Tema 1. Desarrollo utilizando patrones de diseño

Desarrollo de Software

Curso -

3º - Grado Ingeniería Informática

Dpto. Lenguajes y Sistemas Informáticos
ETSIIT
Universidad de Granada

3 de marzo de 2023

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

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

- 1987: Llevada la idea a un congreso (Beck and Cunningham, 1987)
- 1994: Primer libro ("Gang of Four") (Gamma et al., 1994a)
  Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice (Appleton, 2000).
- 1993: Se retiran a la nieve para pensar, Kent Beck y Grady Booch organizan:
  - Patrón arquitectónico y objeto software (creatividad)
  - Crean el "HillSide Group" (The HillSide Group).
  - 1996: Nuevo libro, patrones en todos lo niveles de abstracción/etapas (Buschmann et al., 1996)

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

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

Fueron definidos por GoF como:

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

Y por Go5 como:

> *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).*

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

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

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

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

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

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

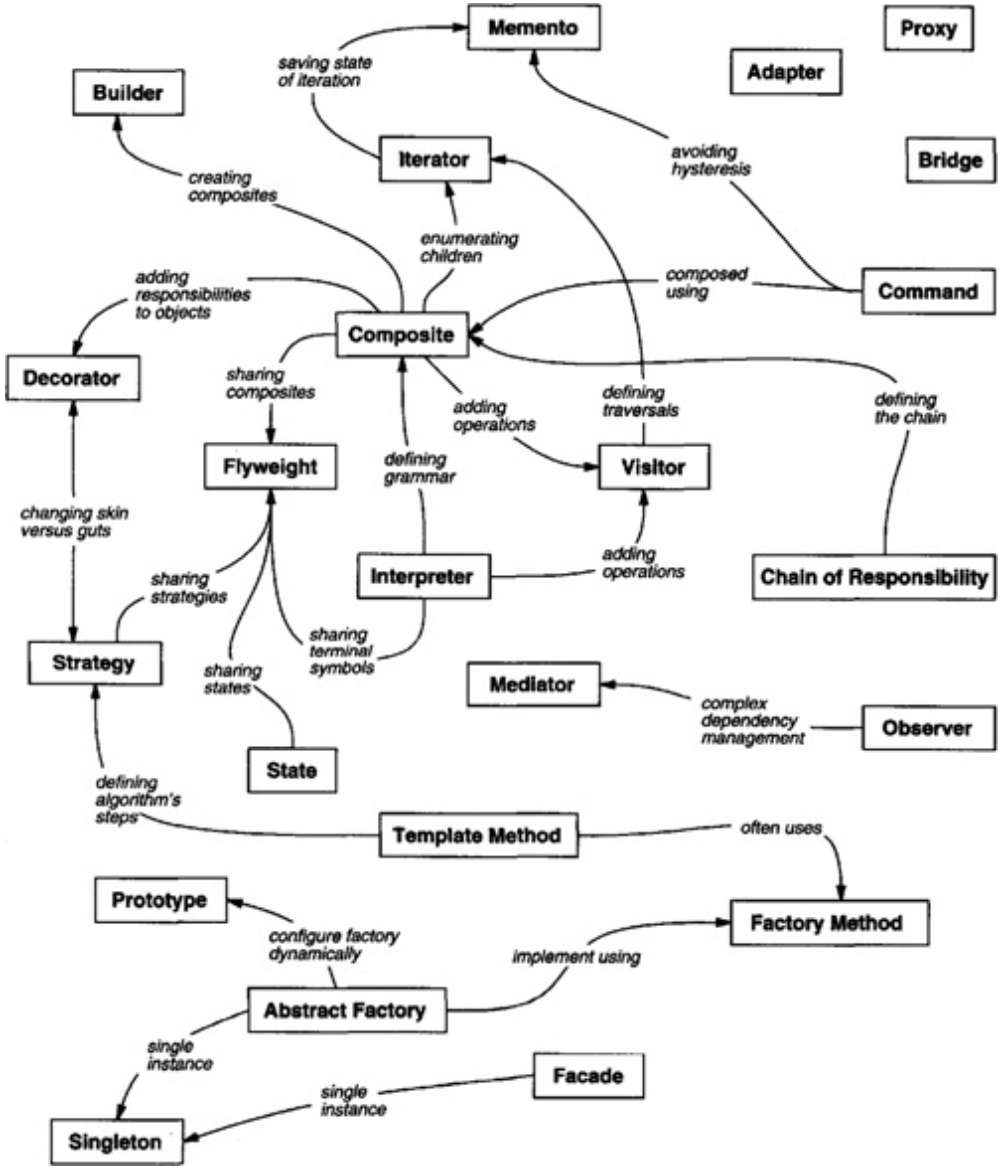


Figura 2: Relación entre los patrones de diseño. [Fuente: (Gamma et al., 1994b), p. 23]

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

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

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

- Idea: separar el objeto de la aplicación (el modelo) de la interacción con el usuario, para aumentar su flexibilidad y reusabilidad.
- La GoF utiliza el diseño MVC para identificar e introducir los tres primeros patrones de su libro (Gamma et al., 1994a).
- MVC hoy se considera un patrón arquitectónico.

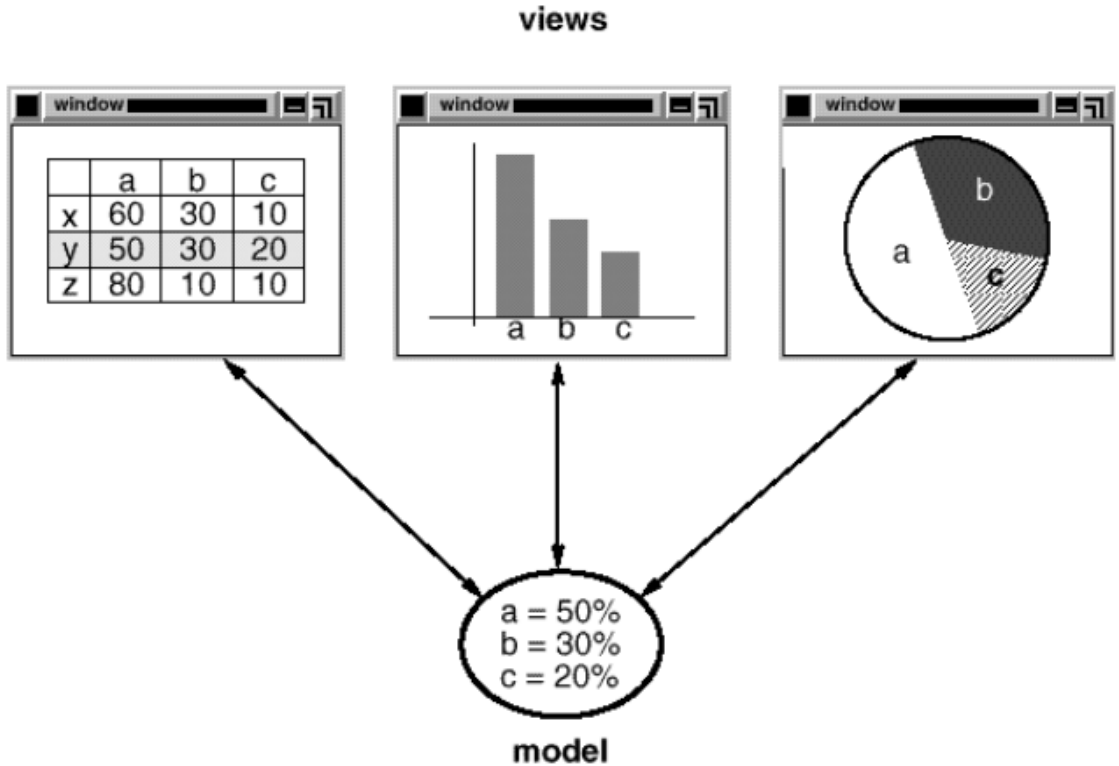


Figura 3: Ejemplo de modelo con tres vistas, en un diseño MVC. [Fuente: (Gamma et al., 1994b), p. 15]

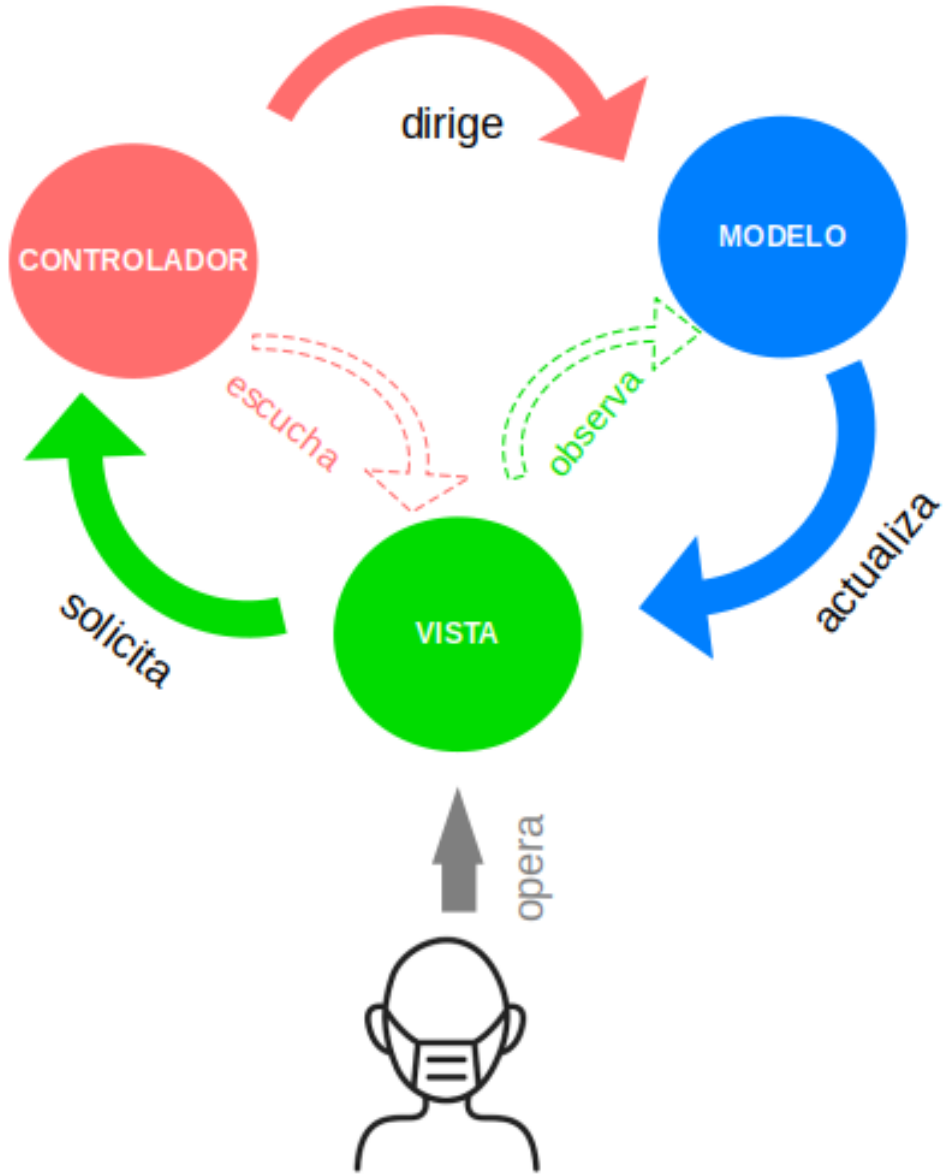


Figura 4: Ejemplo de diagrama relacional de la terna de clases en el diseño MVC.

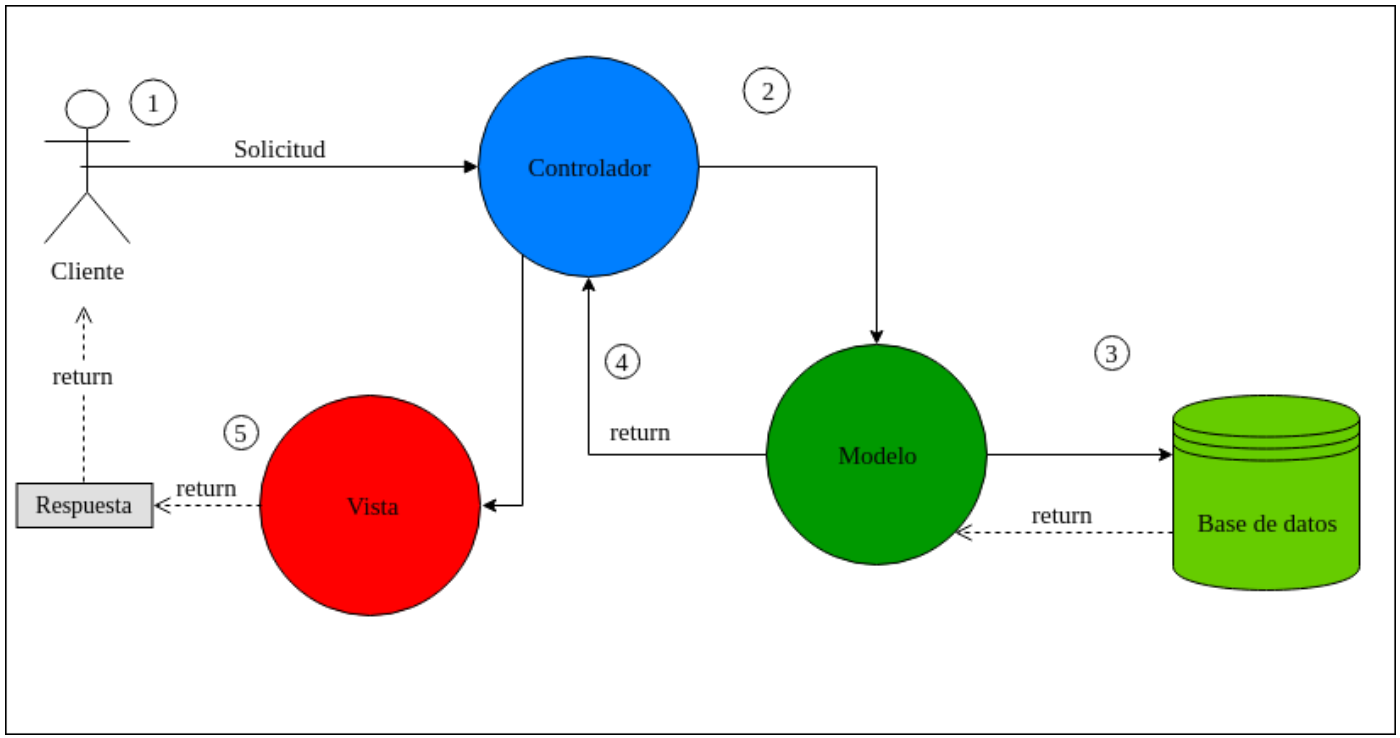


Figura 5: Ejemplo más complejo de diagrama relacional en el diseño MVC.

Patrones que incluye:

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

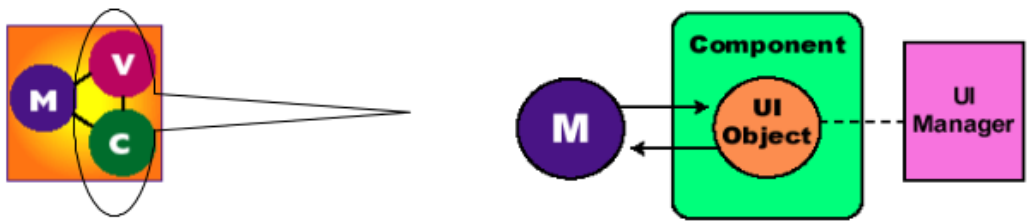


Figura 6: Ejemplo más complejo de diagrama relacional en el diseño MVC con el patrón estrategia.

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

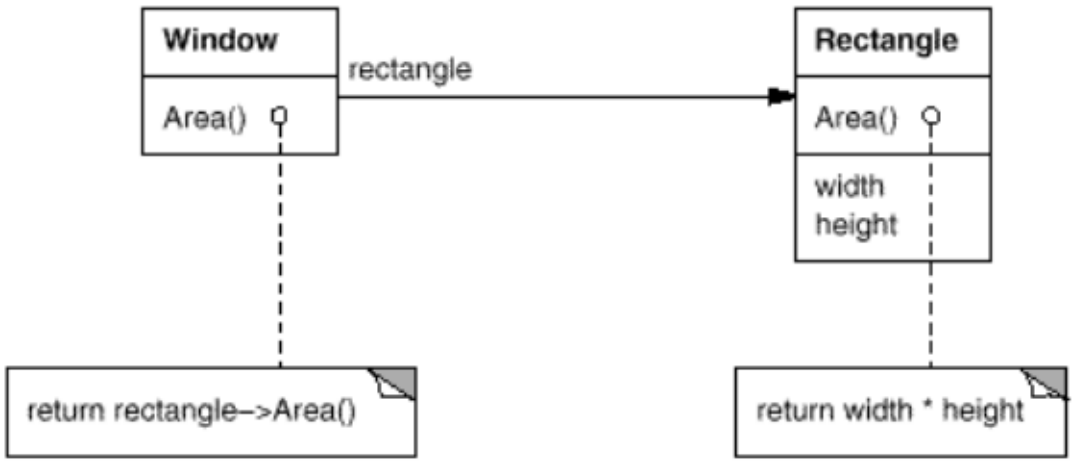


Figura 7: Ejemplo de uso de delegación (Gamma et al., 1994a, p.33) p. 33.

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

- *Abstract Factory* (Gamma et al., 1994b) pp. 99-109 (Figura 31)
- *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 68)
- *Strategy* (Gamma et al., 1994b) pp. 349-359
- *Template Method* (Gamma et al., 1994b) pp. 360-365

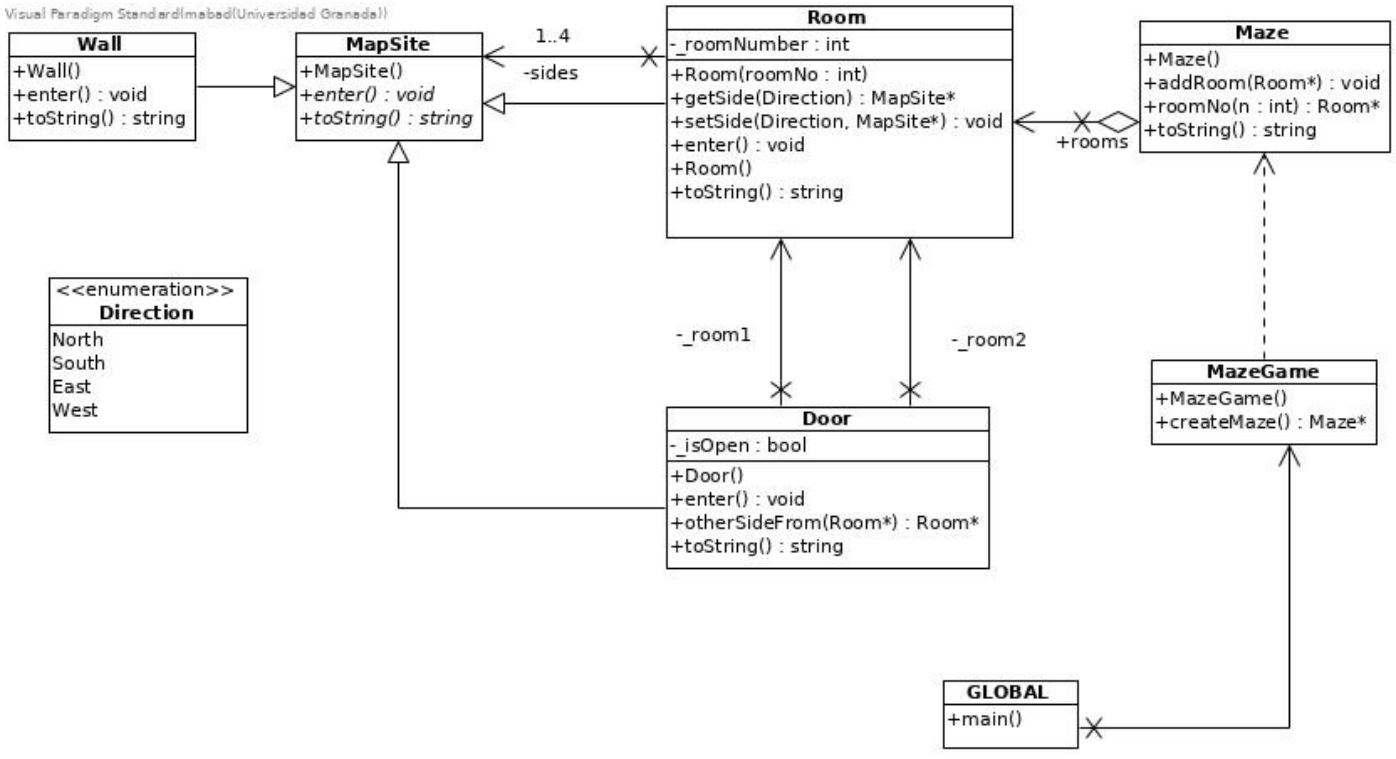
Veremos además los siguientes otros patrones:

- *Prototype* (Gamma et al., 1994b) pp. 133-143 (Figura 40)
- *Builder* (Gamma et al., 1994b) pp. 110-120
- *Visitor* (Gamma et al., 1994b) pp. 366-381 (Figura 70)
- *Facade* (Gamma et al., 1994b) pp. 208-217
- *Filtro de intercepción* (Figura 26)

Usados cuando necesitamos crear objetos dentro de un marco de trabajo o una librería software pero desconocemos la clase de estos objetos ya que depende de la aplicación concreta.
Usaremos como ejemplo comparativo el juego del laberinto:

Figura 8: 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].

2. Estudio del catálogo GoF de patrones de diseño
Patrones creacionales *Factoría Abstracta*, *Método Factoría*, *Prototipo* y *Builder*



Una factoría es simplemente un objeto con algún método (método factoría) para crear objetos (ver Figura 9), pero el concepto se concreta de forma distinta según el tipo de lenguaje OO:

- 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.
- Lenguajes OO híbridos, como Java o C++.- Las factorías no pueden considerarse generalizaciones de los llamados "constructores", pues estos son métodos especiales que, además de seguir reglas sintácticas diferentes al resto de los métodos, no permiten polimorfismo, debiéndose explicitar la clase concreta que se quiere crear.

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
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);
```

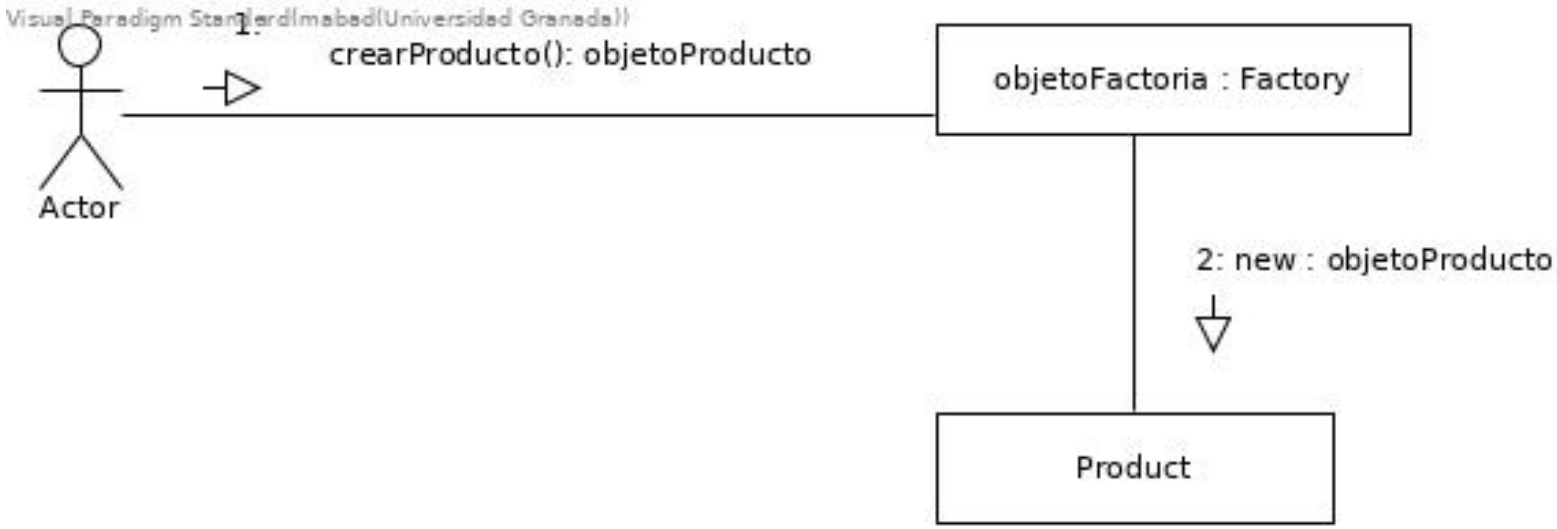


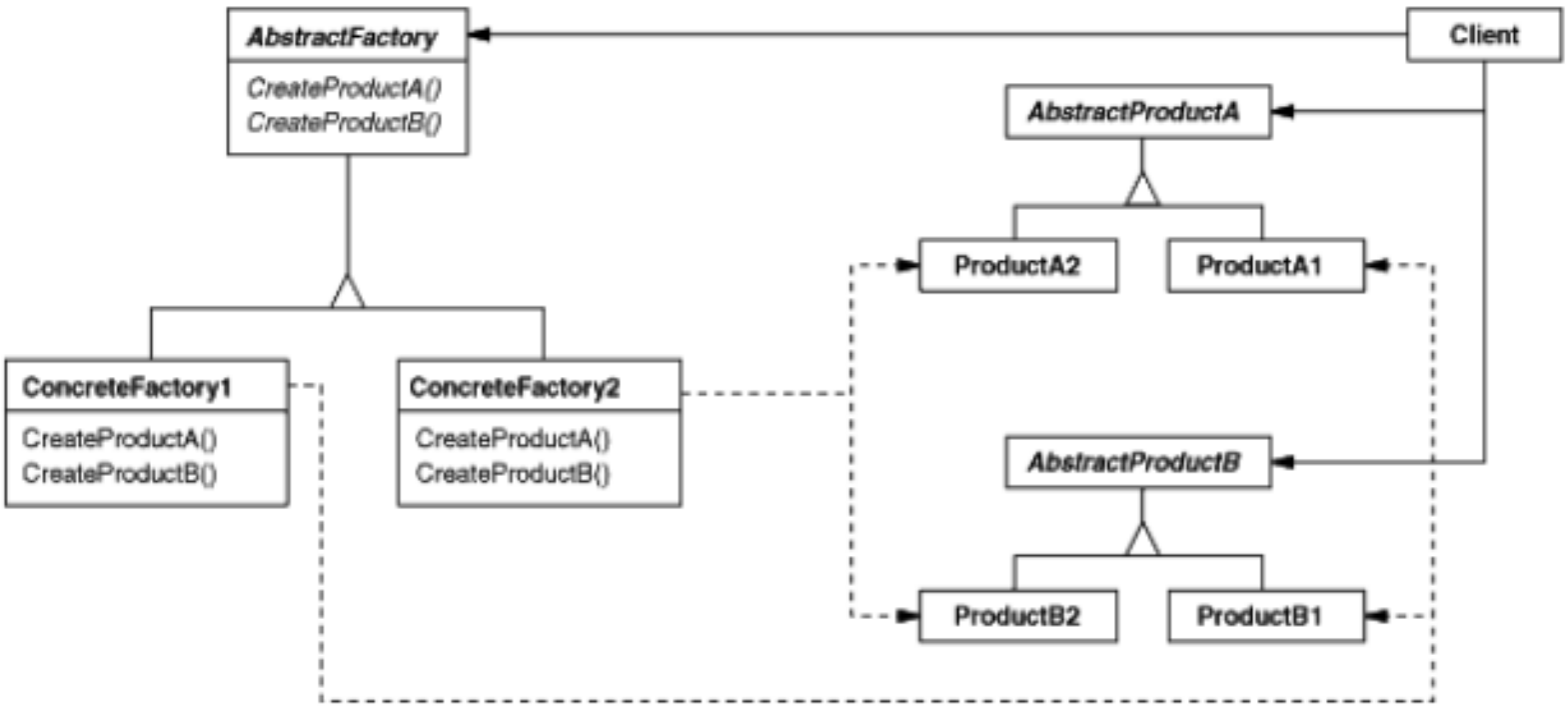
Figura 9: Diagrama de comunicación que muestra un ejemplo de creación de una instancia de la clase *Product* mediante el método factoría *crearProducto* de la clase *Factory*.

Cómo crear laberintos encantados sin cambiar el código

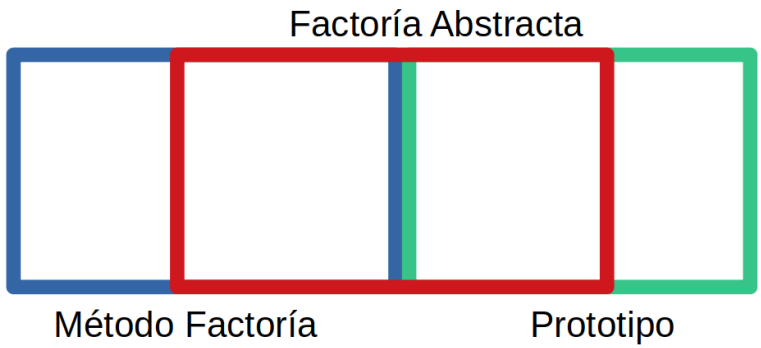
- *Método Factoría*.- El método *createMaze* llamará a métodos comunes ligados dinámicamente para crear las habitaciones, puertas y muros necesarios (métodos factoría). Añadiríamos una subclase de MazeGame, EnchantedMazeGame, que redefiniera esos métodos factoría (Figura 38).
- *Factoría Abstracta* con *Método Factoría*.-El método *createMaze* incluirá un parámetro (la *Factoría Abstracta*), para crear todas las habitaciones, puertas y muros necesarios, de forma que según la factoría, se crearán de un tipo u otro (Figura 33).
- *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, una factoría con prototipos (de muro, habitación y puerta) que podrían ser de distintos tipos de laberintos, creándose laberintos más variados (laberintos "mezcla") (Figura 41).
- *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 (Figura 11).

- Recomendado cuando en una aplicación tenemos líneas, temáticas o familias "paralelas" de clases y se prevé que puedan añadirse nuevas líneas, es decir las mismas clases pero con alguna variación.
- 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.
- NO está recomendado si se piensan agregar nuevas clases a líneas ya creadas.

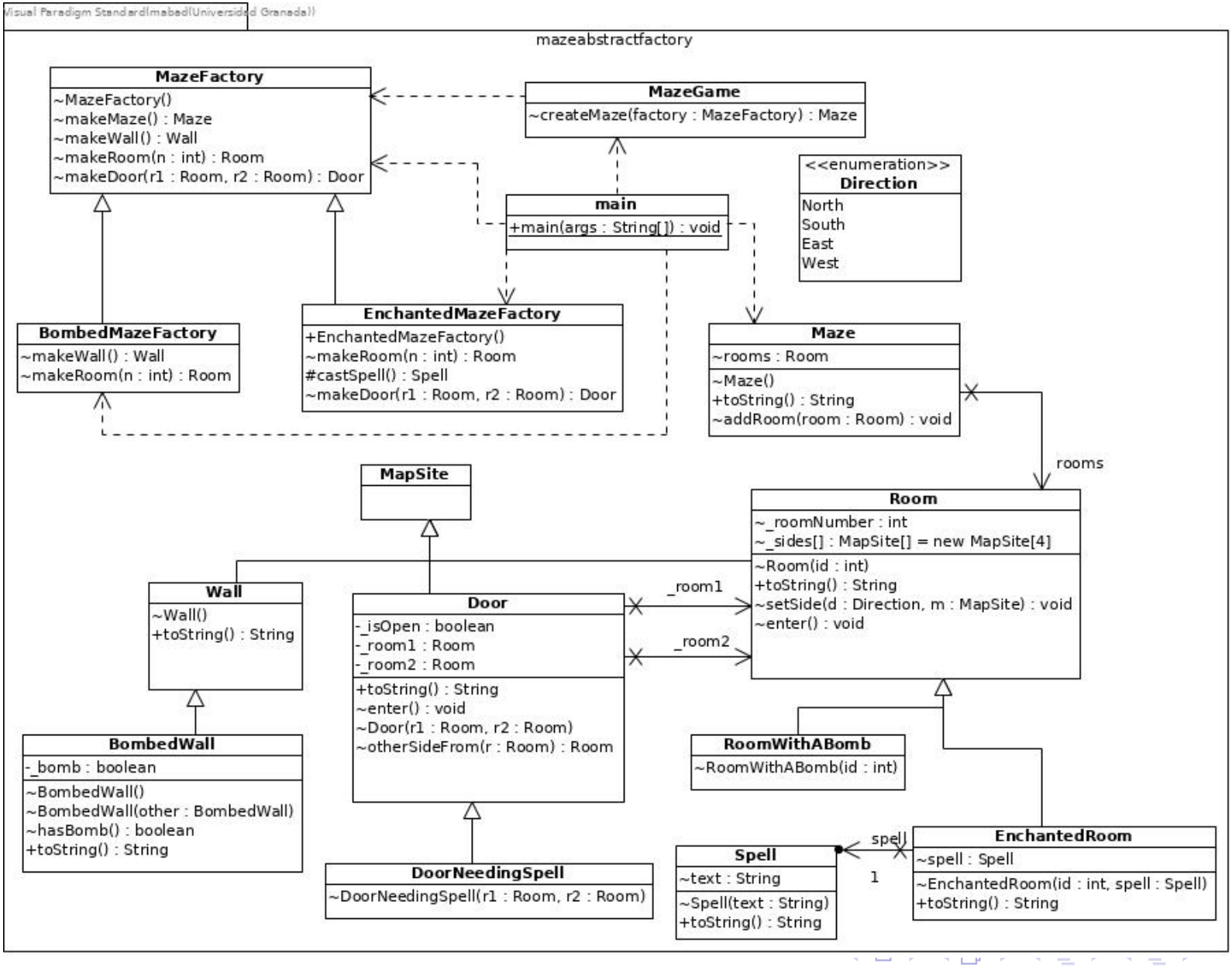
Patrón *Factoría Abstracta*



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).



Patrón / *Factoría Abstracta* / Ejemplo laberinto

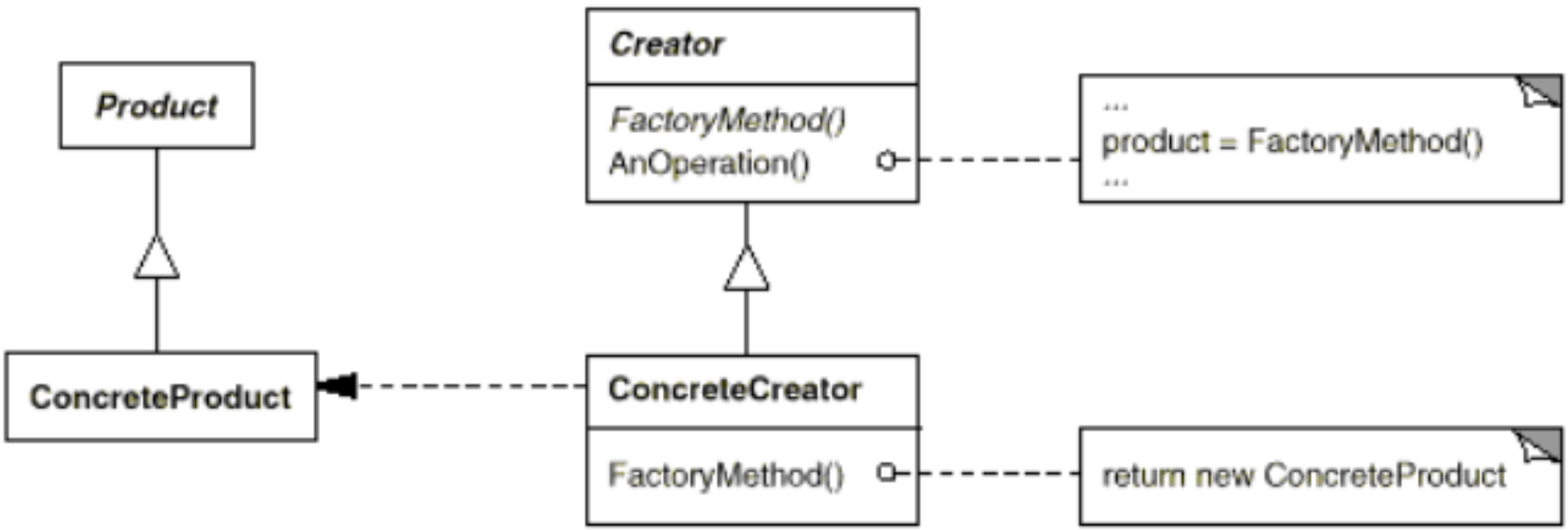


Un ejemplo en c++ (asumiendo ligadura dinámica –métodos virtuales–) de implementación del método *createMaze*, que es independiente de la factoría concreta usada:

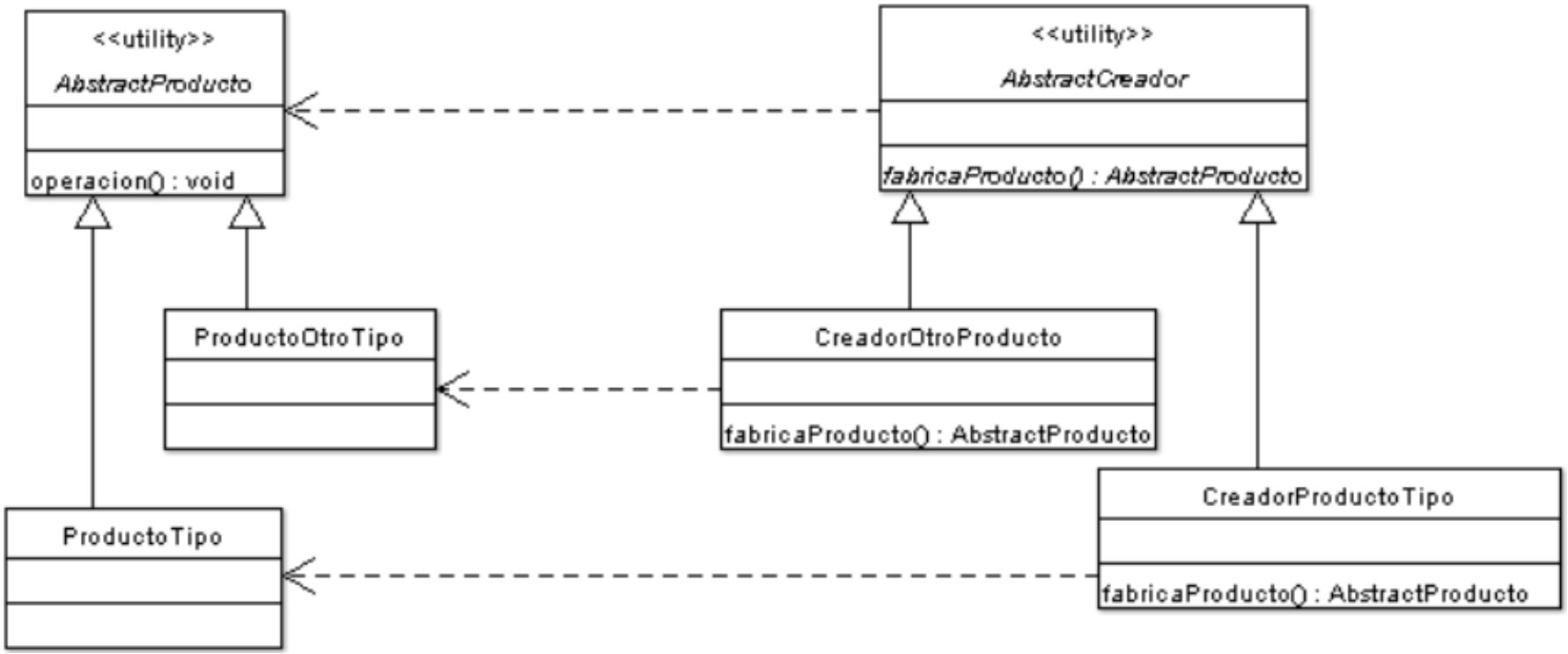
```
Maze* MazeGame::createMaze (MazeFactory& factory) {
Maze* aMaze = factory.makeMaze();
Room* r1 = factory.makeRoom(1);
Room* r2 = factory.makeRoom(2);
Door* aDoor = factory.makeDoor(r1, r2);
aMaze->addRoom(r1);
aMaze->addRoom(r2);
r1->setSide(North, factory.makeWall());
r1->setSide(East, aDoor);
r1->setSide(South,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

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

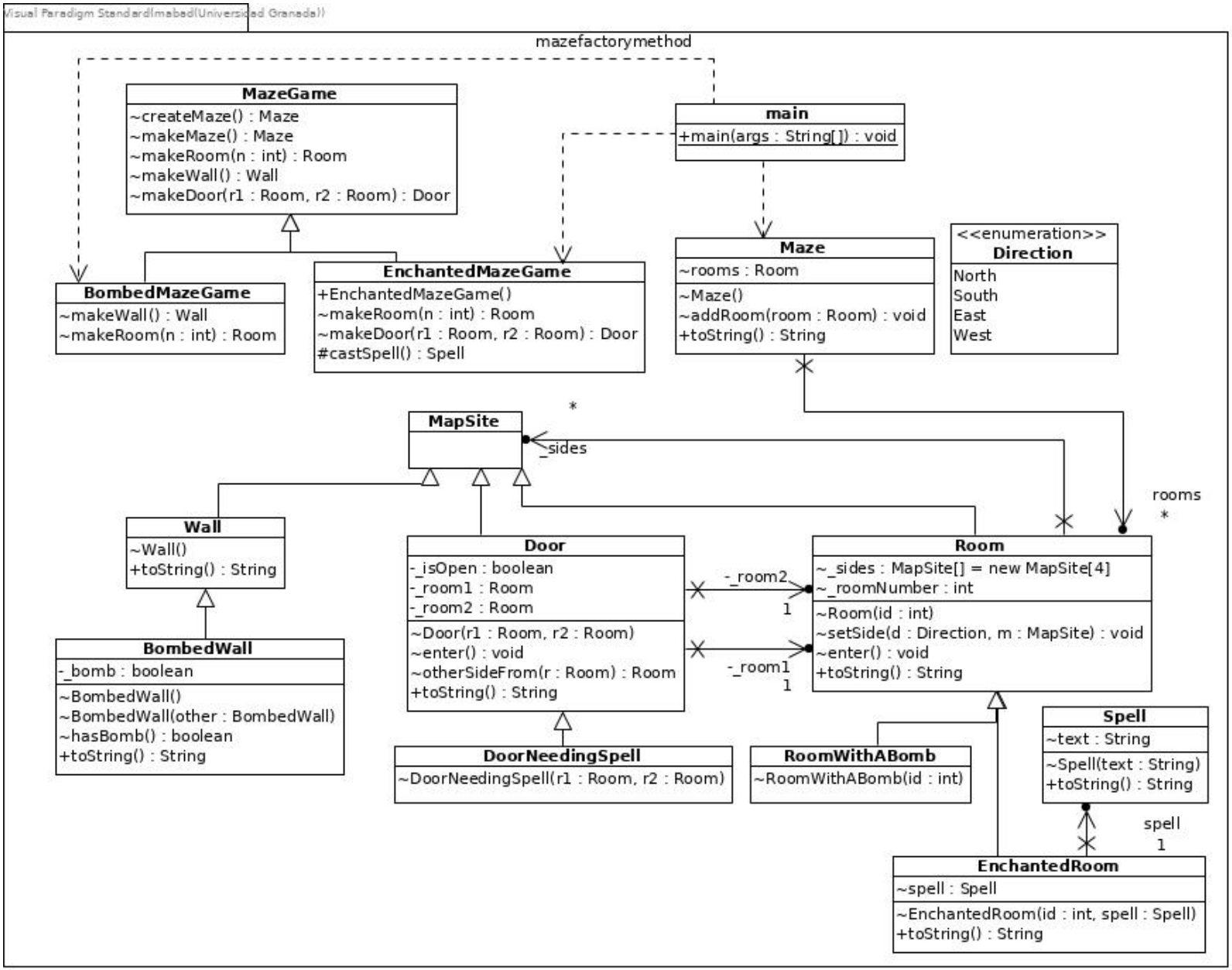
Patrón *Método Factoría* (una sólo interfaz –signatura– del método factoría)



Patrón *Método Factoría*. Otro diagrama de clases (más complejo) del patrón Método Factoría



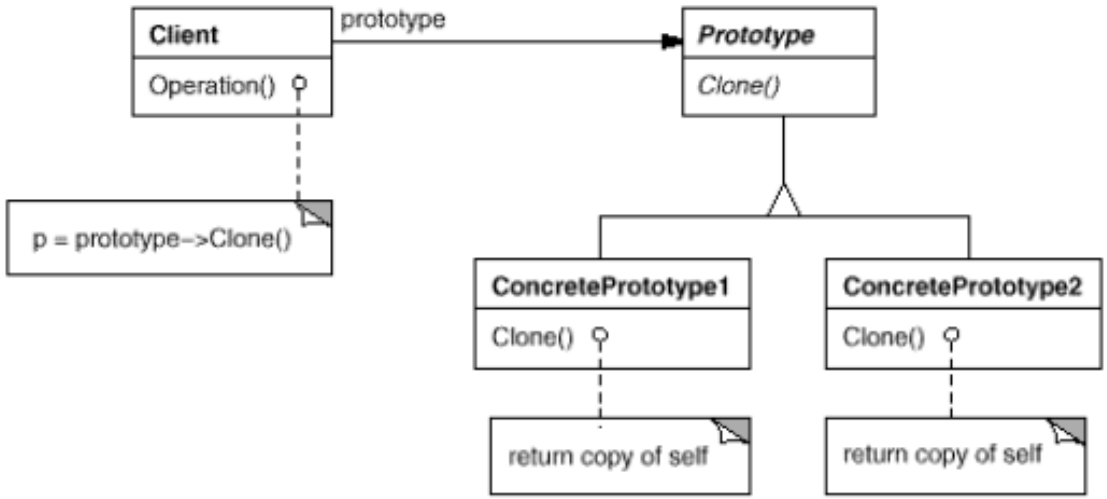
Patrón *Método Factoría* Ejemplo laberinto



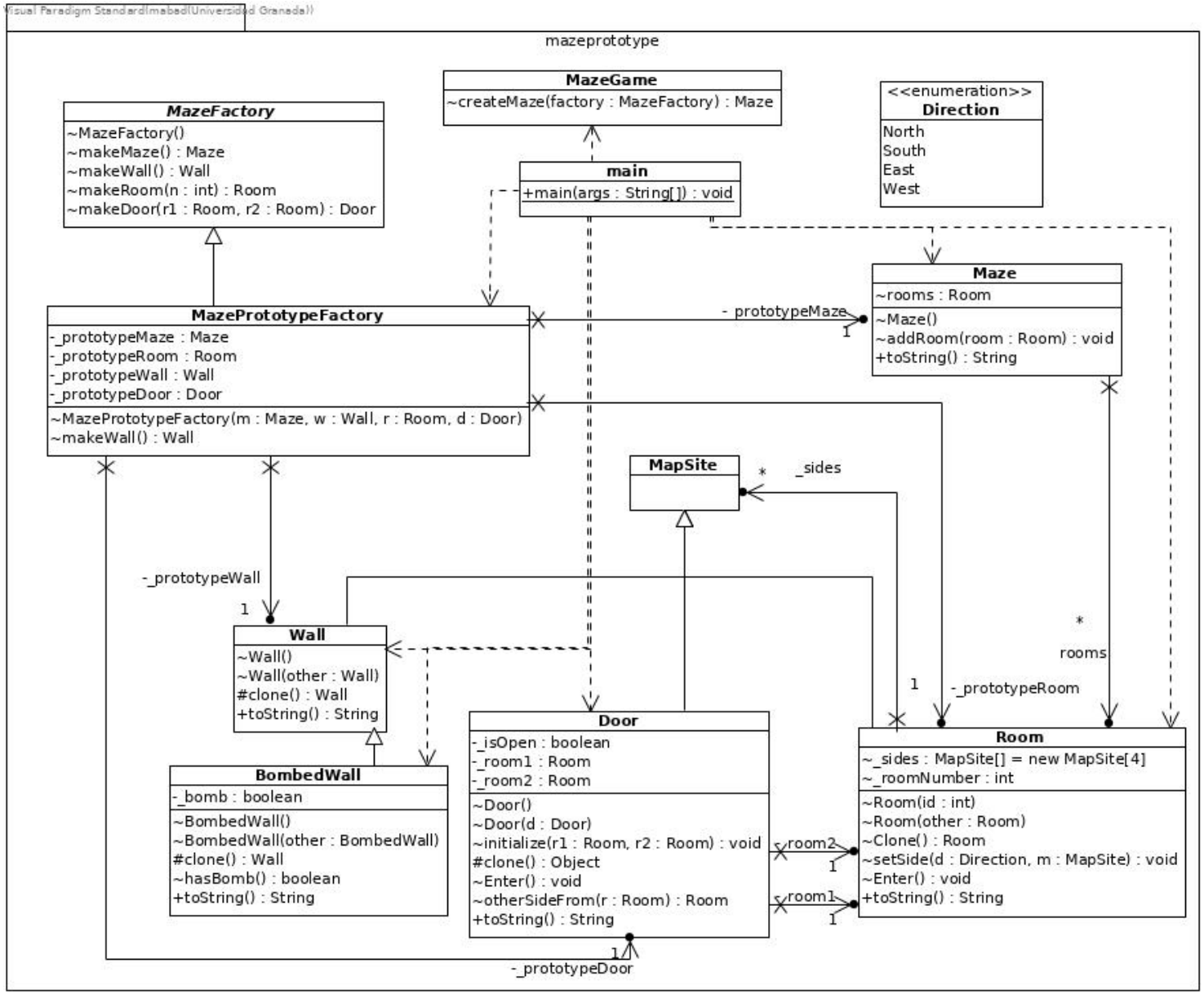
Un ejemplo en c++ (asumiendo ligadura dinámica –métodos virtuales–) de implementación del método *createMaze* con el patrón *Método Factoría*:

```
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;
}
```

- Cuando se usan métodos factoría que crean por delegación, i.e. usando prototipos o métodos de clonación
- Flexibilidad: No tenemos que crear la jerarquía de clases factoría sino que basta con una sola clase factoría que tenga un sólo método para crear cada objeto dentro de una jerarquía (método *operation* en la Figura 40), que considera que las clases de todos los objetos que se quieren reutilizar en la aplicación heredan todas de la clase *Prototype*



Patrón *Prototipo* Ejemplo laberinto



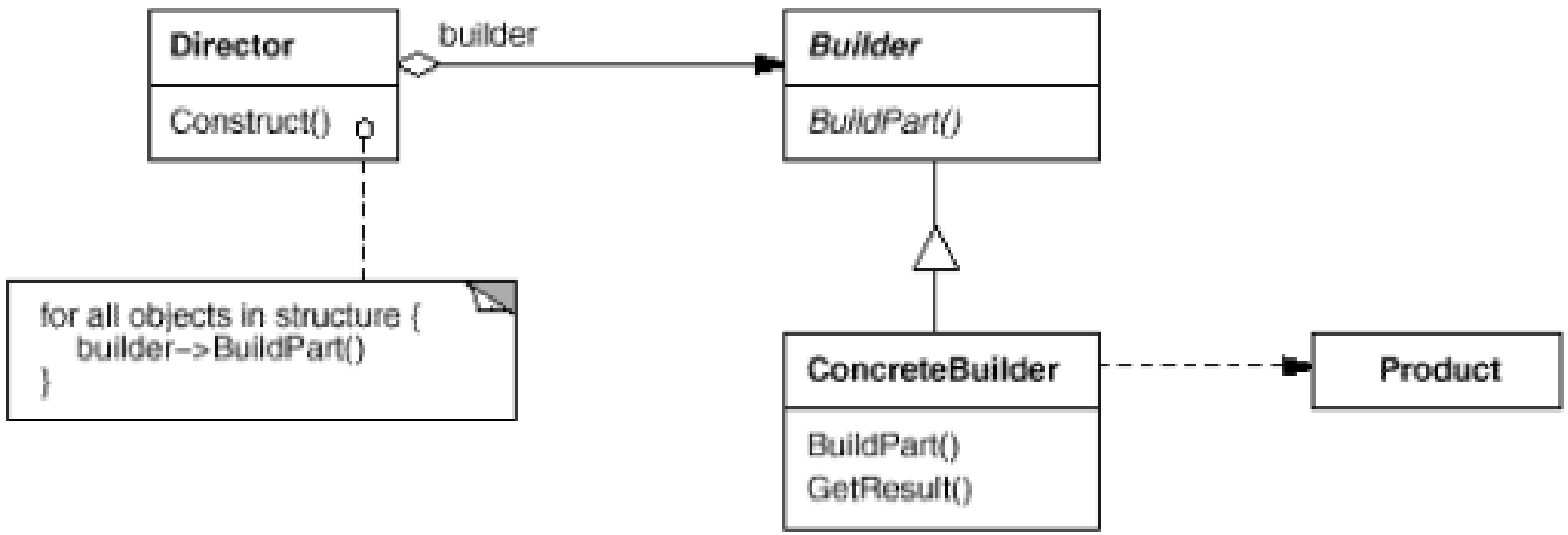
La implementación en c++ del método *createMaze* para el patrón *Prototype* junto con el de *Abstract Factory* sería la misma que cuando se usa la factoría abstracta con el patrón *Método Factoría*.
Ejemplo en c++ de implementación de la creación de dos factorías distintas de prototipos:

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

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

- Cuando queremos crear un objeto de cierta complejidad formado por componentes que puedan cambiar, así como ensamblarse y representarse de distintas formas, siendo común el algoritmo que se necesita para construir las partes y ensamblarlas.
- Cada objeto concreto builder, compone y ensambla de forma distinta, pero todos comparten los métodos, llamados por un 'director de montaje".

Patrón *Builder*



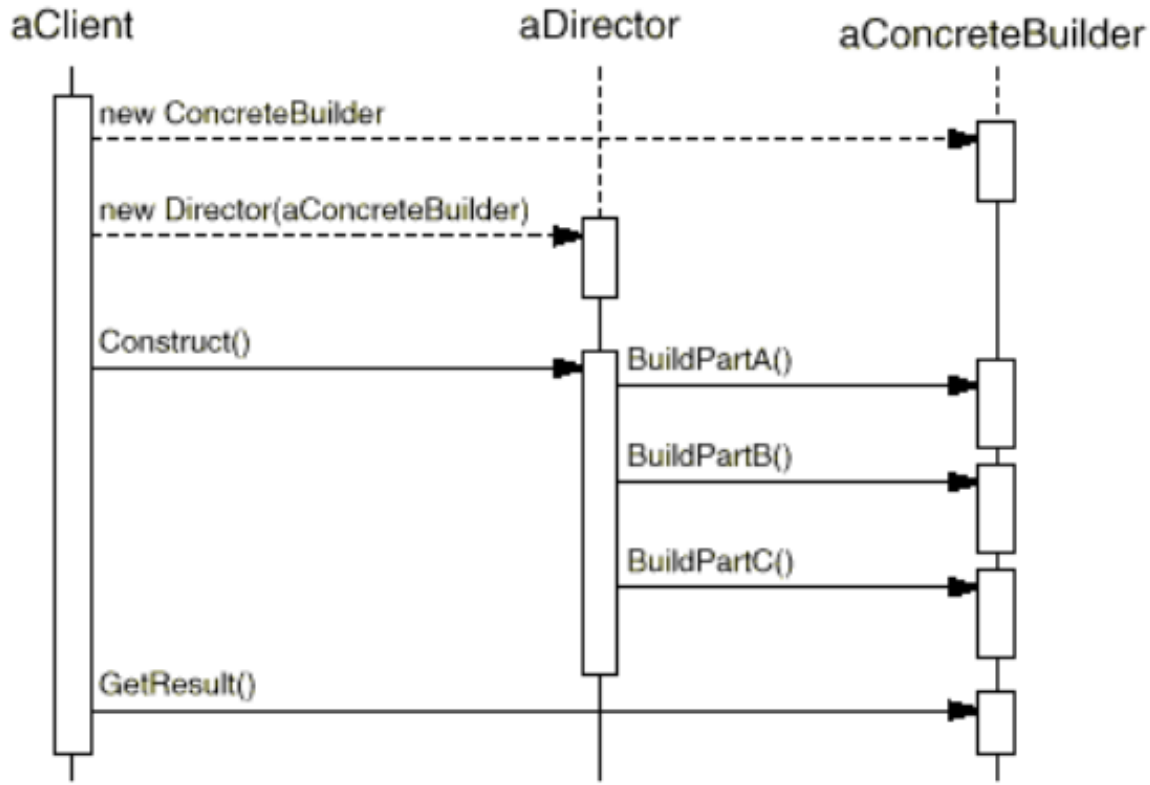


Figura 10: Estructura del patrón Builder [Fuente: (Gamma et al., 1994b), p. 113]

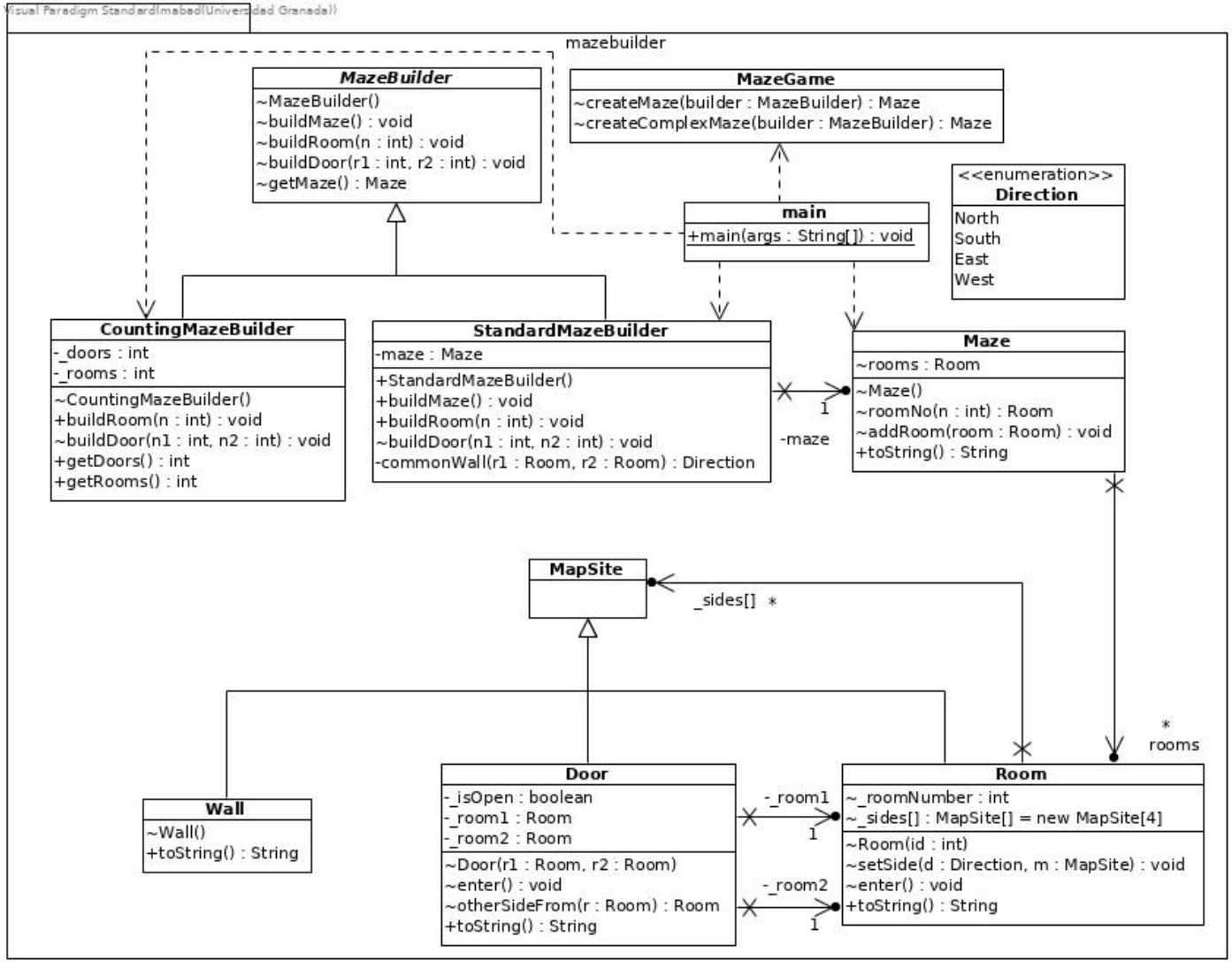


Figura 11: Diagrama de clases correspondiente a la aplicación del patrón "Builder" en el ejemplo del juego del laberinto de GoF (Gamma et al., 1994a).

Ejemplo en c++ (asumiendo ligadura dinámica) de implementación del método *createMaze* con el patrón *Builder*:

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

Cómo construir otro tipo de laberinto, por ejemplo un laberinto complejo:

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

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

CRITERIO DE CALIDAD: Bajo acoplamiento

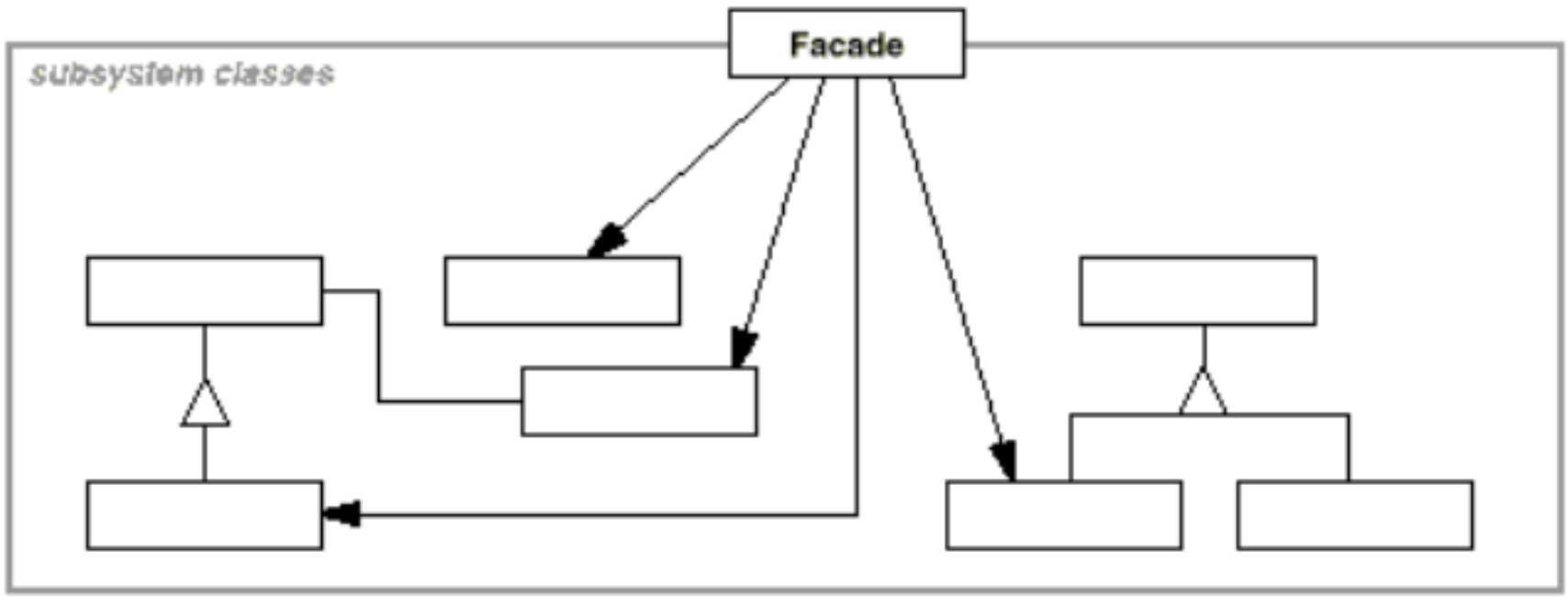
El bajo acoplamiento en diseño OO se refiere a que haya pocas dependencias entre clases pertenecientes a subsistemas distintos.

CRITERIO DE CALIDAD: Alta cohesión

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

Patrón *Fachada*

Proporciona una interfaz única de un subsistema, reduciendo acoplamiento



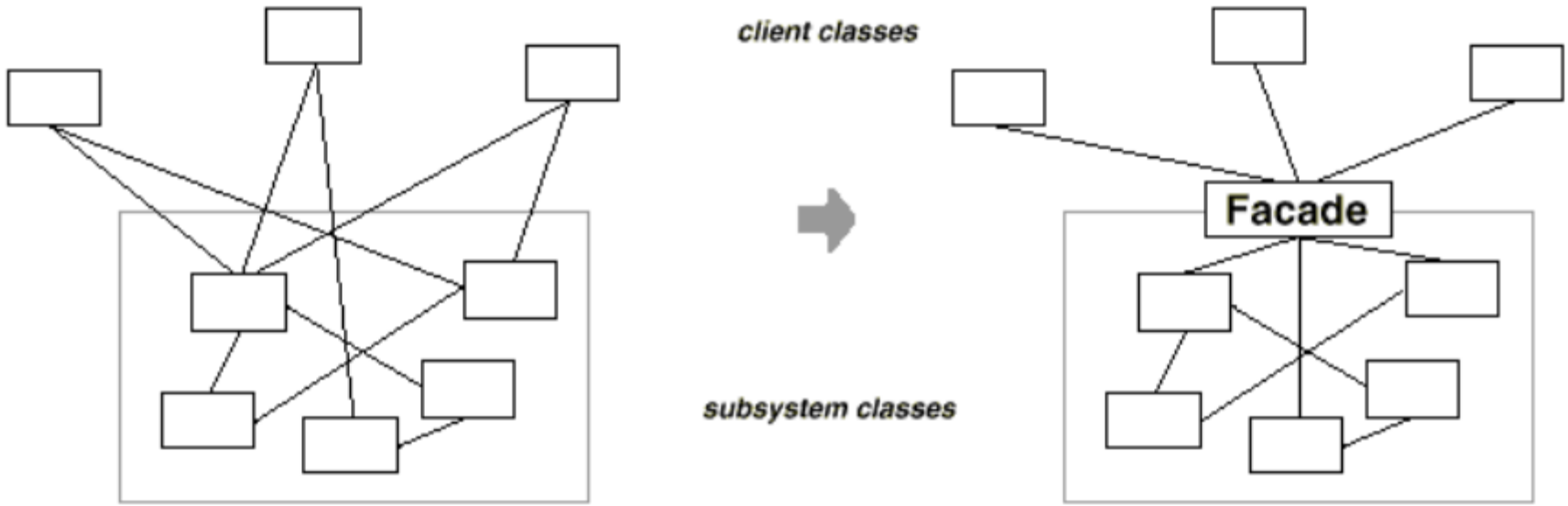


Figura 12: Un esquema de diagrama de clases antes y después de la aplicación del patrón Fachada [Fuente: (Gamma et al., 1994b, pg. 208)].

Patrón *Composición*

Qué hace: Permite tratar a objetos compuestos y simples de la misma forma, mediante una interfaz común a todos, que define un objeto compuesto de forma recursiva.
Cuándo usarlo: cuando el cliente o usuario de esos objetos no quiera distinguir si son compuestos o simples.

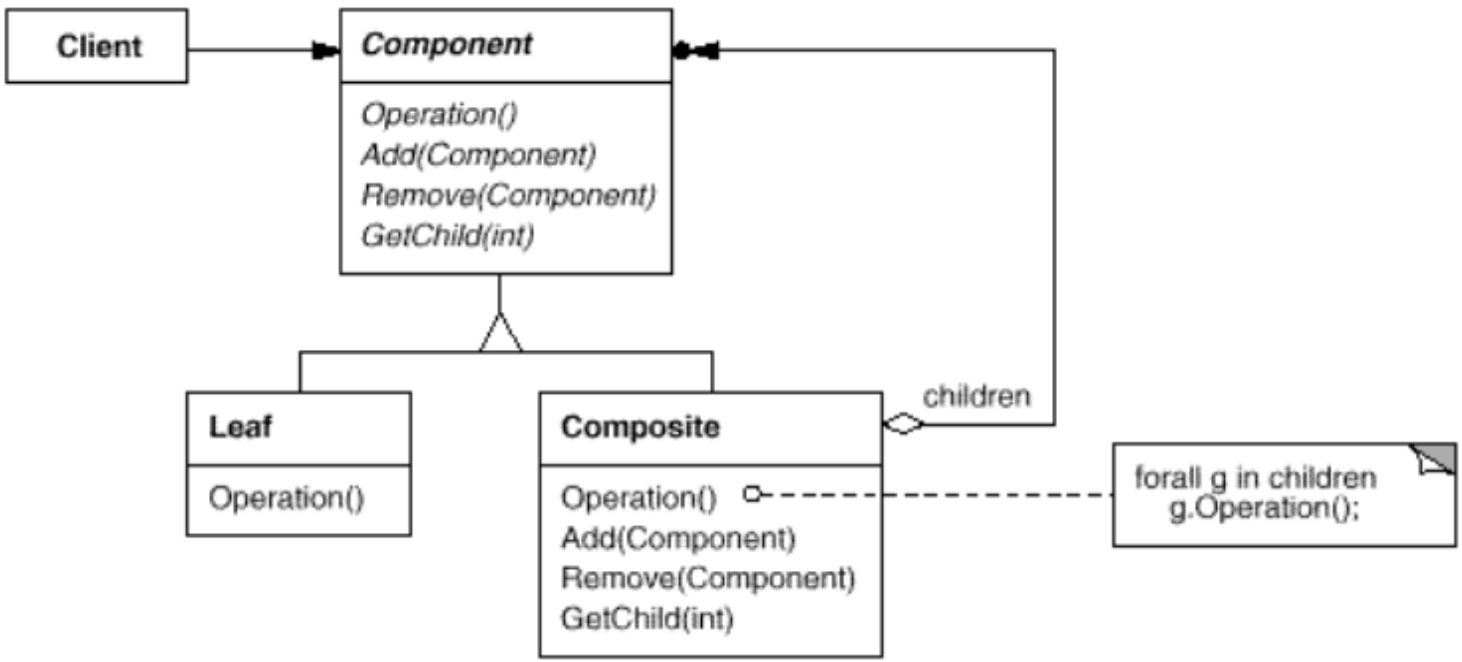


Figura 13: Estructura del patrón Composición [Fuente: (Gamma et al., 1994b), p. 185].

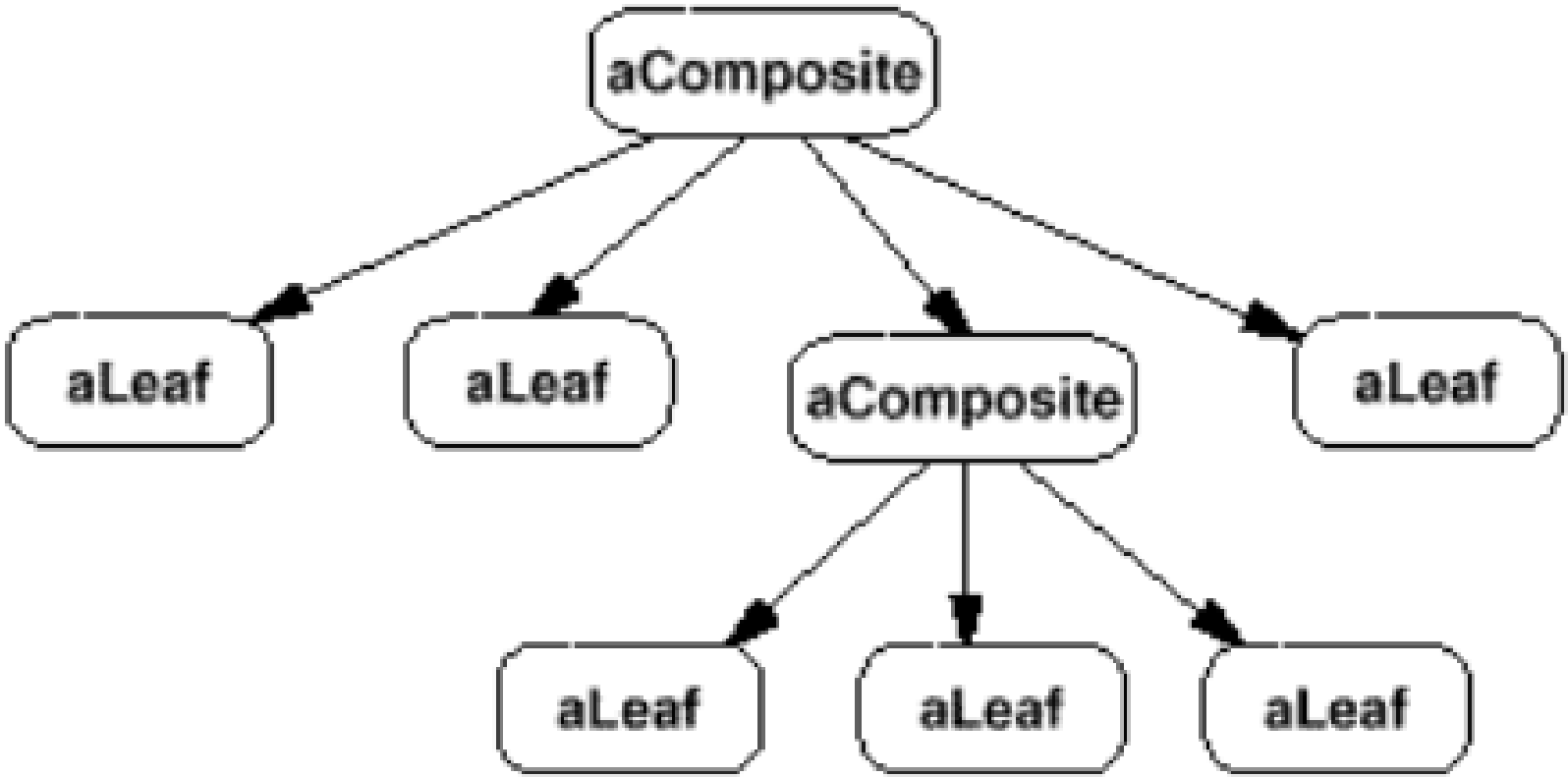


Figura 14: Un ejemplo de jerarquía de objetos donde se ve la relación entre objetos compuestos y simples [Fuente: (Gamma et al., 1994b), p. 185].

Patrón *Composición*

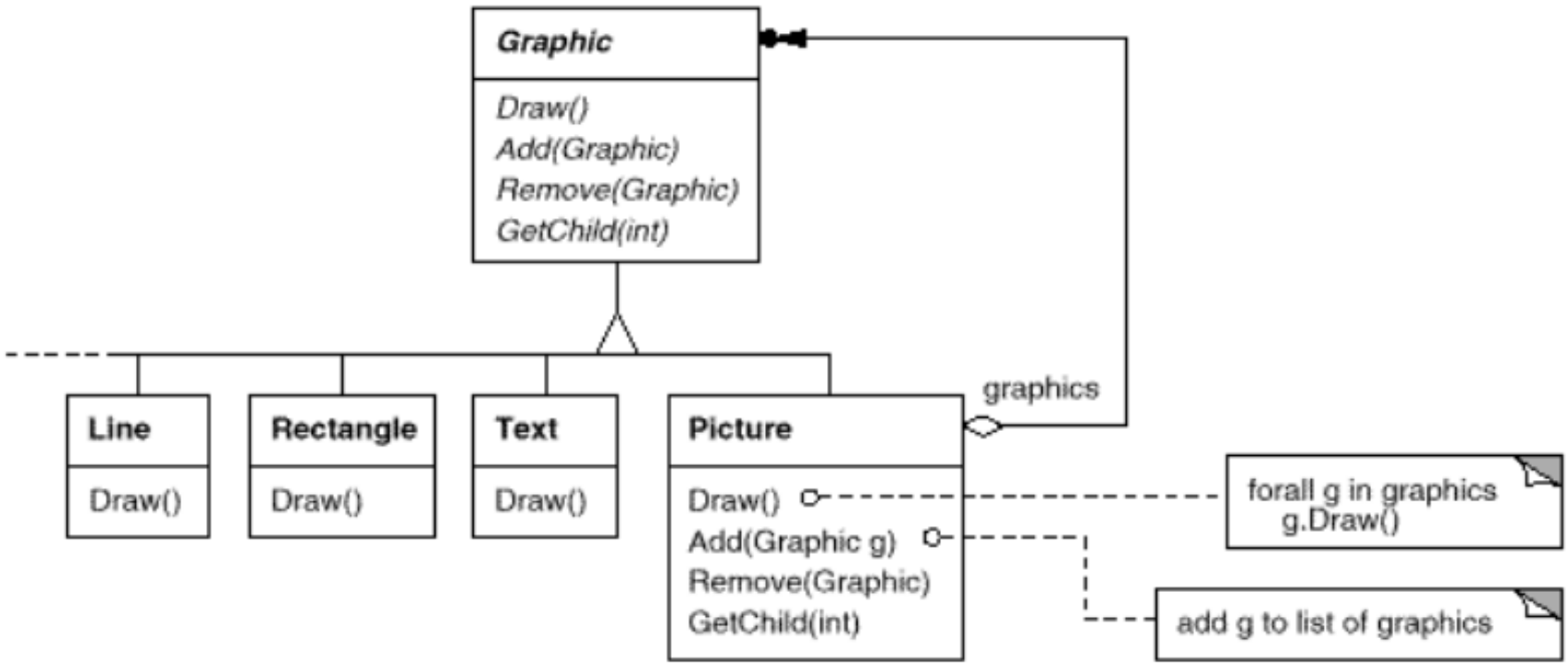


Figura 15: Estructura del patrón Composición en el ejemplo de componentes gráficos. [Fuente: (Gamma et al., 1994b), p. 183].

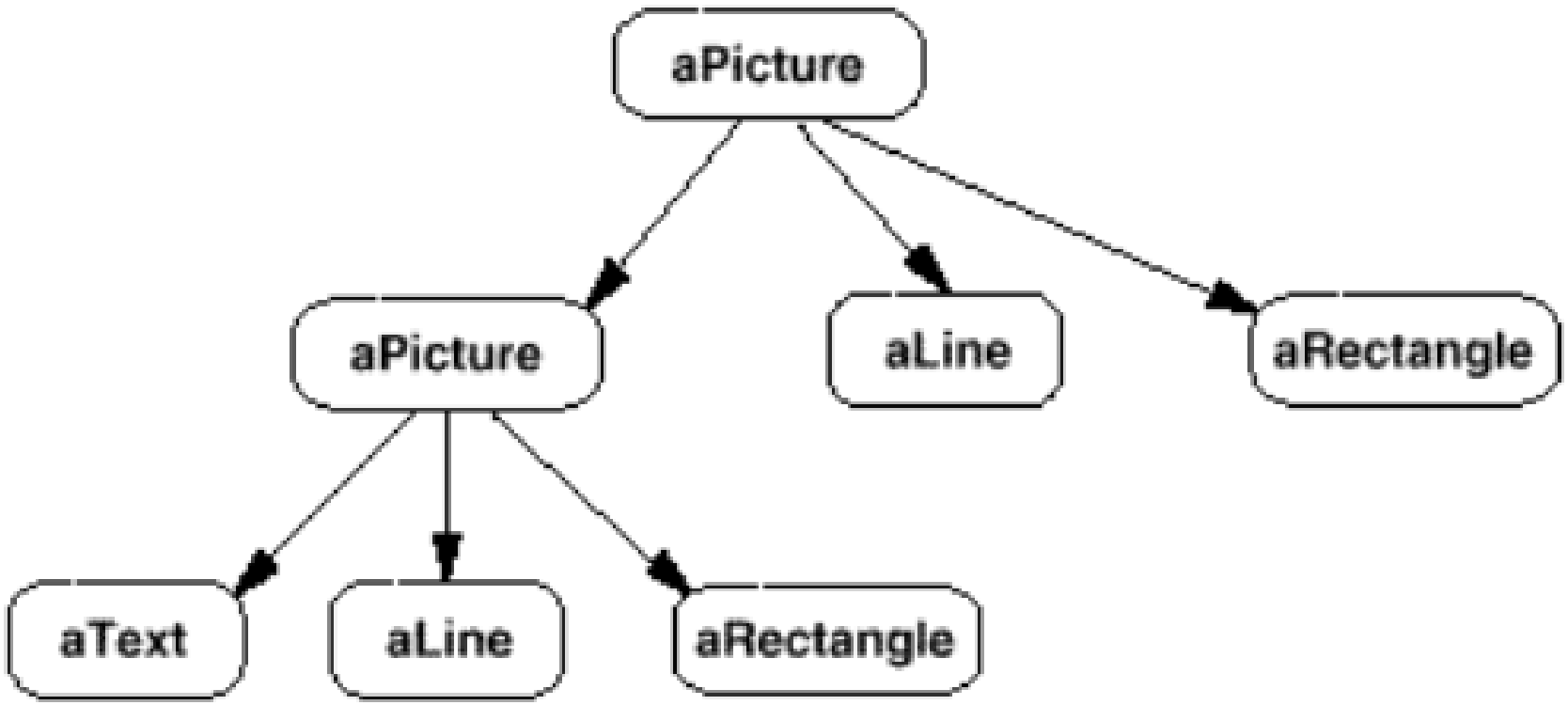


Figura 16: Una de jerarquía de objetos en el ejemplo de componentes gráficos [Fuente: (Gamma et al., 1994b), p. 184].

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

Patrón *Decorador* (o *Envoltorio –Wrapper–*)

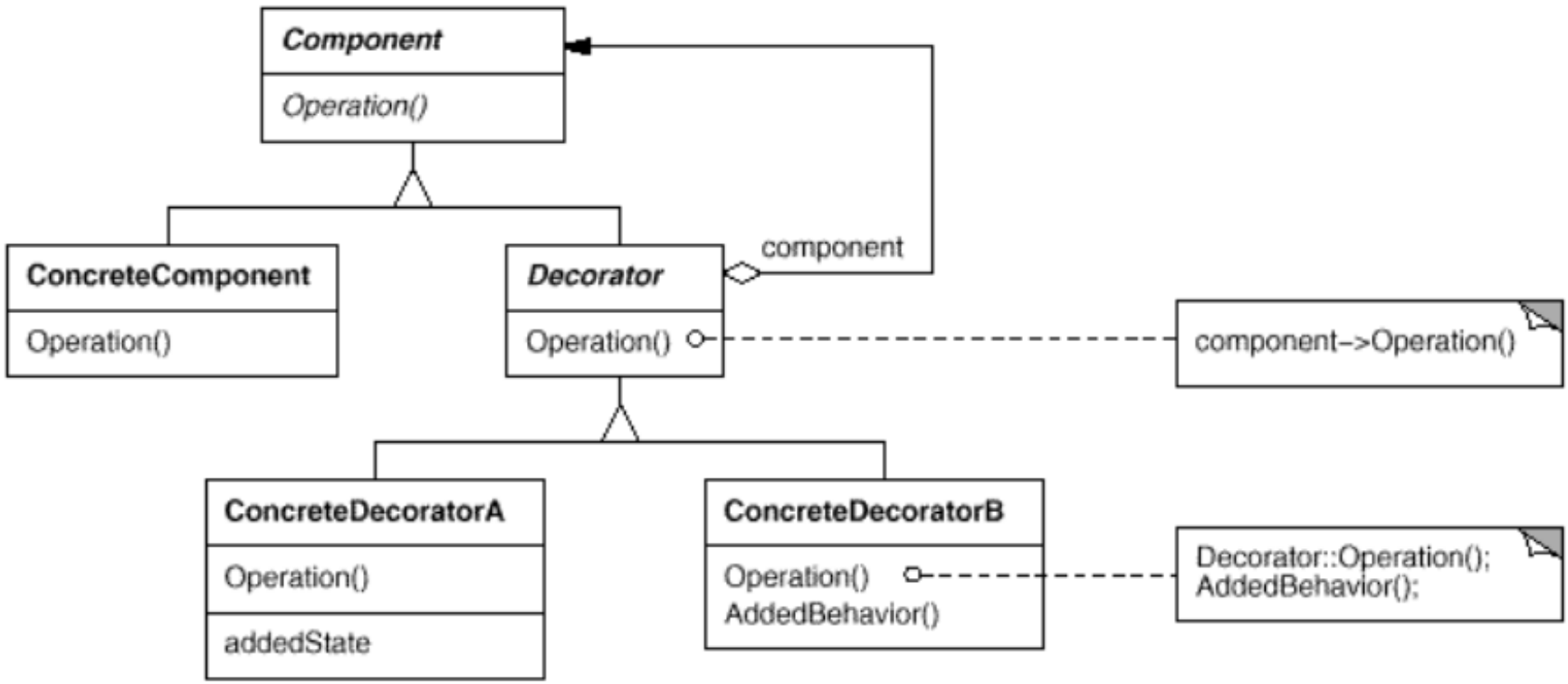


Figura 17: Estructura del patrón decorator [Fuente: (Gamma et al., 1994b), p. 199].

Patrón *Decorador* (o *Envoltorio –Wrapper–*)

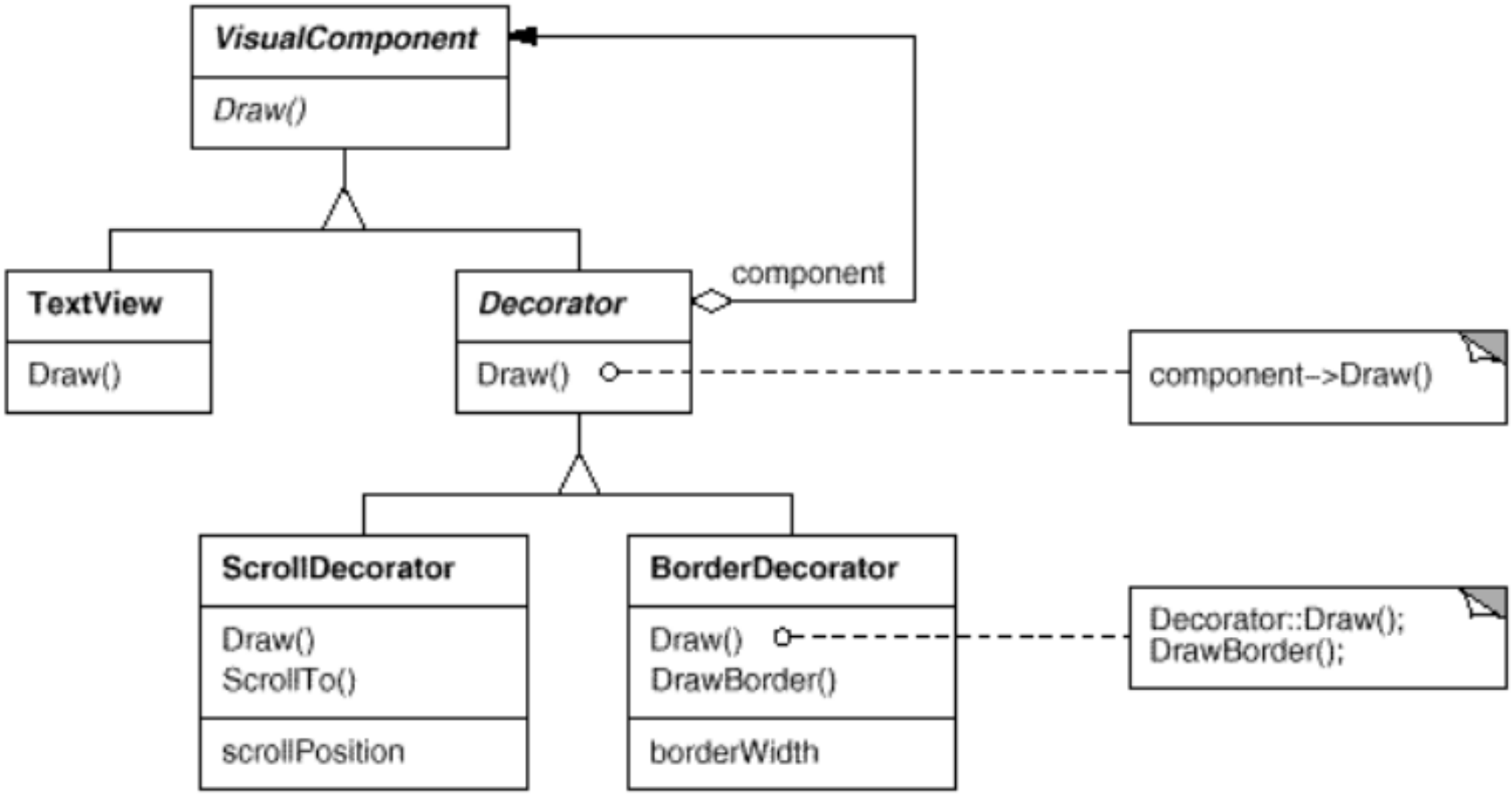


Figura 18: Ejemplo de estructura del patrón *decorator* aplicada a la presentación gráfica de un texto [Fuente: (Gamma et al., 1994b), p. 198].

Patrón *Decorador* (o *Envoltorio –Wrapper–*)

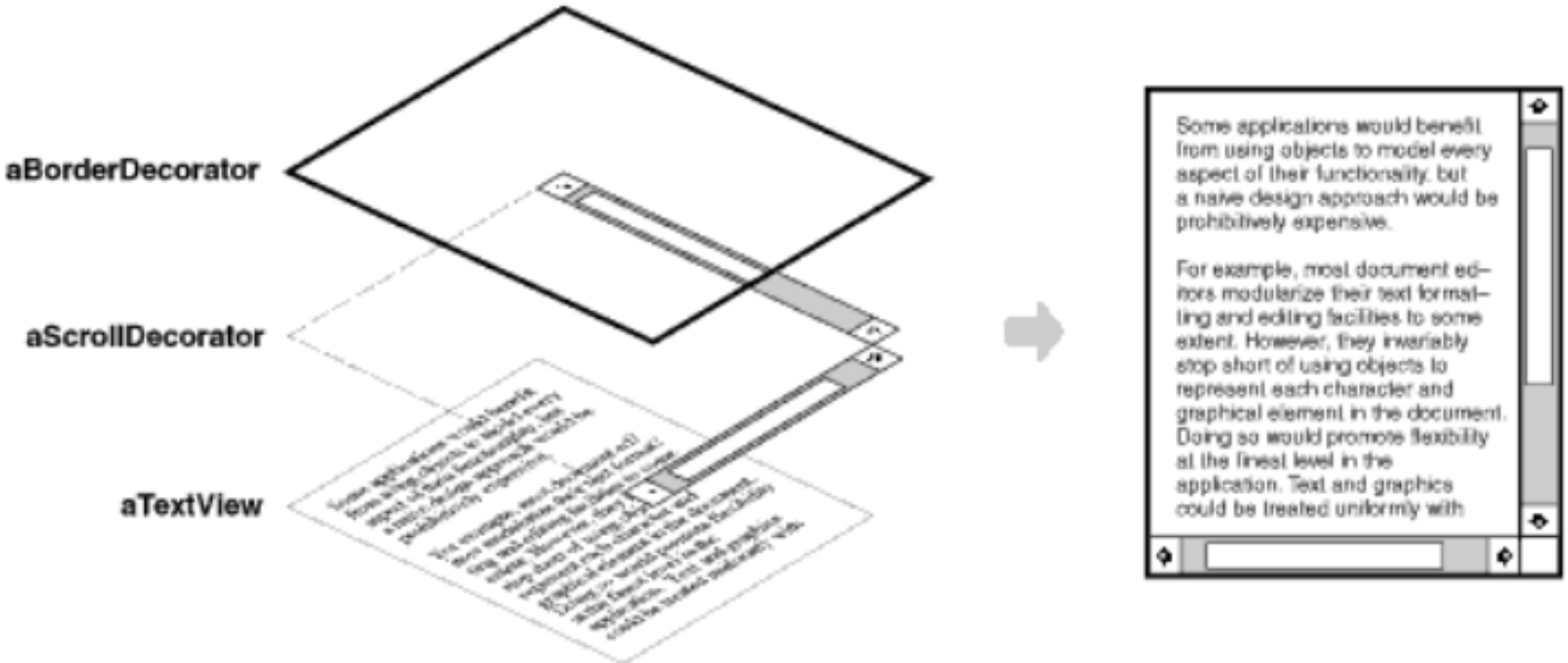


Figura 19: Aspecto de un cuadro de texto y sus decoradores [Fuente: (Gamma et al., 1994b, 197)].

Clase UserBasicDecorator:

```
class UserBasicDecorator < SimpleDelegator

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

Clase UserDecorator:

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

Sin patrón decorador:

```
<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 %>
```

Con decorador simple:

```
<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 %>
```

Con decorador:

```
<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 %>
```

Relación con el patrón *Composición*:

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

Pueden coexistir

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

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

Versiones:

- Patrón *Adaptador* en el ámbito de clases
- Patrón *Adaptador* en el ámbito de objetos

Requiere del uso de herencia múltiple

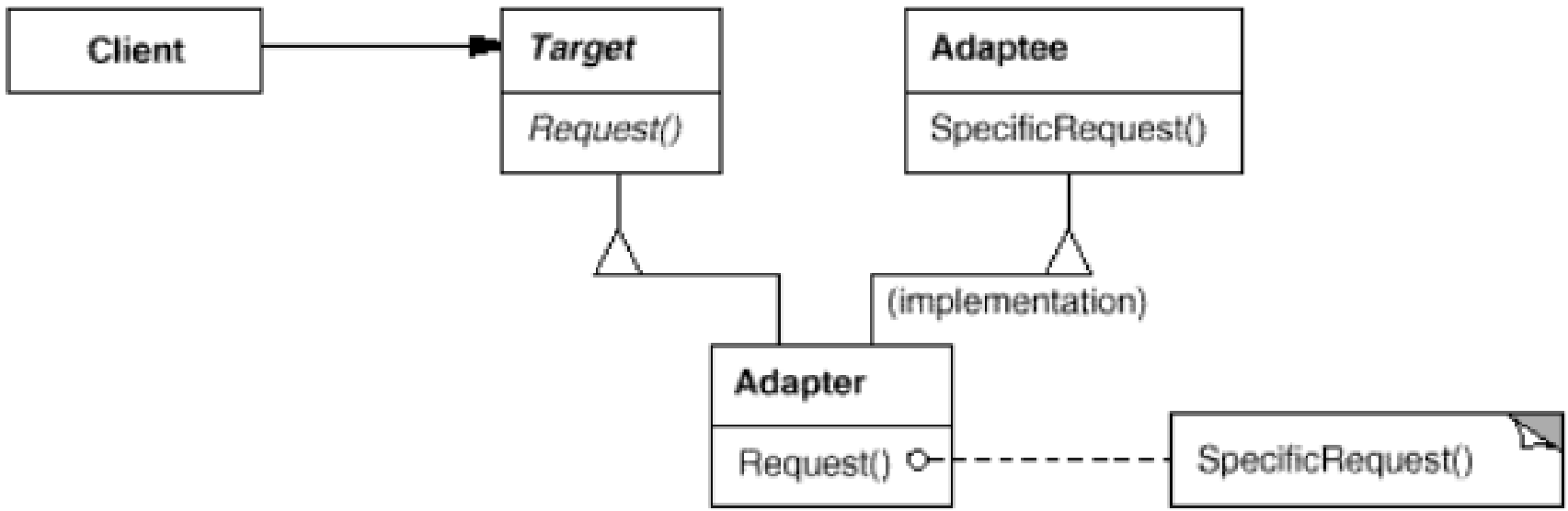


Figura 20: Estructura del patrón adaptador, en un ámbito de clase [Fuente: (Gamma et al., 1994b, pg. 159)].

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

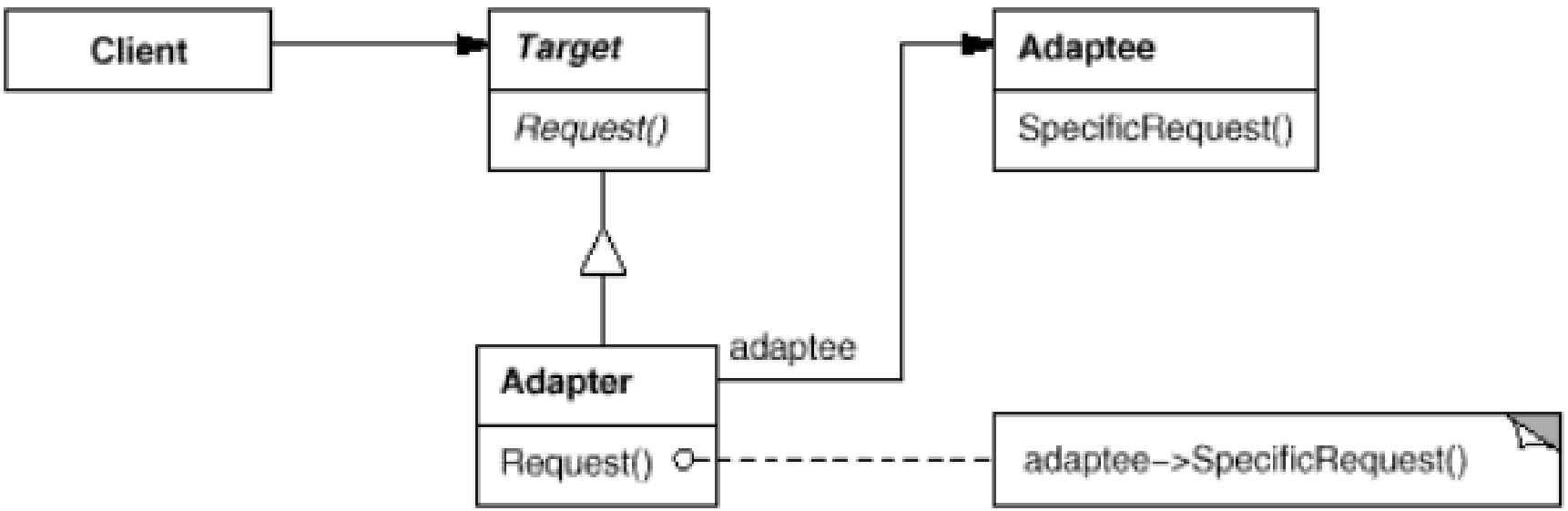


Figura 21: Estructura del patrón adaptador en un ámbito de objeto [Fuente: (Gamma et al., 1994b, pg. 159)].

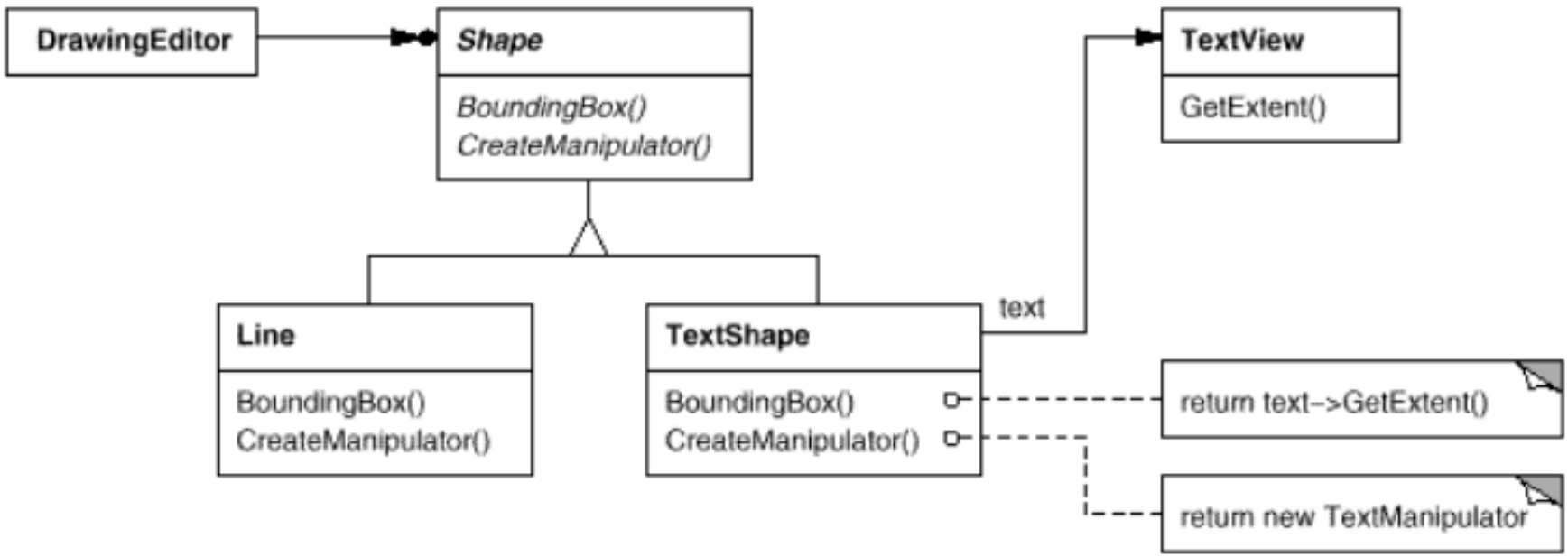
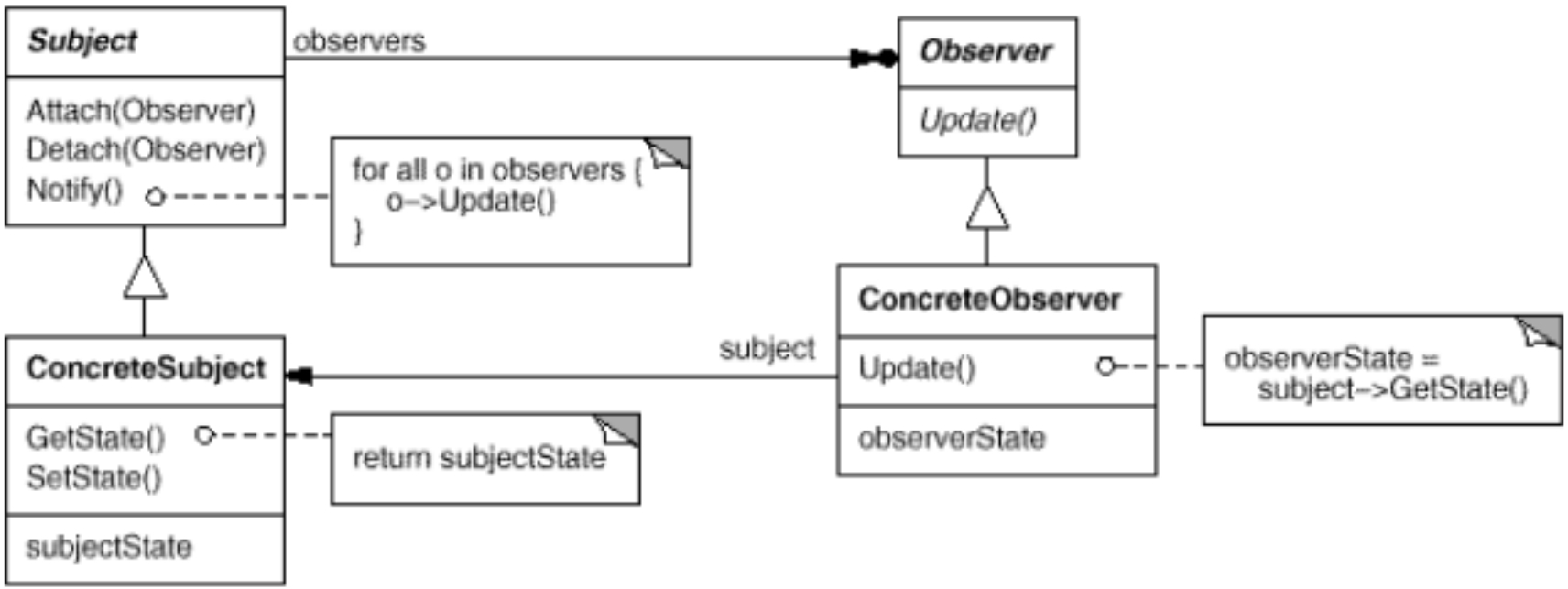


Figura 22: Estructura del patrón "Adapter" aplicado a un editor de dibujo [Fuente: (Gamma et al., 1994b, pg. 158)].

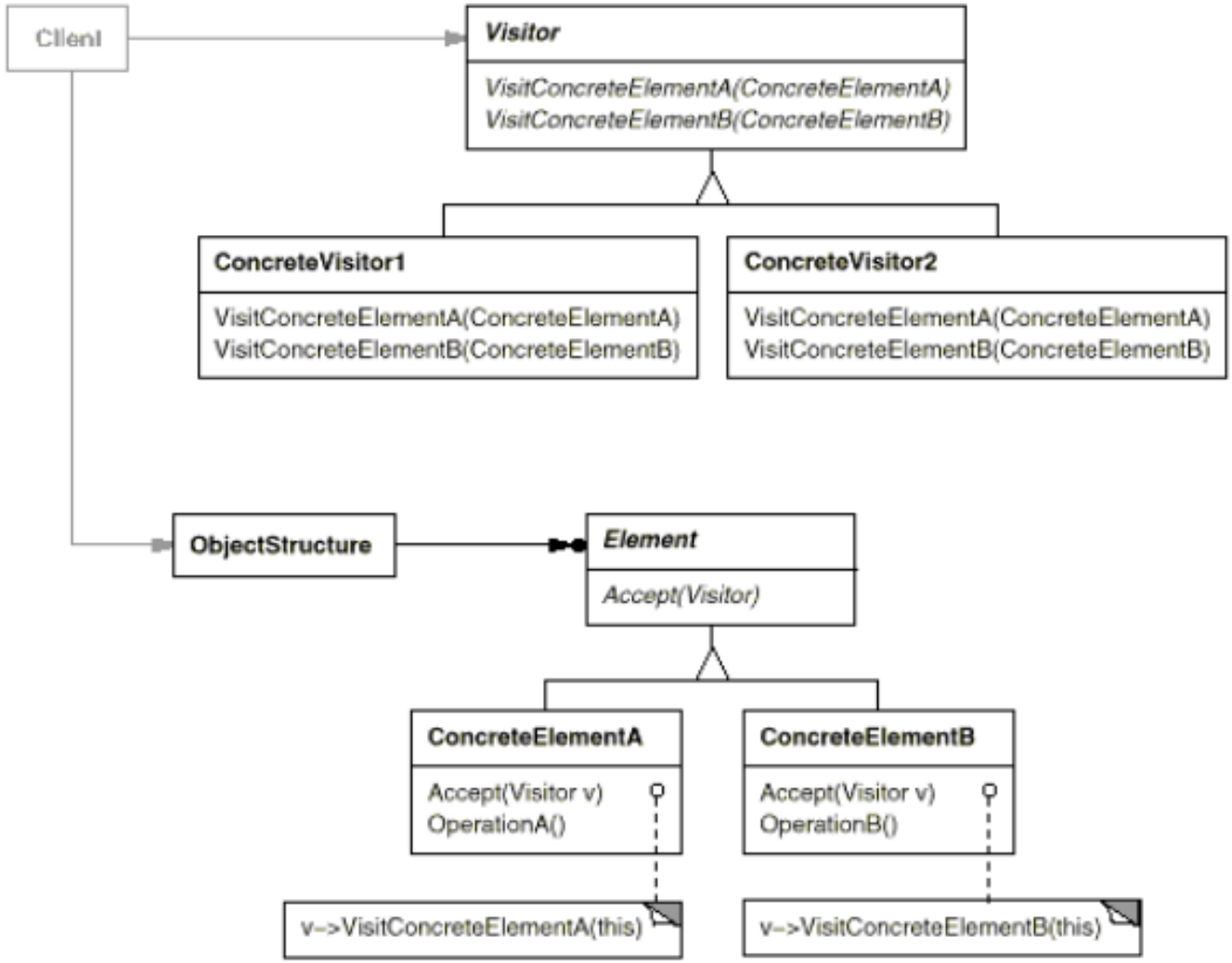
- Otros nombres *Dependientes* o *Publicar-Subscribir*
- Cuando un objeto ("sujeto observable") tiene varios objetos que dependen de él, garantiza la consistencia entre ellos manteniendo un bajo acoplamiento
- Funcionamiento:
  - No hay comunicación directa entre el sujeto observable y sus observadores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

Patrón *Observador*



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

Patrón *Visitante*



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

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

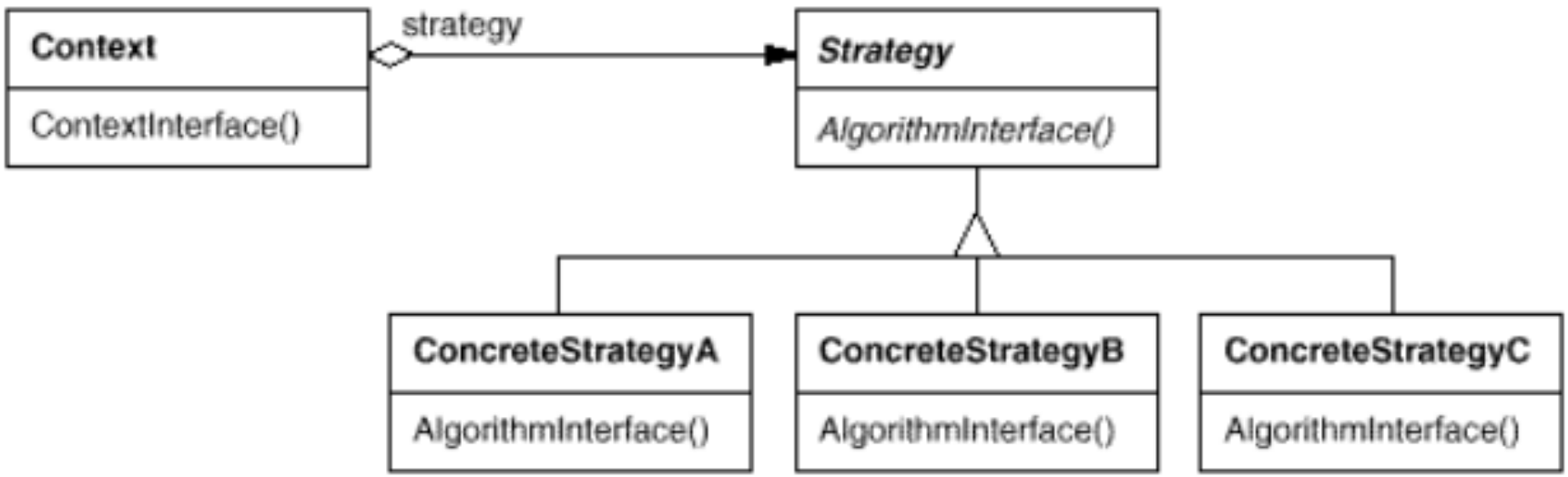


Figura 23: Estructura del patrón *Estrategia* [Fuente: (Gamma et al., 1994b, pg. 351)]

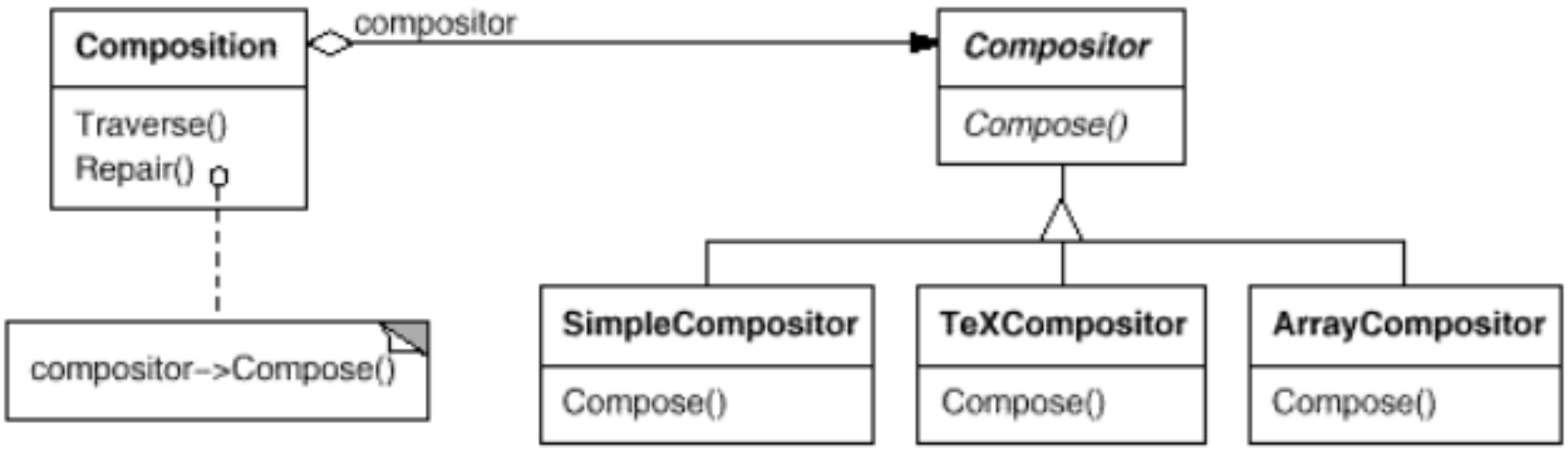


Figura 24: Estructura del patrón *Estrategia* en un ejemplo de visualización de textos [Fuente: (Gamma et al., 1994b, pg. 349)]

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

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

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

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

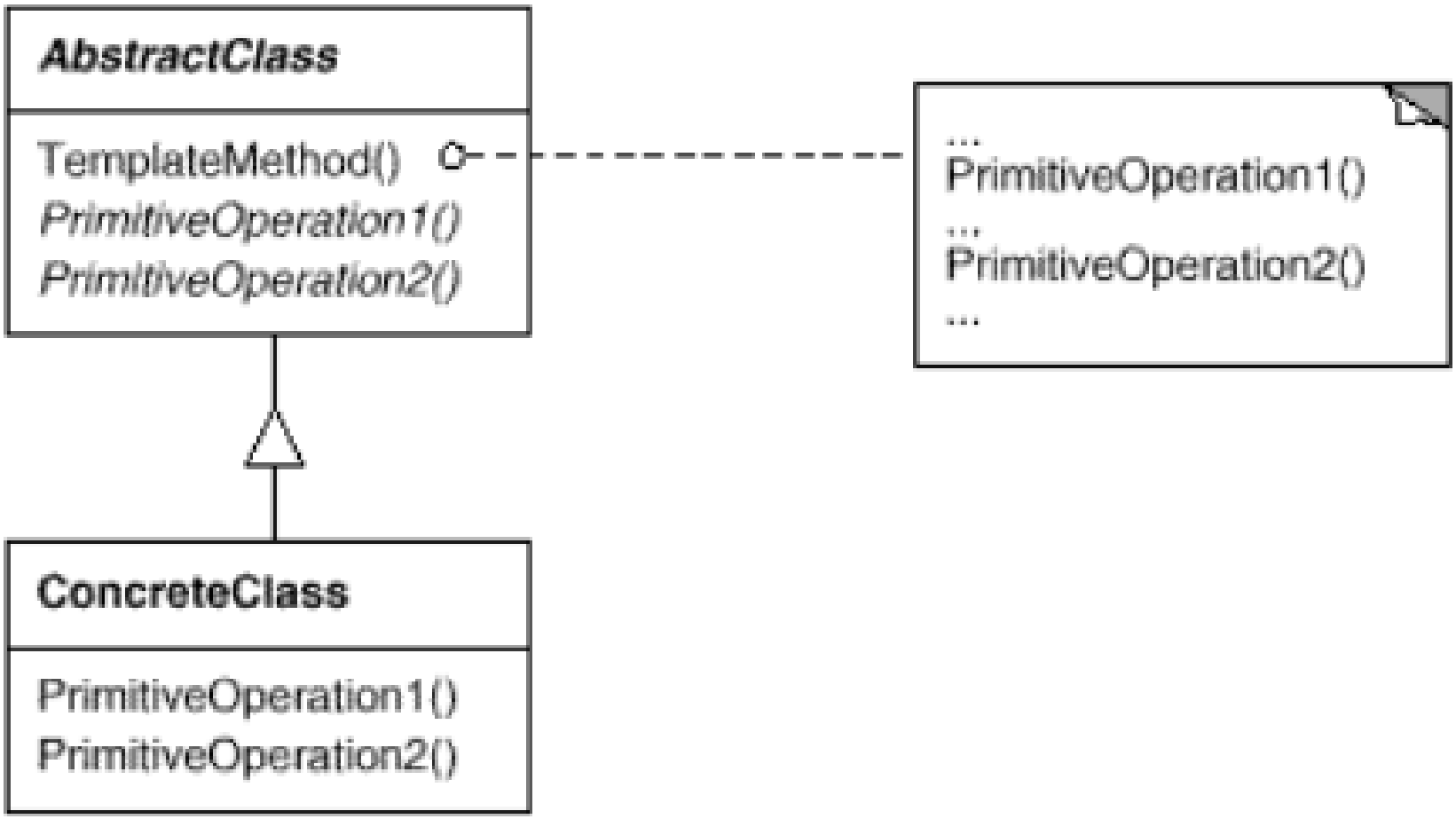


Figura 25: Estructura del patrón "Método Plantilla" [Fuente: (Gamma et al., 1994b, pg. 362)]

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

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

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

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

hookOperation no hace nada en la clase padre:

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

Las subclases lo extienden:

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

- 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.
- ermite añadir un componente de filtrado antes de la petición de un servicio (pre-procesamiento) o después su respuesta (post-procesamiento).

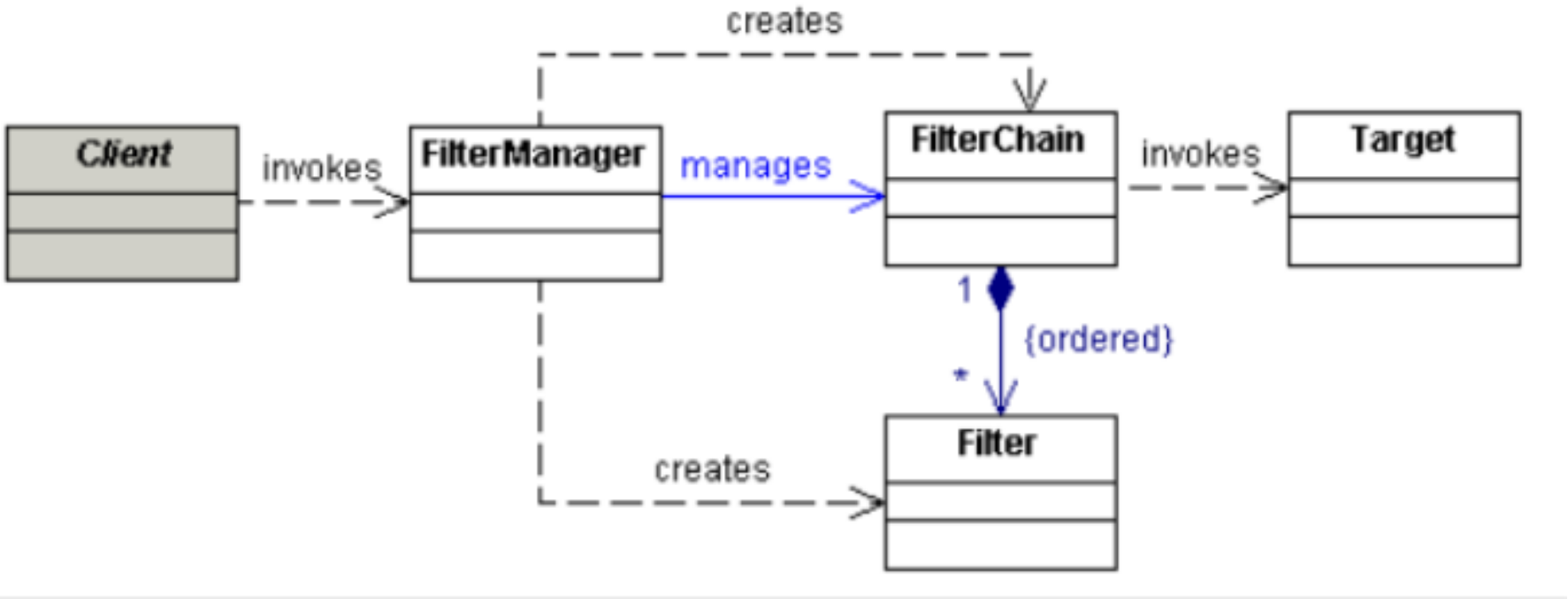


Figura 26: Diagrama de clases correspondiente al patrón interceptor de filtros.
[Fuente: Patrón Filtros de intercepción.]

- *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*).

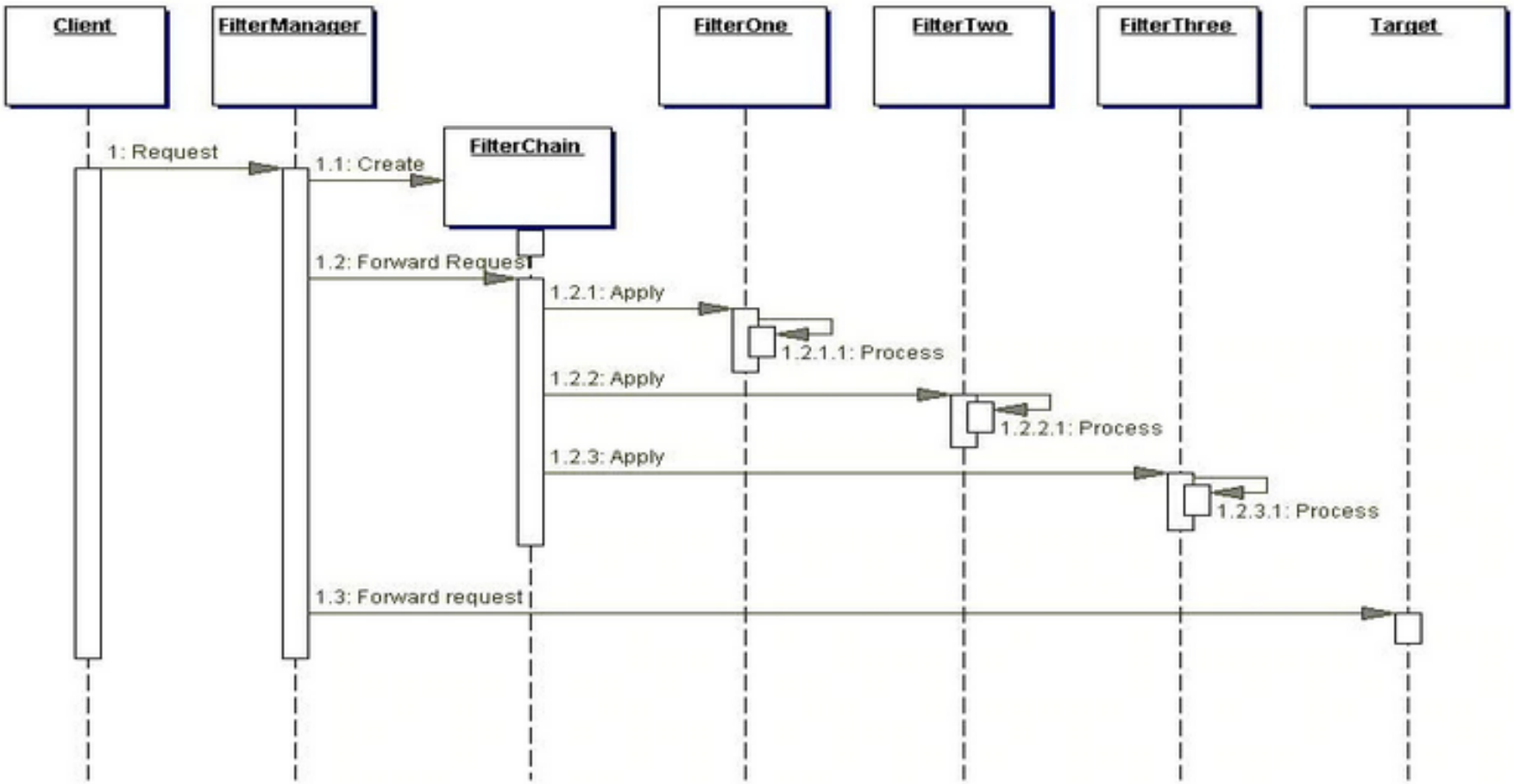


Figura 27: Ejemplo de diagrama de secuencias del estilo arquitectónico *Filtros de intercepción*. [Fuente: https://www.oracle.com/technetwork/java/interceptingfilter-142169.html.]

Un *estilo arquitectónico* (o patrón arquitectónico) define:

- Tipos de componentes y conectores
- Conjunto de restricciones sobre la forma en la que pueden combinarse estos elementos

Algunos de ellos se han intentado clasificar:

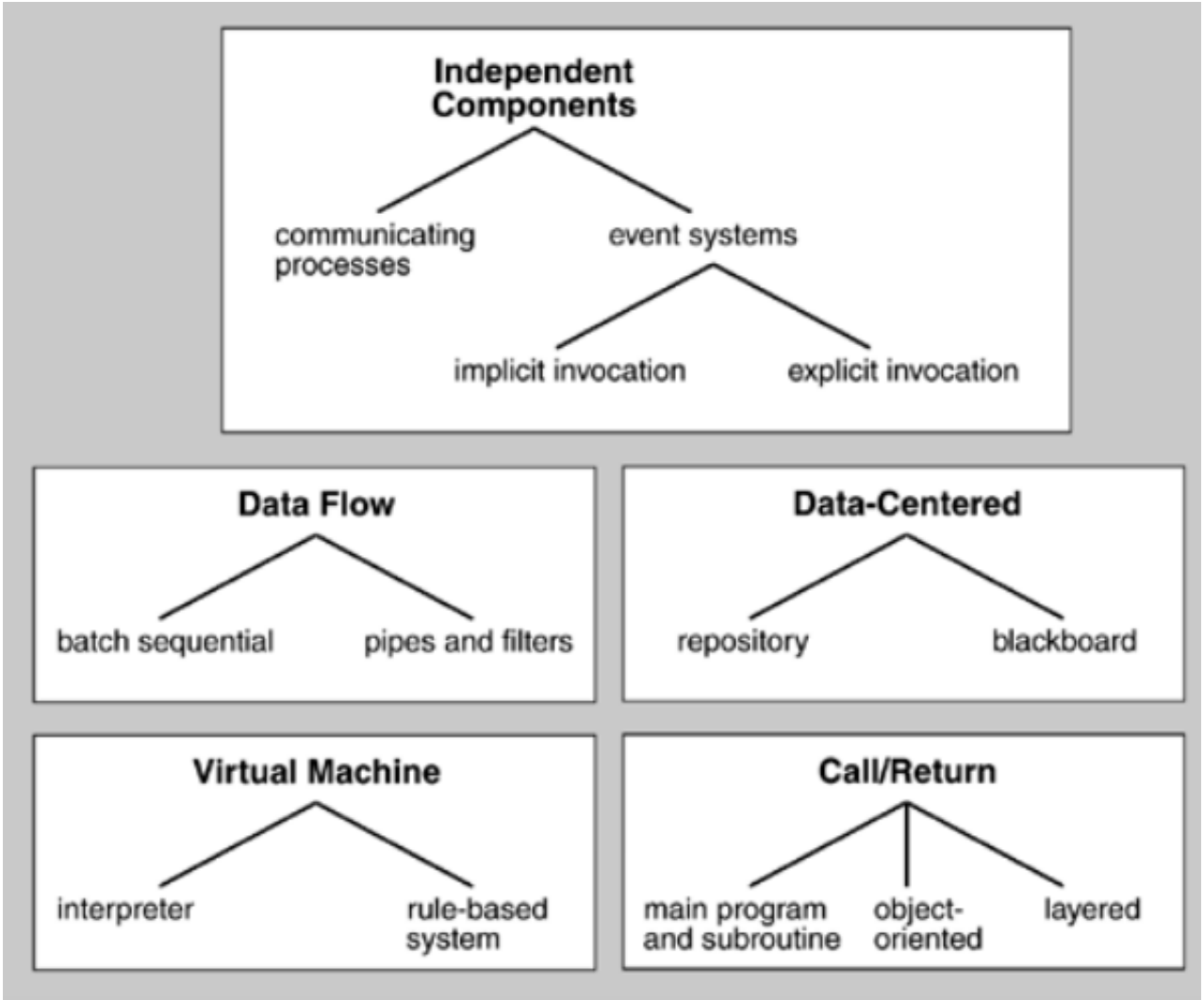


Figura 28: Diagrama de componentes usando el estilo arquitectónico *Control de procesos* para el sistema de control de la velocidad de crucero. [Fuente: (Bass et al., 2003)]

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

Componentes:

- 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
- Tubería: Transportan la corriente de datos (data stream)

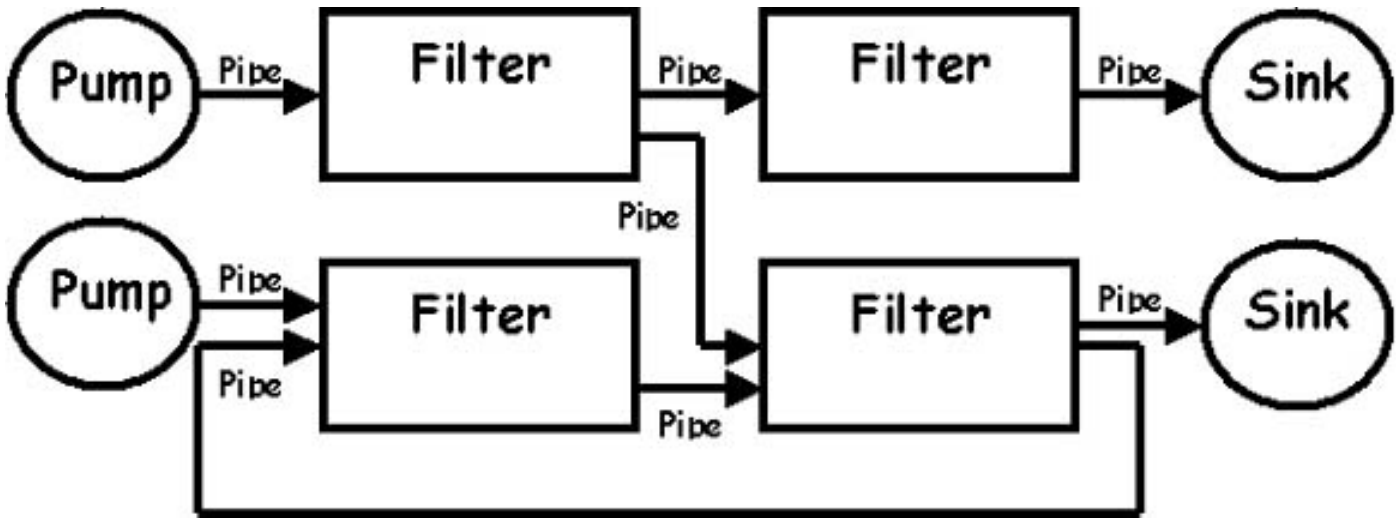


Figura 29: Estructura del estilo arquitectónico *Tubería y filtro*. [Fuente: (Garfixia, Accessed March 4, 2020)]

Ejemplo: shell de Unix:

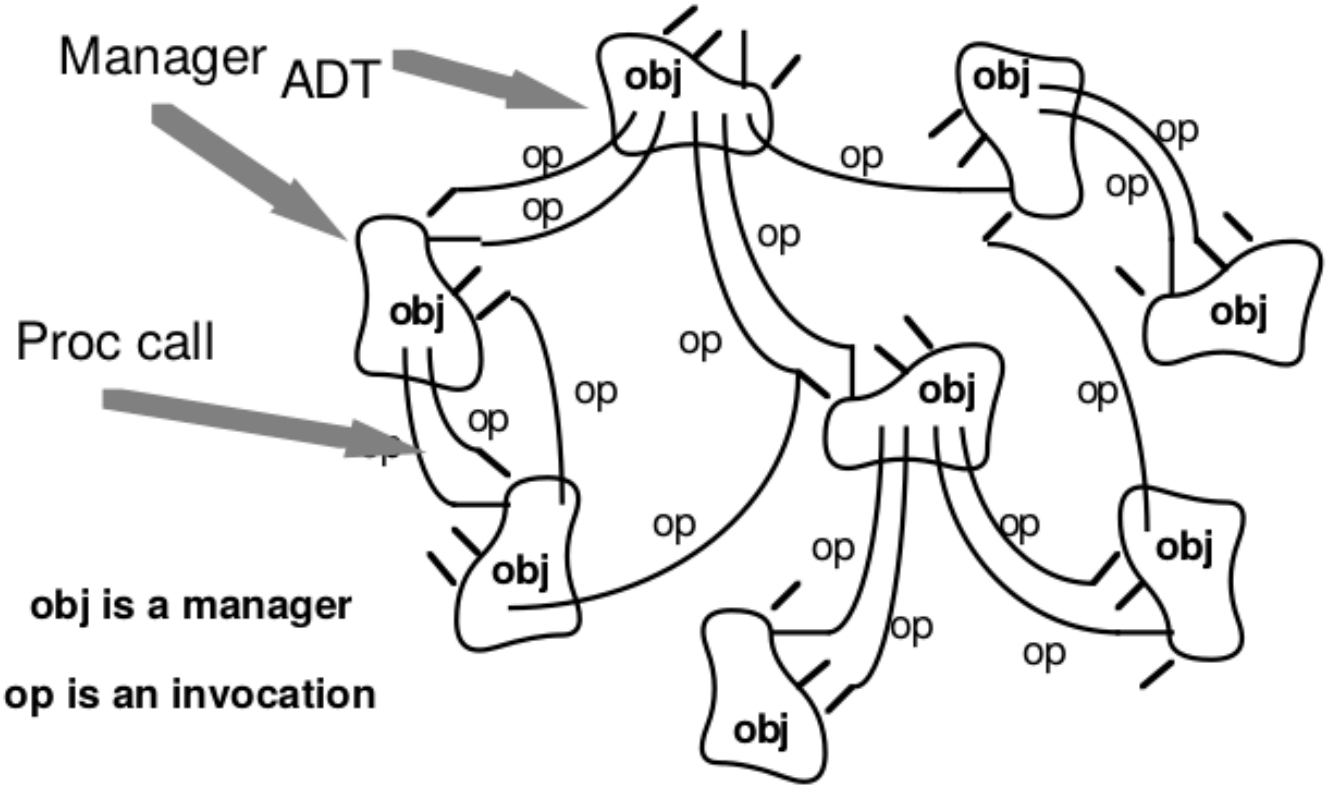


Figura 30: Ejemplo del estilo arquitectónico *Abstracción de datos y organización OO*. [Fuente: (Shaw and Garlan, 1996, pg. 23)]

Estilos de Llamada y Retorno. (2) El estilo *Basado en eventos*

Variantes:

- Invocación implícita (estilo *Manejador de eventos* o *Publicar/subscribir*)
- Invocación explícita

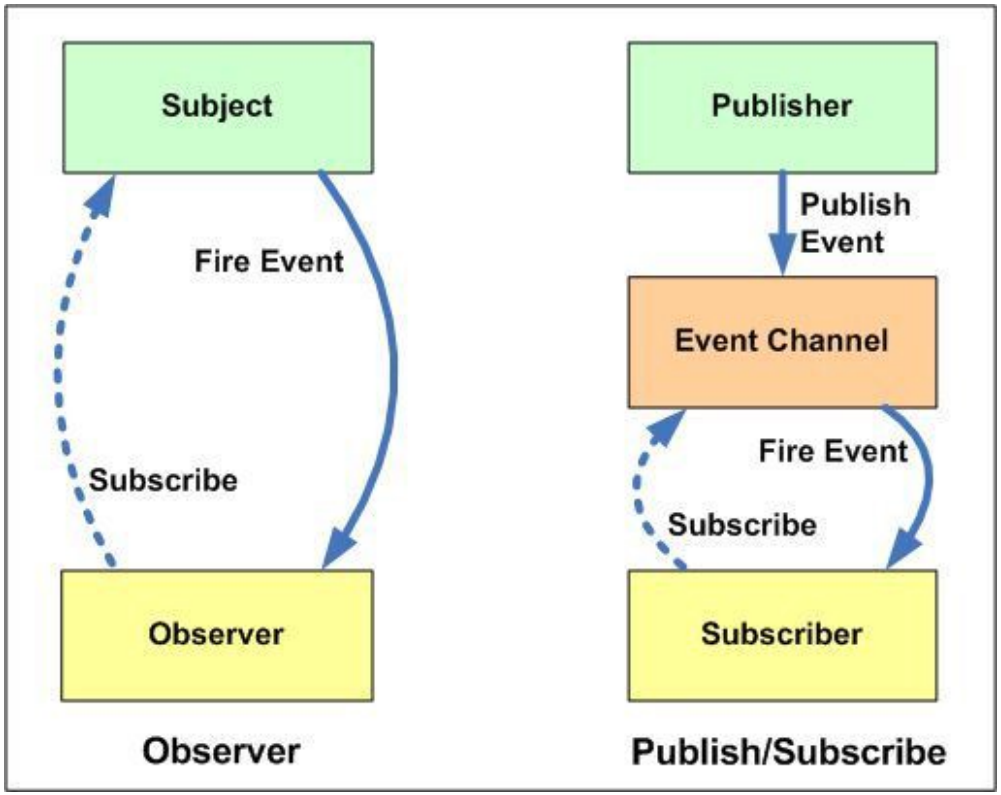


Figura 31: Diferencia entre el estilo arquitectónico *Basado en eventos* y el patrón de diseño *Observador*. [Fuente: https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c

Estilos de Llamada y Retorno. (2) El estilo *Basado en eventos*

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

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

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.

Variantes:

- Controlador ligero o Modelo activo
- Controlador pesado o Modelo pasivo

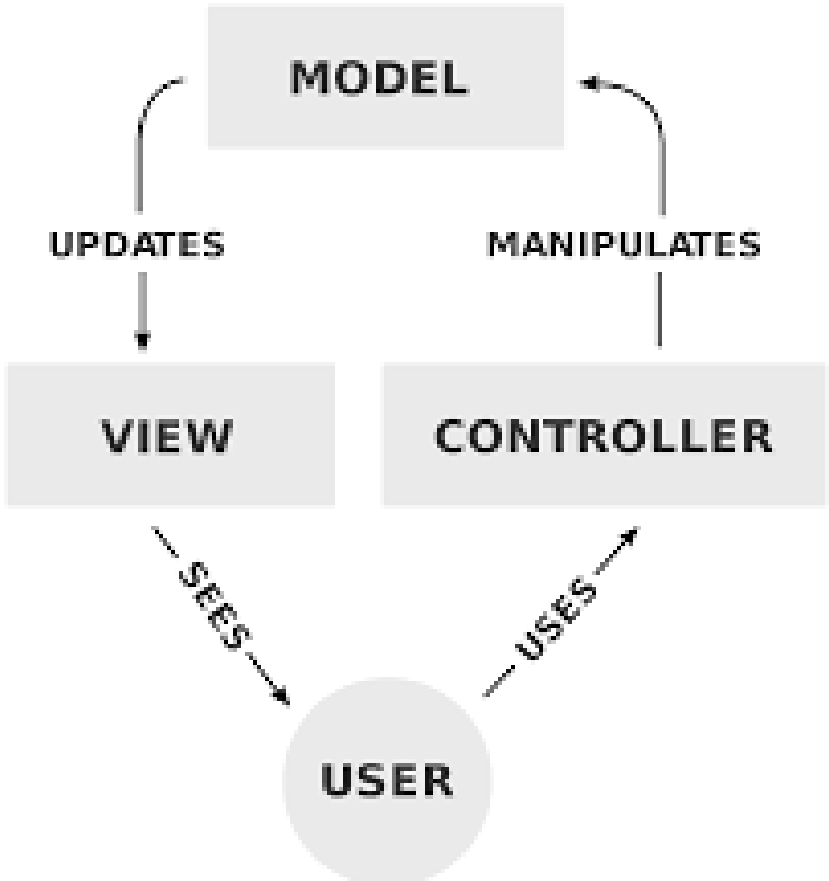


Figura 32: Estructura del estilo arquitectónico MVC de Controlador ligero (ver más abajo). [Fuente: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller]

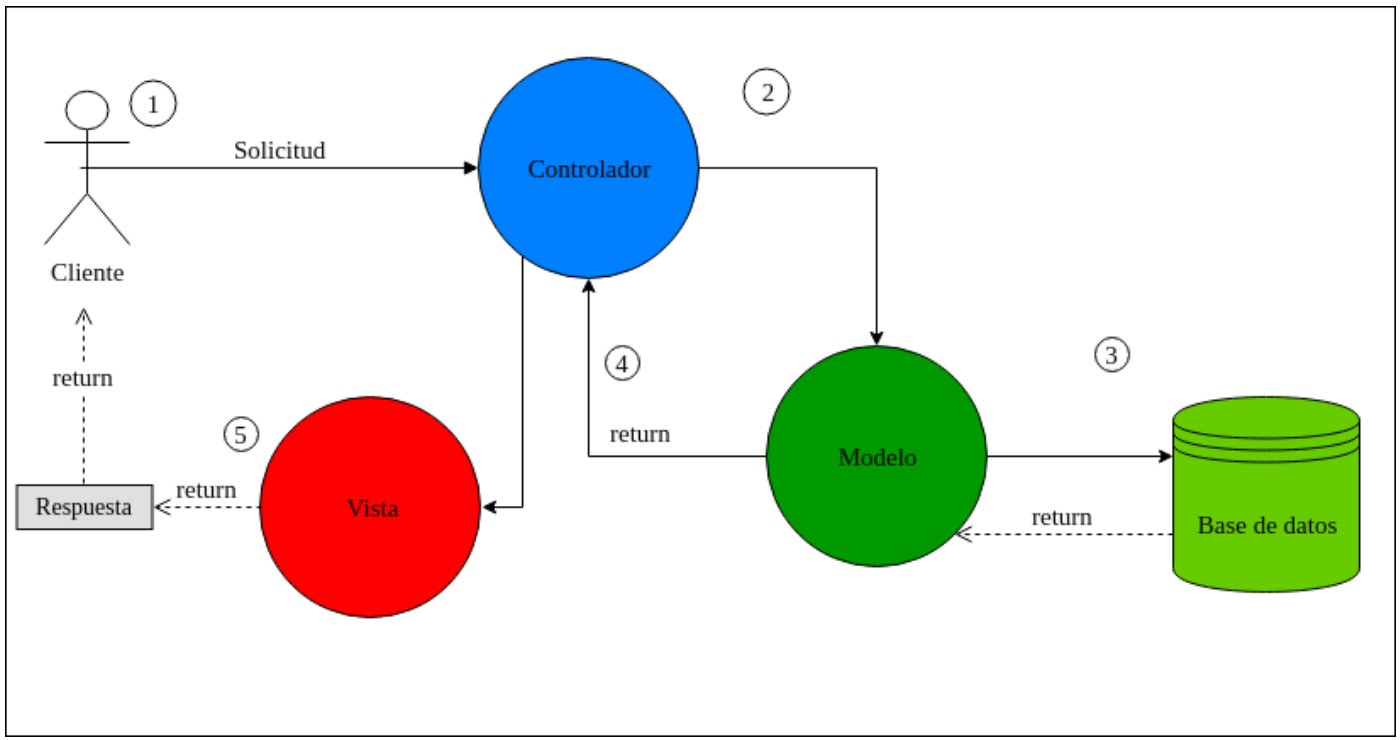


Figura 33: Estructura del estilo arquitectónico MVC de Controlador pesado. [Fuente: https://articulosvirtuales.com/articles/educacion/que-es-el-modelo-vista-controlador-mvc-y-como-funciona]

Figura 34: Implementación del estilo MVC hecha por Java SWING.

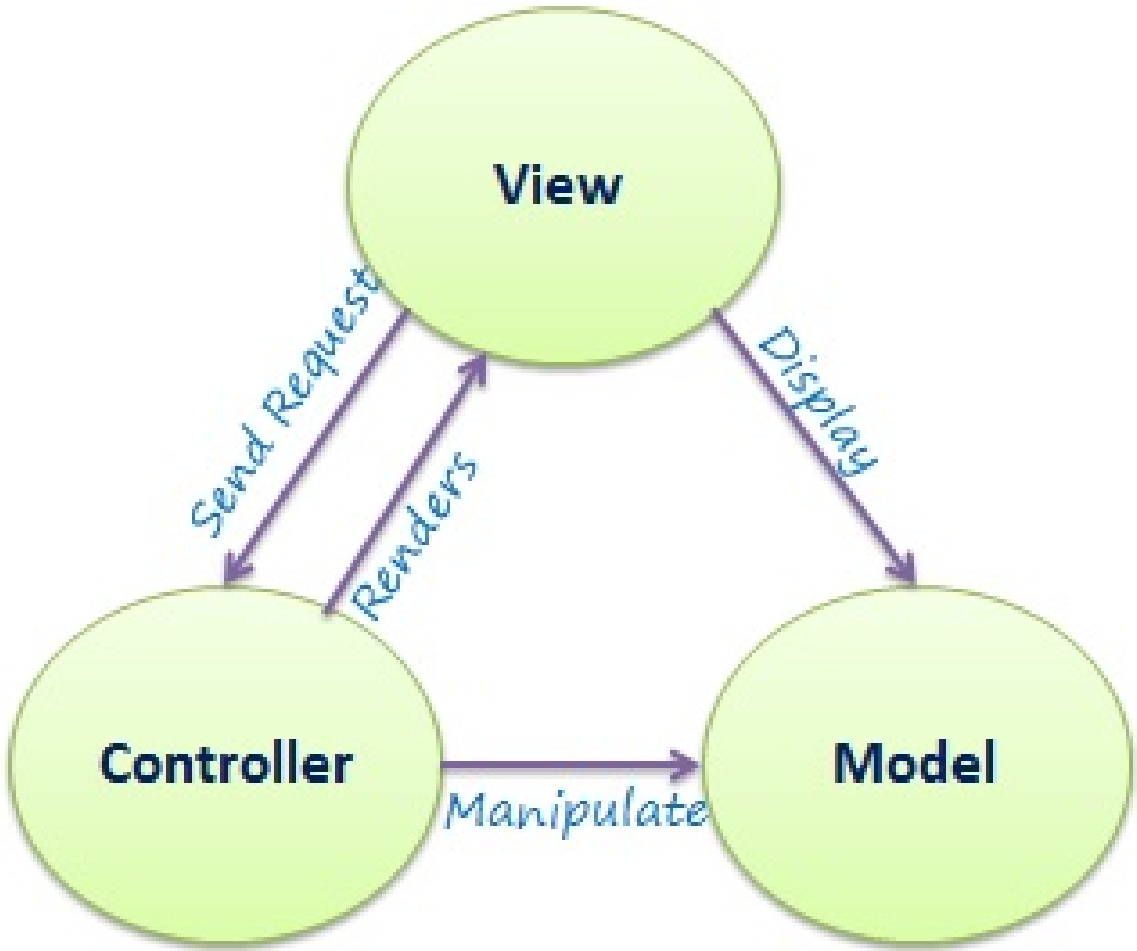


Figura 35: Estructura del estilo arquitectónico MVC pesado implementado por Angular JS. [Fuente: https://w3tutoriels.com/angularjs/angularjs-mvc/]

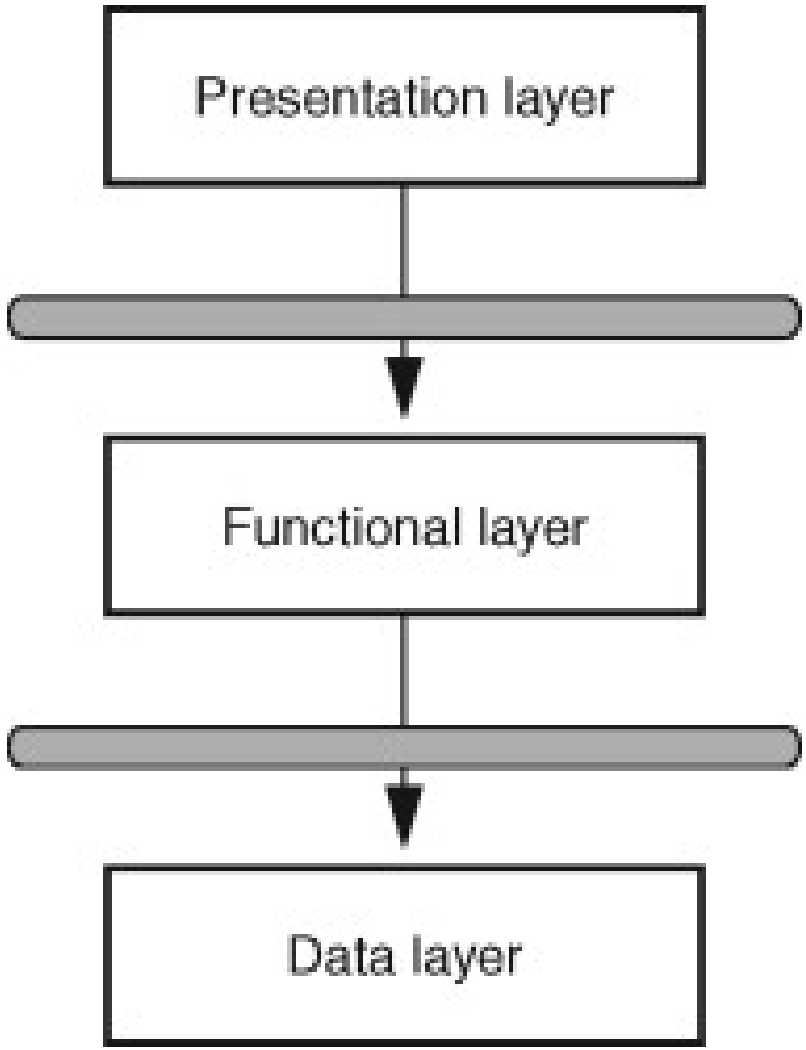


Figura 36: Estructura del estilo arquitectónico *Sistema por capas* con sólo tres capas. [Fuente: (Züllighoven, 2005)]

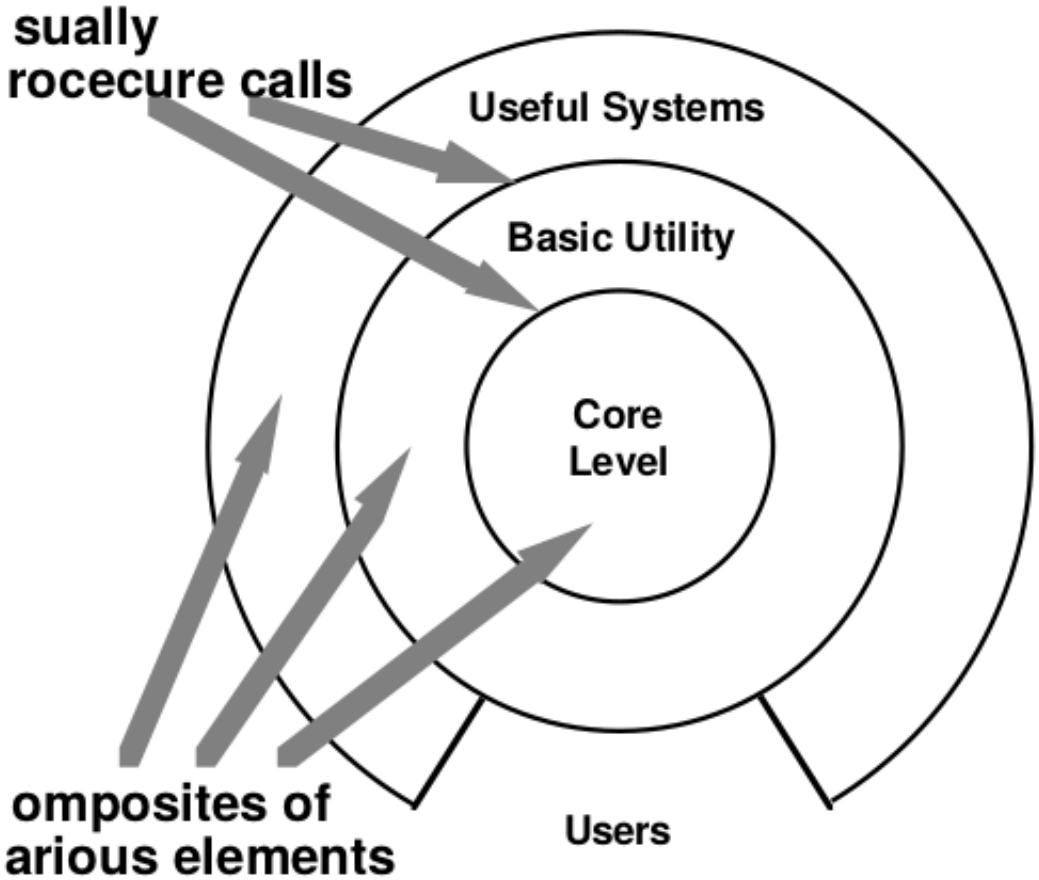


Figura 37: Estructura del estilo arquitectónico *Sistema por capas* genérico. [Fuente: (Shaw and Garlan, 1996, pg. 25)]

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).

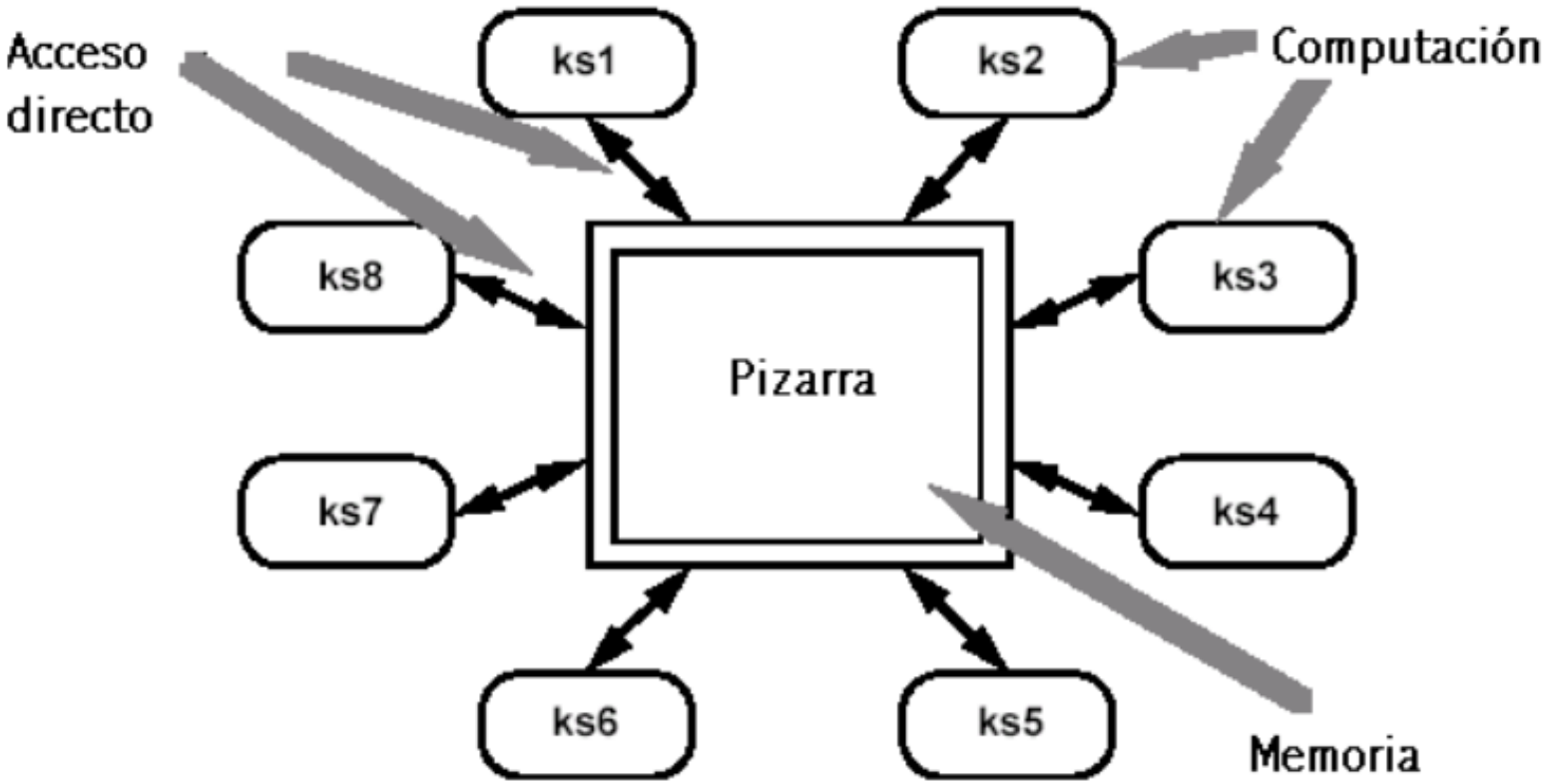


Figura 38: Relación entre la pizarra y las fuentes de conocimiento en el estilo arquitectónico *Pizarra*. [Fuente: (Shaw and Garlan, 1996, pg. 26)]

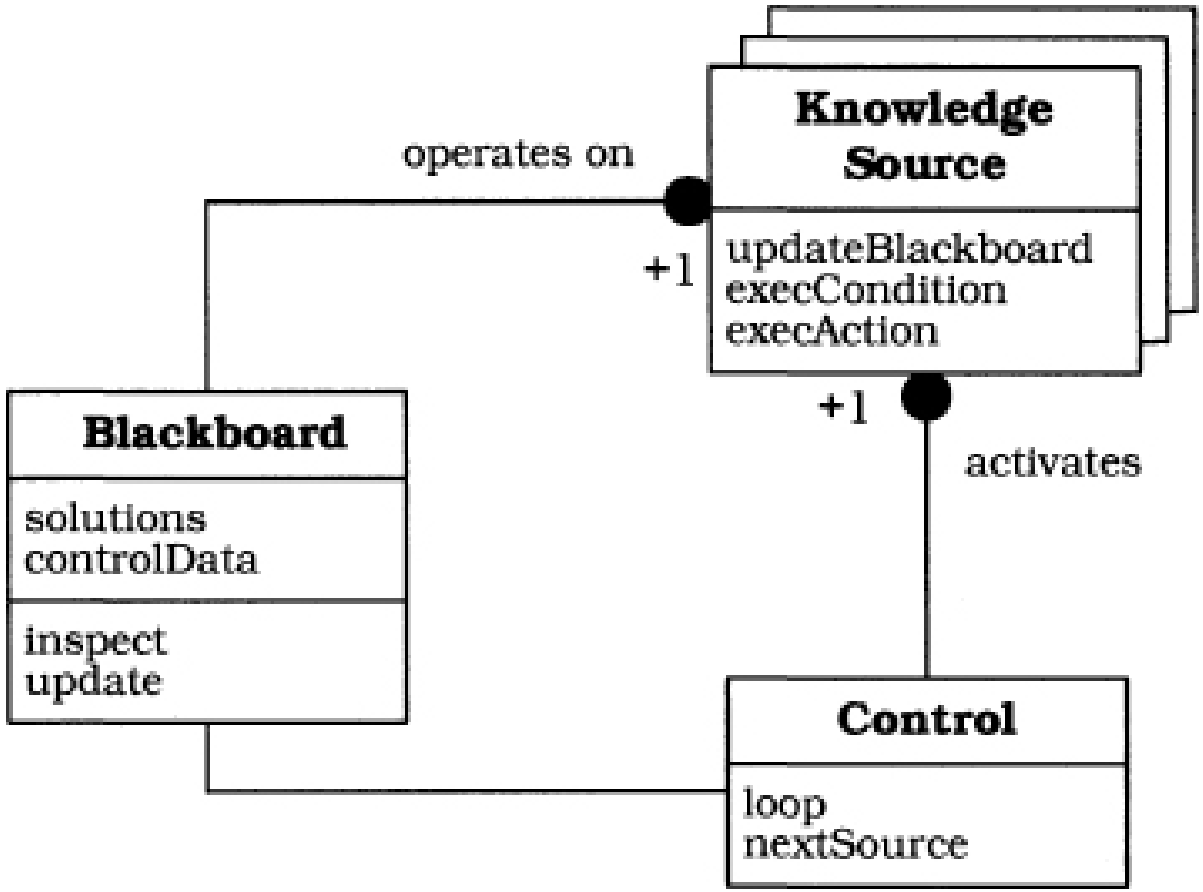


Figura 39: Diagrama de clases para representar el estilo arquitectónico *Pizarra*. [Fuente: (Buschmann et al., 1996, pg. 79)]

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

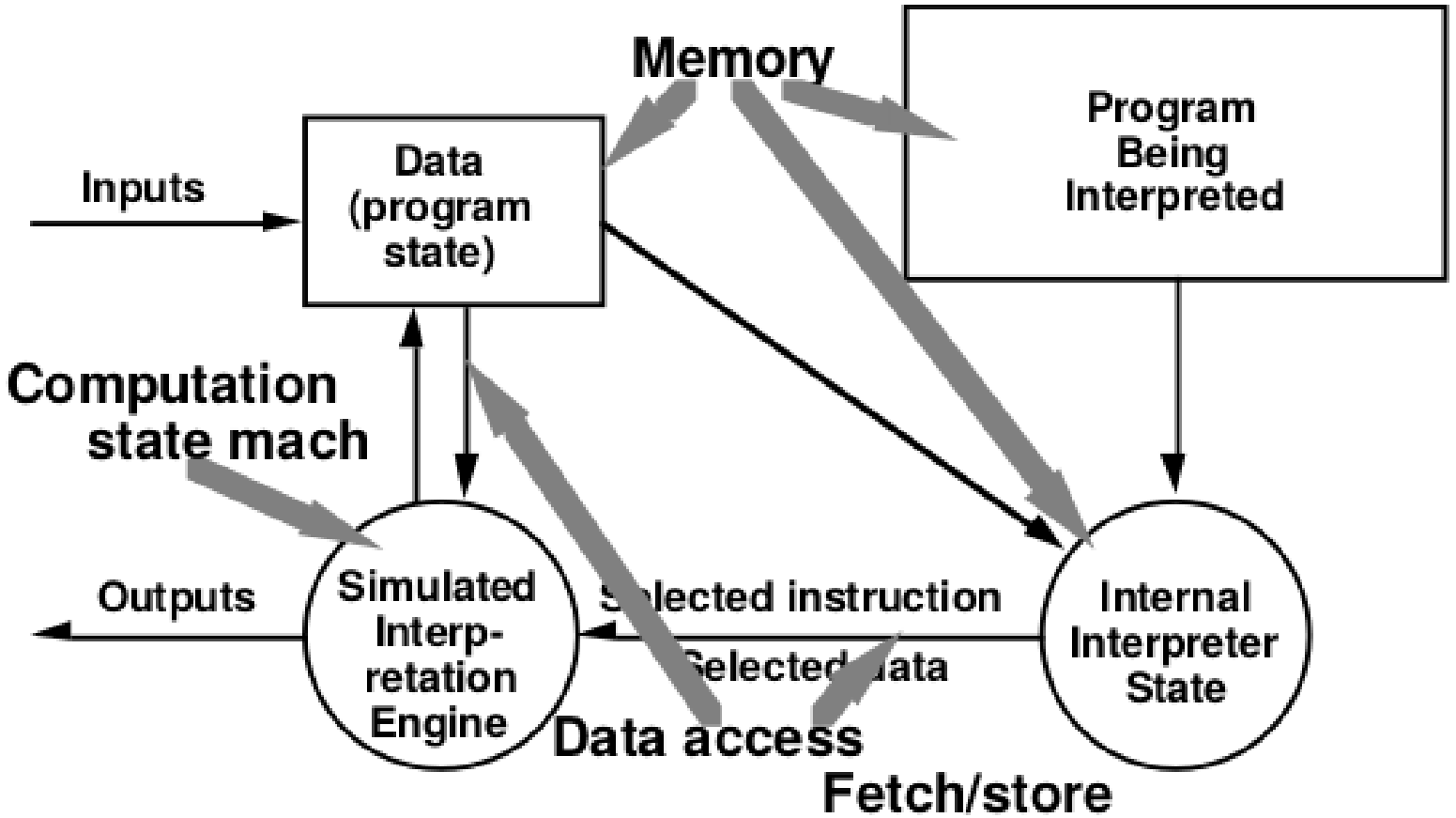


Figura 40: Estructura del estilo arquitectónico *Intérprete*. [Fuente: (Shaw and Garlan, 1996, pg. 27)]

Variantes:

- *Ciclo abierto* (Figure 41).- 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 42).- 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 43)
  - *Control de procesos preventivo* (Figure 44)

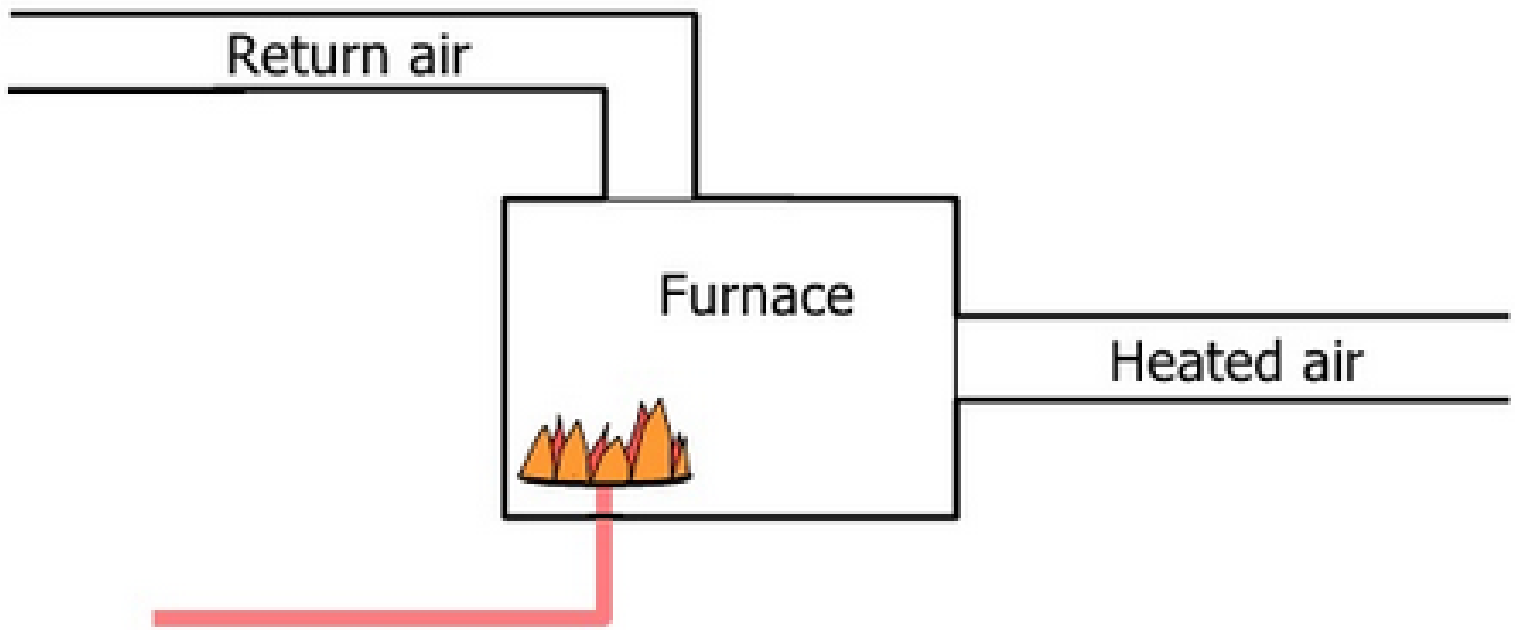


Figura 41: Ejemplo de sistema con estilo arquitectónico *Control de procesos de ciclo abierto*. [Fuente: (Shaw and Garlan, 1996, pg. 29)]

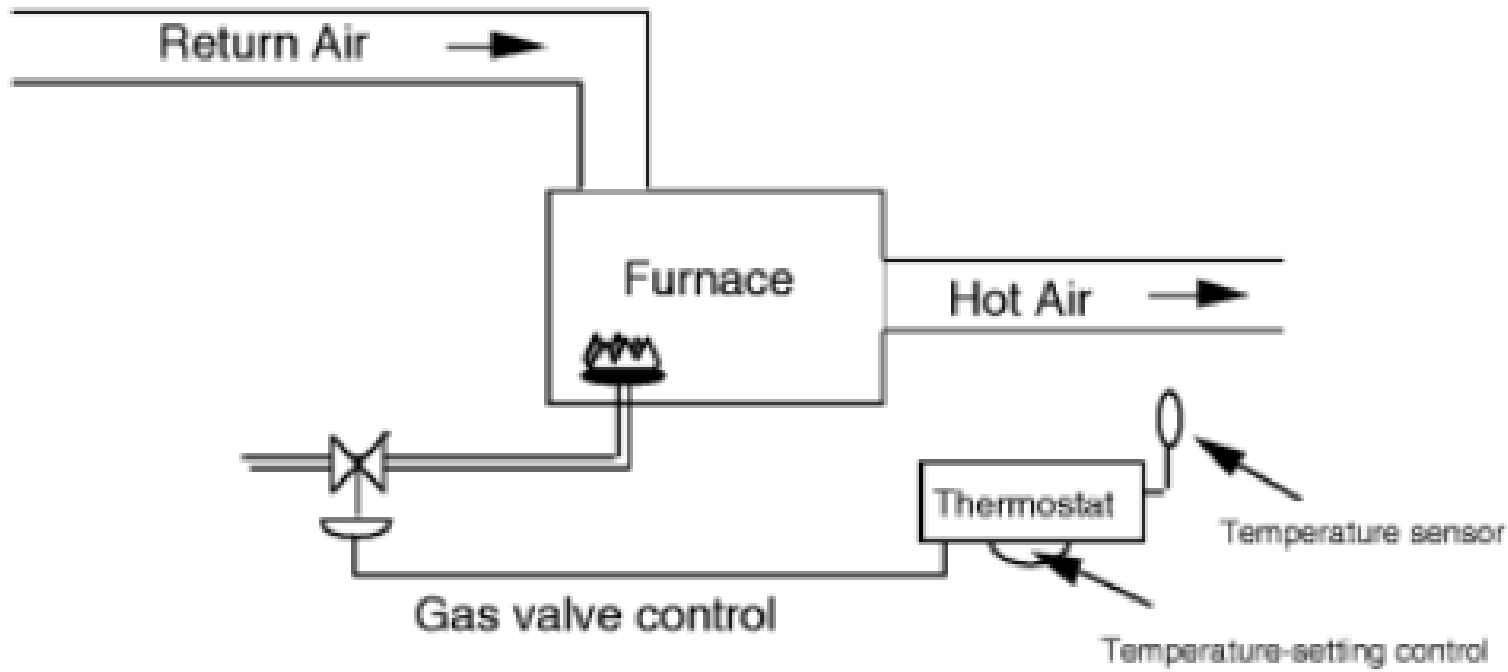


Figura 42: Ejemplo de sistema con estilo arquitectónico *Control de procesos* de ciclo cerrado. [Fuente: (Shaw and Garlan, 1996, pg. 29)]

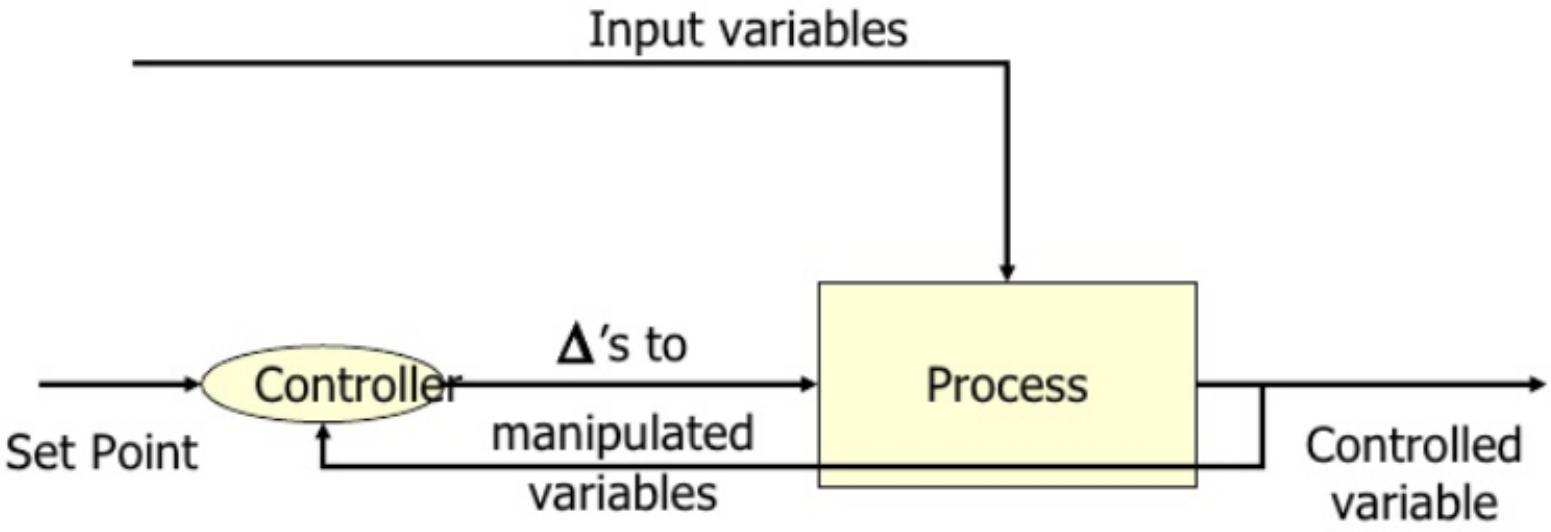


Figura 43: Estructura del estilo arquitectónico *Control de procesos retroalimentado*. [Fuente: (Shaw and Garlan, 1996, pg. 29)]

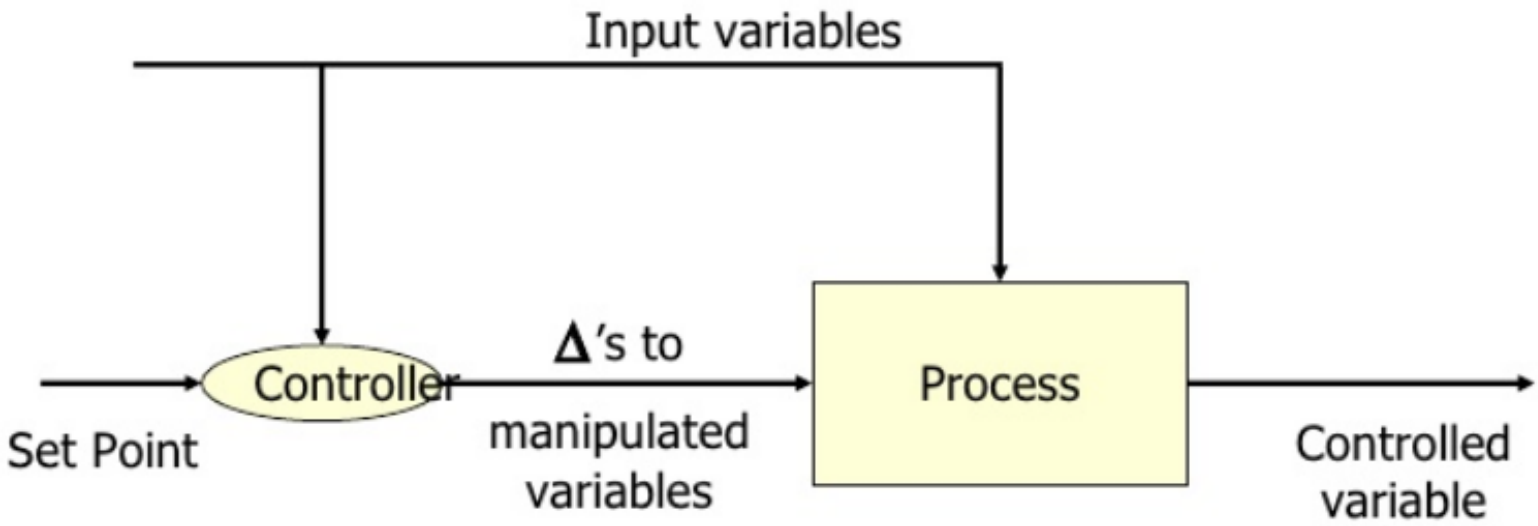


Figura 44: Estructura del estilo arquitectónico *Control de procesos preventivo*. [Fuente: (Shaw and Garlan, 1996, pg. 30)]

Ejemplo control velocidad de crucero (*preventivo*): diagrama de bloques

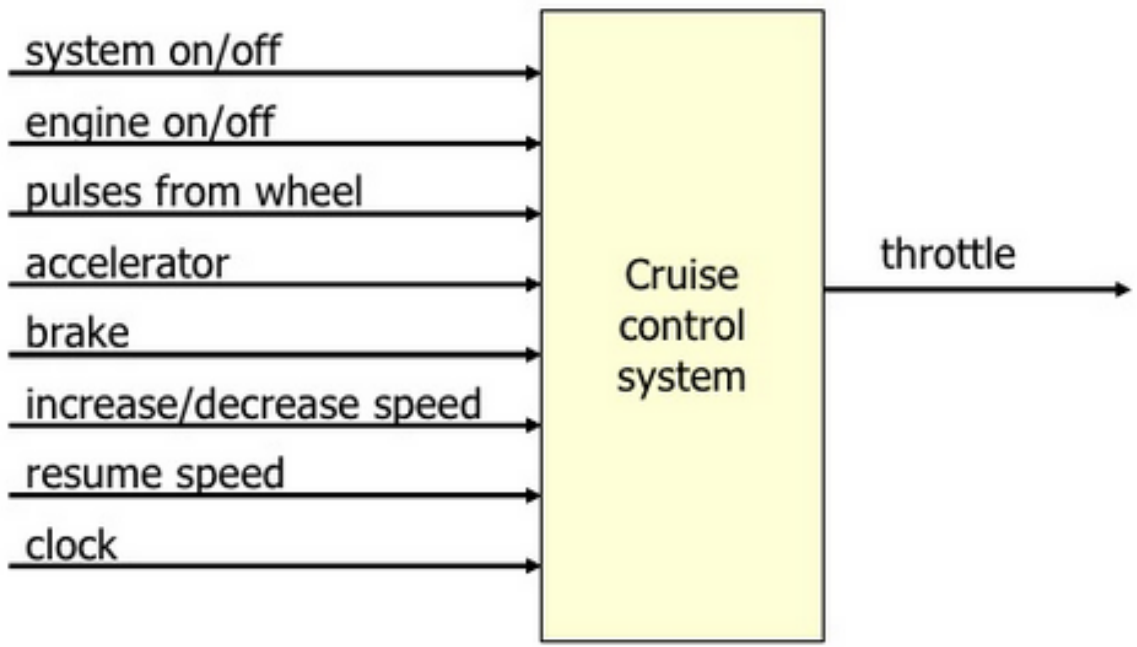


Figura 45: Diagrama de bloques del sistema de control de la velocidad de crucero. [Fuente: (Shaw and Garlan, 1996, pg. 52)]

Solución OO (Booch)

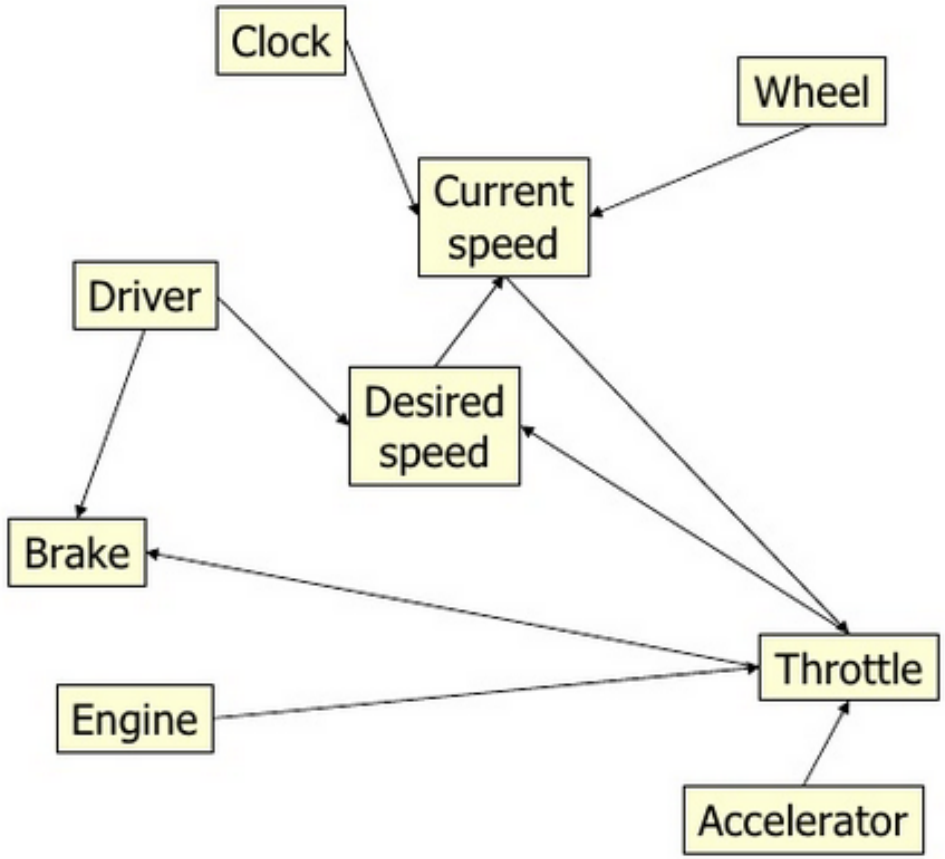


Figura 46: Diagrama de clases para el sistema de control de la velocidad de crucero. [Fuente: (Phillips et al., 1999)]

Diagrama del estilo arquitectónico

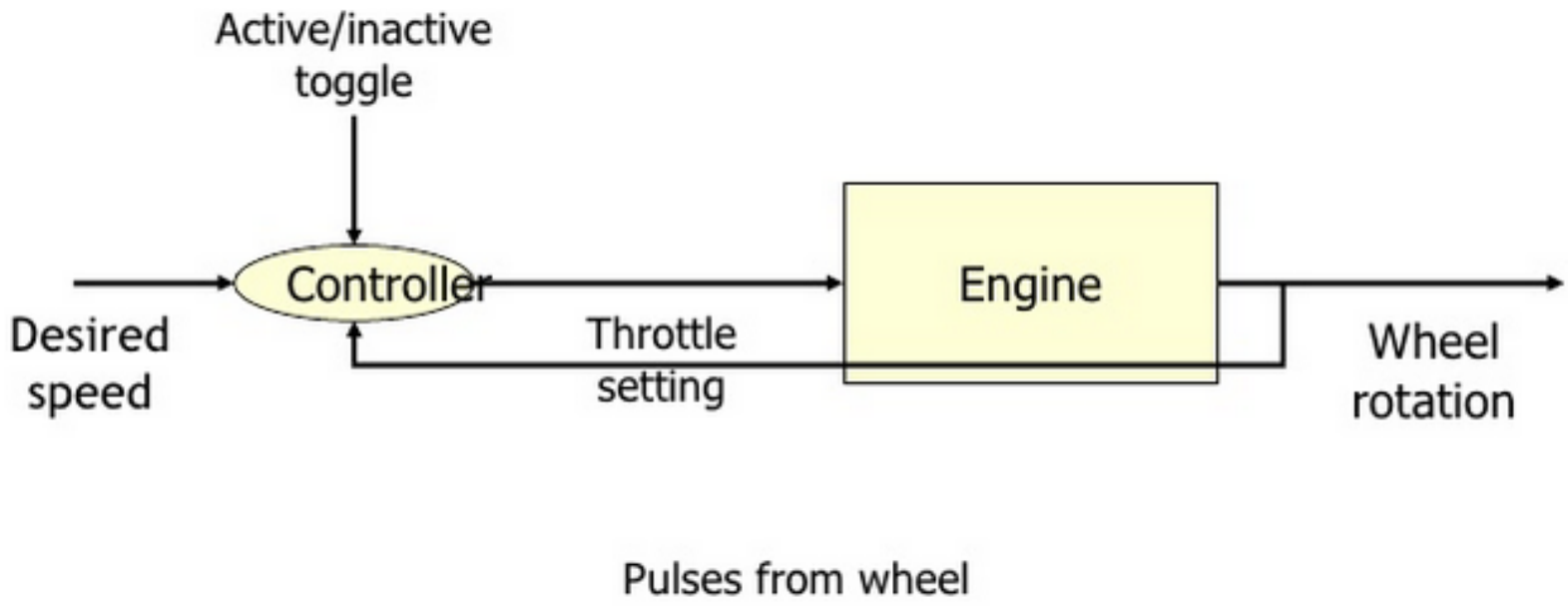


Figura 47: Diagrama del estilo arquitectónico *Control de procesos* para el sistema de control de la velocidad de crucero. [Fuente: (Phillips et al., 1999)]

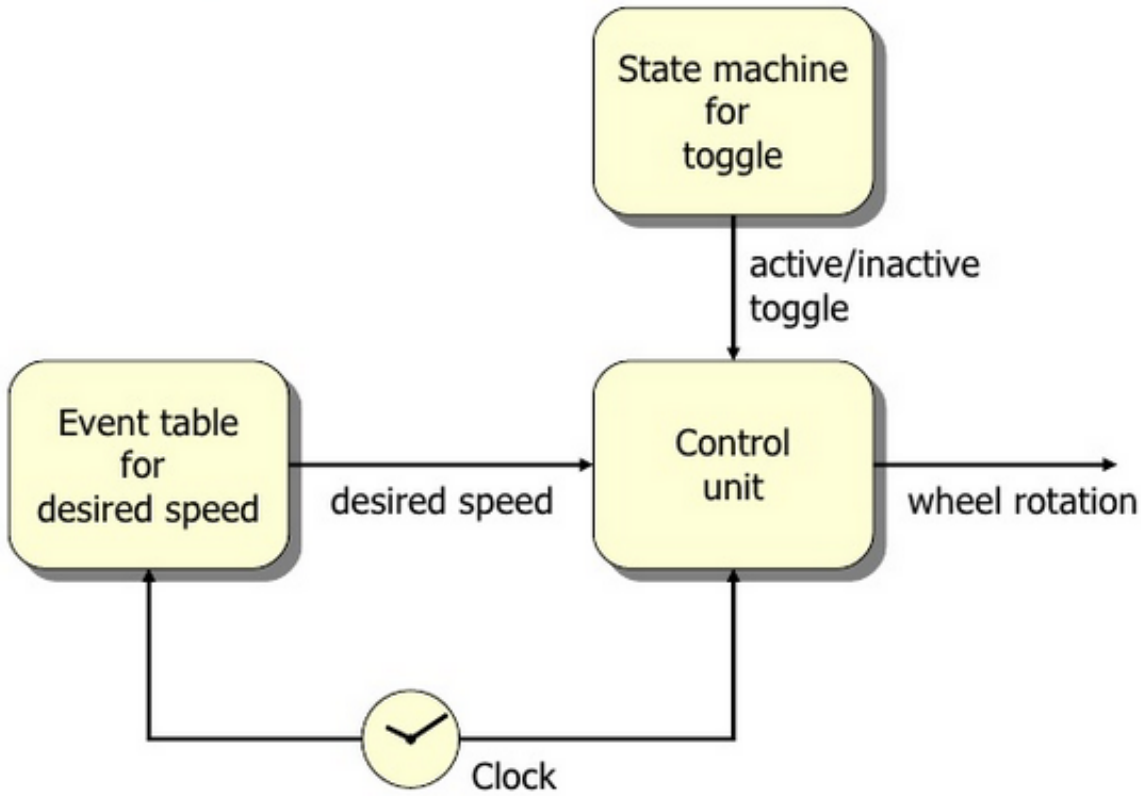


Figura 48: Diagrama de componentes). [Fuente: (Phillips et al., 1999)]

- Estilo *Organización programa principal/subrutinas*
- Estilo *Sistema de transición de estados*
- Estilo de *Procesos distribuidos*
  - Estilo *Peer-to-Peer*
  - Estilo *Cliente/Servidor*
    - *Arquitectura Orientada a Servicios*
      - *Arquitectura Basada en Recursos: REST (Representational State Transfer)*
      - *Arquitectura de Computación en la Nube*
- Estilos específicos de un dominio

Relación de la arquitectura orientada a servicios (SOA) con otros estilos arquitectónicos

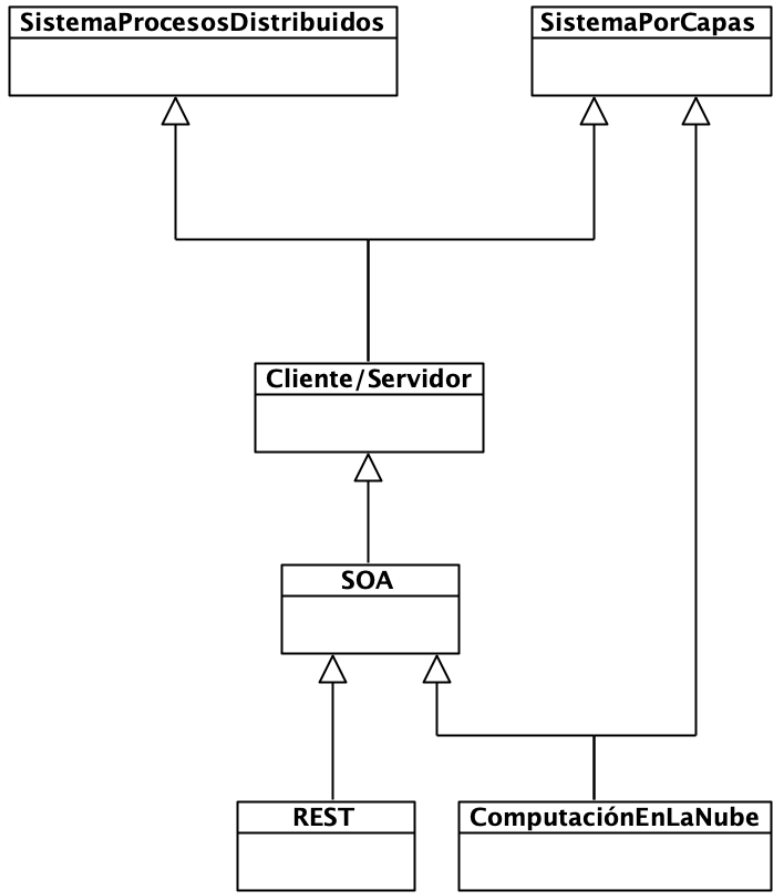


Figura 49: Relación de distintos superestilos y subestilos arquitectónicos de la arquitectura orientada a servicios (SOA).

- REST: REpresentational State Transfer
- Se basa en el concepto de recurso: cualquier cosa con una URI (Uniform Resource Identifier)
- Los servicios y los proveedores de servicios deben ser recursos
- Características necesarias para cumplir REST:
  1. Arquitectura cliente-servidor (como cualquier SOA)
  2. Sin estado (como cualquier SOA)
  3. Cacheable
  4. Interfaz uniforme
  5. Sistema por capas

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 2: Restricciones adicionales de REST para hacer las interfaces uniformes.

REST e interfaz uniforme

- Lo único que establece REST es que se use una interfaz uniforme
- De forma convencional se asocian los métodos recurso de REST con los métodos GET/PUT/POST/DELETE del protocolo HTTP
- NO es necesario para implementar REST seguir esta convención. Lo único que se exige es la uniformidad de la interfaz
- Se suele usar XML y mucho más JSON como formatos de datos transmitidos
- Para 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
- Además, se eligen los cuatro métodos HTTP y se asocia cada uno de ellos a una de las cuatro operaciones CRUD: triple asociación entre los conceptos CRUD, REST y HTTP

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 [1]
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 3: Las cuatro operaciones HTTP más habituales en REST.

[1] 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.

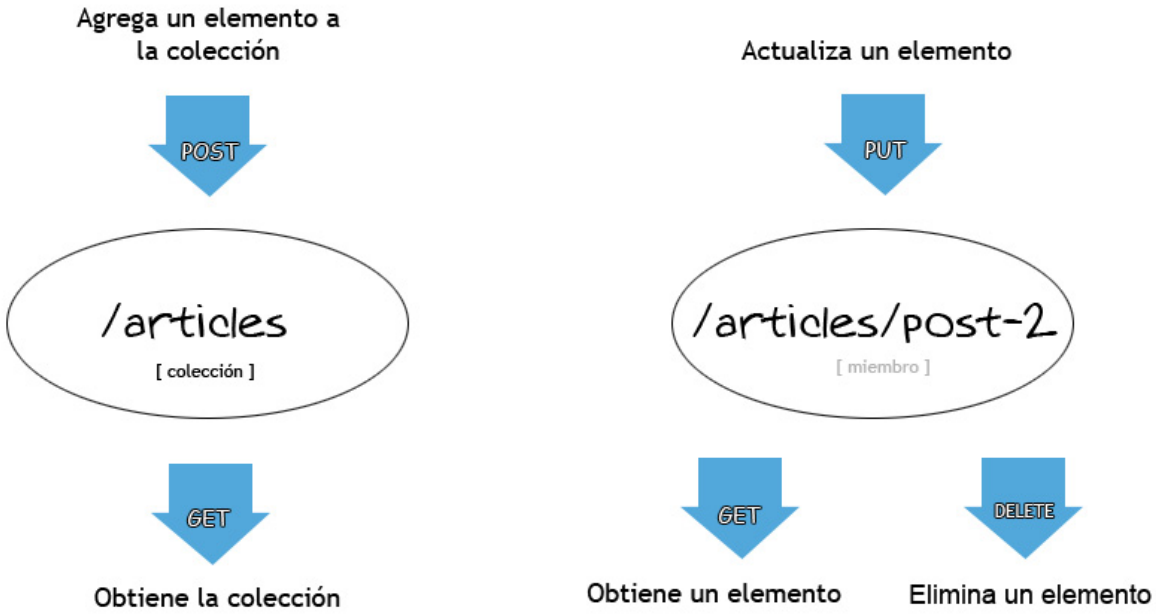


Figura 50: Ejemplo de uso de las funciones REST [Fuente: (Los Santos Aransay, 2009)]

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

Tabla 4: Ejemplo de la relación entre CRUD, HTTP y REST sobre «project».

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 5: 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.

- Combinación de estilos.- los estilos se combinan de forma que en un sólo 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.
  - Componentes compartidos.- Un mismo componente puede formar 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.
- Confusión entre estilos arquitectónicos y patrones de diseño.- 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.
- Estilos híbridos.- Algunos estilos surgen como combinación de otros.

Referencias I

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

Referencias II

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.

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

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

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