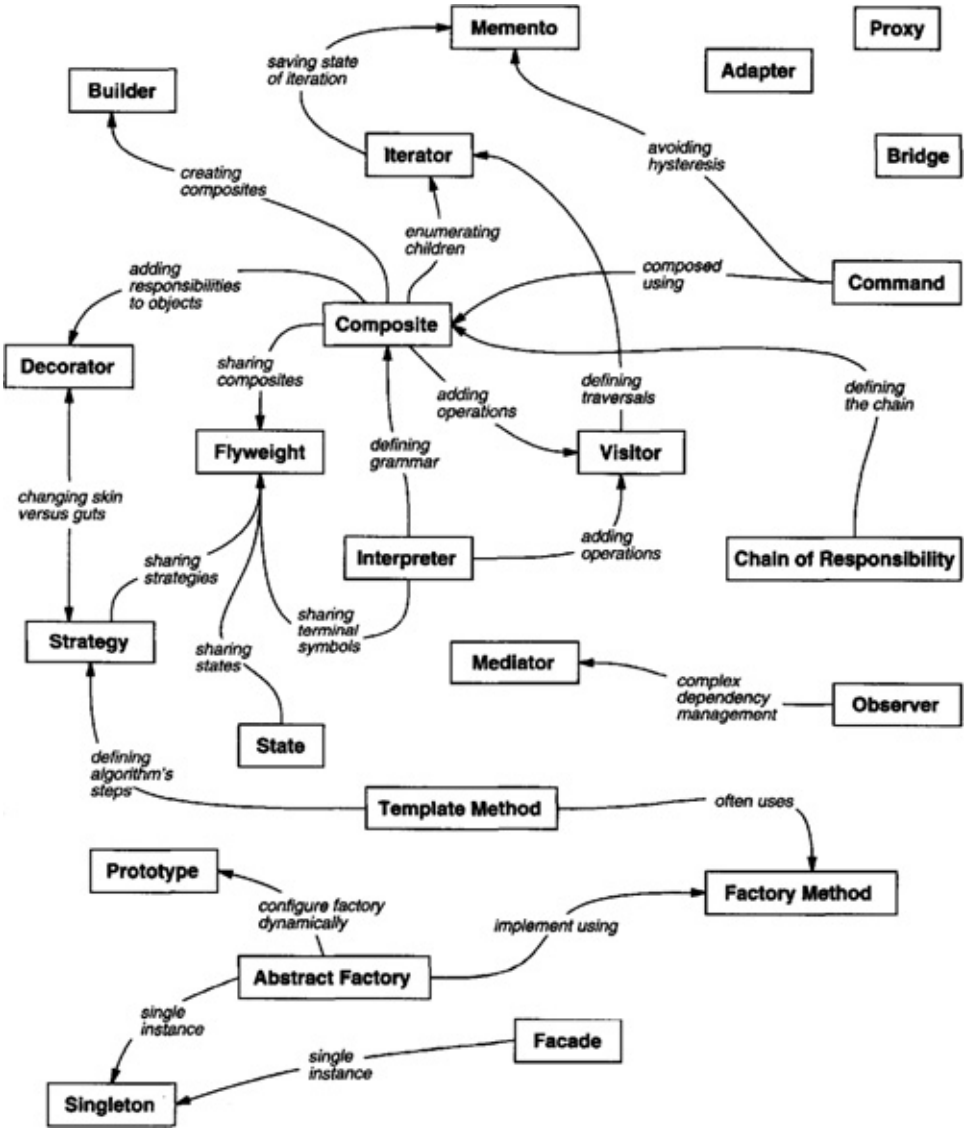

Figura 1.3: Relación entre los patrones de diseño. [Fuente: (Gamma et al., 1994b), p. 23]

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.

- 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 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	el nombre dado al patrón
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 a partir de tres clases 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 (salida) y otro para el controlador (captura la petición del usuario y la pasa al

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

modelo y/o la vista). Pero la idea es siempre la misma, separar el objeto de la aplicación (el modelo) de la interacción con el usuario (entradas y salidas). 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. Debe ser además válido en configuraciones multiusuario donde el mismo modelo es accesible simultáneamente por varios usuarios, con distintas modalidades de interfaces. 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. Y lo mismo ocurre con el framework para aplicaciomnes web Ruby on Rails. La GoF utiliza el diseño MVC para identificar e introducir los tres primeros patrones de su libro (Gamma et al., 1994a), que se refieren a continuación:

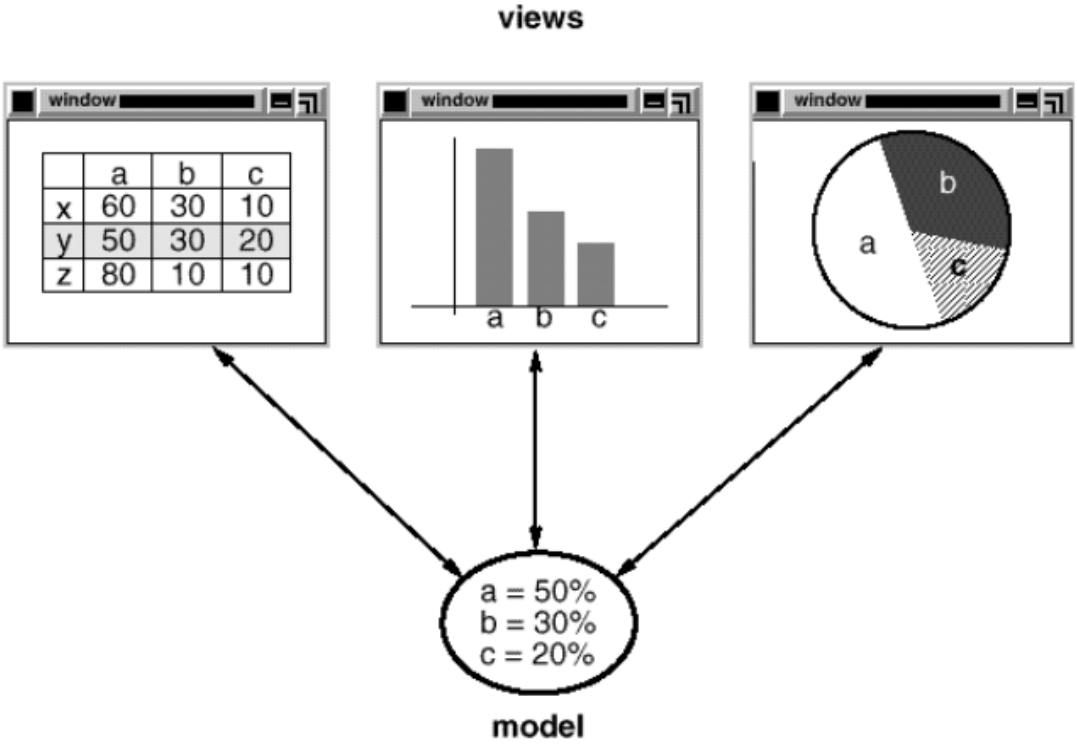

Figura 1.4: Ejemplo de modelo con tres vistas, en un diseño MVC. [Fuente: (Gamma et al., 1994b), p. 15]

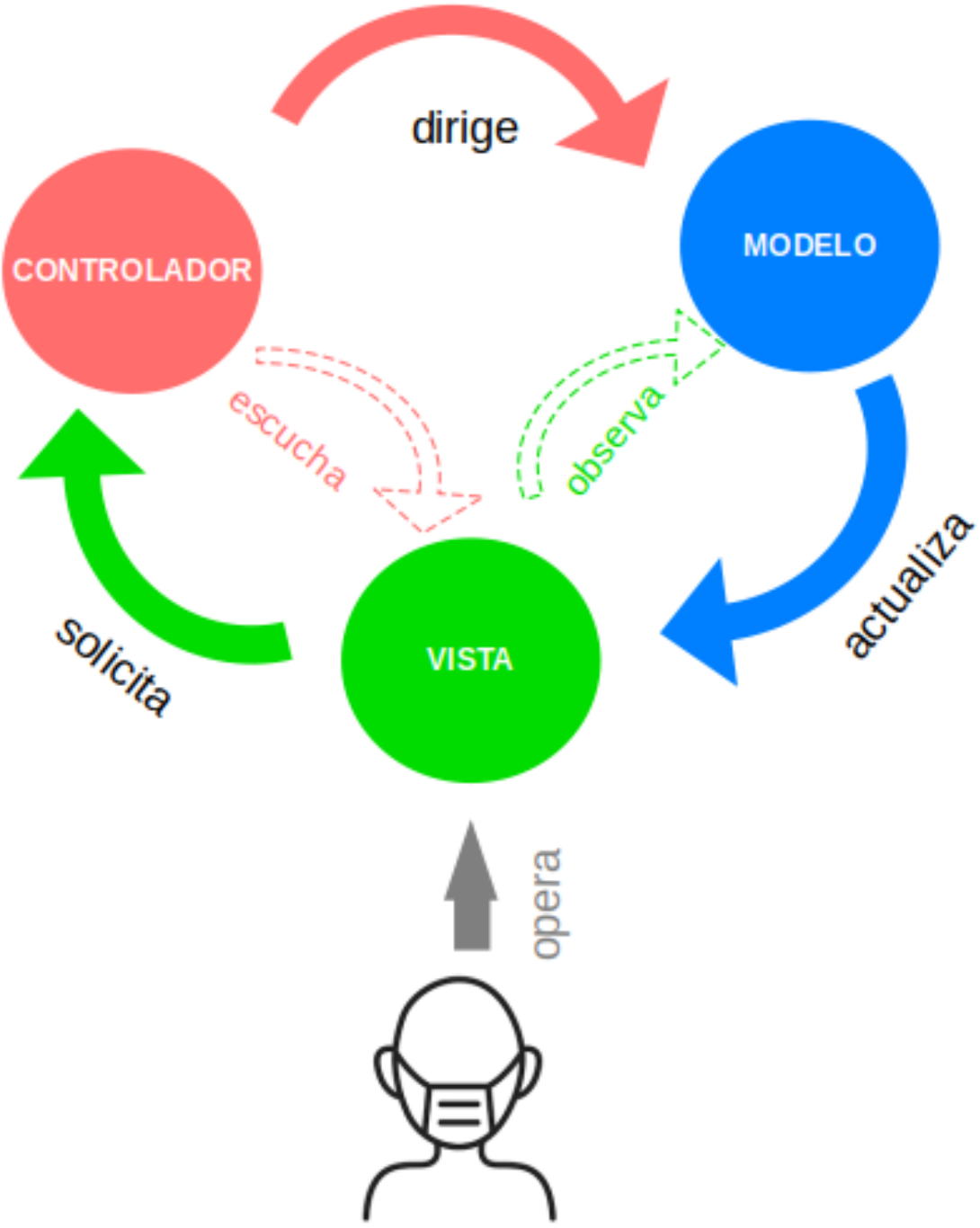

Figura 1.5: Ejemplo de diagrama relacional de la terna de clases en el diseño MVC.

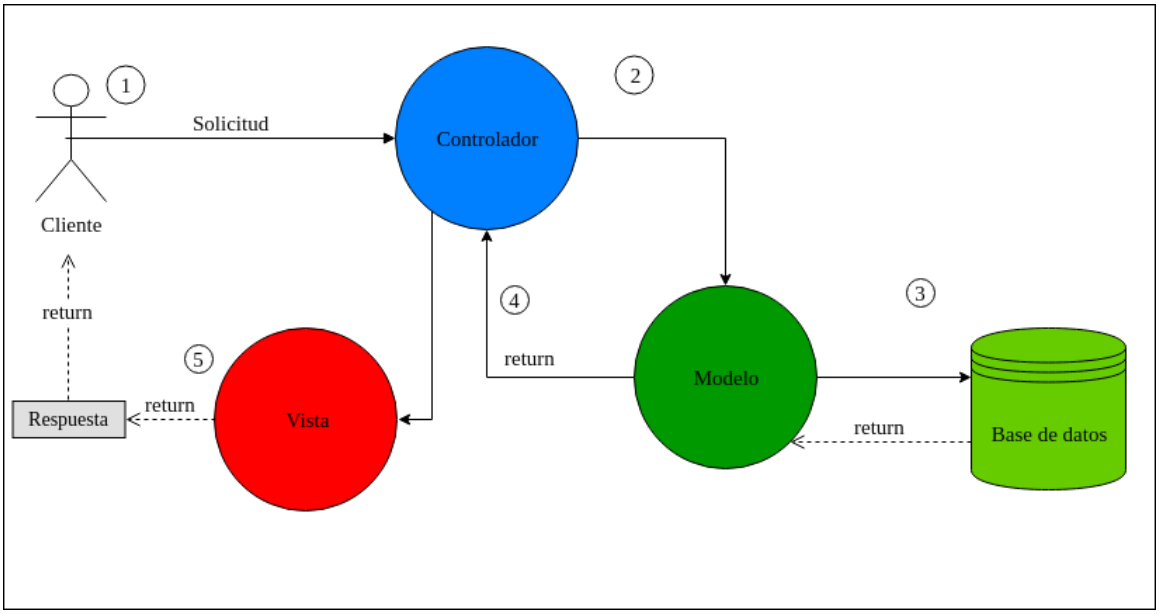

Figura 1.6: Ejemplo más complejo de diagrama relacional en el diseño MVC. En este caso consideramos que el sistema es multiusuario, con distintas vistas para los distintos usuarios, pero un único modelo.

- *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 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 Ruby 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.*- En el diseño MVC puede haber 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 in-

cluso 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 (véase Figura 1.7.

Figura 1.7: Ejemplo aún más complejo de diagrama relacional en el diseño MVC. En este caso consideramos que el sistema es multiusuario, con distintas vistas para los distintos usuarios, pero un único modelo, como en la Figura 1.6 pero ahora además con varias instancias del controlador.

Otros patrones de MVC son el Método Factoría para especificar la clase controlador y el Decorador, para, por ejemplo, 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 suele haber una sola instancia de ellos y (2) 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 segundo el patrón *singleton* y uno del tercero el patrón *flyweight*.
- 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.

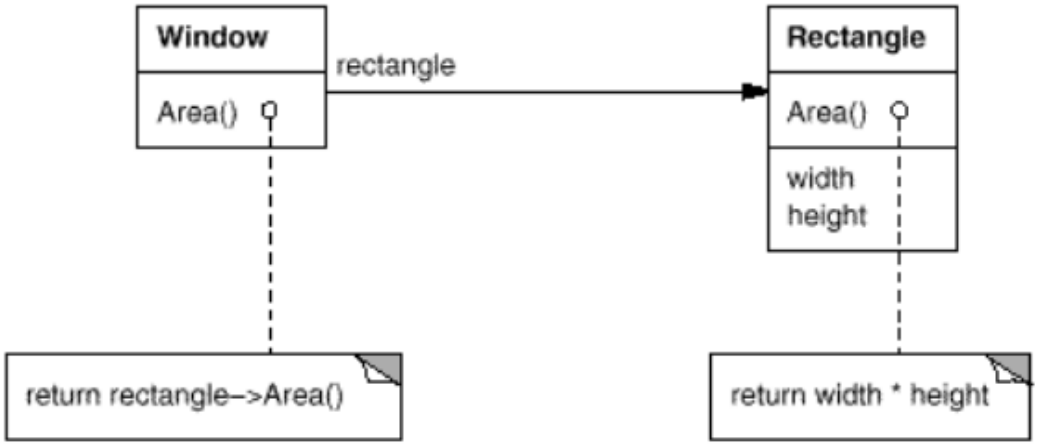

Figura 1.8: Ejemplo de uso de delegación (Gamma et al., 1994a, p.33) p. 33.

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 1.12)
- *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 1.37)
- *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 1.19)

- *Builder* (Gamma et al., 1994b) pp. 110-120
- *Visitor* (Gamma et al., 1994b) pp. 366-381 (Figura 1.38)
- *Facade* (Gamma et al., 1994b) pp. 208-217

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

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. GoF (Gamma et al., 1994a) también presentan un quinto patrón creacional, el patrón *Singleton*, que exige que una clase solo 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 cualquiera 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. Usaremos como ejemplo comparativo el juego del laberinto de GoF (Gamma et al., 1994a), pp. 94-95 (ver Figuras 1.9 y 1.10).

Figura 1.9: Ejemplo de un laberinto según este juego. Las puertas cerradas no son accesibles según 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].

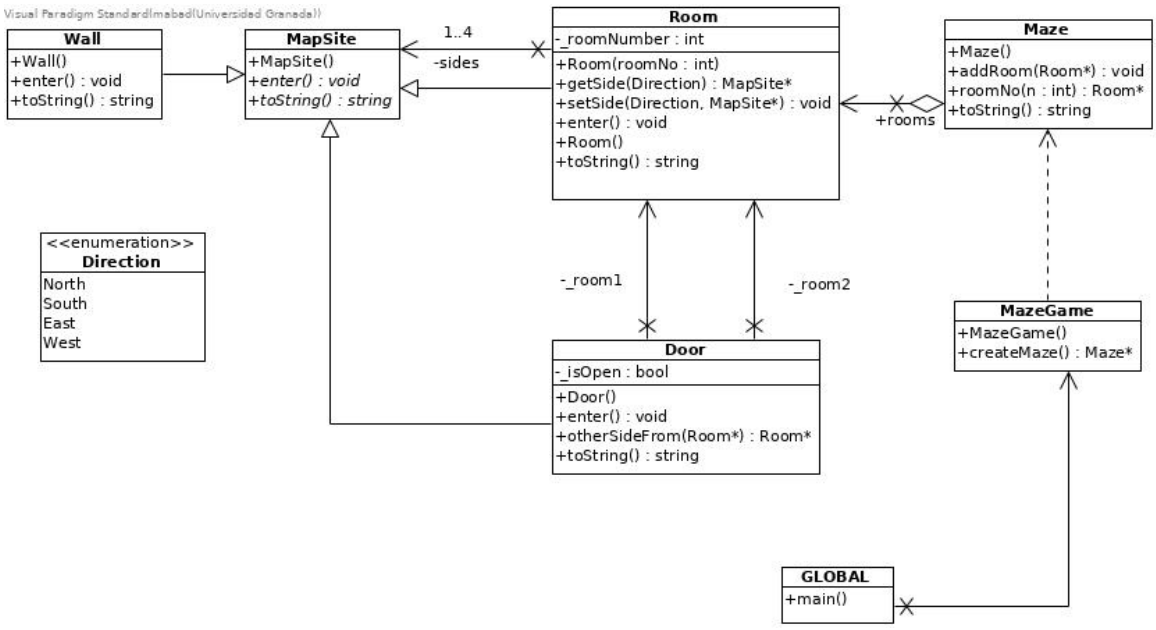

Figura 1.10: Diagrama de clases correspondiente al ejemplo del juego del laberinto de GoF (Gamma et al., 1994a).

Un ejemplo en c++ de implementación del método para crear un laberinto (*createMaze*) muestra cómo los lenguajes OO con métodos constructores especiales (i.e., los "falsos métodos" *new*, que no permiten polimorfismo ni ligadura dinámica) hacen un código muy poco flexible al posible uso de variantes en los objetos que se instancian (los "productos").

```
Maze* MazeGame::createMaze () {
Maze* aMaze = new Maze;
Room* r1 = new Room(1);
Room* r2 = new Room(2);
Door* theDoor = new Door(r1, r2);
aMaze->addRoom(r1);
aMaze->addRoom(r2);
r1->setSide(North, new Wall);
r1->setSide(East, theDoor);
r1->setSide(South, new Wall);
r1->setSide(West, new Wall);
r2->setSide(North, new Wall);
r2->setSide(East, new Wall);
r2->setSide(South, new Wall);
r2->setSide(West, theDoor);
return aMaze;
}
```

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 1.11). 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 ni ligadura dinámica, debiéndose explicitar la clase concreta que se quiere crear. Por tanto, el constructor de copia en Java o C++, tampoco puede considerarse un método factoría. Un ejemplo de método factoría por delegación[2] es el método de clonación (clone).

En el siguiente ejemplo se declaran los métodos factoría que clonan objetos:

```
//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);
   }
```

Ahora puede verse que el método clone se invoca de la misma manera en una clase y su subclase:

```
// 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();
```

Mientras que si copiamos objetos usando directamente el constructor de copia, el código no se podría compartir independientemente de la clase a instanciar, pues es específico de cada clase:

```
// Copia de objetos sin metodo factoria en una clase distinta
// 1. Creamos dos objetos de una clase y su subclase
```

[2]La clase delega su responsabilidad en una instancia suya.

```
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);
```

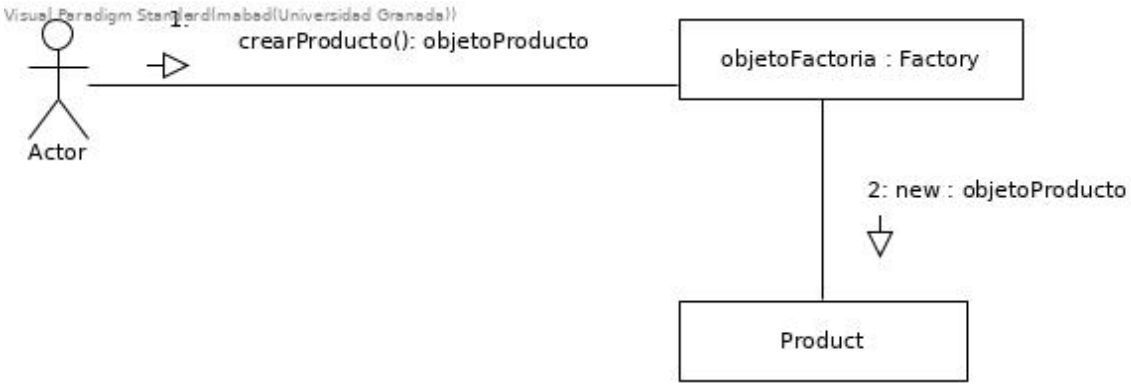

Figura 1.11: Diagrama de comunicación que muestra un ejemplo de creación de una instancia de la clase *Product* mediante el método factoría *crearProducto* de la clase *Factory*.

Los patrones creacionales tienen el objeto de permitir un código más flexible y reusable ante el posible cambio de las clases que se necesitan para crear los objetos en un programa. Por ejemplo, si quisiéramos cambiar las puertas por puertas que se abren con una palabra mágica (DoorNeedingSpelling, subclase de Door), y habitaciones por habitaciones encantadas (EnchantedRoom, subclase de Room) y quisiéramos hacer un código flexible que mantuviera el mismo método *createMaze*, podríamos usar uno de los siguientes cuatro patrones creacionales:

- *Método Factoría*.- El método *createMaze* llamará a métodos comunes ligados dinámicamente (funciones virtuales en caso de c++) para crear las habitaciones, puertas y muros necesarios, es decir, a métodos factoría que deberán crearse. Además añadiríamos una subclase de MazeGame, EnchantedMazeGame, que redefiniera esos métodos factoría (ver Figura 1.18).

- *Factoría Abstracta* con *Método Factoría*.-El método *createMaze* incluirá un parámetro (la *Factoría Abstracta*), que será usado para crear todas las habitaciones, puertas y muros necesarios, de forma que según la factoría, se crearán de un tipo u otro (ver Figura 1.15).

- *Prototipo*.- El método *createMaze* incluirá varios parámetros con los prototipos de los distintos componentes del laberinto que serán copiados para crear todas las habitaciones, puertas y muros necesarios. Si se combina con el patrón *Factoría Abstracta*, tendría un único argumento, la factoría abstracta, y se implementaría como cuando se

usa el patrón *Método Factoría*, pero esta factoría tendría prototipos (de muro, habitación y puerta) que podrían ser de distintos tipos de laberintos, creándose laberintos más variados (laberintos "mezcla") (ver Figura 1.20).

- *Builder*.- El método *createMaze* incluirá un parámetro capaz de crear un laberinto completo usando las operaciones de añadir muros, habitaciones y puertas al laberinto, y después *createMaze* puede usar la herencia para hacer cambios en las distintas partes del laberinto o en la forma que se construye (ver Figura 1.23).

Patrón *Factoría Abstracta* (*kit*)

Recomendado cuando en una aplicación tenemos líneas, temáticas o familias "paralelas" de objetos que se necesitan producir ("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 1.13) 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 1.13). 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).

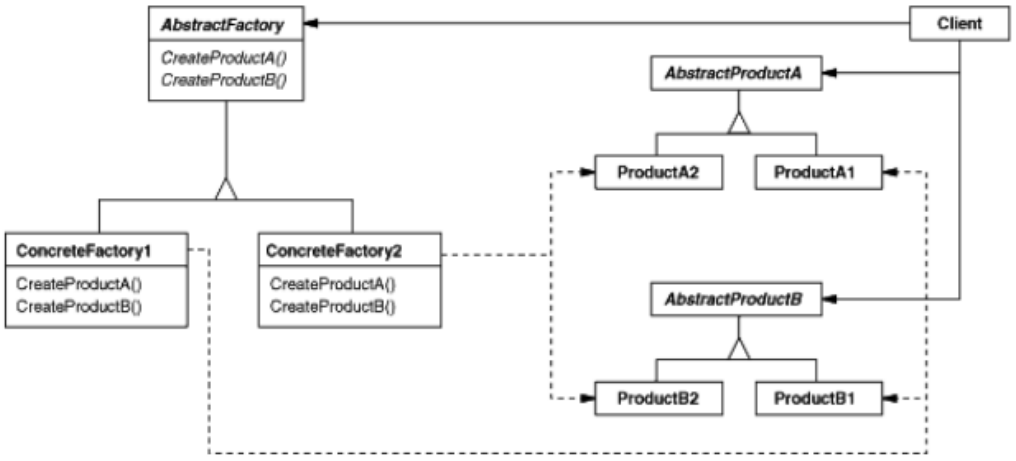

Figura 1.12: Estructura del patrón Factoría abstracta [Fuente: (Gamma et al., 1994b), p. 101]

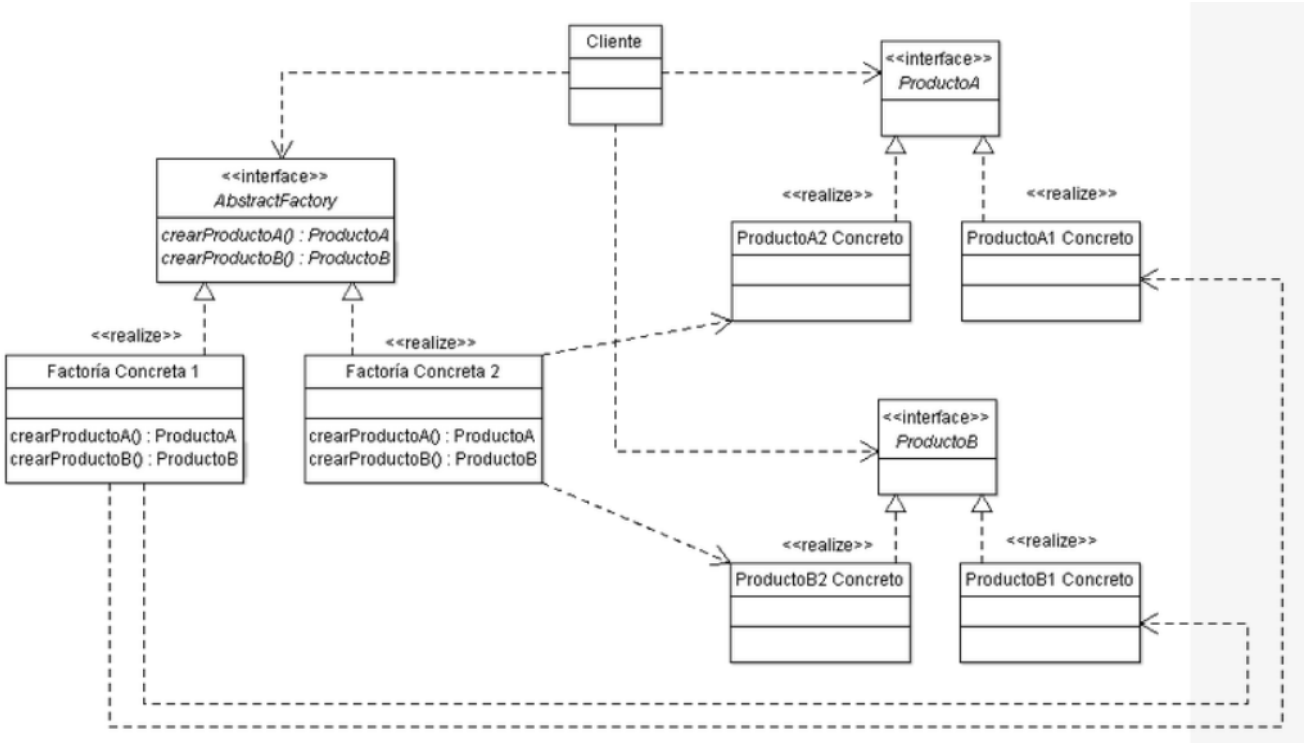

Figura 1.13: Otro diagrama de clases (más ampliado) del patrón *Factoría Abstracta*. [Fuente: Abstract Factory].

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 siempre por clonación a partir de todas las posibles clases que se quieran poder instanciar (prototipos) (ver Figura 1.14).

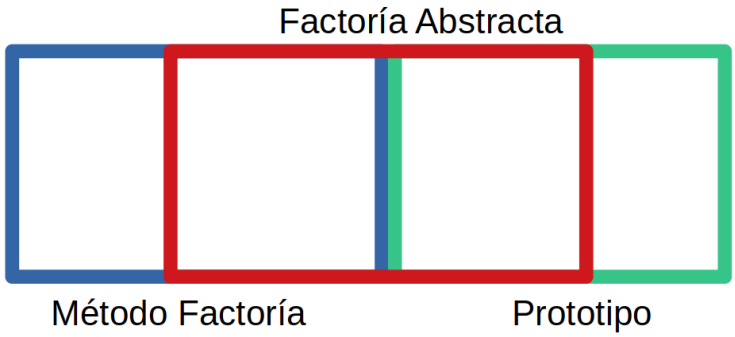

Figura 1.14: Relación de coexistencia entre los patrones *Factoría Abstracta*, *Método Factoría* y *Prototipo*.

En la Figura 1.15 puede verse el resultado de aplicar este patrón al juego del laberinto, usando métodos factoría.

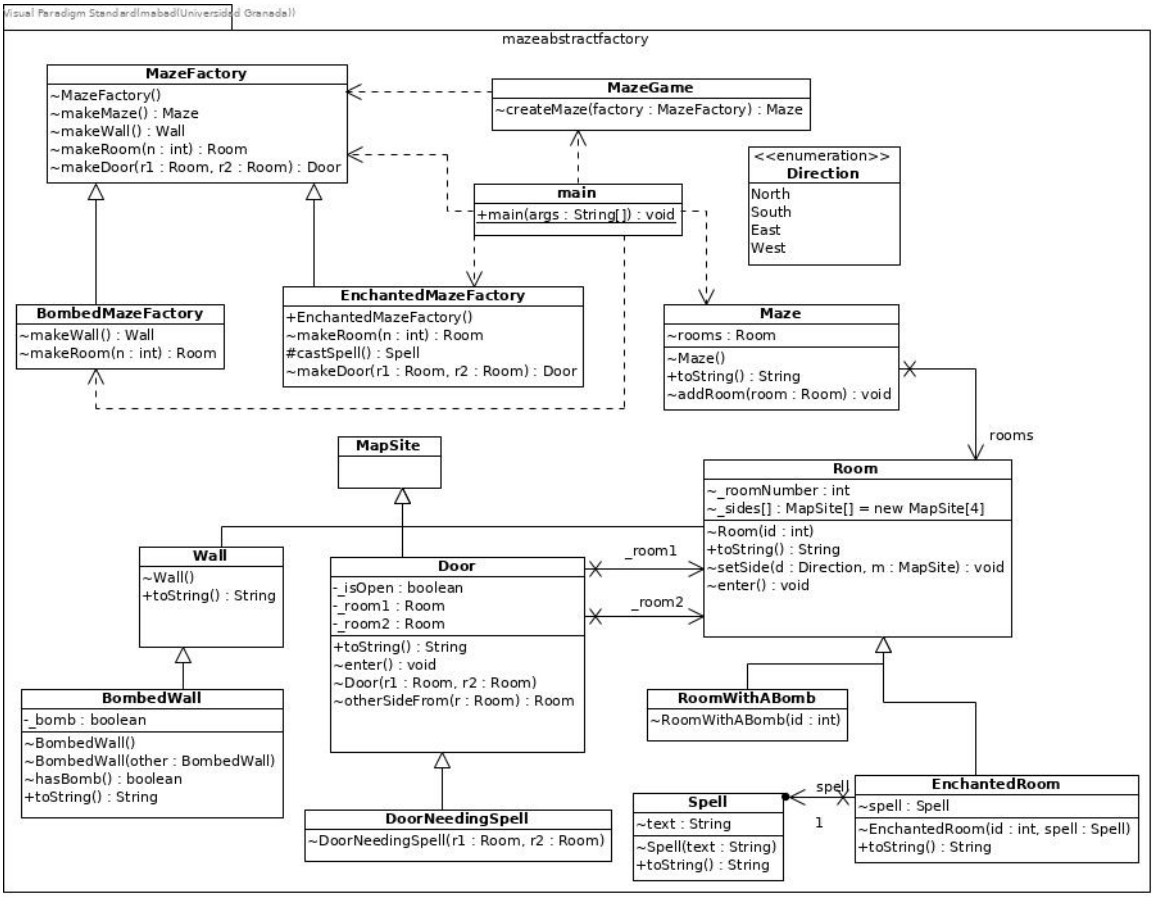

Figura 1.15: Diagrama de clases correspondiente a la aplicación del patrón *Abstract Factory* en el ejemplo del juego del laberinto de GoF (Gamma et al., 1994a). Este patrón se usa aquí junto con el patrón *Factory Method*.

Un ejemplo en c++ (asumiendo ligadura dinámica –métodos virtuales–) de implementación del método *createMaze*, que es independiente de la factoría concreta usada, puede verse a continuación.

```
Maze* MazeGame::createMaze (MazeFactory& factory) {
Maze* aMaze = factory.makeMaze();
Room* r1 = factory.makeRoom(1);
Room* r2 = factory.makeRoom(2);
Door* aDoor = factory.makeDoor(r1, r2);
aMaze->addRoom(r1);
aMaze->addRoom(r2);
r1->setSide(North, factory.makeWall());
r1->setSide(East, aDoor);
r1->setSide(South, factory.makeWall());
r1->setSide(West, factory.makeWall());
r2->setSide(North, factory.makeWall());
r2->setSide(East, factory.makeWall());
```

```
r2->setSide(South, factory.makeWall());
r2->setSide(West, aDoor);
return aMaze;
}
```

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 1.12 y 1.13).

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 1.17), 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 1.17) con un método de operación común a los posibles subtipos de productos (método *operacion* en Figura 1.17). Para cada nuevo producto (clases *ProductoTipo*, *ProductoOtroTipo* en la Figura 1.17) deberán crearse creadores o factorías concretas (*CreadorProductoTipo* y *CreadorOtroProducto* en la Figura 1.17)."

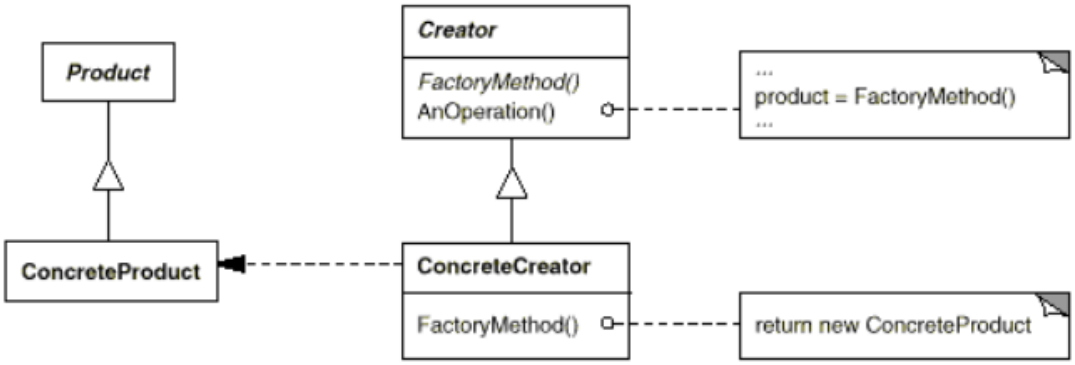

Figura 1.16: Diagrama de clases del patrón Método Factoría. [Fuente: (Gamma et al., 1994b, p. 122)].

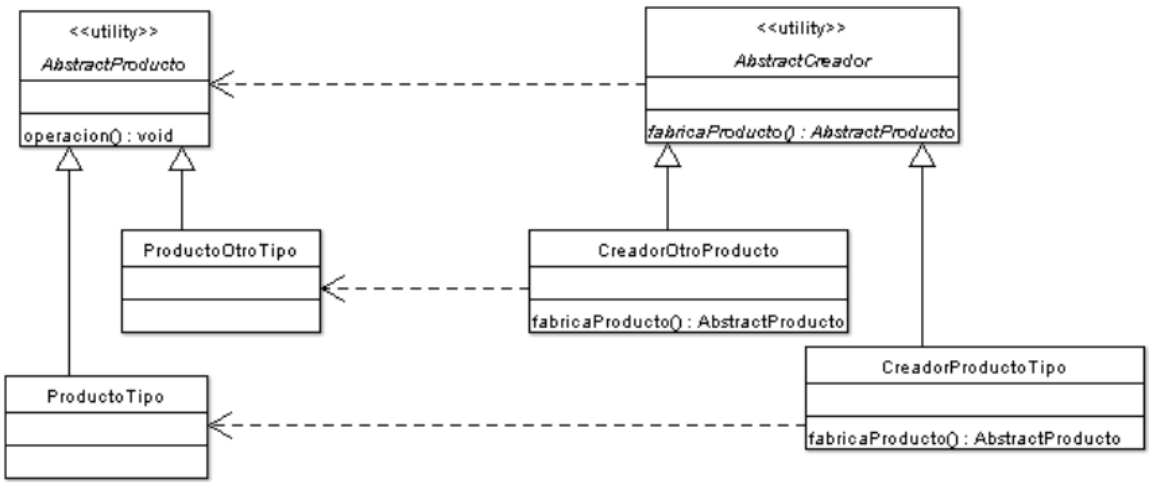

Figura 1.17: Otro diagrama de clases (más ampliado) del patrón Método Factoría. [Fuente: Factory Method].

En la Figura 1.18 puede verse el resultado de aplicar este patrón al juego del laberinto.

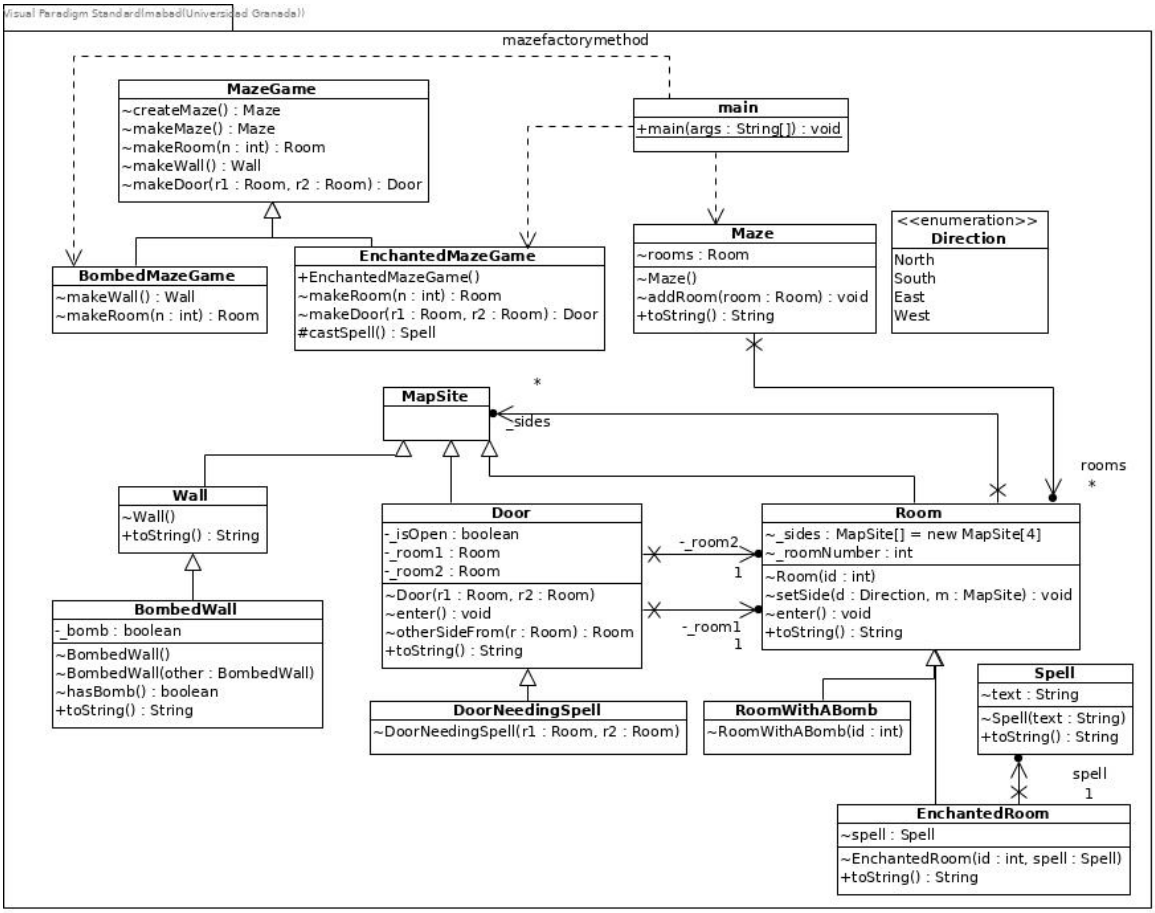

Figura 1.18: Diagrama de clases correspondiente a la aplicación del patrón *Factory Method* en el ejemplo del juego del laberinto de GoF (Gamma et al., 1994a).

Un ejemplo en c++ (asumiendo ligadura dinámica –métodos virtuales–) de implementación del método *createMaze* con el patrón *Método Factoría*, puede verse a continuación.

```
Maze* MazeGame::createMaze () {
Maze* aMaze = makeMaze();
Room* r1 = makeRoom(1);
Room* r2 = makeRoom(2);
Door* theDoor = makeDoor(r1, r2);
aMaze->addRoom(r1);
aMaze->addRoom(r2);
r1->setSide(North, makeWall());
r1->setSide(East, theDoor);
r1->setSide(South, makeWall());
r1->setSide(West, makeWall());
r2->setSide(North, makeWall());
r2->setSide(East, makeWall());
r2->setSide(South, makeWall());
r2->setSide(West, theDoor);
```

```
return aMaze;
}
```

Para construir variantes del laberinto, por ejemplo, un laberinto encantado, se declararía una subclase de MazeGame, EnchantedMazeGame, que redefiniera los métodos para costruir puertas y habitaciones, sin necesidad de redefinir el método *createMaze*.

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 1.19) 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.

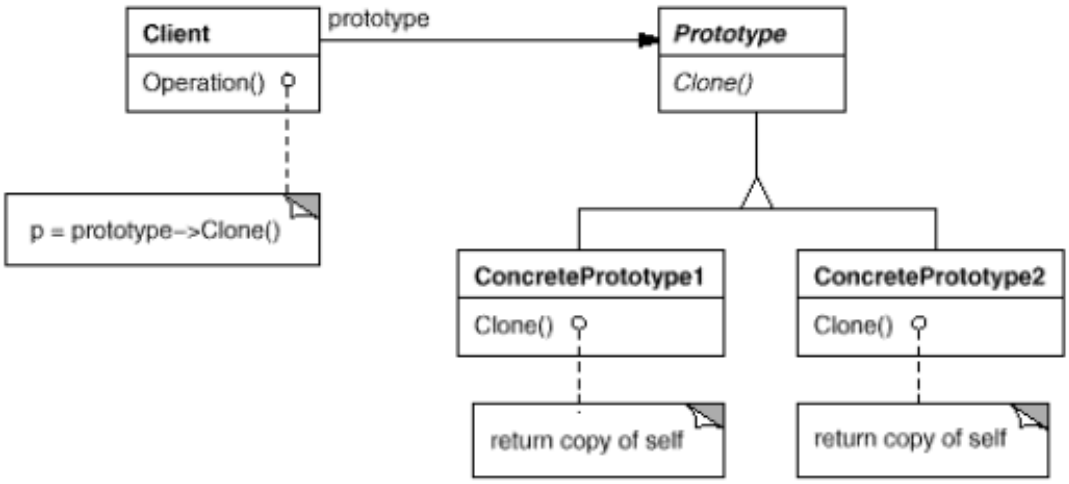

Figura 1.19: Estructura del patrón Prototype [Fuente: (Gamma et al., 1994b), p. 135]

En la Figura 1.20 puede verse el resultado de aplicar este patrón junto con el de *Factoría abstracta* al juego del laberinto.

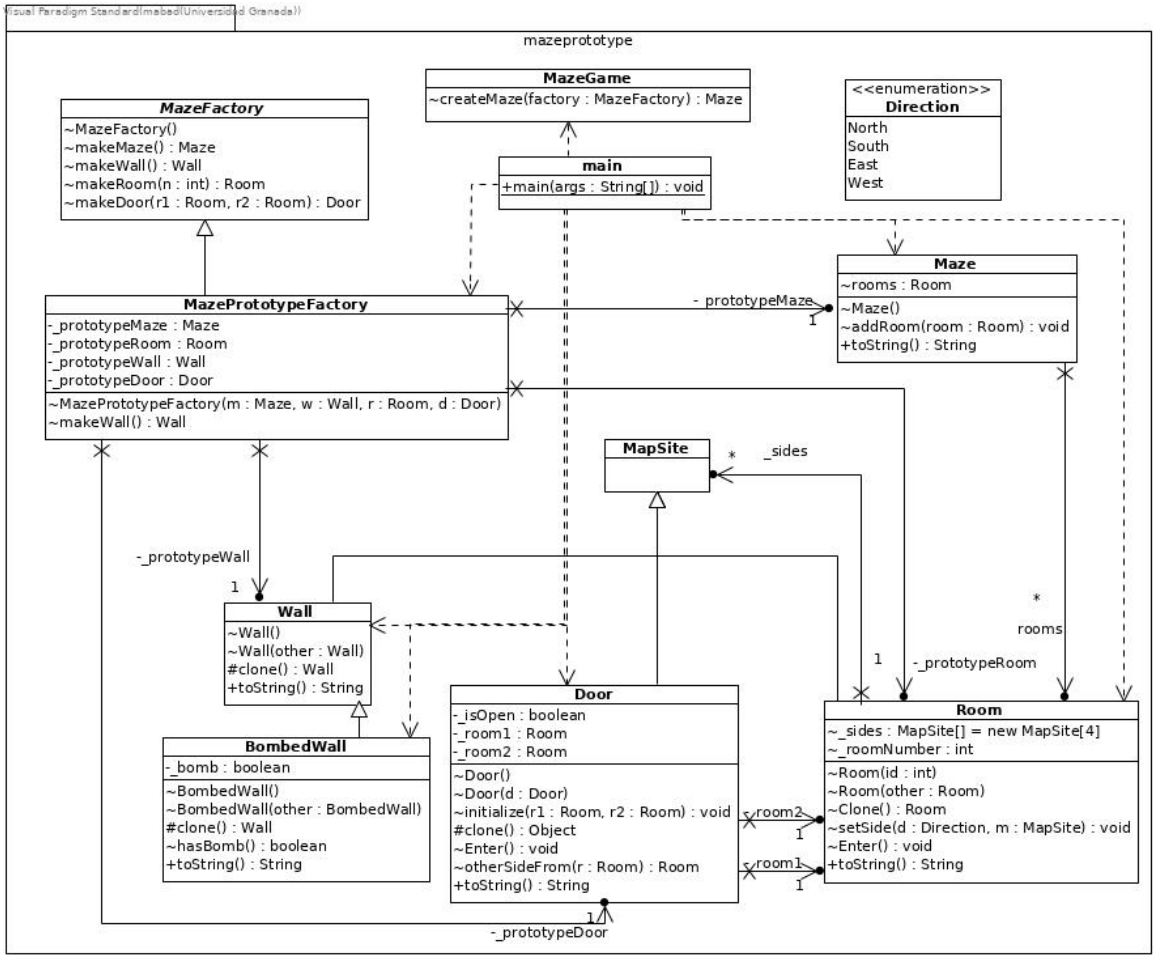

Figura 1.20: Diagrama de clases correspondiente a la aplicación del patrón *Prototype* junto con el de *Abstract Factory* en el ejemplo del juego del laberinto de GoF (Gamma et al., 1994a).

La implementación en c++ del método *createMaze* para el patrón *Prototype* junto con el de *Abstract Factory* sería la misma que cuando se usa la factoría abstracta con el patrón *Método Factoría*. Un ejemplo en c++ de implementación de la creación de dos factorías distintas de prototipos aparecen a continuación.

```
MazePrototypeFactory simpleMazeFactory(
new Maze, new Wall, new Room, new Door
);
```

```
MazePrototypeFactory bombedMazeFactory(
new Maze, new BombedWall,
new RoomWithABomb, new Door
);
```

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 1.21 muestra el diagrama de clases con la estructura del patrón y la Figura 1.22 un diagrama de secuencias con la forma en la que cooperan el *Director* y el *Builder* con el cliente.

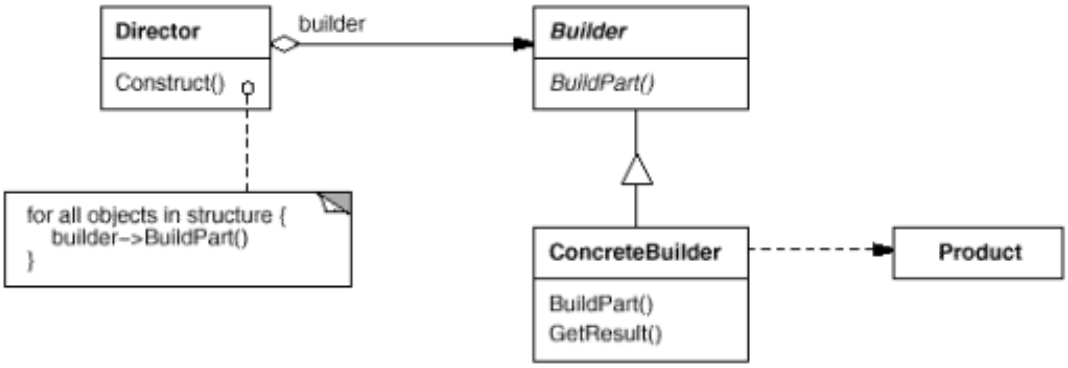

Figura 1.21: Estructura del patrón Builder [Fuente: (Gamma et al., 1994b), p. 112]

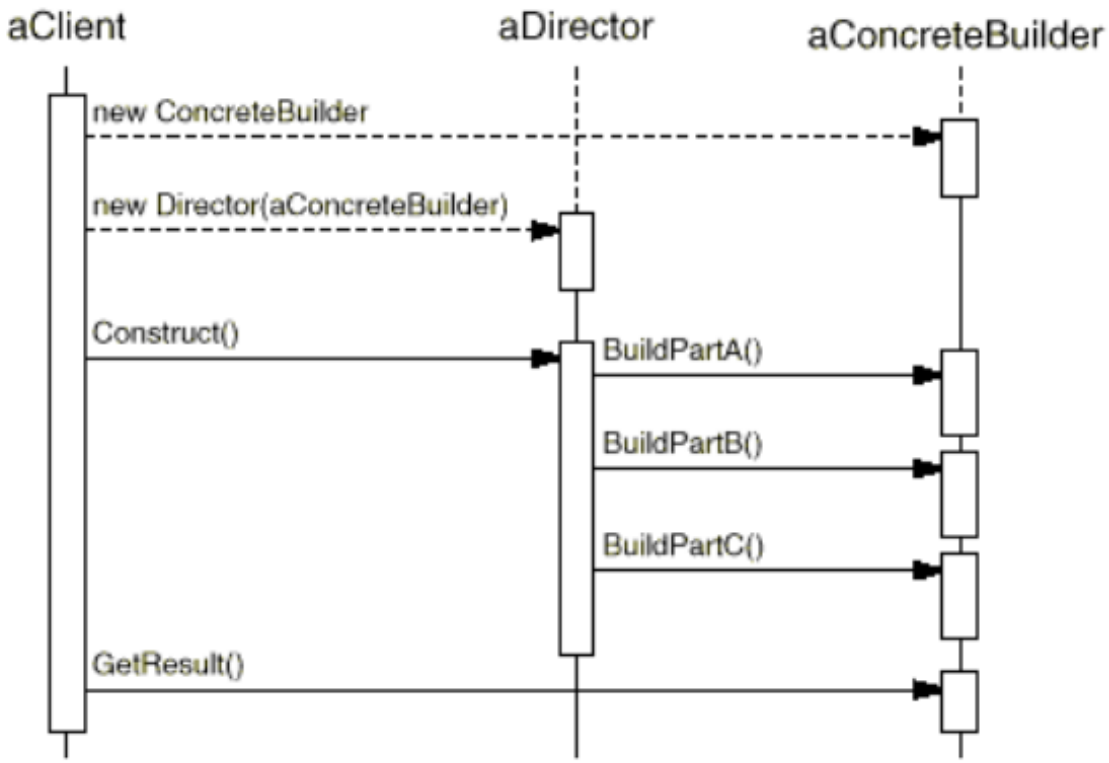

Figura 1.22: Estructura del patrón Builder [Fuente: (Gamma et al., 1994b), p. 113]

En la Figura 1.23 puede verse el resultado de aplicar este patrón al juego del laberinto.

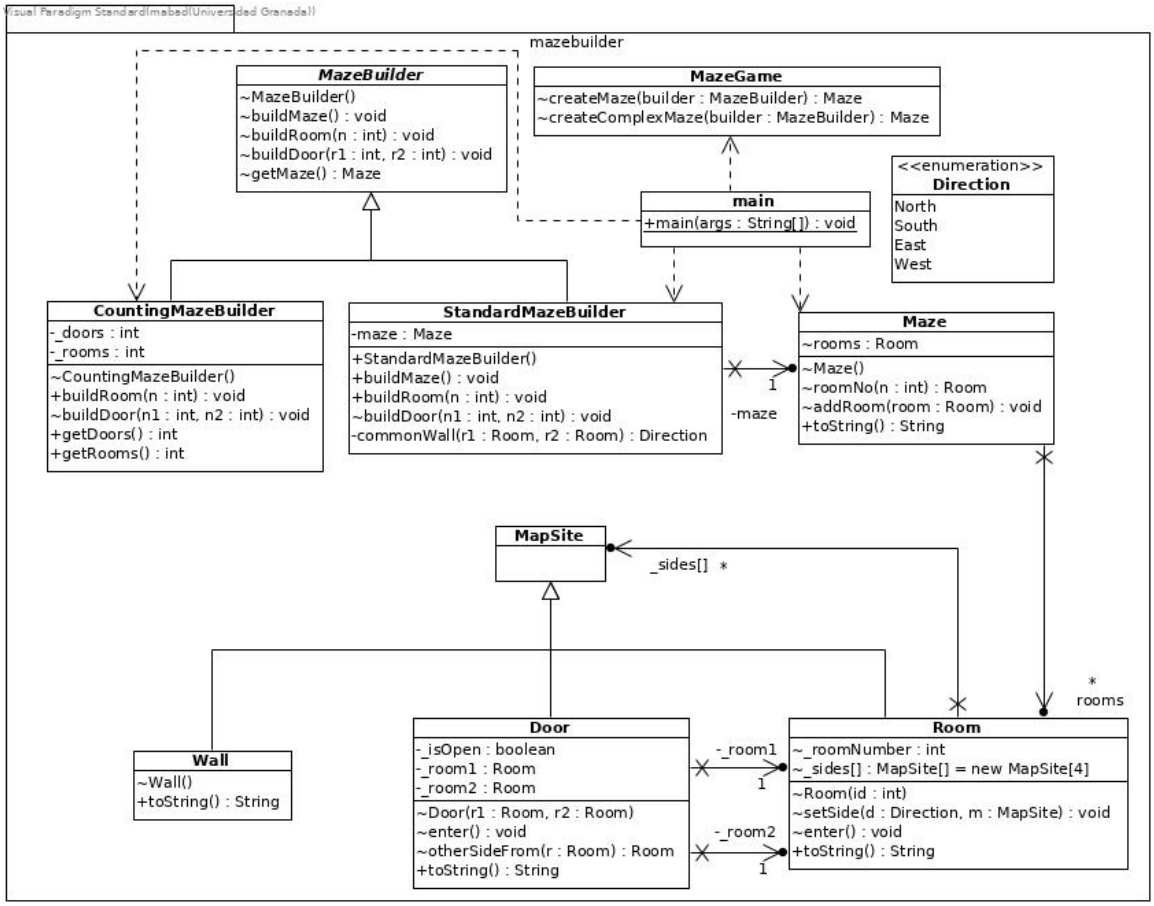

Figura 1.23: Diagrama de clases correspondiente a la aplicación del patrón "Builder" en el ejemplo del juego del laberinto de GoF (Gamma et al., 1994a).

Un ejemplo en c++ (asumiendo ligadura dinámica –métodos virtuales–) de implementación del método *createMaze* con el patrón *Builder* puede verse a continuación.

```
Maze* MazeGame::createMaze (MazeBuilder& builder) {
builder.buildMaze();
builder.buildRoom(1);
builder.buildRoom(2);
builder.buildDoor(1, 2);
return builder.getMaze();
}
```

Si necesitamos construir otro tipo de laberinto, por ejemplo un laberinto complejo, se puede hacer como aparece a continuación:

```
Maze* MazeGame::createComplexMaze (MazeBuilder& builder) {
builder.buildRoom(1);
// ...
builder.buildRoom(1001);
return builder.getMaze();
```

```
}
```

CRITERIO DE CALIDAD 1.3.1: Bajo acoplamiento

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

CRITERIO DE CALIDAD 1.3.2: 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 1.24 puede verse un esquema de un diagrama de clases antes y después de la aplicación del patrón y en la Figura 1.25 la estructura de un diagrama con el patrón ya aplicado.

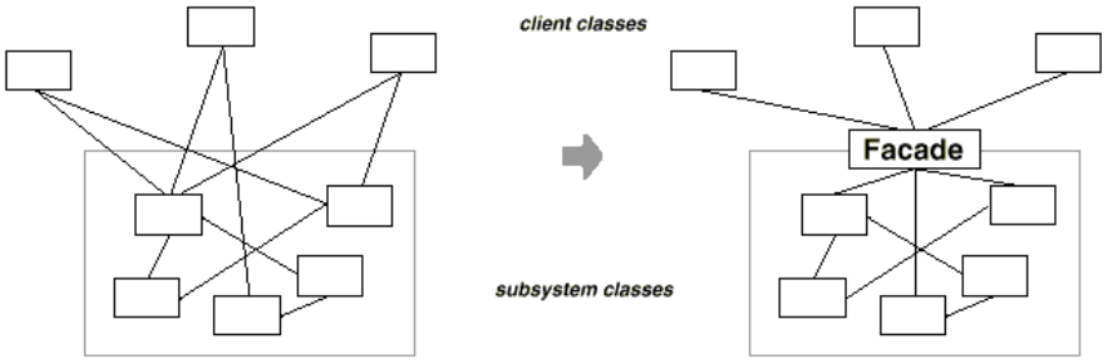

Figura 1.24: Un esquema de diagrama de clases antes y después de la aplicación del patrón Fachada [Fuente: (Gamma et al., 1994b), p. 208].

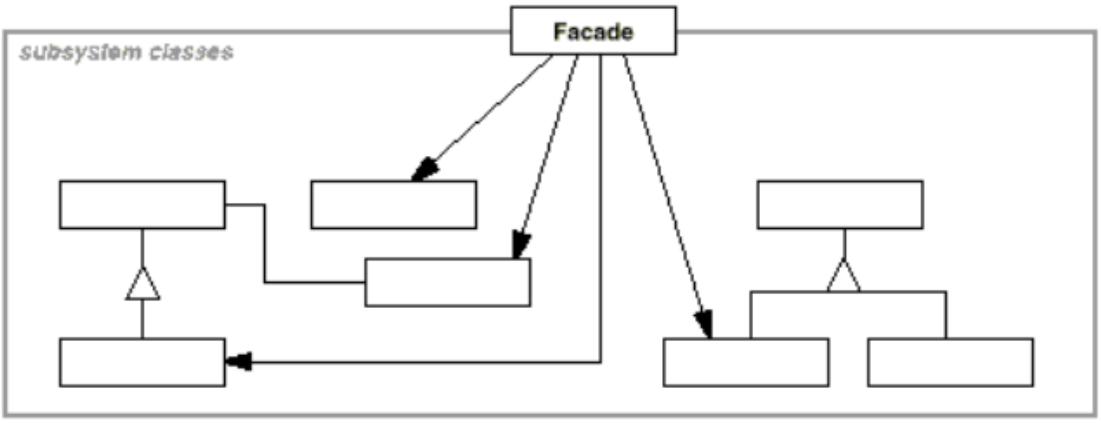

Figura 1.25: Estructura del patrón Fachada [Fuente: (Gamma et al., 1994b), p. 210].

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 1.26 muestra la estructura del patrón, mientras que la Figura 1.27 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 1.28 con el diagrama de clases que representa la estructura del patrón y en la Figura 1.29 con una jerarquía posible de elementos gráficos según el patrón.

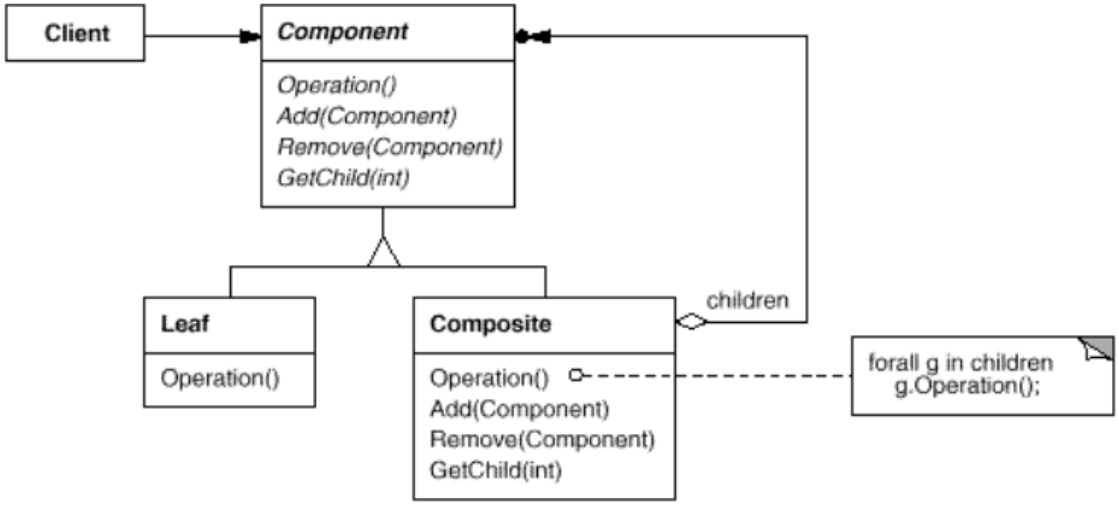

Figura 1.26: Estructura del patrón Composición [Fuente: (Gamma et al., 1994b), p. 185].

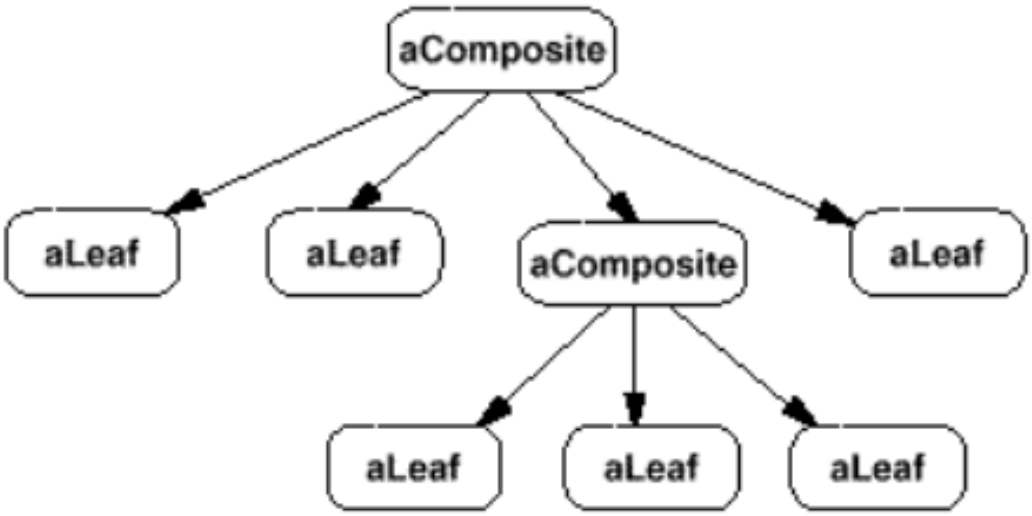

Figura 1.27: Un ejemplo de jerarquía de objetos donde se ve la relación entre objetos compuestos y simples para el patrón "Composición" [Fuente: (Gamma et al., 1994b), p. 185].

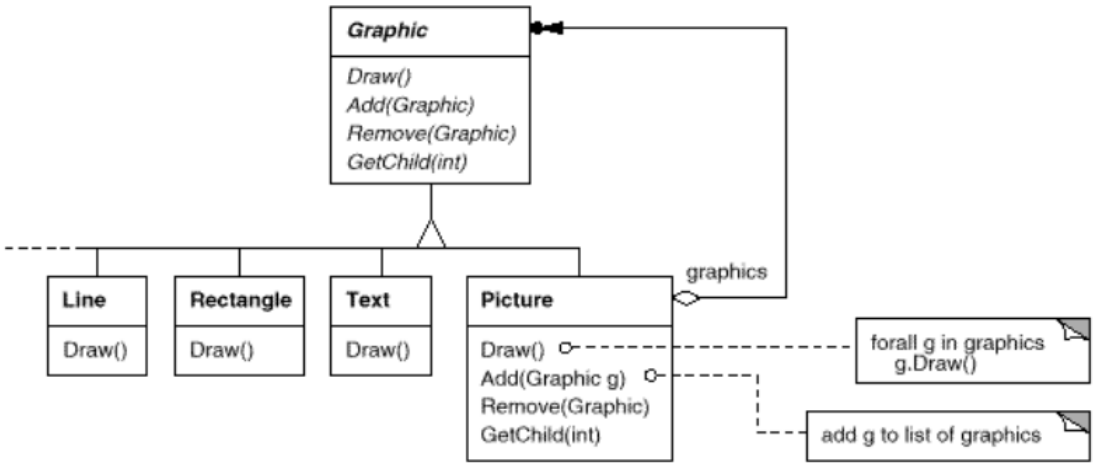

Figura 1.28: Estructura del patrón Composición en el ejemplo de componentes gráficos. [Fuente: (Gamma et al., 1994b, pg. 183)].

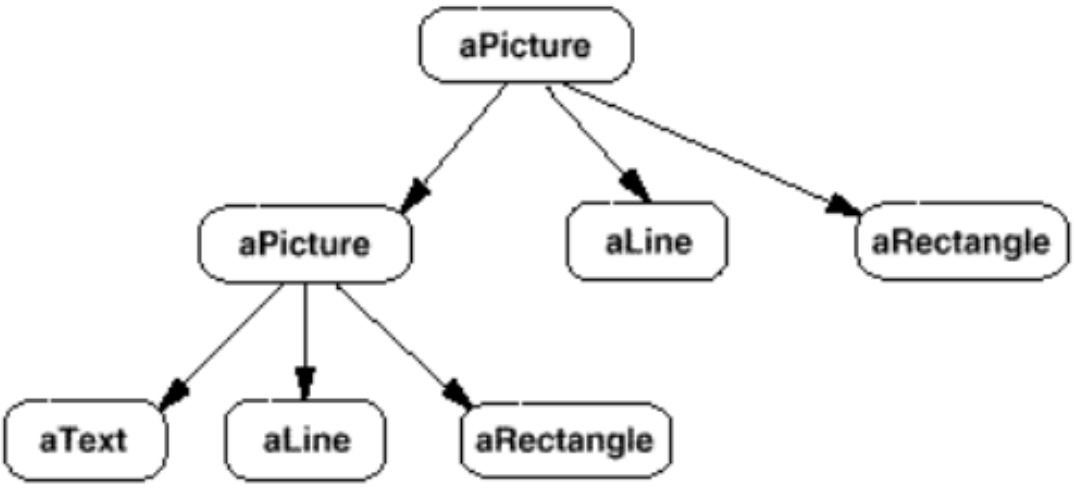

Figura 1.29: Una jerarquía de objetos en el ejemplo de componentes gráficos [Fuente: (Gamma et al., 1994b, pg. 184)].

Flutter 1.3.1. Patrón compuesto en los widgets de Flutter

Otro ejemplo de uso de este patrón es la relación entre la mayoría de los *widgets*, los distintos elementos (clases) que definen las distintas partes estáticas de la interfaz de usuario en Flutter. La Figura 1.30 muestra un ejemplo de diagrama de clases con la relación entre los distintos widgets de una aplicación Flutter.

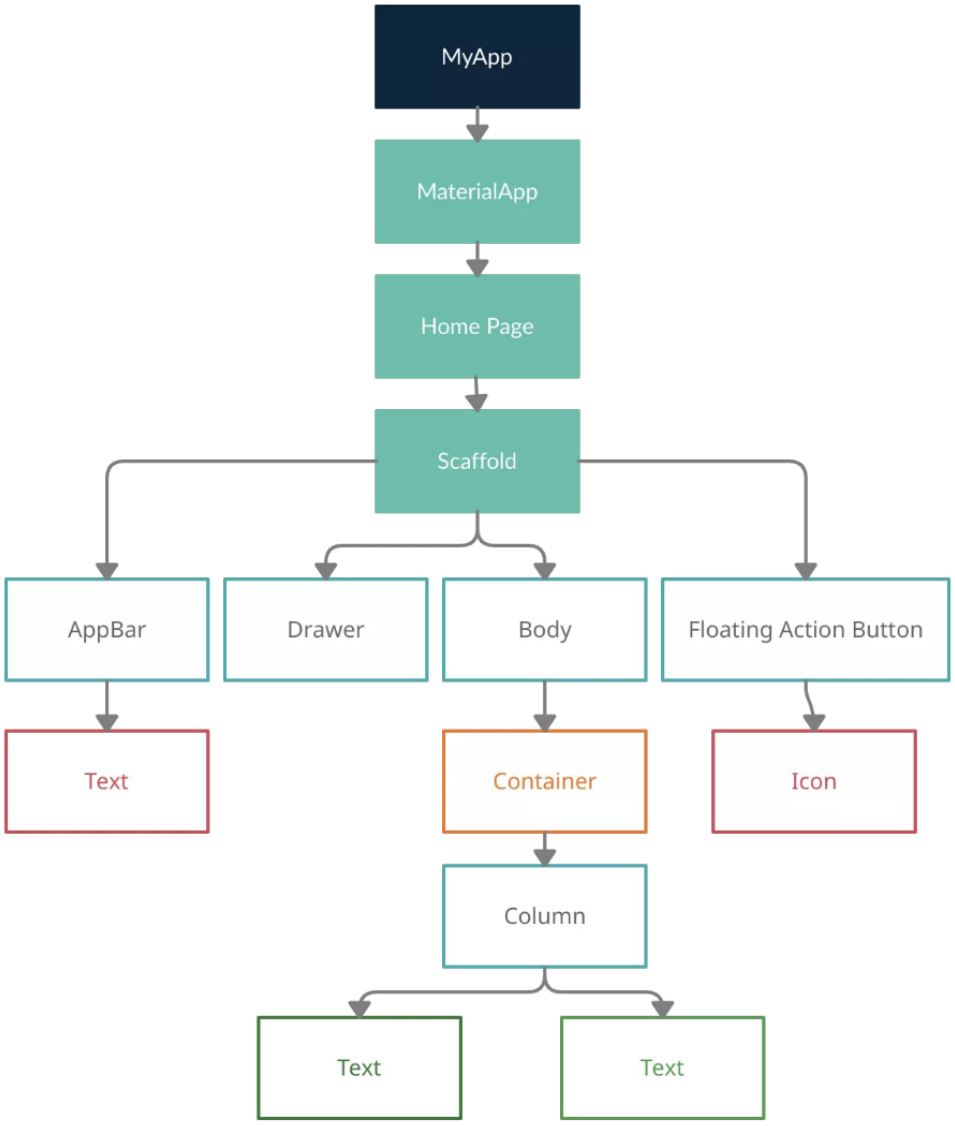

Figura 1.30: Un ejemplo de árbol de objetos widgets en una app Flutter [Fuente: https://www.fluttercampus.com/tutorial/5/flutter-widgets/].

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 solo 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 1.31 muestra la estructura del patrón.

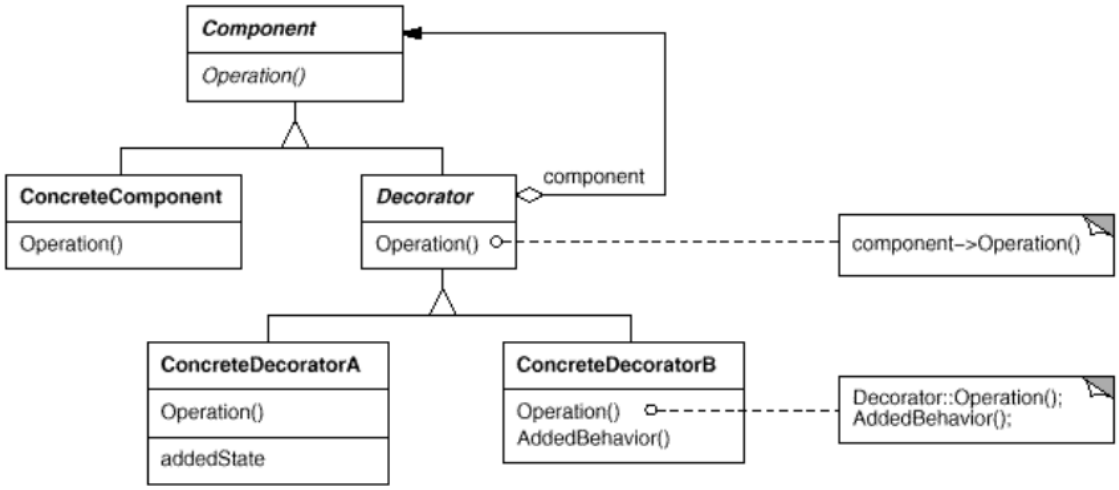

Figura 1.31: Estructura del patrón decorator [Fuente: (Gamma et al., 1994b), p. 199].

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 1.32 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 1.33.

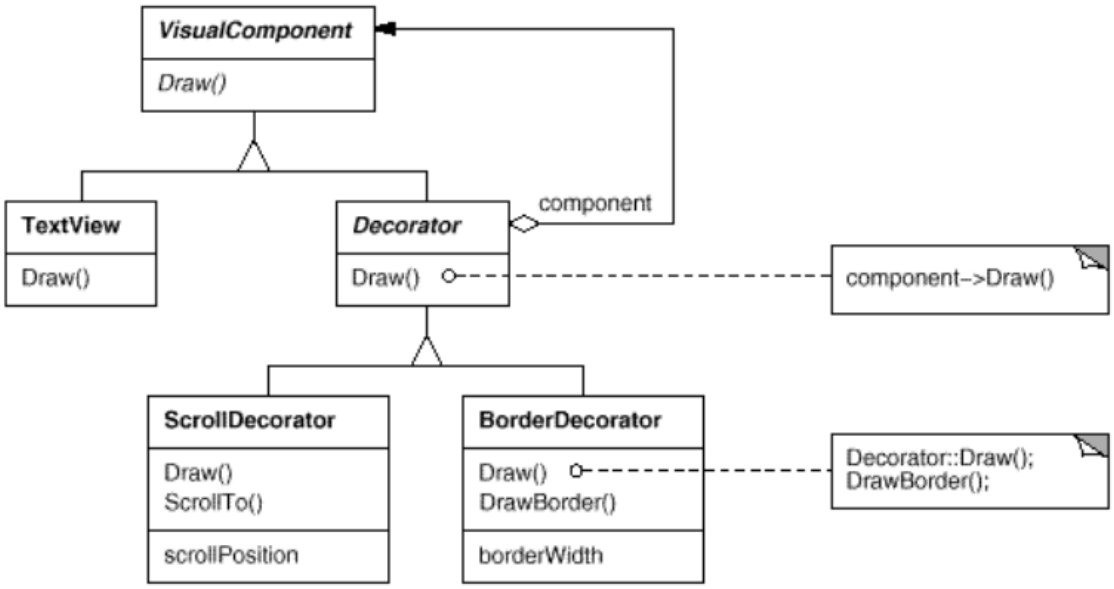

Figura 1.32: Ejemplo de estructura del patrón decorator aplicada a la presentación gráfica de un texto [Fuente: (Gamma et al., 1994b), p. 198].

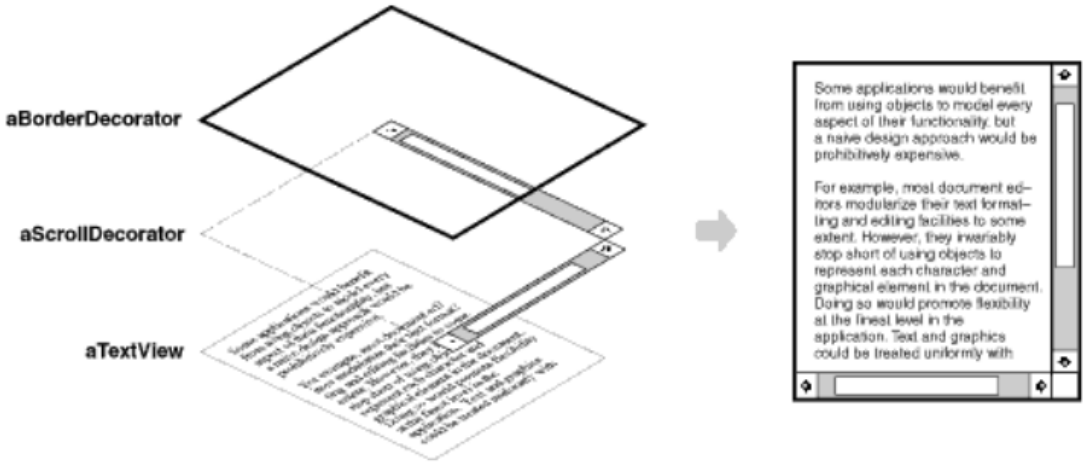

Figura 1.33: Aspecto de un cuadro de texto y sus decoradores [Fuente: (Gamma et al., 1994b, 197)].

En los fragmentos de código siguiente se puede ver una vista en Ruby on Rails que no usa el patrón decorador (Cuadro de código 1.3), que usa un decorador simple (listado 1.4) y que usa otro más complejo (Cuadro de código 1.5) para mostrar información sobre el usuario actual. Se enfatiza en amarillo el código clave. Las clases decoradoras (en Ruby) aparecen primero (Cuadros de código 1.1 y 1.2).

```
class UserBasicDecorator < SimpleDelegator
```

```
# new methods can call methods on the parent implicitly
def info
"#{ name } #{ lastname }"
end
end
```

Cuadro de código 1.1: Clase UserBasicDecorator

```
class UserDecorator < UserBasicDecorator
# new methods can call methods on the parent implicitly
def info
"#{ name } #{ lastname } (created at: #{ created_at })"
end
end
```

Cuadro de código 1.2: Clase UserDecorator

```
{view/sessions/welcome.html.erb}
<h1>Clados on Rails</h1>
<% if logged_in? %>

<h3><%= "#{ current_user.name } #{ current_user.lastname }" %>
(joined: <%= current_user.created_at.strftime("%A, %d %b %Y %l:%M %p")
    %>)</h3>
<%= render partial: "home_page/index" %>
<%else %>
<%= button_to "Login", '/login', method: :get%>
<%end %>
```

Cuadro de código 1.3: Sin patrón decorador

```
{view/sessions/welcome.html.erb}
<h1>Clados on Rails</h1>
<% if logged_in? %>
<% user = UserSimpleDecorator.new(current_user) %>
<h3><% user.info %></h3>
<%= render partial: "home_page/index" %>
<%else %>
<%= button_to "Login", '/login', method: :get%>
<%end %>
```

Cuadro de código 1.4: Con decorador simple

```
{view/sessions/welcome.html.erb}
<h1>Clados on Rails</h1>
<% if logged_in? %>
 user = UserDecorator.new(current_user) %>
<h3><% user.info %></h3>}
<%= render partial: "home_page/index" %>
<%else %>
<%= button_to "Login", '/login', method: :get%>
<%end %>
```

Cuadro de código 1.5: Con decorador

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 solo 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 1.34) y de objeto (ver Figura 1.35).

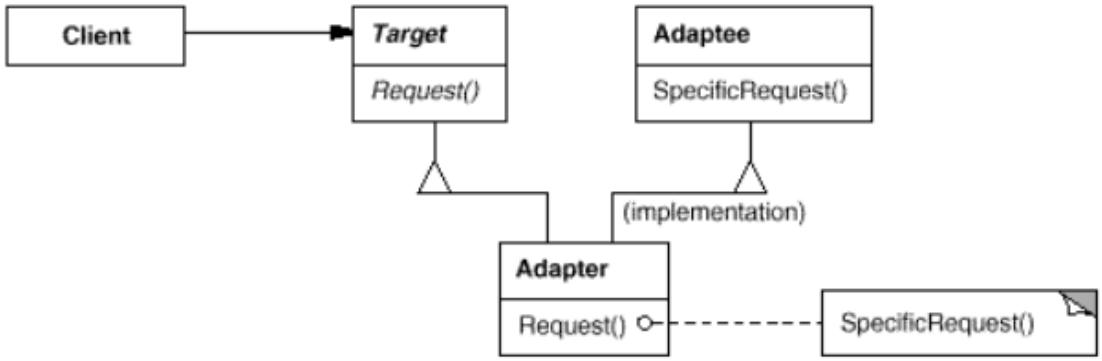

Figura 1.34: Estructura del patrón adaptador, en un ámbito de clase [Fuente: (Gamma et al., 1994b, pg. 159)].

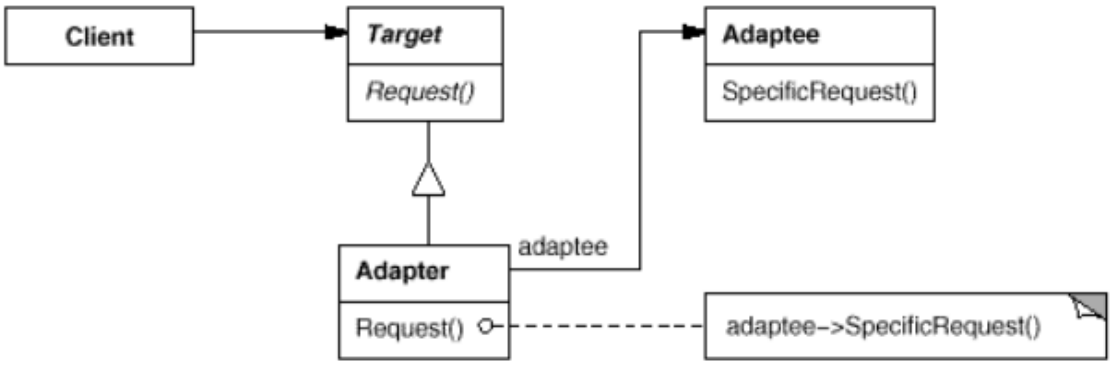

Figura 1.35: Estructura del patrón adaptador en un ámbito de objeto [Fuente: (Gamma et al., 1994b, pg. 159)].

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 1.36 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.

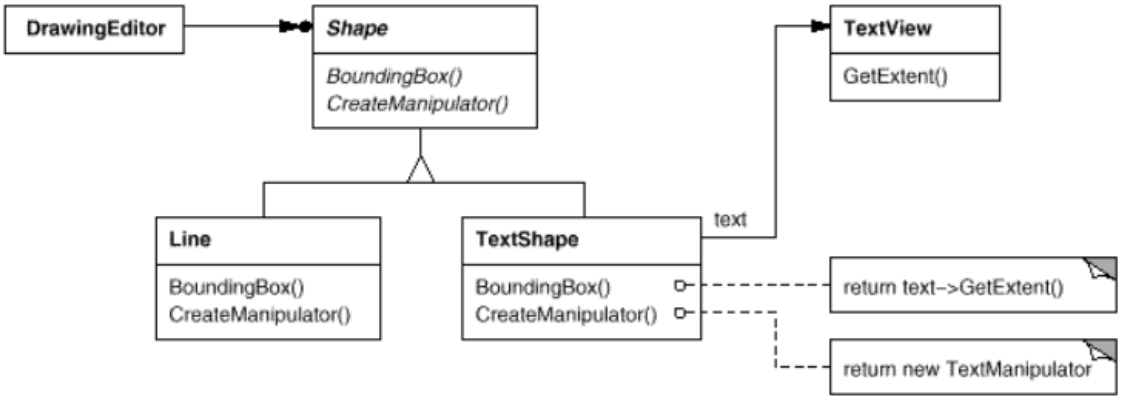

Figura 1.36: Estructura del patrón "Adapter" aplicado a un editor de dibujo [Fuente: (Gamma et al., 1994b, pg. 158)].

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 1.37 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 1.37).

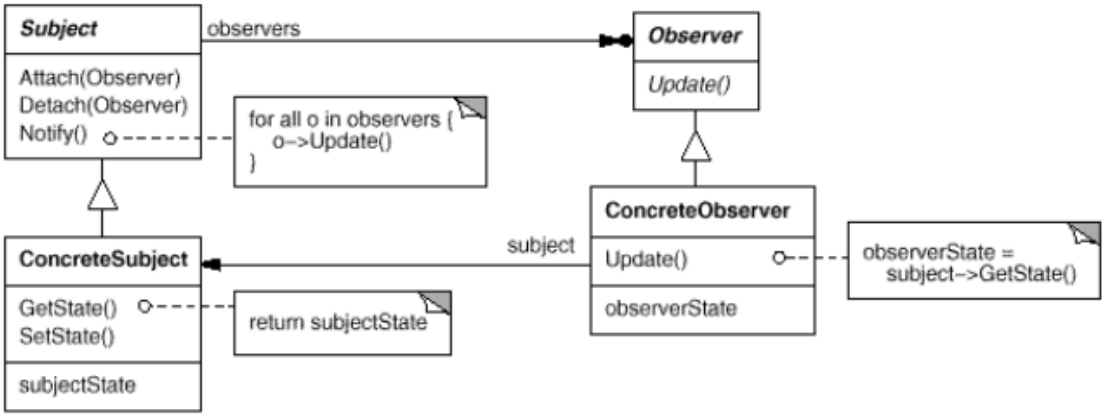

Figura 1.37: Estructura del patrón Observador [Fuente: (Gamma et al., 1994b), p. 328]

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.

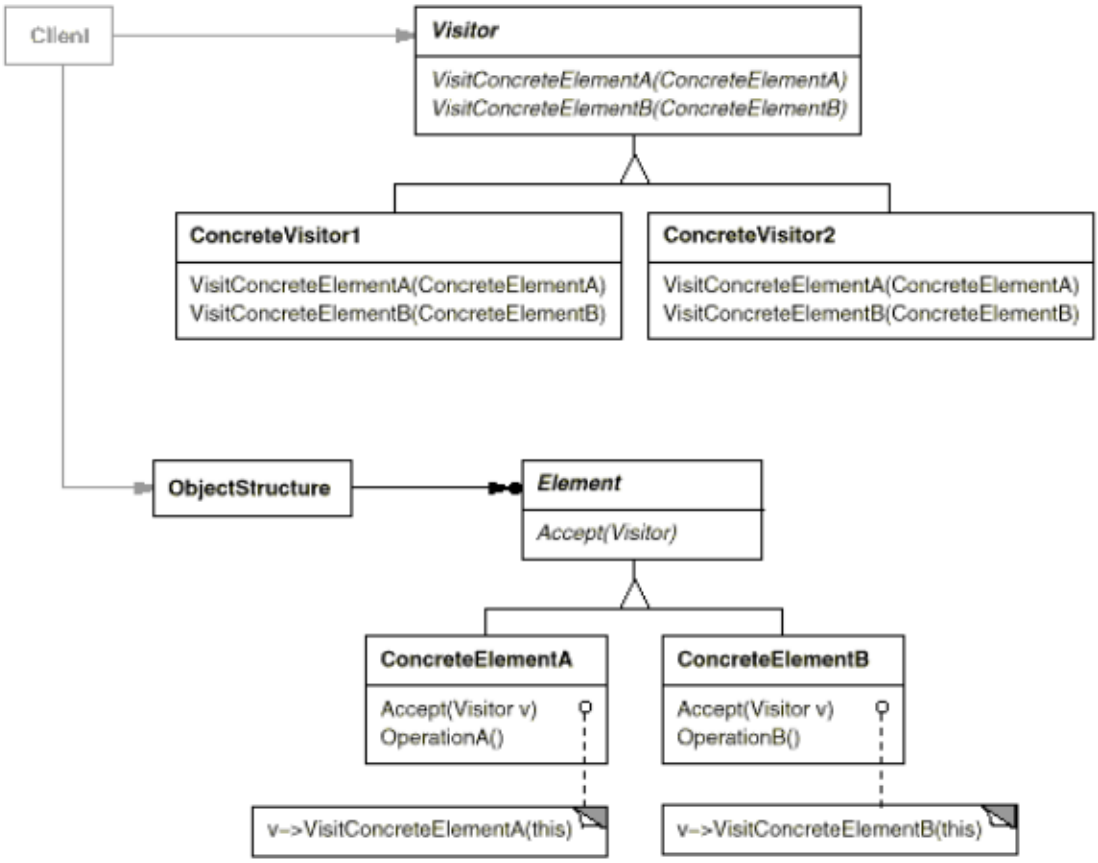

Figura 1.38: Estructura del patrón "Visitante" [Fuente: (Gamma et al., 1994b, pg. 369)]

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 1.39 muestra la estructura del patrón. La Figura 1.40 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.

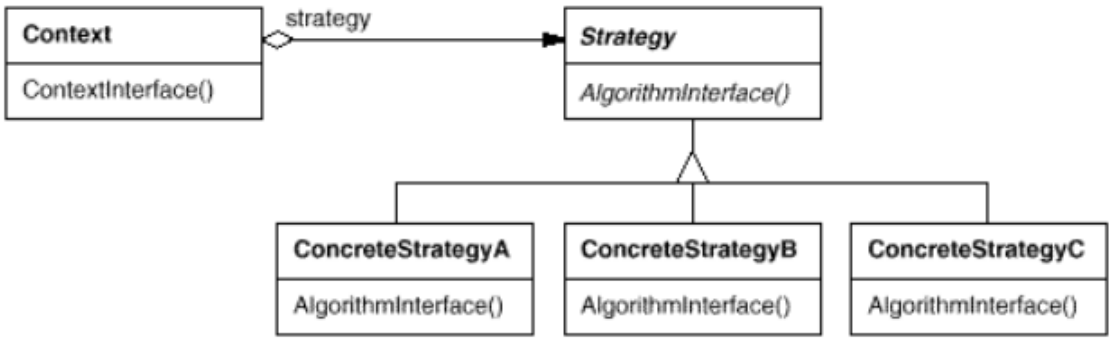

Figura 1.39: Estructura del patrón "Estrategia" [Fuente: (Gamma et al., 1994b, pg. 351)]

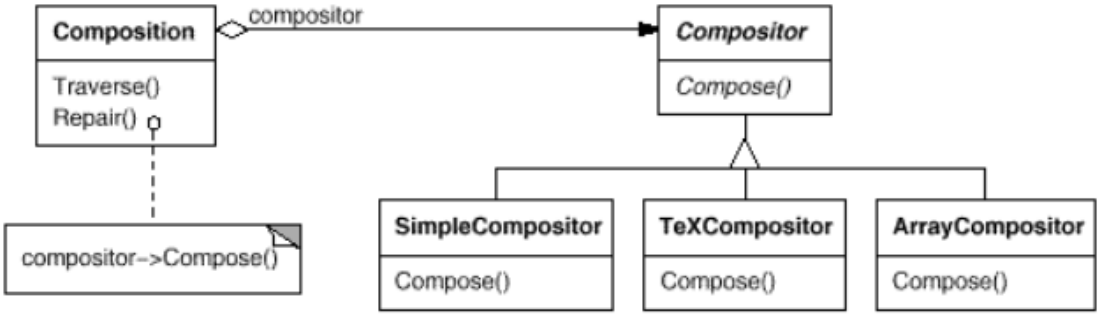

Figura 1.40: Estructura del patrón "Estrategia" en un ejemplo de visualización de textos [Fuente: (Gamma et al., 1994b, pg. 349)]

Patrón *Método Plantilla*

Un método plantilla es un método de una clase abstracta[3] 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 1.41 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)]:

- Métodos concretos de las subclases *ConcreteClass*
- Métodos implementados en la clase *AbstractClass*

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

- 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

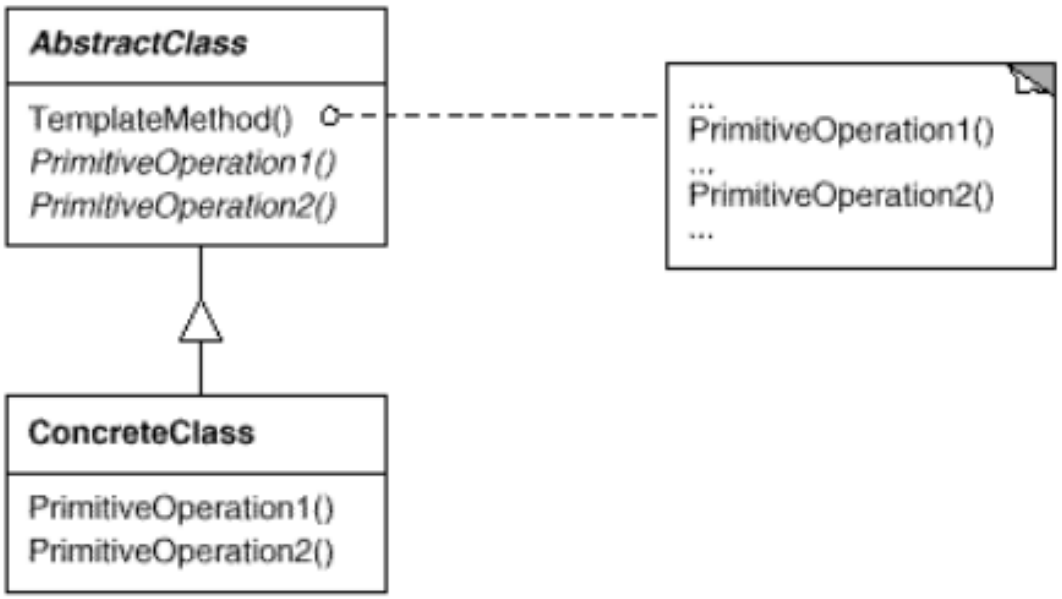

Figura 1.41: Estructura del patrón "Método Plantilla"[Fuente: (Gamma et al., 1994b, pg. 362)]

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* 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*

Algunos patrones surgen como combinación de otros. Este patrón surge por la combinación de otros dos, que, a su vez, se pueden considerar tanto patrones de diseño como estilos arquitectónicos. El patrón *Filtros de intercepción* (ver Figura 1.42) 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 arquitectónico *Tubería y Filtro*, 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[4]. 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 1.42, 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).

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

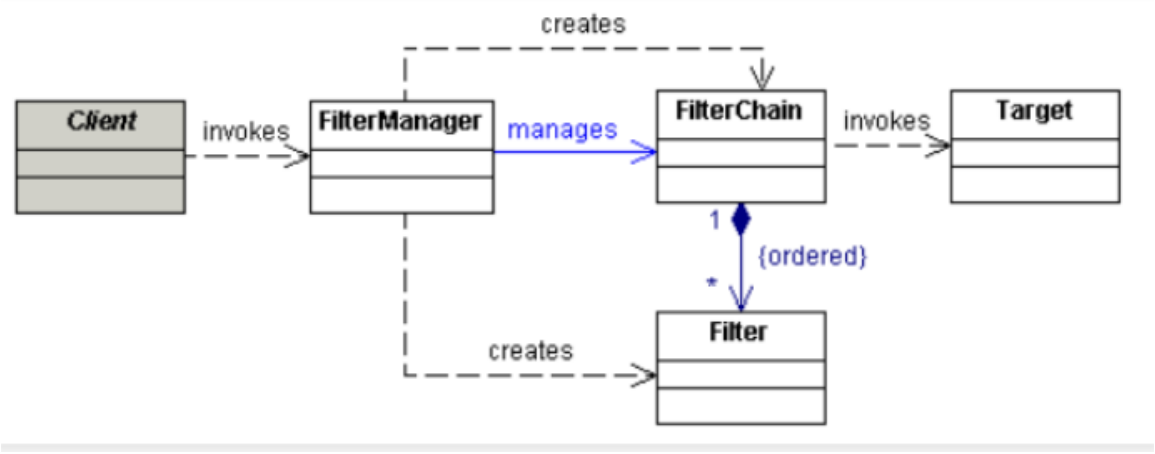

Figura 1.42: Diagrama de clases correspondiente al patrón interceptor de filtros. [Fuente: Patrón Filtros de intercepción.]

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*).

La Figura 1.43 muestra un ejemplo de diagrama de secuencias donde se usa este patrón.

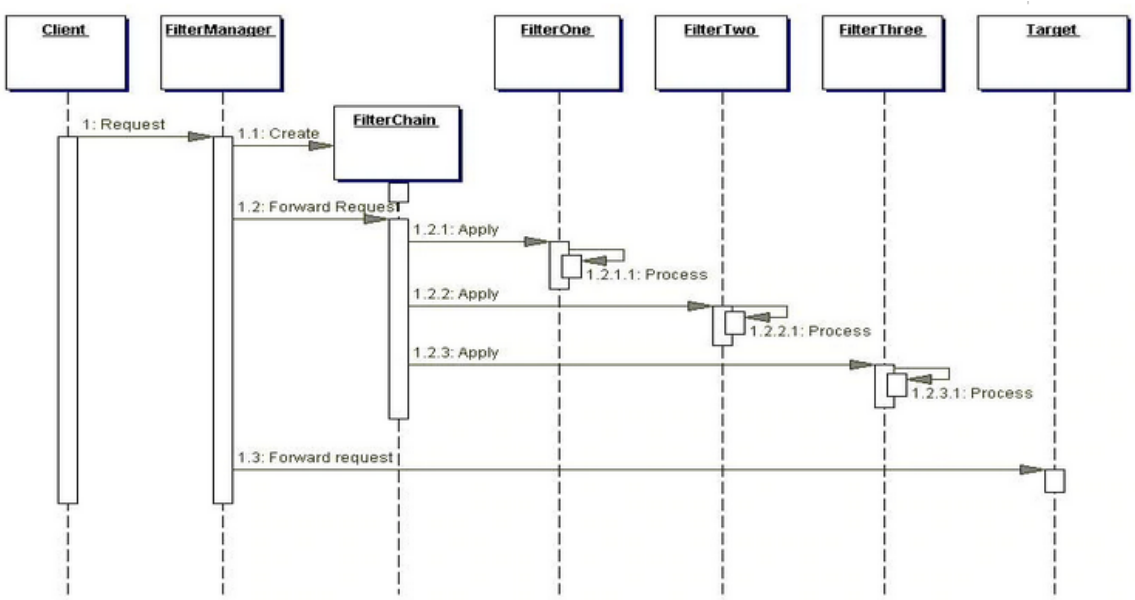

Figura 1.43: Ejemplo de diagrama de secuencias del estilo arquitectónico *Filtros de intercepción*. [Fuente: https://www.oracle.com/technetwork/java/interceptingfilter-142169.html.]

1.4. Patrones o estilos arquitectónicos

Un *estilo arquitectónico* (o patrón arquitectónico) define tipos de componentes y conectores y un conjunto de restricciones sobre la forma en la que pueden combinarse estos elementos, en el nivel más alto de abstracción de un sistema software. Una definición más formal es:

> Un *patrón arquitectónico* expresa un esquema de organización estructural fundamental en un sistema software. Proporciona un conjunto de subsistemas predefinidos, especificando sus responsabilidades, e incluye reglas y guías para organizar las relaciones entre ellos (Buschmann et al., 1996).

Algunos de ellos se han intentado clasificar, como puede apreciarse en la Figura 1.44. Esta clasificación puede ayudar a entender los estilos y relacionar unos con otros.

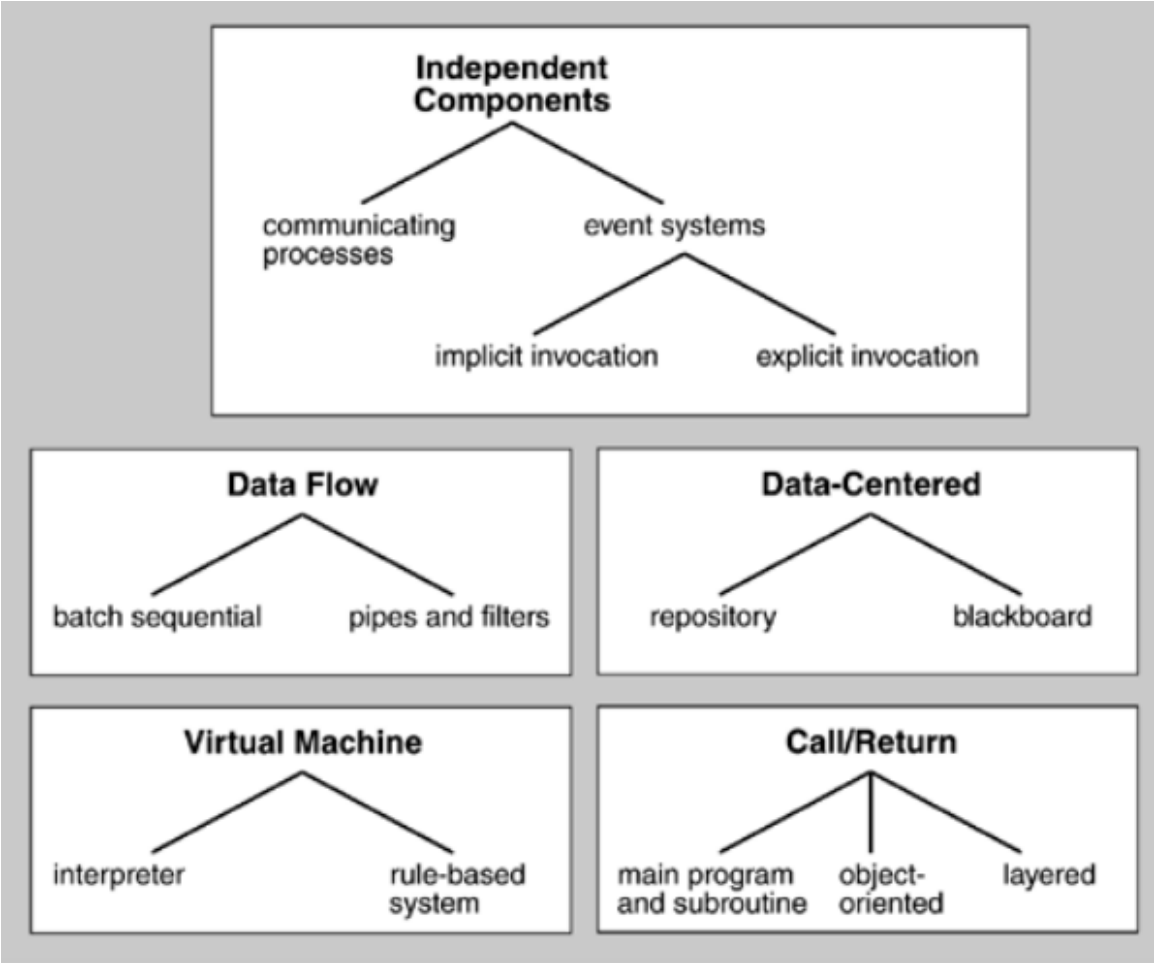

Figura 1.44: Clasificaciones de distintos estilos arquitectónicos *Control de procesos* para el sistema de control de la velocidad de crucero. [Fuente: (Bass et al., 2003)]

1.4.1. Estilos de Flujo de Datos: el estilo *Tubería y filtro*

Los estilos de Flujo de Datos «enfatizan la reutilización y la modificabilidad. Es apropiada para sistemas que implementan transformaciones de datos en pasos sucesivos.» (Reynoso and Kicillof, 2004).

En el estilo *Tubería y filtro* los componentes son los *filtros* (filter) y los conectores son las *tuberías* (pipes). Las tuberías transportan la corriente de datos (data stream). El filtro recibe una corriente de datos de entrada y los va procesando de forma incremental, realizando con ellos una transformación y empezando a poner los datos ya transformados en la salida aún cuando todavía no haya consumido todo los datos de entrada que le llegan. Un ejemplo aparecen en la Figura 1.45.

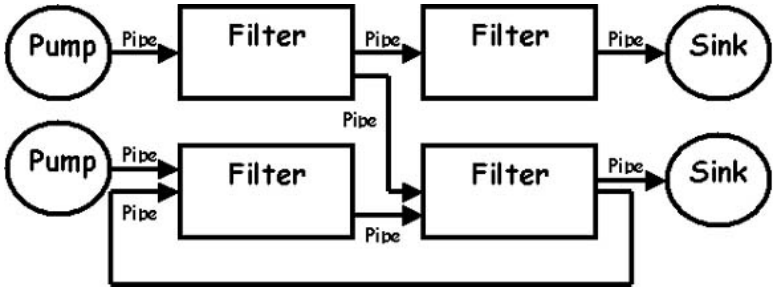

Figura 1.45: Estructura del estilo arquitectónico *Tubería y filtro*. [Fuente: (Garfixia, Accessed March 4, 2020)]

Invariantes del estilo

- Los filtros son entidades independientes, no comparten su estado con otros filtros
- Los filtros no conocen a sus filtros vecinos, tan sólo pueden imponer restricciones a los datos que le entran y garantizar que cumple con restricciones a los datos que saca como salida. Ni siquiera debería afectar a la salida final del sistema la forma en la que cada filtro realizan el procesamiento incremental de los datos

Algunos subestilos o tipos más epecíficos del estilo *Tubería y filtro* son:

- Estilo *Tubería lineal* (pipelines).- Todos los filtros están en una única secuencia (ver Figura 1.46)
- Estilo *Tubería limitada*.- Restringe la cantidad máxima de datos que pueden estar en un momento dado en una tubería
- Estilo *Secuencial por lotes* (sequential batch).- Es un caso extremo en el que cada filtro procesa todos los datos de entrada en conjunto, como una única entidad. En este caso las tuberías realmente no tienen sentido porque no se requiere una corriente de datos. Realmente éste se considera un estilo independiente

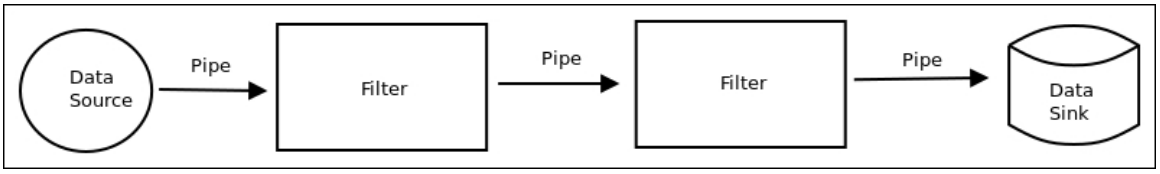

Figura 1.46: Estructura del estilo arquitectónico *Tubería y filtro*. [Fuente: (Balachandran Pillai, 2017)]

El mejor ejemplo de este estilo lo proporciona Unix en su intérprete de comandos (shell). Para ello proporciona una notación que representa las tuberías, es decir, que conecta los componentes (el carácter "|") junto con mecanismos de ejecución para implementar las tuberías.

Por ejemplo, para obtener el total de ficheros en el directorio actual que contienen la palabra "Maze" en su nombre:

```
ls | grep Maze | wc -l
```

Otros ejemplos se dan en el procesamiento de señales o en la programación paralela.

Ventajas

- Diseño fácil: Los sistemas con este estilo son muy fáciles de ser entendidos
- Reusabilidad: Se pueden construir filtros por la unión de dos filtros, siempre que haya acuerdo en la información que pasa de uno a otro
- Mantenibilidad, incluyendo mejoras: Se pueden añadir nuevos filtros o reemplazar los que ya existen
- Permiten análisis especializado, como rendimiento y análisis de interbloqueos
- Concurrencia: cada filtro puede realizar una tarea y actuar en paralelo con otros

Inconvenientes

- No permiten proceso interactivo, sólo programación por lotes
- Bajo rendimiento: Si no se programan bien se pueden producir cuellos de botella en los filtros más lentos
- Complejidad en la programación: para que el rendimiento sea alto, requieren de una programación compleja

1.4.2. El estilo *Abstracción de Datos y Organización OO*

Los estilos de Llamada y Retorno «enfatizan la modificabilidad y la escalabilidad. Son los estilos más generalizados en sistemas en gran escala.» (Reynoso and Kicillof, 2004). Algunos ejemplos son las arquitecturas basadas en componentes, las orientadas a objetos (que se ven a continuación), el estilo basado en eventos que se verá en la siguiente Sección1.4.4, la arquitectura Modelo-Vista-Controlador (Sección 1.4.3) y el estilo en capas (Sección 1.4.5).

El estilo *Abstracción de datos y organización OO* transparenta las características modulares cuando se usa un lenguaje de programación modular u OO. Así, en este estilo, la representación de los datos y las operaciones que éstos pueden hacer se encapsulan: Los componentes son instancias de tipos abstractos de datos (TADs) en lenguajes modulates u objetos en lenguajes OO. Los conectores son las vías de comunicación entre los componentes (asociaciones, en terminología OO). Una conexión concreta se hace operativa cuando se realiza entre ellos alguna llamada o invocación (o envío de mensajes, en terminología OO) a procedimientos o funciones (o métodos, en terminología OO). A los componentes

también se les llama gestores por ser los encargados de preservar la integridad de los datos que contienen.

En la Figura 1.47 se muestra un ejemplo. "op" es la invocación (procedure call) que se hace para conectar un objeto con otro.

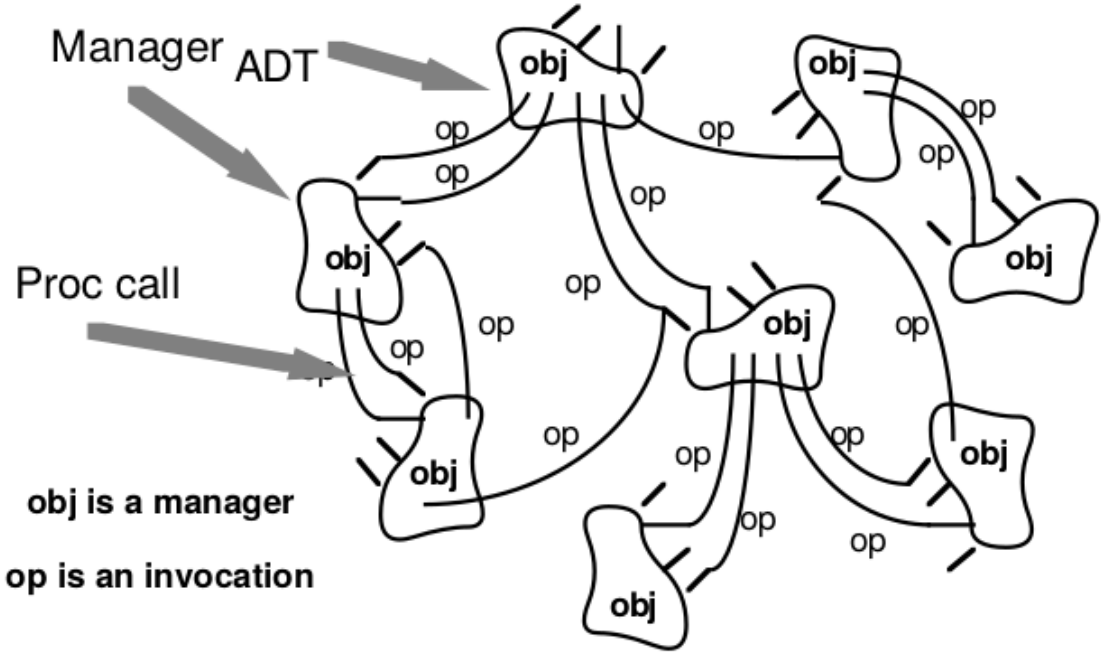

Figura 1.47: Ejemplo del estilo arquitectónico *Abstracción de Datos y Organización OO*. [Fuente: (Shaw and Garlan, 1996, pg. 23)]

Invariantes del estilo

- Encapsulamiento: Datos y operaciones sobre ellos están encapsulados
- El objeto es responsable de preservar la integridad de su estado
- Ocultación: La implementación de las operaciones están ocultas a otros objetos. Los datos también pueden ocultarse

Ventajas

- Gracias al ocultamiento de las operaciones es posible cambiar la implementación de éstas sin que afecte a sus clientes
- Gracias al encapsulamiento, los diseñadores pueden descomponer la resolución de un poblema como un conjunto de agentes con responsabilidades específicas que cooperan para la resolución del mismo

Inconvenientes

- Los componentes deben conocerse entre ellos para poder interaccionar (piénsese como alternativa en el estilo *Tubería y filtro*)

- Puede haber efectos colaterales entre objetos que están conectados entre sí directa o indirectamente

1.4.3. El estilo *Modelo-Vista-Controlador* (MVC)

Fue descrito por primera vez para su aplicación en Smalltalk (Reenskaug, 1979). La idea fundamental es separar la funcionalidad del sistema (*lógica del negocio* o *dominio de la aplicación*) de la interfaz de usuario. En la actualidad está ampliamente extendido. Ejemplos de uso son Smalltalk y Ruby on Rails.

Se compone de tres elementos:

- Modelo: Conjunto de clases que representan la lógica de negocio de la aplicación (clases deducidas del análisis del problema). Encapsula la funcionalidad y el estado de la aplicación.

- Vista: Representación de los datos contenidos en el modelo. Para un mismo modelo pueden existir distintas vistas.

- Controlador: Es el encargado de interpretar las ordenes del usuario. Mapea la actividad del usuario con actualizaciones en el modelo. Puesto que el usuario ve la vista y los datos originales están en el modelo, el controlador actúa como intermediario y mantiene ambos coherentes entre sí.

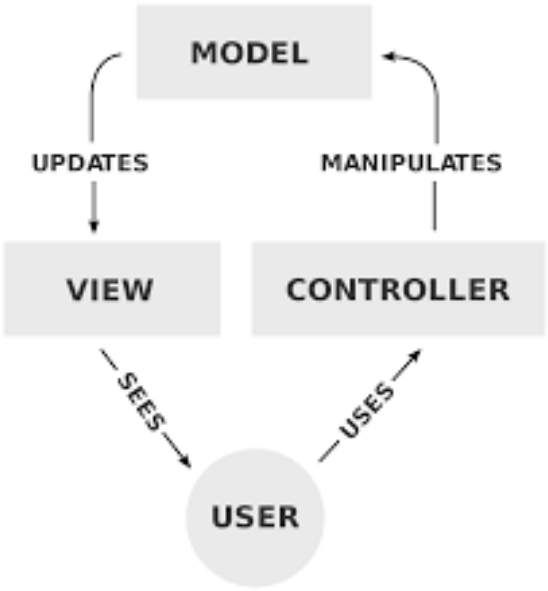

Figura 1.48: Estructura del estilo arquitectónico MVC de Controlador ligero (ver más abajo). [Fuente: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller]

Ventajas

- El desarrollo es más sencillo y limpio.
- Facilita la evolución por separado de vista y modelo.
  - Cualquier modificación que afecte al dominio de la aplicación, implica una modificación sólo en el modelo. Las modificaciones a las vistas no afectan al modelo, simplemente se modifica la representación de la información, no su tratamiento.
- Incrementa la reutilización y la flexibilidad.

Incovenientes

- Curva de aprendizaje muy pendiente, pues muchas veces este estilo combina tecnologías múltiples que los desarrolladores
- Puede añadir complejidad superflua si la aplicación no lo requiere

Variantes Hay principalmente dos variantes en este estilo:

- Controlador ligero o Modelo activo: La actualización de las vistas es hecha directamente desde el modelo (Figura 1.48). Para mantener la independencia de ambas, en esta opción es necesario el uso a su vez del estilo *Basado en Eventos*, estilo que veremos a continuación, y que es equivalente al patrón de diseño *Observador* que ya vimos.

- Controlador pesado o Modelo pasivo: el controlador conoce las vistas e interactúa con ellas para notificarles que se han hecho cambios en el modelo de forma que las vistas pedirán los datos al modelo (lo cuál puede hacerse directa o indirectamente).

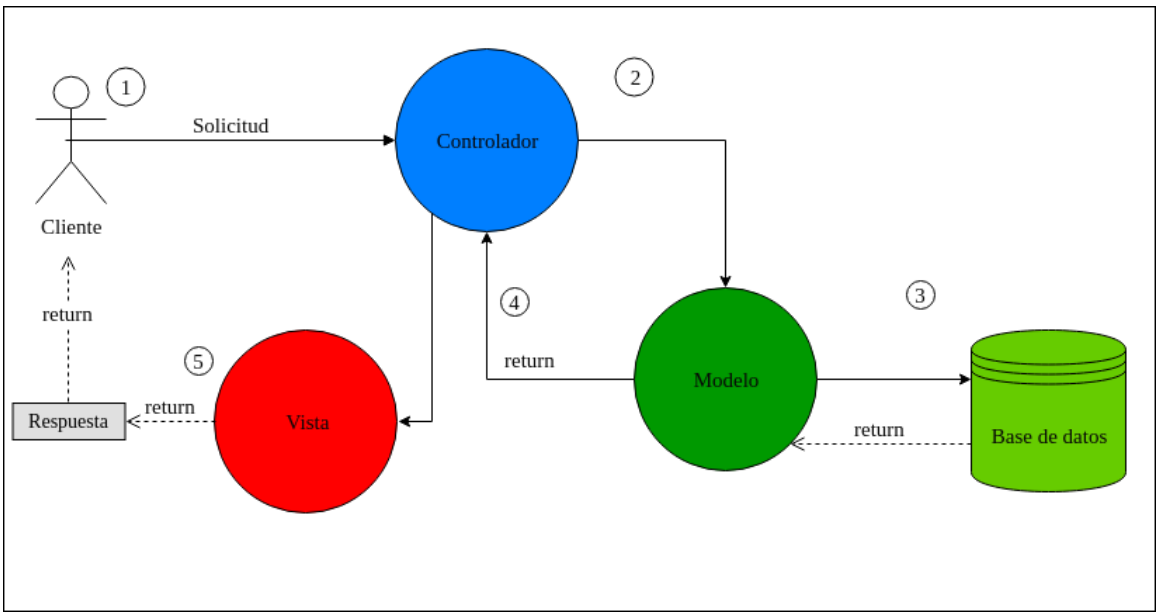

Figura 1.49: Estructura del estilo arquitectónico MVC de Controlador pesado. [Fuente: https://articulosvirtuales.com/articles/educacion/que-es-el-modelo-vista-controlador-mvc-y-como-funciona]

Un ejemplo de implementación del estilo MVC del controlador ligero con el estilo de eventos de invocación implícita es el que hace la librería Java SWING. En ella, se ha pasado del clásico MVC a una *arquitectura de modelo separable* (separable model architecture) en la que el controlador y la vista están fuertemente acoplados en una única clase llamada *UIObject* (o *UIDelegate*) (Ver Figura 1.50)[5].

Figura 1.50: Implementación del estilo MVC hecha por Java SWING.

[5]El elemento UI Manager es usado para almacenar información general a la apariencia de todos los componentes (color de fondo por defecto, tipo de letra por defecto, tipos de bordes, etc.), estando todos manejados por el mismo UI Manager.

Un ejemplo de utilización del estilo de controlador pesado es el que realiza el entorno de trabajo Angular JS para desarrollo Web, escrito en JavaScript (ver Figura 1.51).

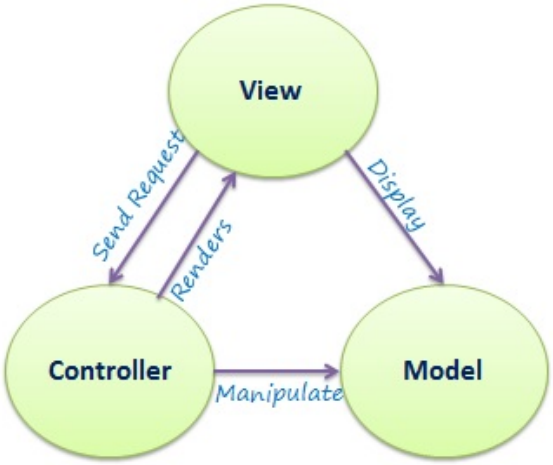

Figura 1.51: Estructura del estilo arquitectónico MVC pesado implementado por Angular JS. [Fuente: https://w3tutoriels.com/angularjs/angularjs-mvc/]

Existe una alternativa intermedia. Los controladores son suficientemente pesados para interactuar con las vistas, es decir, no lo hace el modelo directamente. Sin embargo, son las más ligeros posible, en el sentido de que todo lo que pueda hacer el modelo, se lo dejan al modelo, y de que tienen responsabilidades simples, pareciéndose al modelo en ese sentido. Este es el caso de Ruby on Rails (RoR), que tiene la máxima de *Fat models, skinny controllers*. Así, para cada clase del modelo, habrá un controlador, que hará lo mínimo: manejar el tráfico de datos, básicamente manejar la petición del usuario según la ruta, pasarla al modelo y devolver al usuario el resultado (vista).

RoR 1.4.1. ¡CUIDADO!: controladores pesados

Cuando tenemos controladores pesados en RoR, es porque están haciendo tareas que pertenecen al modelo y no seguimos las convenciones de Rails (la primera es: *Convention Over Configuration*).

A continuación podemos ver un ejemplo de un controlador (fichero *p*rojects_controllers.rb) para la clase **Project**:

```
class ProjectsController < ApplicationController

  before_action :set_project, only: %i[ show edit update destroy ]

  # GET /projects or /projects.json
  def index
    @projects = Project.all
  end
```

```
  # GET /projects/1 or /projects/1.json
  def show
  end

  # GET /projects/new
  def new
    @project = Project.new
  end

  # GET /projects/1/edit
  def edit
  end

  # POST /projects or /projects.json
  def create
    @project = Project.new(project_params)

    respond_to do |format|
      if @project.save
        format.html { redirect_to project_url(@project), notice: "Project
     was successfully created." }
        format.json { render :show, status: :created, location: @project
   }
      else
        format.html { render :new, status: :unprocessable_entity }
        format.json { render json: @project.errors, status: :
   unprocessable_entity }
      end
    end
  end
```

```
  # PATCH/PUT /projects/1 or /projects/1.json
  def update
    respond_to do |format|
      if @project.update(project_params)
        format.html { redirect_to project_url(@project), notice: "Project
     was successfully updated." }
        format.json { render :show, status: :ok, location: @project }
      else
        format.html { render :edit, status: :unprocessable_entity }
        format.json { render json: @project.errors, status: :
   unprocessable_entity }
      end
    end
  end

  # DELETE /projects/1 or /projects/1.json
  def destroy
    @project.destroy

    respond_to do |format|
      format.html { redirect_to projects_url, notice: "Project was
```

```
    successfully destroyed." }
      format.json { head :no_content }
    end
  end

  private
    # Use callbacks to share common setup or constraints between actions.
    def set_project
      @project = Project.find(params[:id])
    end

    # Only allow a list of trusted parameters through.
    def project_params
      params.require(:project).permit(:name, :team, :info)
    end
end
```

1.4.4. El estilo *Basado en Eventos*

Una alternativa a la llamada a procedimientos propia del estilo *Abstracción de datos y organización OO* (a partir de ahora, estilo OO, para simplificar) es la posibilidad de no hacer una invocación directa a un procedimiento o función (un método en OO) sino de forma indirecta. Para ello, los componentes con métodos cuya invocación dependa del estado de otros componentes, deben subscribirse a una lista de partes interesadas en este otro componente. Cuando en este otro componente se produce un cambio de estado (evento), lo retransmite o anuncia a los subscriptores de la lista, o lo publica y otro componente recoge la publicación para transmitirla a los subscriptores. Los subscriptores reciben el anuncio y realizan las operaciones necesarias a partir de la información que les ha sido comunicada. Este estilo también está considerado dentro del grupo de estilos *Peer-to-Peer* (Reynoso and Kicillof, 2004).

Variantes Hay dos variantes dentro de este estilo:

- Invocación implícita (estilo *Manejador de Eventos* o *Publicar/subscribir*).- El componente que anuncia un evento no conoce quiénes son sus clientes a la escucha. La forma de comunicación es mediante un tercer componente intermediario (manejador de evento) que es el que recoge los cambios publicados y sí conoce a los componentes que están interesados para retransmitírselos. Ejemplos de este estilo son usados por la librería SWING para la GUI en Java o por Flutter. En el caso de SWING, los que manejan el evento son objetos de la clase **ActionListener. En Flutter, para los eventos con varios subscriptores (listeners) la suscripción se puede hacer a través de la clase Listenable.**.
- Invocación explícita (patrón *Observador*[6]).- El componente en el que se produce el evento tiene él mismo la lista de subscriptores y les anuncia a todos el cambio que se

[6]Obsérvese que éste es también propuesto como patrón de diseño. Este es un ejemplo de la fuerte relación que hay entre ellos, hasta tal punto de que muchos autores no hacen tal distinción, considerando además

ha producido.

En la Figura 1.52 puede verse la diferencia entre el subestilo de *invocación explícita* u *Observador* (también un patrón de diseño) y el subestilo *Manejador de Eventos*.

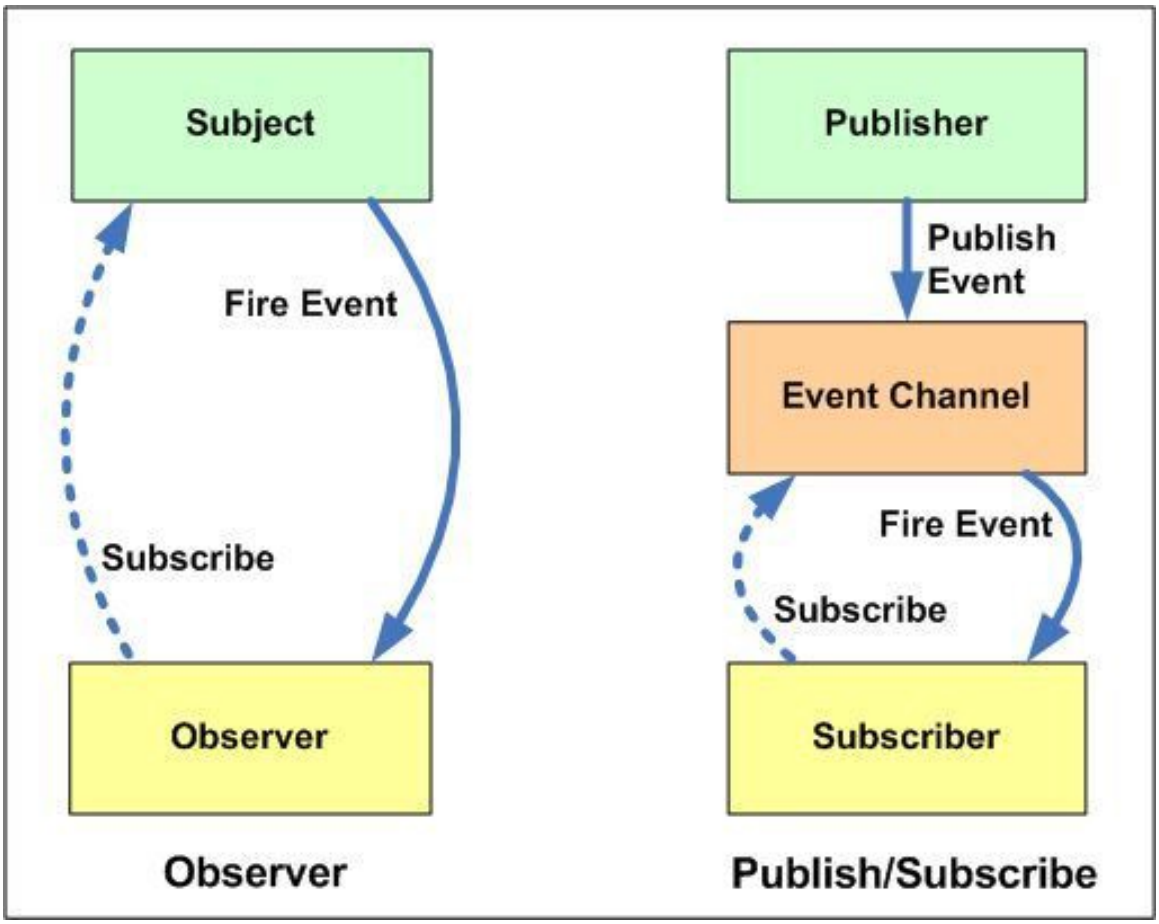

Figura 1.52: Diferencia entre los estilos arquitectónicos por invocación implícita (*Manejador de Eventos* o *Publicar/subscribir*) y el de invocación implícita (*Observador)*. [Fuente: https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c

```
public class Observable
{
    public ActionEvent<Object, Object> onStateChange;

    public void eventRelease()
    {
        // do something
        // choose appropriate args
        onStateChange.invoke(null, null);
    }
```

que diferentes patrones pueden aplicarse sobre un mismo sistema de forma jerárquica.

```
    /* versus ***********************************************/
    Observator[] observers;
    public void informObservators()
    {
        foreach (Observer observer in observers)
            // choose appropriate args
            observer.updateState(null, null);
    }
}

public class Observer
{
    public Observer(Observable observable)
    {
        observable.onStateChange += updateState;
    }
    /* versus ***********************************************/
    public void updateState(Object arg1, Object arg2)
    {

    }
}
```

Un ejemplo de uso del estilo de invocación implícita son los depuradores software. Cuando el depurador se para en un punto de interrupción (breakpoint), se anuncia esta parada. Los que reciben el anuncio operan en consecuencia, por ejemplo, el editor de textos para ponerse en la línea de parada y la ventana con el estado de las variables se actualiza.

Ventajas

- Reusabilidad: Pueden añadirse componentes al sistema sólo añadiéndolos a la lista de subscriptores
- Evolución más sencilla del sistema: Los componentes pueden reemplazarse (cambiando incluso su interfaz) sin que ellos afecte a las interfaces del resto de componentes con los que se relaciona (pues lo hacen de forma indirecta)

Inconvenientes

- Pérdida de control: Los componentes no saben realmente cómo afecta lo que hacen en otros componentes, ni siquiera el orden en el que los cambios se producen en los otros componentes (sobre todo en el estilo *Manejador de eventos*
- Difícil comprobación de integridad: Derivada de la pérdida de control también se hace difícil establecer las condiciones de ejecución de una operación concreta, frente a la tradicional llamada a procedimientos (estilo OO o estilo *Organización programa principal/subrutinas* (ver más abajo) para los que bastan las precondiciones y post-condiciones de una función o procedimiento para conocer su comportamiento en un momento determinado.

1.4.5. El estilo *Sistema por Capas*

En este estilo el sistema se organiza por capas, de forma que cada capa sirve a la capa superior y es cliente de la capa inferior. Cuando una capa sólo puede ver a las adyacentes podemos hablar de que se trata de una máquina virtual de cara a su capa cliente. La Figura 1.53 proporciona un ejemplo con sólo tres capas, mientras que la Figura 1.54 proporciona un diagrama genérico.

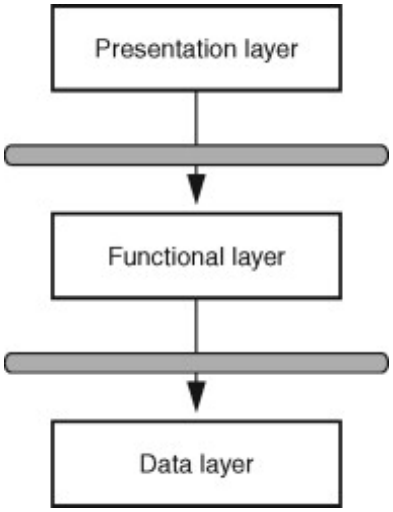

Figura 1.53: Estructura del estilo arquitectónico *Sistema por Capas* con sólo tres capas. [Fuente: (Züllighoven, 2005)]

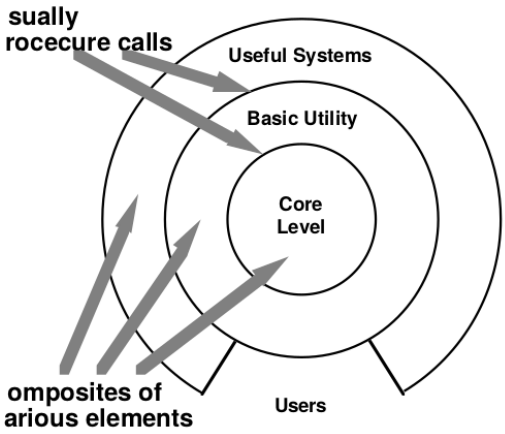

Figura 1.54: Estructura del estilo arquitectónico *Sistema por Capas* genérico. [Fuente: (Shaw and Garlan, 1996, pg. 25)]

Un ejemplo de este estilo es la forma en la que interactúa todo el software de un ordenador (capas: firmware y sistema operativo, utilidades, aplicaciones de usuario). Otro son las capas de un Sistema de Gestión de Bases de Datos (SGBD) (capas: física –gestor de datos y

metadatos en disco–, lógica –gestor de operaciones de consulta y actualización de datos como entidades provistas de significado para el usuario–, de aplicación: gestión de vistas y solicitudes de usuario).

Ventajas

- Soportan diseños basados en niveles de abstracción incremental.

- Reusabilidad: permiten la sustitución de la implementación de una capa por otro código, siempre que se mantega su interfaz (igual que en el estilo OO.

Incovenientes

- Difícil adaptación: No todos los sistemas pueden ser concebidos de forma jerárquica. Incluso si lo hacen pueden perder eficiencia frente a un estilo que permita mayor cohesión entre componentes.

Otro ejemplo es la arquitectura de Flutter, como puede verse en la Figura 1.55.

Flutter 1.4.1. Arquitectura en capas de Flutter

Cada capa es independiente de la otra y podría ser reemplazada (Ver Figura 1.55). La capa que tiene acceso a las operaciones básicas del Sistema Operativo (la capa *embedder*), está escrita en el lenguaje más apropiado para cada plataforma. En la actualidad está en Java y C++ para Android, en Objective-C y Objective-C++ para iOS y macOS y en C++ para Windows y Linux. Con esta capa, se puede desarrollar una aplicación Flutter independiente o un módulo para ser integrado en otra aplicación. La capa intermedia (la capa *máquina Flutter*) está escrita casi por completo en C++ y proporciona toda la implementación Flutter de bajo nivel. Es accesible desde la capa superior (la capa *framework Flutter*) mediante clases Dart que envuelven el código C++ a través de dart:ui. El framework –la última capa–, es la capa con la que interactúan normalmente los desarrolladores, y está escrita en Dart. A su vez, está compuesta por capas, como puede verse en la figura.

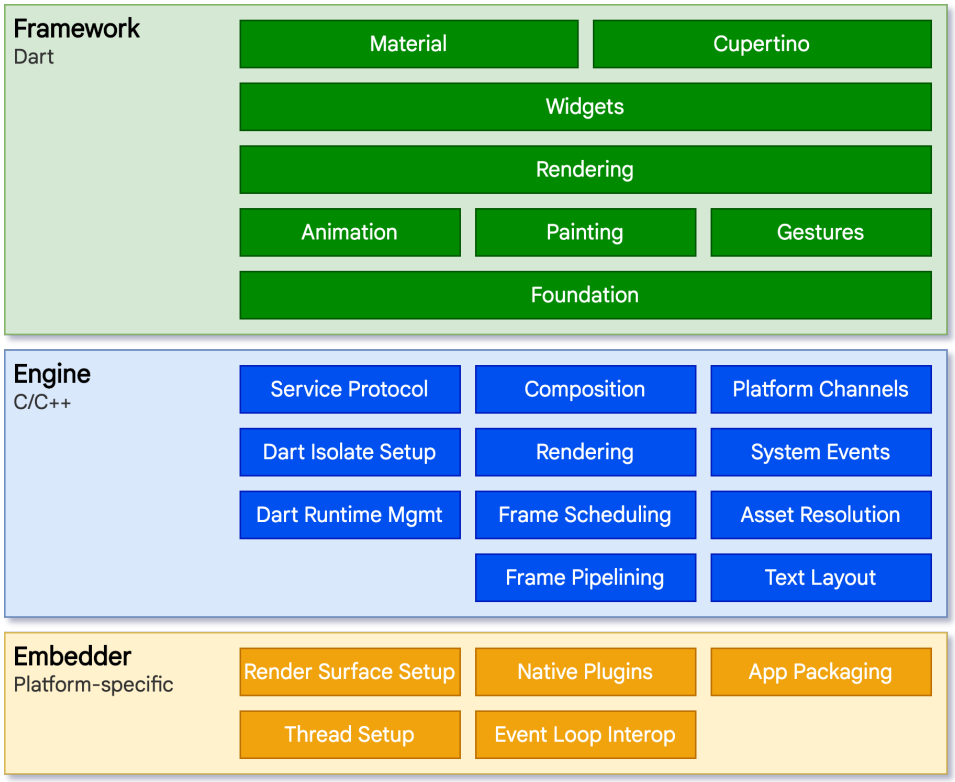

Figura 1.55: Estructura en capas de la arquitectura de Flutter. [Fuente: https://docs.flutter.dev/resources/architectural-overview#architectural-layers]

1.4.6. Estilos Centrados en Datos. El estilo *Repositorio*

Los estilos Centrados en Datos «enfatizan la integrabilidad de los datos. Se estiman apropiados para sistemas que se fundan en acceso y actualización de datos en estructuras de almacenamiento.» (Reynoso and Kicillof, 2004).

El estilo *Repositorio* está compuesto por un componente como estructura central de datos (almacén de datos o repositorio) y el resto de componentes, externos, que operan sobre él.

Variantes Se considera que existen dos variantes:

- Estilo *Repositorio Básico*: Cuando es una petición externa hacia un componente externo la que dispara una petición para que se ejecute un proceso concreto de ese componente que a su vez solicitará información al componente central. Un ejemplo es el uso de una base de datos distribuida.

- Estilo *Pizarra* (blackboard): Es el propio estado en el que se encuentra el componente central el que dispara el proceso a realizarse en un componente externo o *fuente de conocimiento* (knowledge source). Se utiliza en problemas para los que no existen estrategias para encontrar soluciones deterministas.

En una Pizarra, existen varios subsistemas especializados (*fuentes de conocimiento*, del inglés *knowledge sources, ks*) cuyo conocimiento es unido para encontrar una posible solución parcial o aproximada. Un ejemplo de la relación entre la pizarra y las *ks* en el estilo *Pizarra* puede verse en la Figure 1.56. Además existe un tercer componente, el componente de control, que permiten seguir y centralizar una estrategia para llegar a la solución (Buschmann et al., 1996). La Figura 1.57 usa un diagrama de clases para representar las relaciones entre los tres tipos de componentes.

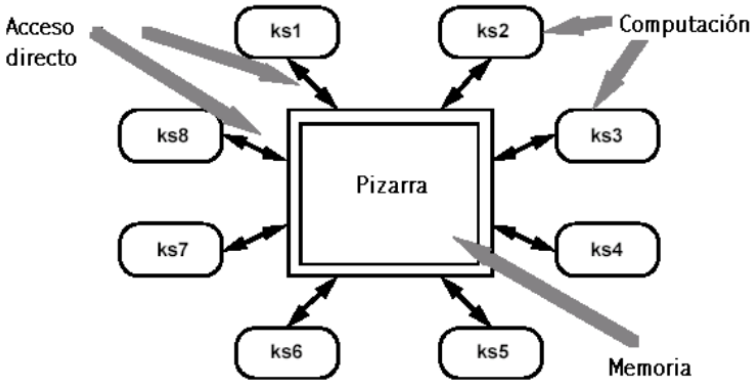

Figura 1.56: Relación entre la pizarra y las fuentes de conocimiento en el estilo arquitectónico *Pizarra*. [Fuente: (Shaw and Garlan, 1996, pg. 26)]

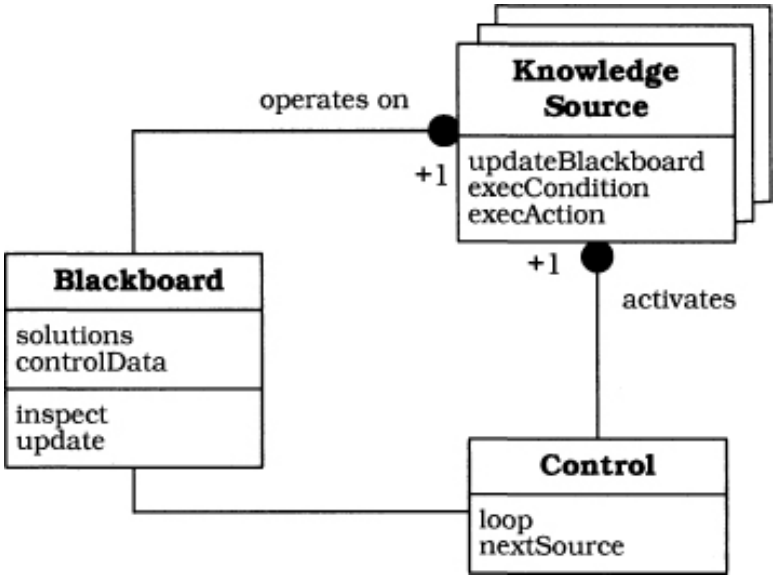

Figura 1.57: Diagrama de clases para representar el estilo arquitectónico *Pizarra*. [Fuente: (Buschmann et al., 1996, pg. 79)]

Un ejemplo de aplicación del estilo *Pizarra* es en el reconocimiento de voz, donde la solución no es determinista y donde cada fuente de conocimiento requiere un tipo distinto de información (ondas acústicas, fonemas, palabras, sintagmas, etc.).

1.4.7. Estilos de Código Móvil. El estilo *Intérprete*

Los estilos de Código Móvil enfatizan la portabilidad. Algunos ejemplos son los intérpretes, los sistemas basados en reglas y los procesadores de lenguaje de comandos (Reynoso and Kicillof, 2004).

El estilo *Intérprete* permite crear una máquina virtual que sirva de puente entre el nivel de computación básico que está disponible (programa objeto) para interactuar con el hardware y el nivel lógico que es entendido por la semántica de una aplicación.

Usualmente está formado por cuatro componentes:

- El motor que interpreta (o intérprete en sí)
- El contenedor con el pseudocódigo a ser interpretado
- Una representación del estado en el que se encuentra el motor de interpretación
- Una representación del estado en el que se encuentra el programa que está siendo simulado

La Figure 1.58 muestra la estructura del estilo *Intérprete*.

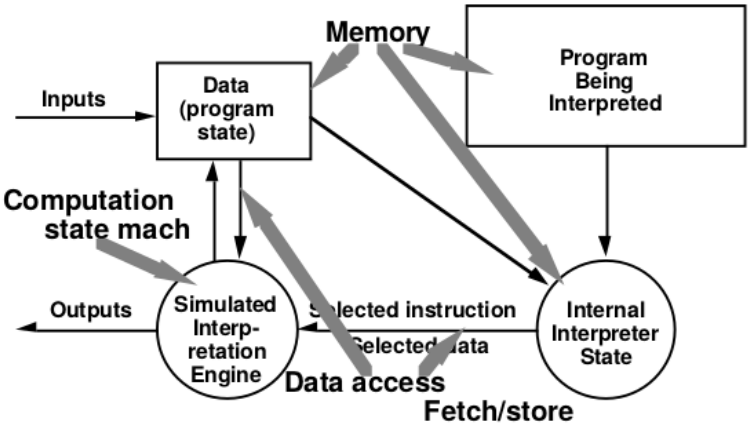

Figura 1.58: Estructura del estilo arquitectónico *Intérprete*. [Fuente: (Shaw and Garlan, 1996, pg. 27)]

Algunos ejemplos de lenguajes que usan intérpretes (lenguajes interpretados) actuales son Python, PHP y R.

Un ejemplo de intérprete más sofisticado son la máquinas virtuales, usadas por ejemplo para Java (JVM) o para Ruby (RubyVM) y disponibles para cualquier plataforma. En este caso, el código de entrada son los bytecodes (ficheros .class resultados de la compilación de los .java o los .rb respectivamente).

1.4.8. Estilos heterogéneos. Estilo *Control de procesos*

Su propósito es el de mantener las propiedades de salida (variables controladas) de un proceso en un nivel o cercano a un nivel de referencia (set point). Se usa sobre todo en el entorno industrial.

Variantes Existen algunas variantes:

- *Ciclo abierto* (Figure 1.59).- En muy pocos casos, cuando todo el proceso es completamente predecible, no es necesario vigilar el proceso (controlar el estado de las variables y reaccionar en consecuencia).

- *Ciclo cerrado* (Figure 1.60).- Es necesario supervisar el sistema para corregir la salida según el cambio en los valores de las variables de entrada.

    - *Control de procesos retroalimentado* (Figure 1.61)
    - *Control de procesos preventivo* (Figure 1.62)

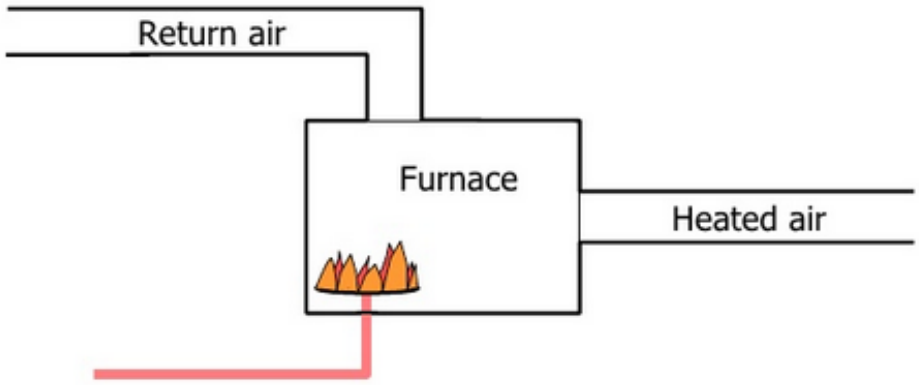

Figura 1.59: Ejemplo de sistema con estilo arquitectónico *Control de procesos de ciclo abierto*. [Fuente: (Shaw and Garlan, 1996, pg. 29)]

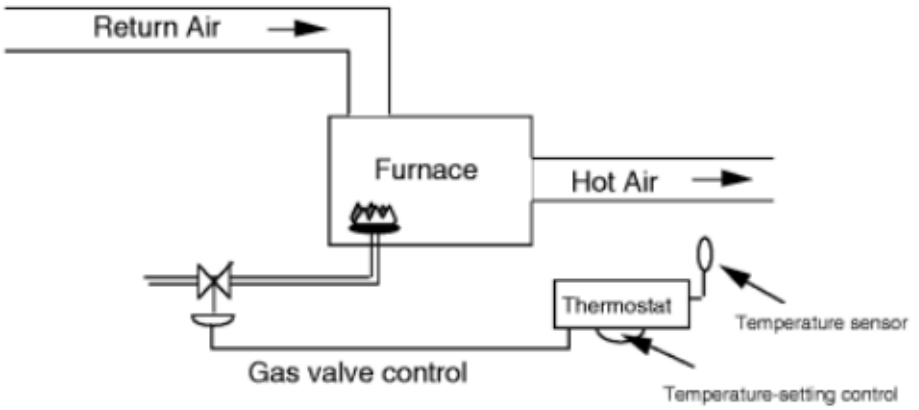

Figura 1.60: Ejemplo de sistema con estilo arquitectónico *Control de procesos de ciclo cerrado*. [Fuente: (Shaw and Garlan, 1996, pg. 29)]

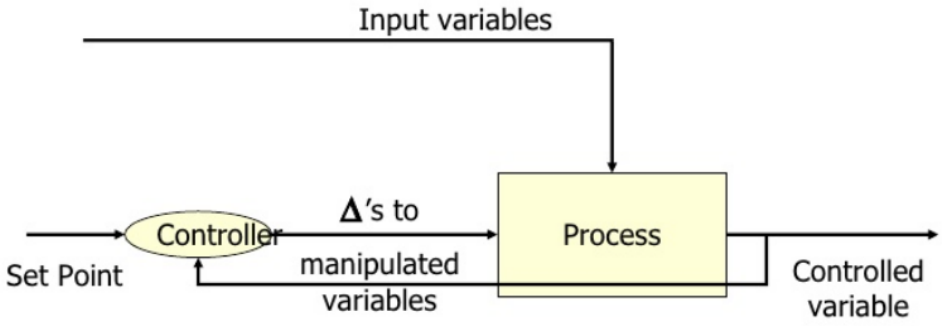

Figura 1.61: Estructura del estilo arquitectónico *Control de procesos retroalimentado*. [Fuente: (Shaw and Garlan, 1996, pg. 29)]

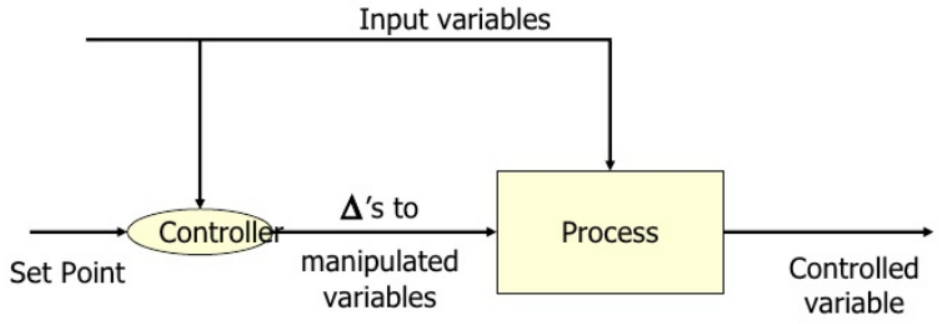

Figura 1.62: Estructura del estilo arquitectónico *Control de procesos preventivo*. [Fuente: (Shaw and Garlan, 1996, pg. 30)]

La Figura 1.63 muestra un ejemplo de un sistema de control de procesos retroalimentado: un sistema de control de la velocidad de crucero. En concreto es un diagrama de bloques del sistema.

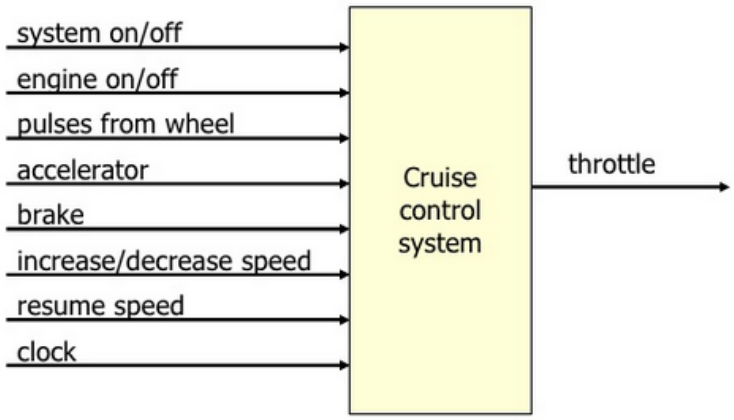

Figura 1.63: Diagrama de bloques del sistema de control de la velocidad de crucero. [Fuente: (Shaw and Garlan, 1996, pg. 52)]

Solución mediante arquitectura OO (*Abstracción de datos y organización OO*)
Ejemplo de solución propuesta por la arquitectura de Booch: 1.64

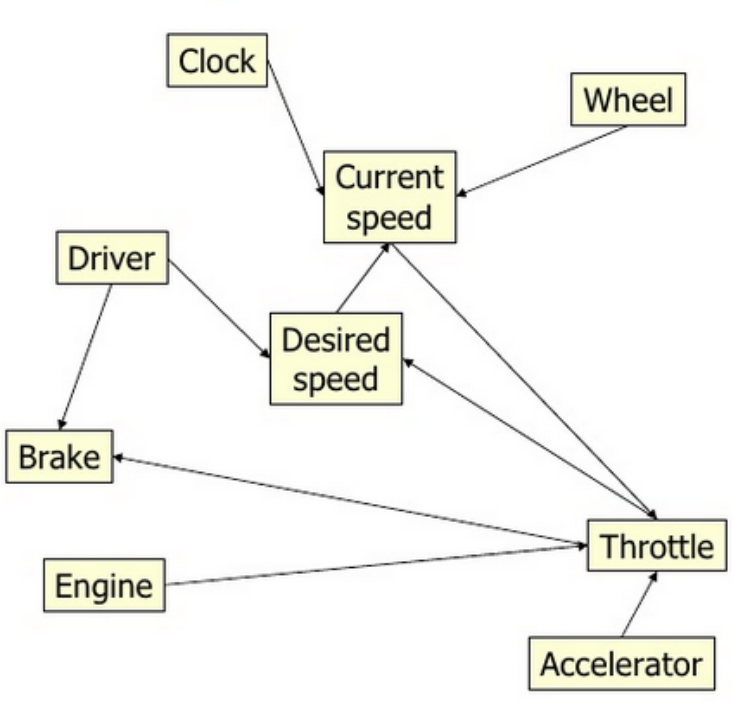

Figura 1.64: Diagrama de clases para el sistema de control de la velocidad de crucero. [Fuente: (Phillips et al., 1999)]

Diagrama del estilo arquitectónico *Control de procesos* para el sistema de control de la velocidad de crucero propuesto por David Garlan. 1.66

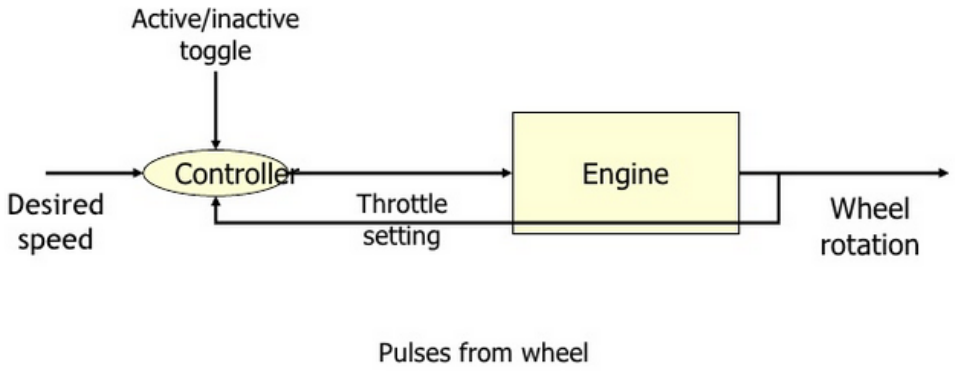

Figura 1.65: Diagrama del estilo arquitectónico *Control de procesos* para el sistema de control de la velocidad de crucero. [Fuente: (Phillips et al., 1999)]

Diagrama de componentes usando el estilo arquitectónico *Control de procesos* para el sistema de control de la velocidad de crucero propuesto por David Garlan. 1.66

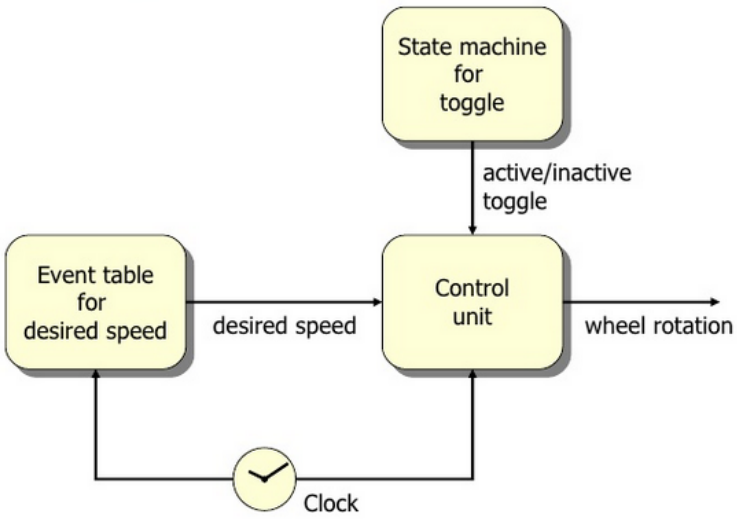

Figura 1.66: Diagrama de componentes). [Fuente: (Phillips et al., 1999)]

1.4.9. Otros estilos arquitectónicos

Hay otros muchos paradigmas o estilos arquitectónicos, que pueden ser aplicados de forma simultánea. Algunos de ellas se describen a continuación:

- Estilo *Organización programa principal/subrutinas*.- Es el estilo que proyecta directamente la forma de funcionamiento de los lenguajes de programación que son usados por el sistema cuando éstos no permiten modularidad. Pertenece a la categoría de estilos de *Llamada y Retorno*.

- Estilo *Sistema de Transición de Estados*.- Utilizado, por ejemplo, en sistemas que definen estados y transiciones entre ellos para pasar de un estado a otro.

- Paradigma de *Programación reactiva*.- Es compatible con otros paradigmas de programación, como por ejemplo funcional e imperativa o procedural. Se refiere al uso de programación asíncrona para actualizar en tiempo real el estado de un sistema como consecuencia de otros cambios de estado ocurridos.

  Un ejemplo es el usado de forma interna en los sistemas para la construcción de interfaces de usuario JavaScript React, de Facebook y el framework Flutter, de Google, que se inspiró en la primera. En el Cuadro FLUTTER 1.4.2 se describe el estilo según se aplica en Flutter para crear interfaces de usuario reactivas.

Flutter 1.4.2. Interfaces de usuario reactivas con Flutter

Se trata de un sistema pseudo-declarativo, en el cual, el desarrollador debe proporcionar un mapeo entre el estado de la aplicación y el de la interfaz. Cada vez que cambia el estado de la aplicación, el framework debe actualizar la interfaz en tiempo de ejecución. Está especialmente indicado para interfaces de usuario muy complejas, de forma que se ahorra mucho tiempo al desarrollador especificando qué aspecto debe tener la interfaz ante cualquier interacción del usuario y cambio del modelo (estado del sistema).

PREGUNTA 1.4.1. Programación reactiva y Manejador de eventos

¿Qué diferencia hay entre ambos estilos?

- Estilo de *Procesos Distribuidos*.- Dentro de éstas existen arquitecturas diversas según criterios diversos, como la topología que relaciona los procesos o el protocolo de comunicación entre procesos. Por ejemplo, atendiendo a la topología de la red podemos tener estilos arquitectónicos en anillo o en estrella.

  - Estilo *Peer-to-Peer (P2P)*.- Se refiere a todas las arquitecturas de componentes independientes en las que cada componente o nodo tiene las mismas capacidades y responsabilidades. Dos ejemplos en software de intercambio de archivos son eMule y BitTorrent.
  - Estilo *Cliente/Servidor*.- El estilo *Cliente/Servidor* es un tipo particular de arquitectura de procesos distribuidos de uso muy generalizado en el que al menos entran en juego dos nodos con responsabilidades y capacidades muy distintas: el cliente (pudiendo haber varios) y el servidor. El servidor representa a los procesos que dan servicio y el cliente a los procesos que reciben el servicio. El servidor no conoce a los clientes por adelantado, mientras que el cliente conoce al servidor y accede a él mediante llamada a procedimientos remota (remote procedure call, RPC). También se considera un subtipo del estilo *Sistema por Capas* con solo dos capas aunque realmente puede haber más capas dentro del servidor o del cliente. Dentro de este estilo, podemos encontrar al menos dos subtipos de arquitecturas:

- *Arquitectura Orientada a Servicios* (del inglés *Services Oriented Architecture (SOA)*).- Arquitectura en la cual, los componentes están fuertemente desacoplados[7], de forma que unos proporcionan (servidores) y otros consumen (clientes) servicios, sin que éstos conozcan la implementación de los servidores (caja negra), logrando la interoperabilidad en la interacción entre máquinas, sistemas software y aplicaciones a través de la red. Además se trata de una arquitectura sin estado[8], que conectan entre sí para proveer nueva funcionalidad, por una interfaz bien definida.
  Esta arquitectura se ha desarrollado especialmente por los servicios Web [9], «ya que poseen un conjunto de características que permiten cubrir todos los principios básicos de la orientación a servicios» (Los Santos Aransay, 2009). En SOA, se consigue el desacoplamiento entre los agentes software que interactúan siguiendo dos principios:
  1. Usan un conjunto reducido de interfaces simples accesibles por todos los proveedores y consumidores.
  2. Usan mensajes decriptivos, entregados a través de las interfaces, más que instructivos, pues solo interesa saber qué servicio demandamos, no cómo se implementa internamente.

  Un ejemplo de protocolo creado bajo esta arquitectura es el *protocolo SOAP (Simple Object Access Protocol)*.- Incluye (1) un protocolo con el mismo nombre (SOAP), basado en XML para pedir los servicios web mediante un protocolo de transporte (HTTP, SMTP, etc.); (2) un directorio donde publicar los servicios Web ofertados con la información necesaria para usarlos (UDDI: Universal Description, Siscovery and Integration); y (3) un lenguaje basado en XML para decribir los servicios Web a publicar en un directorio UDDI: WSDL (Web Sevices Description Language).
  A su vez, dentro del estilo arquitectónico SOA encontramos al menos dos subtipos arquitectónicos:
  - *Arquitectura Basada en Recursos: REST (REpresentational State Transfer)*.- Se basa en el concepto de recurso: cualquier cosa con una URI (Uniform Resource Identifier). Permite distintos formatos de texto (plano, HTML, XML, JSON, etc.), aunque lo más frecuente es usar JSON. También podría usar el protocolo SOAP, basado en XML pero más complicado y requiriendo, por tanto, un mayor ancho de banda. Los servicios y los proveedores de servicios deben ser recursos. Aunque es un subtipo más de SOA, para algunos, tiene otras características que la hacen diferen-

[7] Para ver su relación con un estilo de mayor acoplamiento del que proviene, la *Arquitectura basada en Componentes*, puede consultarse el trabajo de Helmut Petritsch (Petritsch, 2006).

[8] Cada petición del cliente al servidor debe contener toda la información necesaria para procesar la petición, sin poder hacer uso de ninguna información de contexto almacenada en el servidor. Por eso la información del estado de una sesión se almacena por completo en el cliente (cookies).

[9] «Conjunto de protocolos, estándares y recomendaciones, definidos por la W3C (World Wide Web Consistorium) y OASIS (Organization for the Advancement of Structured Information Standards).» (Los Santos Aransay, 2009).

te (Reynoso and Kicillof, 2004; Los Santos Aransay, 2009, Consultado en mayo de 2021). Las cinco características necesarias para tratarse de arquitectura REST son:

1. Arquitectura cliente-servidor (como cualquier SOA).

2. Sin estado (como cualquier SOA).

3. Cacheable.- Para mejorar la eficiencia de la red, las respuestas deben ser capaces de ser etiquetadas como cacheables o no cacheables. La caché del cliente podrá usar una respuesta anterior si era cacheable.

4. Interfaz uniforme.- Esta característica simplifica la arquitectura. Para hacerla uniforme, REST aplica cuatro restricciones a sus interfaces, que se muestran en Tabla 1.3. Puede consultarse el Cuadro PARA PROFUNDIZAR 1.4.1 para más detalles sobre la forma más convencional de uso REST para garantizar la uniformidad de sus interfaces.

5. Sistema por capas.- Se permiten capas entre el cliente y el servidor, de forma que desde una capa solo se tiene acceso a las adyacentes. Por ejemplo, se pueden usar capas intermedias para mejorar la seguridad o el rendimiento (servidores proxy, caché, puertas de enlace, ...).

◇ *Arquitectura de Computación en la Nube*.- Se trata de un tipo de *Sistema por Capas*, también orientado a servicio, como SOA, pero en el que los servicios finales consisten en recursos computacionales (almacenamiento y procesamiento), y los clientes (demandantes) no manejan directamente las peticiones a un servidor concreto sino que los proveedores les proporcionan lo que necesitan según los recursos disponibles en cada momento. Los servicios además no son horizontales, como en SOA básico, sino que se organizan por capas (v.g. de infraestructura, de plataforma y de aplicación) para proporcionar el servicio demandado por el ciente.

La figura 1.67 muestra la relación entre distintos superestilos y subestilos arquitectónicos de la arquitectura orientada a servicios (SOA).

- Estilos específicos de un dominio.- Como por ejemplo estilos concretos para la aviación, los videojuegos o los sistemas de gestión de vehículos. Al estar el domino restringido el estilo o patrón es más específico y permite generar código a partir de ella de forma automática o semi-automática.

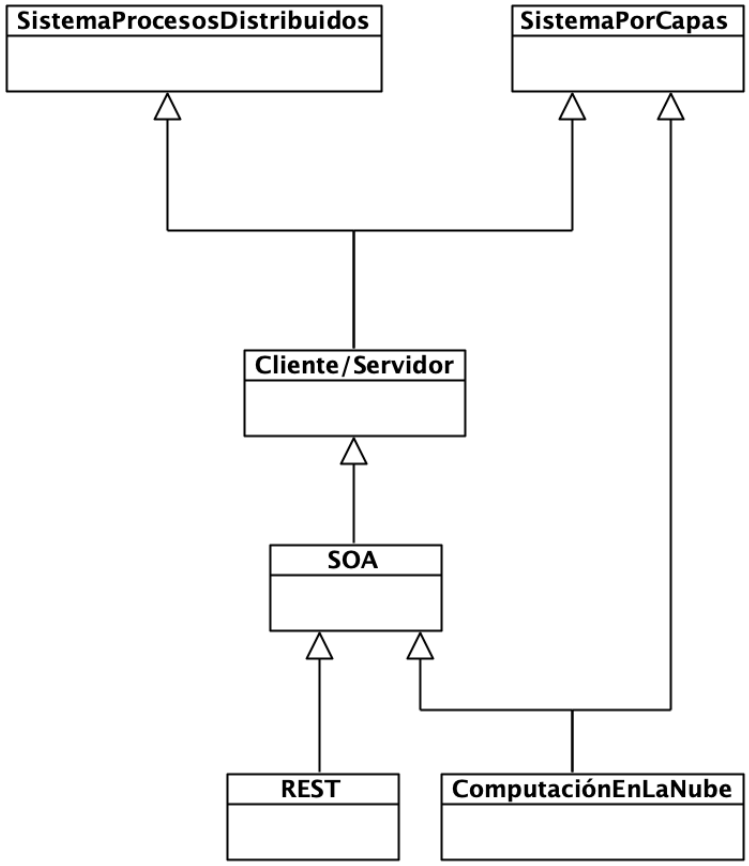

Figura 1.67: Relación de distintos superestilos y subestilos arquitectónicos de la arquitectura orientada a servicios (SOA).

Restricción	Descripción
URI	Identificación de los recursos
Representaciones	Manipulación de los recursos mediante representaciones, siendo una *representación*, el estado de un recurso en un momento concreto
Mensajes auto-descriptivos	El cliente tiene que entender el formato usado en las distintas representaciones de los recursos (lo que se llama *tipos de medios*, del inglés *media types*)
Hipermedia o hipertexto	Más genérico que usar navegadores y HTML, XML o JSON

Tabla 1.3: Restricciones adicionales de REST para hacer las interfaces uniformes.

PARA PROFUNDIZAR 1.4.1: REST e interfaz uniforme

De forma convencional se asocian los métodos recurso de REST con los métodos GET/PUT/POST/DELETE del protocolo HTTP, aunque lo único que establece REST es que se use una interfaz uniforme.

Así, las interfaces se construyen solo sobre HTTP, lo que las hace muy simples, con siete funciones definidas siguiendo el protocolo HTTP: GET, DELETE, HEAD, OPTION, POST, PUT, PATCH y TRACE, todas idempotentes[a], salvo POST. Las cuatro más usadas en muchas implementaciones REST (RESTFul) pueden verse en la Tabla 1.4. En la Figura 1.68 puede verse un ejemplo de uso.

Pero tampoco es necesario para implementar REST seguir esta convención. Se podría usar POST para actualizar un recurso en vez de PUT. Lo único que se exige es la uniformidad de la interfaz (que se haga de la misma manera con todos los recursos que ofrece el servicio Web).

También de forma convencional se suele usar XML y mucho más JSON como formatos de datos transmitidos. En un intento de hacer más estándar la Web, se recomienda usar los principios REST de forma más restictiva, y por ello a menudo se identifica HTTP con REST. Como además, se eligen los cuatro métodos HTTP y se asocia cada uno de ellos a una de las cuatro operaciones (CRUD o CLAB en español [b]) de persistencia de datos (bases de datos) en un sistema software, hay una triple asociación entre los conceptos CRUD, REST y HTTP, como se ve en la Tabla 1.5 con un ejemplo.

En la Tabla 1.6 se pone un ejemplo en una API de Ruby on Rails junto con las acciones del controlador que provoca cada ruta (URI). Como se ve para la operación *index*, el mismo recurso puede tener asociadas más de una URI.

[a] Una función es idempotente cuando se produce el mismo efecto independientemente del número de veces que sea invocada.

[b] Del inglés Create/Read/Update/Delete.

Operación	Descripción
HTTP GET	Usado para obtener una representación de un recurso. Un consumidor lo utiliza para obtener una representación desde una URI. Los servicios ofrecidos a través de este interfaz no deben contraer ninguna obligación respecto a los consumidores
HTTP DELETE	Se usa para eliminar representaciones de un recurso. La segunda y siguientes veces que se hace, el recurso ya habrá sido borrado (Error 404) y el estado del recurso no cambia pues no existe [10]
HTTP POST	Usado para crear un recurso. NO es idempotente
HTTP PUT	Se usa para actualizar el estado de un recurso. Una vez actualizado, las siguientes repeticiones de esta operación, no provocarán ningún cambio en el recurso

Tabla 1.4: Las cuatro operaciones HTTP más habituales en REST.

CRUD	HTTP	REST
Create	POST	/api/project
Read	GET	/api/project/id
Update	PUT	/api/project/id
Delete	DELETE	/api/project/id

Tabla 1.5: Ejemplo de la relación entre CRUD, HTTP y REST sobre «project».

[10] Una API que permita borrar el último objeto con la función DELETE: `DELETE /item/last` no cumple con la propiedad de idempotencia del método DELETE que exige HTTP y no sería una buena API REST. La solución para borrar de esta manera es usar el protocolo POST.

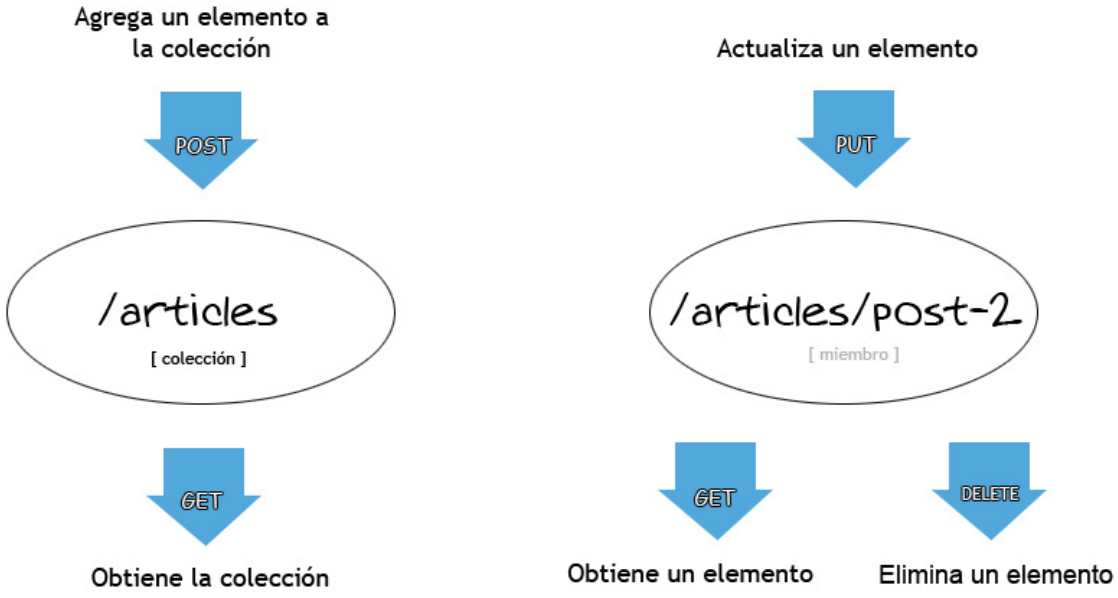

Figura 1.68: Ejemplo de uso de las funciones REST [Fuente: (Los Santos Aransay, 2009)]

CRUD	HTTP	REST	Acción controlador
Create	POST	/taller_rails/project	projects#create
Read	GET	/taller_rails/projects/id	projects#show
Read	GET	/taller_rails/projects	projects#index
Read	GET	/taller_rails	projects#index
Read	GET	/taller_rails/projects/new	projects#new
Read	GET	/taller_rails/projects/id/edit	projects#edit
Update	PUT	/taller_rails/projects/id	projects#update
Update	PATCH	/taller_rails/projects/id	projects#update
Delete	DELETE	/taller_rails/projectid	projects#destroy

Tabla 1.6: Ejemplo de la relación entre CRUD, HTTP, REST y las operaciones del controlador en una aplicación realizada con Ruby on Rails: http://clados.ugr.es/taller_rails.

1.4.10. Combinación de estilos y frontera débil con patrones de diseño

En la realidad, los estilos se combinan de forma que en un solo sistema puede aplicarse más de uno. Se puede considerar que un componente implementa a su vez otro estilo y así sucesivamente, de forma que se relacionan entre ellos de forma jerárquica. Un ejemplo es el estilo *Tubería y Filtro*, como por ejemplo ocurre en las tuberías de Unix, que pueden programarse mediante la línea de comandos. Cada filtro puede implementar a su vez otro estilo arquitectónico.

También es posible que un mismo componente forme parte de más de un estilo, porque tenga conectores de varios estilos, como los estilos *Repositorio*, *Tubería y Filtro* y *Control de Procesos*. Así, la interfaz tendrá una parte específica para cada tipo de conector.

Por otro lado, la frontera entre estilo arquitectónico y patrón de diseño no está siempre clara, en especial cuando bajamos en el nivel de anidamiento entre estilos arquitectónicos en sistemas que implementan más de uno.

Bibliografía

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

Anand Balachandran Pillai. *Software Architecture with Python*. Packt Publishing, 2017. URL https://learning.oreilly.com/library/view/software-architecture-with/9781786468529/ch08s04.html.

Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, 2003.

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.

Garfixia. Pipe-and-filter, Accessed March 4, 2020. URL http://www.dossier-andreas.net/software_architecture/pipe_and_filter.html.

Alberto Los Santos Aransay. Revisión de los servicios web soap/rest: Características y rendimiento. Technical report, Universidad de Vigo, 2009. URL http://www.albertolsa.

com/wp-content/uploads/2009/07/mdsw-revision-de-los-servicios-web-soap_rest-alberto-los-santos.pdf.

Alberto Los Santos Aransay. Rest api tutorial, Consultado en mayo de 2021. URL https://restfulapi.net.

Helmut Petritsch. Service-oriented architecture (soa) vs. component based architecture. Technical report, 2006. URL http://petritsch.co.at/download/SOA_vs_component_based.pdf. Consultado el 13 de febrero 2023.

Greg Phillips, Rick Kazman, Mary Shaw, and Florian Mattes. Process control architectures, 1999. URL https://www.slideshare.net/ahmad1957/process-control.

Trygve Reenskaug. A note on dynabook requirements. Technical report, Xerox PARC, 1979. URL http://folk.uio.no/trygver/1979/sysreq/SysReq.pdf.

Carlos Reynoso and Nicolás Kicillof. Estilos y patrones en la estrategia de arquitectura de microsoft. Technical report, Universidad de Buenos Aires, 2004. URL http://biblioteca.udgvirtual.udg.mx/jspui/handle/123456789/940.

Mary Shaw and David Garlan. *Software Architecture*. Prentice Hall, New Jersey, 1996.

Heinz Züllighoven. *Object-Oriented Construction Handbook*. Science Direct, EE.UU., 2005.