Programación Orientada a Objetos

Patrones de diseño (Design Patterns)

Don't keep reinventing the wheel !

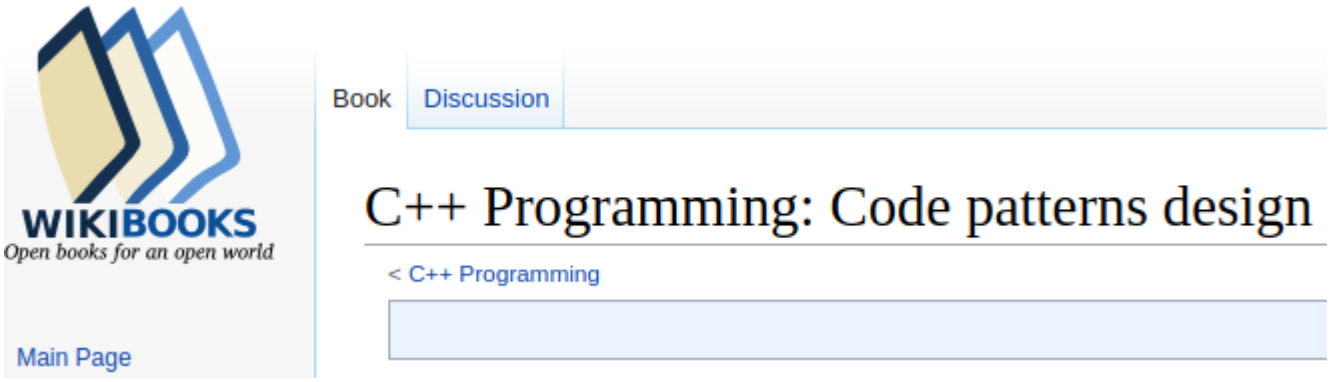
Design Patterns

- Existen elementos en el diseño de software que se repiten una y otra vez en otros diseños
- 1994. Eric Gamma, Richard Helm, Ralph Johnson y John Vlissides publican:
  - *Design Patterns: Elements of Reusable Object-Oriented Software*
  - *23 patterns*
  - *A los autores se les conoce como* ***Gang of Four*** *(GoF)*

Patrones de diseño

- Se aprecian “patrones” / ”pattern” que se repiten cuando diseñamos software
- Diseñar software es difícil, por ello es bueno no partir “de cero”.
- Los patrones son fuente de soluciones de diseño de software
- Si los estudiamos y los conocemos, podremos utilizarlos para solucionar problemas de diseño de software

Patrones de diseño

- Referencias
  - “***Design Patterns: Elements of Reusable Object-Oriented Software***” *Erich Gamma y otros* 1994. (Enorme influencia en el campo de la Ingeniería del Software moderna)
  - 

Patrón de diseño. Definición

- “Una solución reutilizable a un problema de diseño de software”
- En la etapa de diseño
  - Es un concepto utilizado en el desarrollo de software
  - Concretamente en la etapa de diseño del software
- Soluciones elegantes y de calidad a problemas complejos

Patrón de diseño. Contenido

- Un patrón de diseño está compuesto por objetos, clases y las relaciones entre ellos
- Cada patrón está especializado en resolver un problema de diseño concreto en un determinado contexto

Patrón de diseño. Utilidad

- Si estudiamos estos patrones de diseño, podremos aplicarnos a los problemas de diseño a los que nos enfrentemos en el futuro
- Diseños futuros serán:
  - Diseño de software no es tarea sencilla. Los patrones la simplifican.
  - Diseños menos costosos
  - Diseños de más calidad

Elementos esenciales

Elementos esenciales que definen un patrón de diseño concreto:

- **Nombre.** Hará referencia al problema o a la solución que aporta
- **Descripción/Estructura**. Elementos del patrón y sus relaciones, normalmente un conjunto de clases/objetos que cooperan
- **Aplicaciones**. Cuándo y en qué circunstancias aplicar el patrón
- **Consecuencias**. Ventajas e inconvenientes de su uso

Tipos

Los numerosos patrones de diseño que existen se suelen clasificar en 3 grupos según su estructura:

- **Creational Patterns.** Instantiating objects
- **Structural Patterns**. Composing clase structures between many disparate objects
- **Behavioral Patterns**. Object interactions, algorithms, etc.

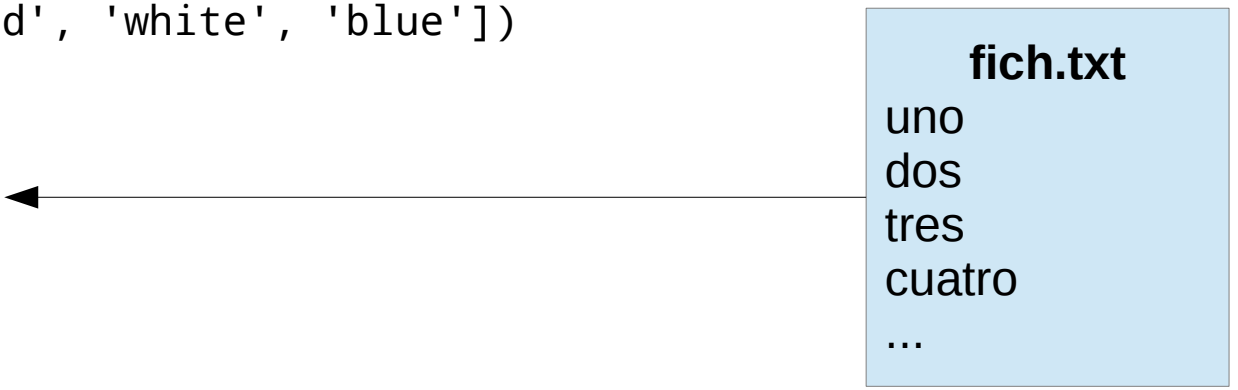
Iterator

- Nombre: **iterador** (*iterator*)
- Descripción: Behavioral pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Aplicaciones: collections, aggregates, containers, lists, vectors, etc.
- Consecuencias:
  - Sencillez de uso
  - Acceso uniforme para todas las colecciones
  - Independencia de la representación interna
- Ejemplos: C++: STL iterators. Python: for..in.. etc.

Iterator

En Python3 (por ejemplo):

```
x=[1,2,True, 4.5,”hola”]
for i in x:
  print i

import itertools as it
counter = count(start=1, step=2)
next(counter) . . .
counter = it.count(start=0.5, step=0.75)
next(counter) . . .

colors = it.cycle(['red', 'white', 'blue'])
next(colors)

fo=open(“fich.txt”)
print fo.next()

Etc...
```



Iterator

Ejemplos en C++: STL

```
list<int>::iterator it;
```

- `it++`
- `it--`
- `*it`
- `(*it).getDNI() ...`
- `etc...`

Observer

- Nombre: **observador, observer, publish-susbscribe, dependents, etc.**
- Descripción/Estructura:
  - Behavioral. Esquema de **clases cooperantes** el cual se da mucho en diseño de software (clases cooperantes: varias clases interaccionan entre sí dentro del patrón)
  - Estructura definida por el par: sujeto-observador
  - Sujeto: los datos
    - El sujeto conoce a los observadores (suscriptores) asignados
    - En su interfaz tiene un método para comunicar cambios
  - Observador: objetos que se nutren de los datos
    - Puede haber varios observadores del mismo sujeto
    - En su interfaz tiene un método para actualizarse

Observer

- Ej. datos *vs* vistas
  - Datos. Es el sujeto. Puede ser una base de datos
  - Vistas (observadores/suscriptores):
    - Estadística
    - Gráfica
    - Hoja de cálculo
    - Texto
    - Etc.

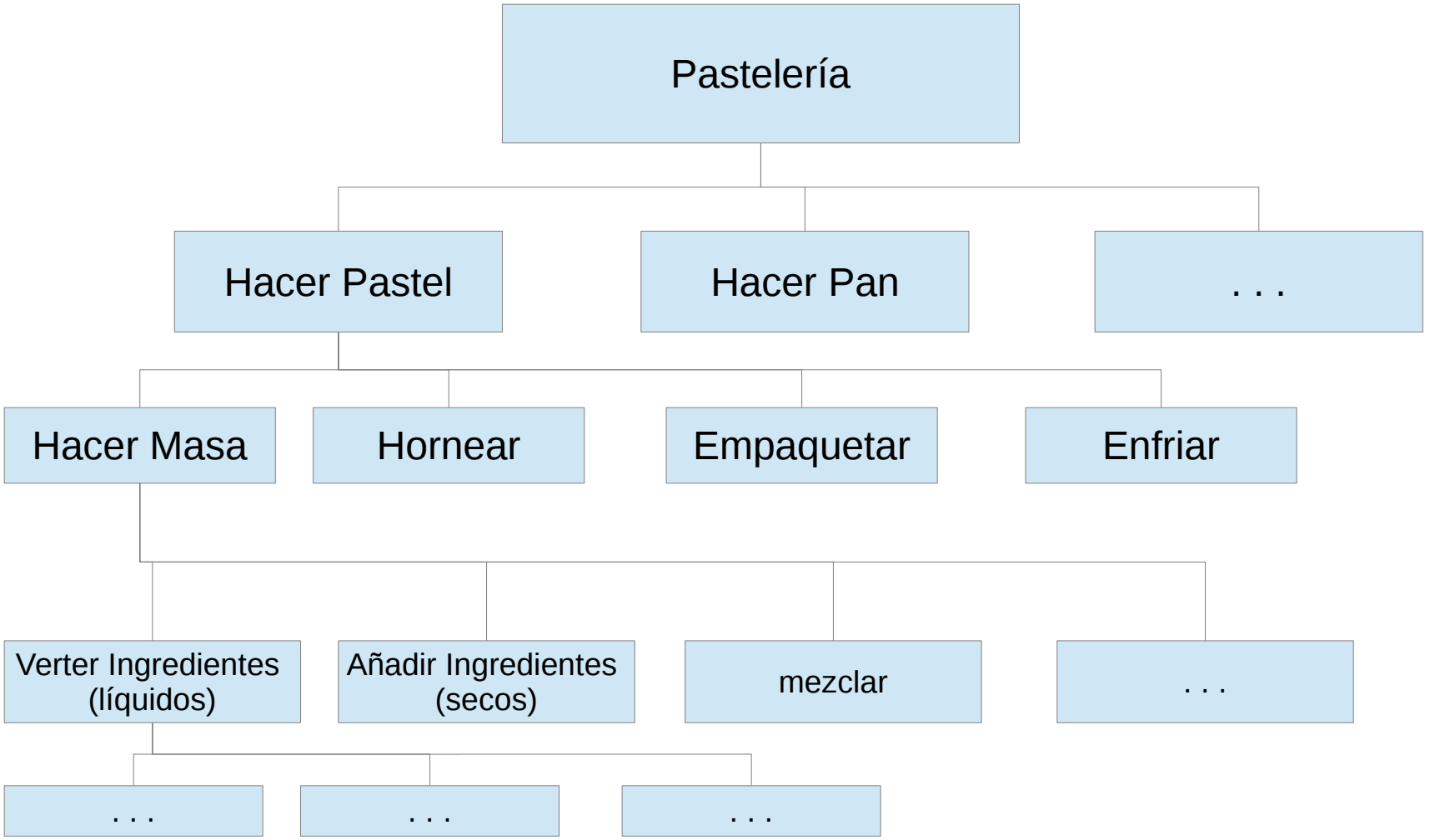
Observer

- Consecuencias:
  - Ambos elementos son independientes
  - Se pueden reutilizar por separado
  - Se pueden añadir observadores nuevos
  - Mezclar datos y observador es un ERROR de diseño
- Aplicaciones:
  - Infinidad de aplicaciones tienen clases cooperantes de este tipo
  - Relación entre modelo y vista en el patrón de diseño MVC (lo veremos posteriormente)

Composite (Objeto Compuesto)

- Things built of similars sub-things
- Bigger objects from small sub-objects which might themselves be made up of smaller sub-sub-objects
- POO es tomar objetos pequeños para construir otros más grandes/complejos/interesantes
- Agrupar componentes, para construir super-componentes ocurre con mucha frecuencia en diseño
- Por eso se debe explotar constantemente para 'forzar' su uso en beneficio del diseño basado en componentes

Composite (Objeto Compuesto)



En cada nivel hay una "lista de tareas" a ejecutar que a su vez pueden ser "listas de tareas". Si creamos objeto "lista de tareas": todas se manejarán igual y se simplifica mucho todo

Composite (Objeto Compuesto)

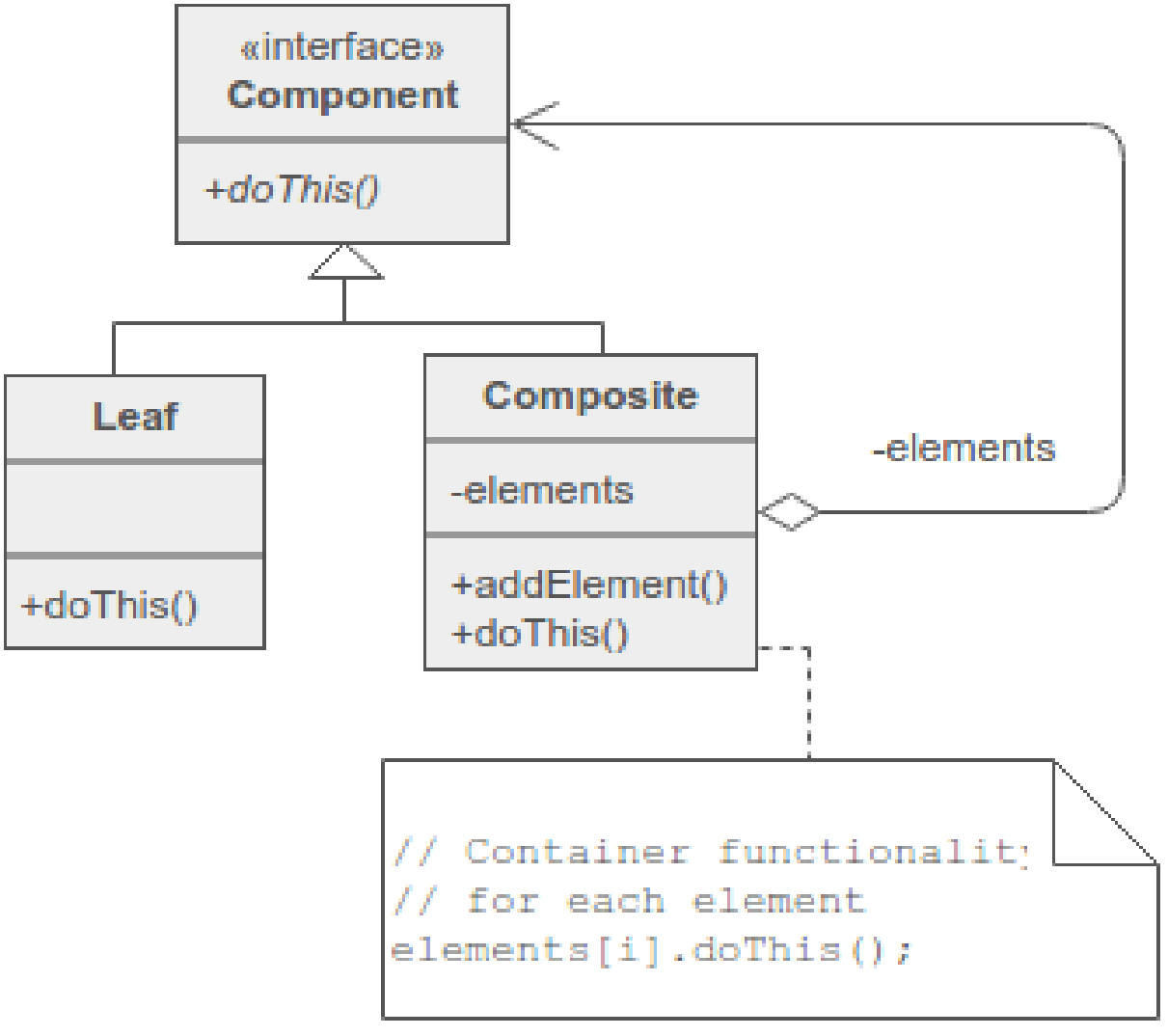
- Ejemplo en una jerarquía de tareas
  - Desde tareas complejas: hacerPan, hacerPastel, etc.
  - Hasta llegar a tareas muy simples: *verterIngredientes, añadirIngredientes, mezclarIngredientes*
  - Ambas comparten la misma interfaz: tiempoTarea, ejecutarTarea, stop, start, pause, etc.
  - Si se observan objetos que se comportan de forma parecida → composite

Composite (Objeto Compuesto)

- Otros ejemplos:
  - Sobre todo cuando en jerarquías: menús de usuario, sistema de ficheros y directorios, objetos en una aplicación que manipule objetos de diferente tipo (productos, figuras, etc.)
  - Empleados de una empresa
  - Contenedores donde cada elemento puede ser un contenedor
  - Etc.

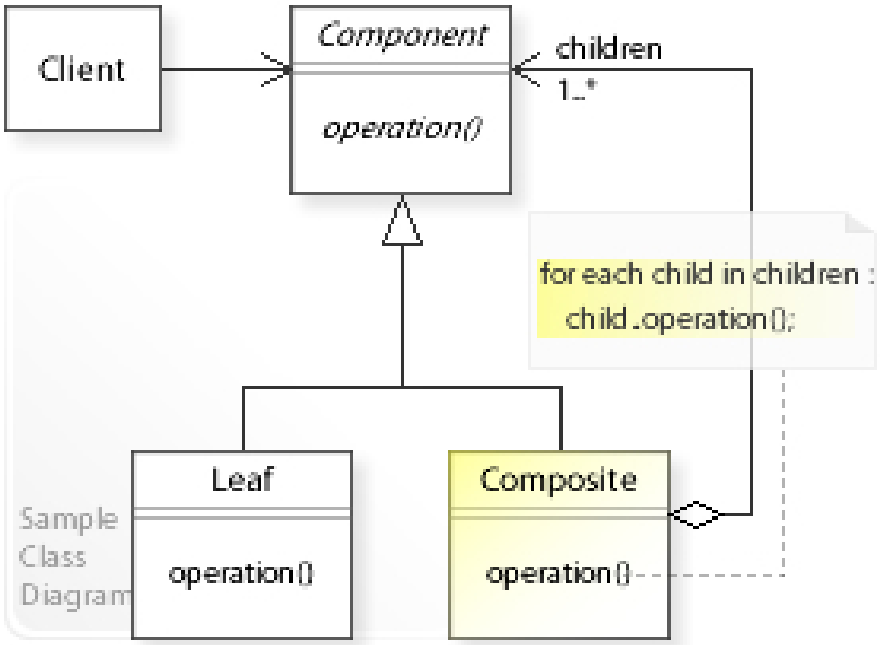
Composite (Objeto Compuesto)

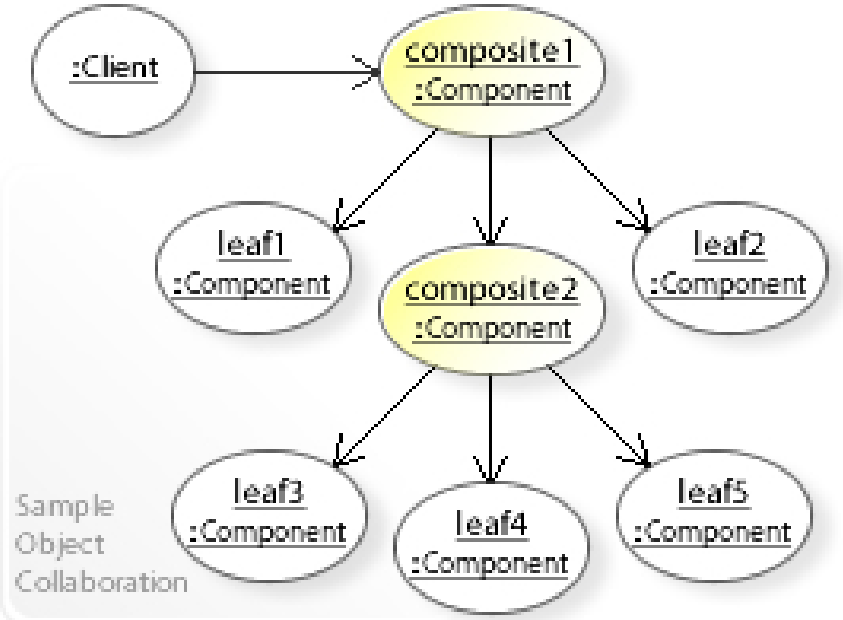
- Nombre: **objeto compuesto, composite**
- Tipo: Structura
- Estructura:

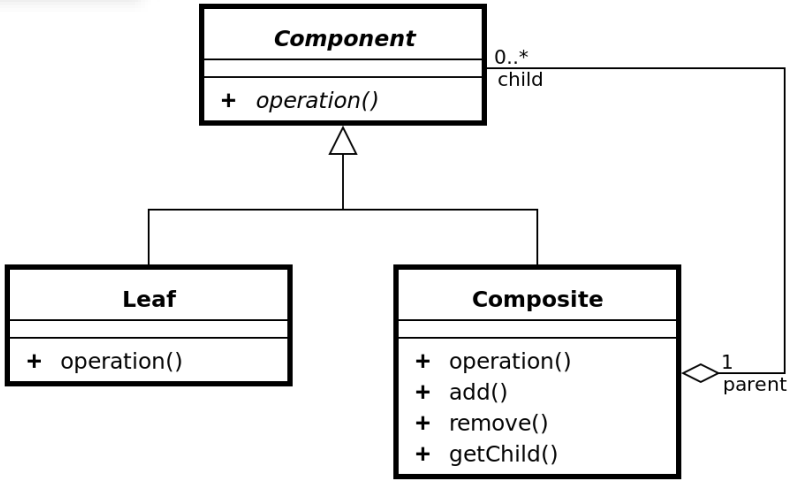


Composite (Objeto Compuesto)

- More UML class and object diagrams







Composite (Objeto Compuesto)

- Nombre: **objeto compuesto, composite**
- Estructura:
  - Se crean jerarquías de manera que se tratan igual a objetos individuales que a los compuestos
  - Clases abstractas abiertas a incorporación de nuevos componentes que se tratarán igual
- Otro ejemplo:
  - Dibujar: línea o rectángulo
  - Igual que Dibujar: dibujo1
  - Siendo dibujo1 = línea + rectángulo + dibujo2
- Los objetos se tratan de manera uniforme sean primitivas o grupos

Composite (Objeto Compuesto)

- Aplicaciones:
  - Allí donde se quiera simplificar la interfaz sean primitivas, sean grupos complejos de objetos
- Consecuencias:
  - Simplificación
  - Interfaz sencilla
  - Acceso uniforme

Strategy (Estrategia)

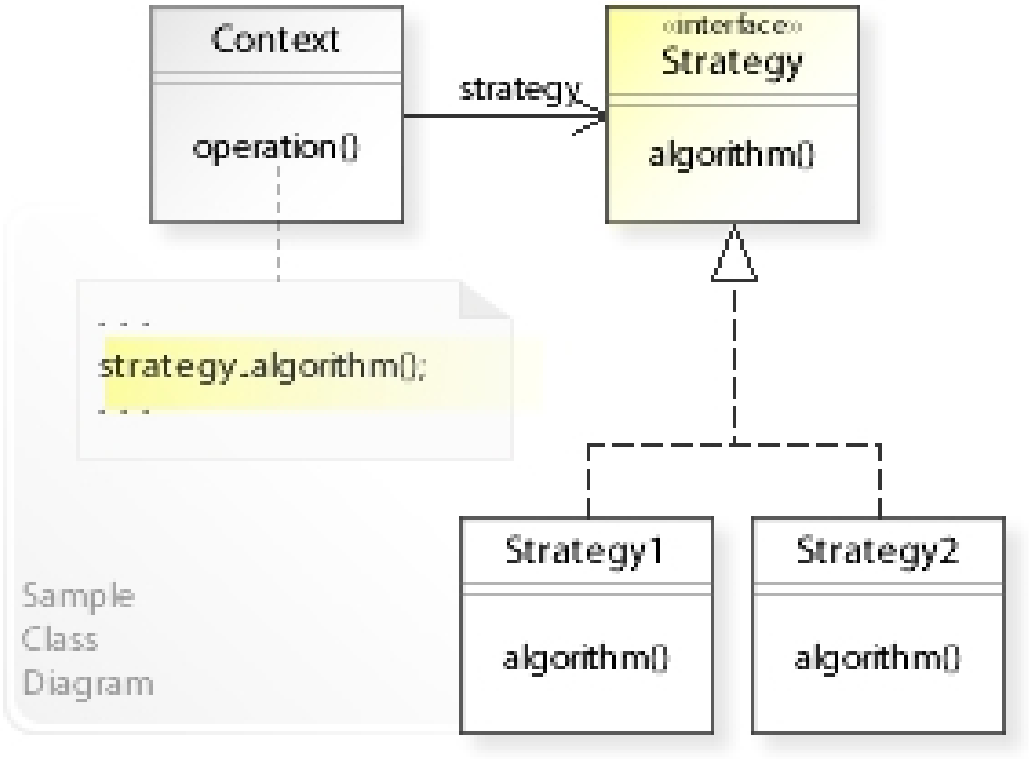
- Nombre: **estrategia, strategy, algorithm**
- Tipo: Behavioral
- Estructura:
  - Familia de algoritmos intercambiables
  - Se prepara una descripción genérica del algoritmo para que posteriormente se instancie con el que convenga
  - Se prepara la “llamada” al algoritmo que será el más conveniente en cada caso
  - Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior.
- Ej. el editor:
  - La forma de presentar el texto dependerá del algoritmo usado: sin formato, con formato, HTML, LaTeX, etc.
  - Será el algoritmo concreto que se instancie para visualizar el que determinará el resultado
  - Otras estrategias: para ordenar, clasificar, filtrar, etc...

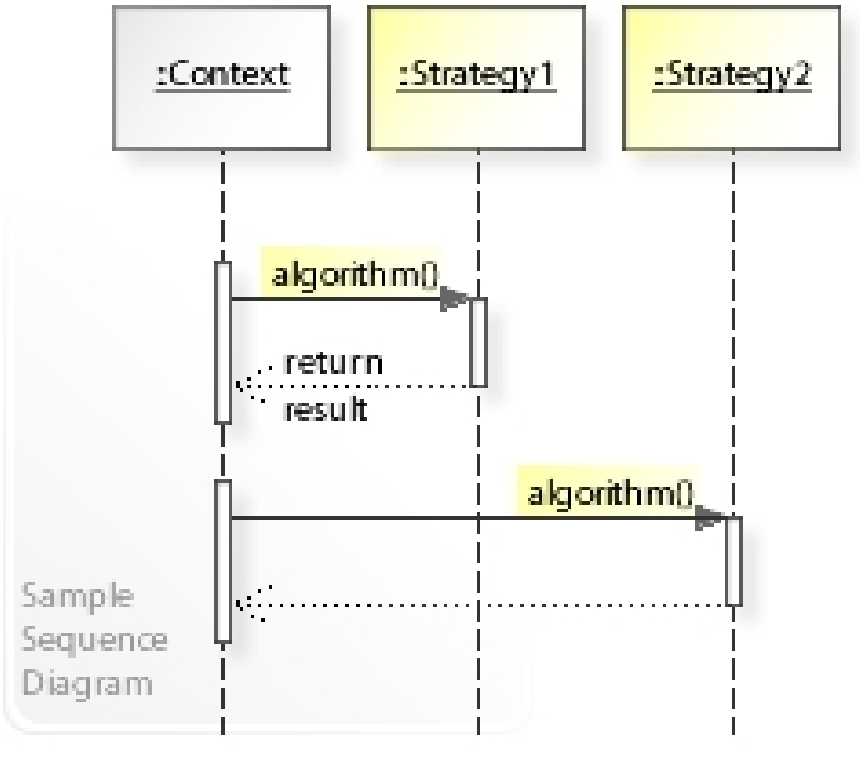
Strategy (Estrategia)

- Cliente
  - El cliente usa la interfaz a la estrategia
  - La estrategia concreta es indiferente
  - Cliente → editor
  - Estrategia → Visor HTML o LaTeX o PDF, etc

Strategy (Estrategia)

- UML class and sequence diagram

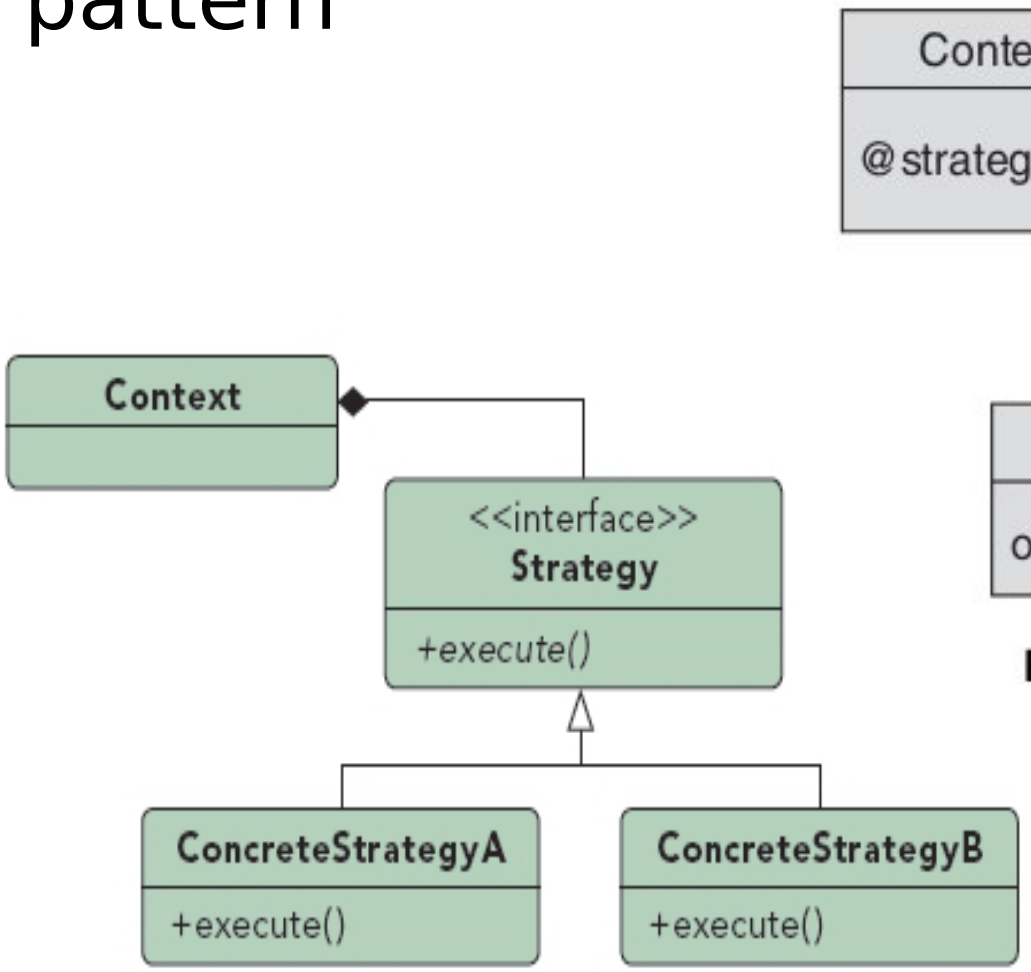




Context class: se refiere a cualquier clase en la que estamos implementando el patrón de diseño: *strategy*

Strategy (Estrategia)

- More UML class diagrams for the same pattern



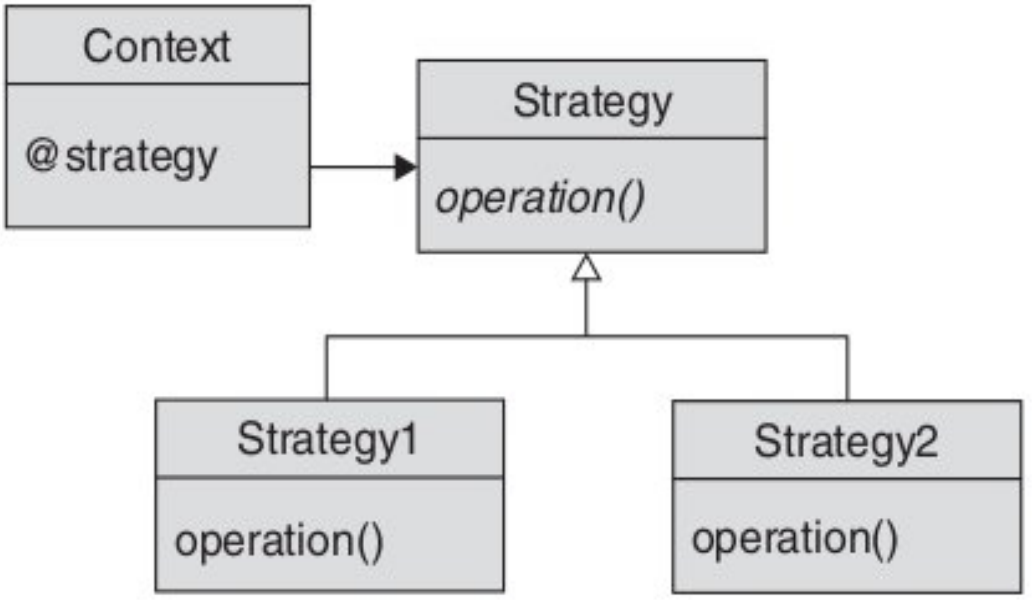


Figure 4-1 The Strategy pattern

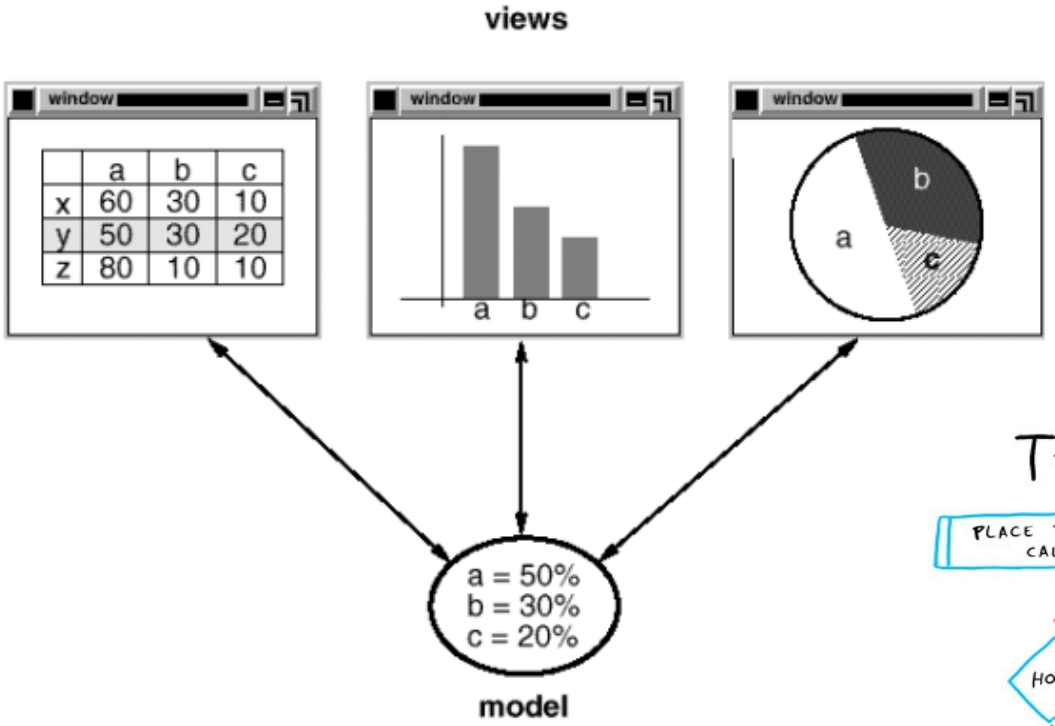
Strategy (Estrategia)

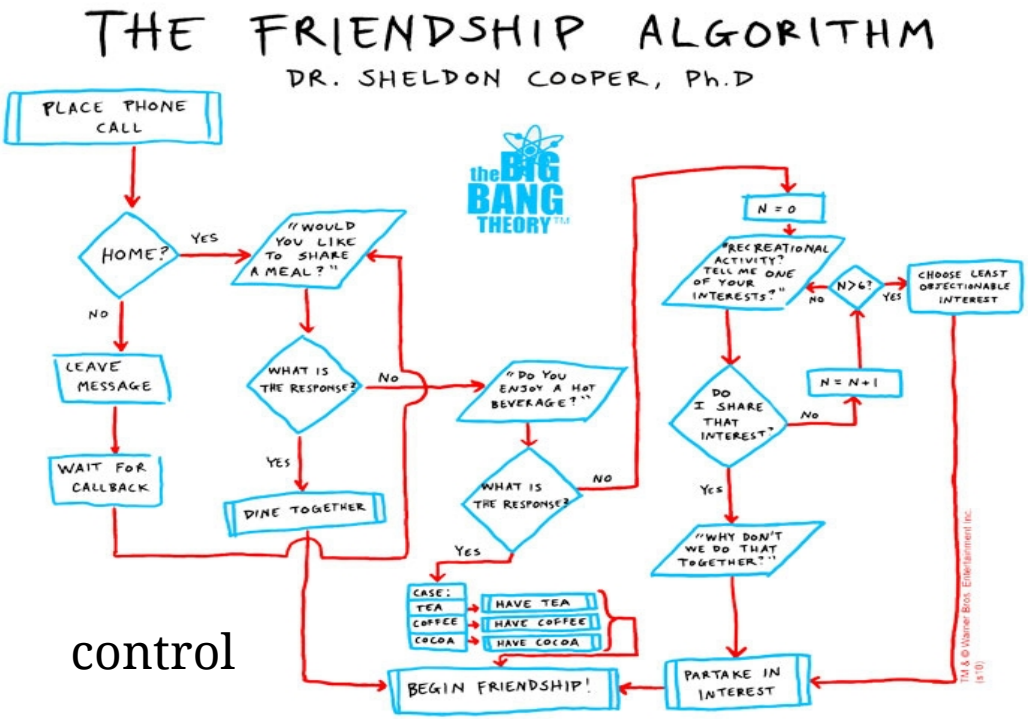
- Aplicaciones:
  - Cuando preveamos distintos comportamientos en un futuro, podemos habilitarlo
  - Estructuras de datos complejas que podrán implementarse en un futuro de otras formas
- Consecuencias:
  - Posibilidad de mejorar eficiencias y rendimientos en el futuro
  - Permitir otras estrategias de solución (más adecuadas a otros casos) distintas a la propuesta inicialmente
  - Facilitar ampliaciones

Model-View-Controller, MVC (Tríada Modelo-Vista-Controlador)

- Nombre: **MVC** (no es un patrón, es una triada de patrones)
- Tipo: Structural
- Estructura:
  - Modelo: objeto de la aplicación
  - Vista: su presentación o representación
  - Controlador: define el modo en que se reacciona ante la entrada; p. ej., del usuario
  - Usa las propiedades de varios patrones de diseño que cooperan: observer, composite, strategy, etc.
- Tiene su origen en el lenguaje Smalltalk-80

Model-View-Controller, MVC (Tríada Modelo-Vista-Controlador)





control

Model-View-Controller, MVC (Tríada Modelo-Vista-Controlador)

- Aplicaciones:
  - Presente en casi todos los *frameworks* de desarrollo modernos
  - Se puede aplicar a cualquier aplicación
- Consecuencias:
  - Simplificación del desarrollo
  - Separar desarrollos
  - Varias presentaciones para un mismo modelo.

Builder

- Tipo: Creational
- Definición:
  - Ayuda en la construcción de objetos complejos (creational)
    - Muchos atributos internos
    - Constructor con muchos parámetros
    - Que deben cumplir ciertas condiciones entre ellos
    - Resulta complejo configurar bien el objeto
    - No se puede construir el objeto en un solo paso
- Solución:
  - Definir objeto intermedio que ayude con la definición del objeto complejo

Builder

Builder se usa cuando el proceso de construcción y configuración de un objeto es muy complejo. Si no se usara el patrón “Builder” tendríamos dos opciones:

a) Constructur con múchos parámetros.

```
A obj(“param1”, “param2”, “param3”, “param4”, “param5”, “param6”, “param7”, “param8”, “param9”, “param10”, “param11”, “param12”);
```

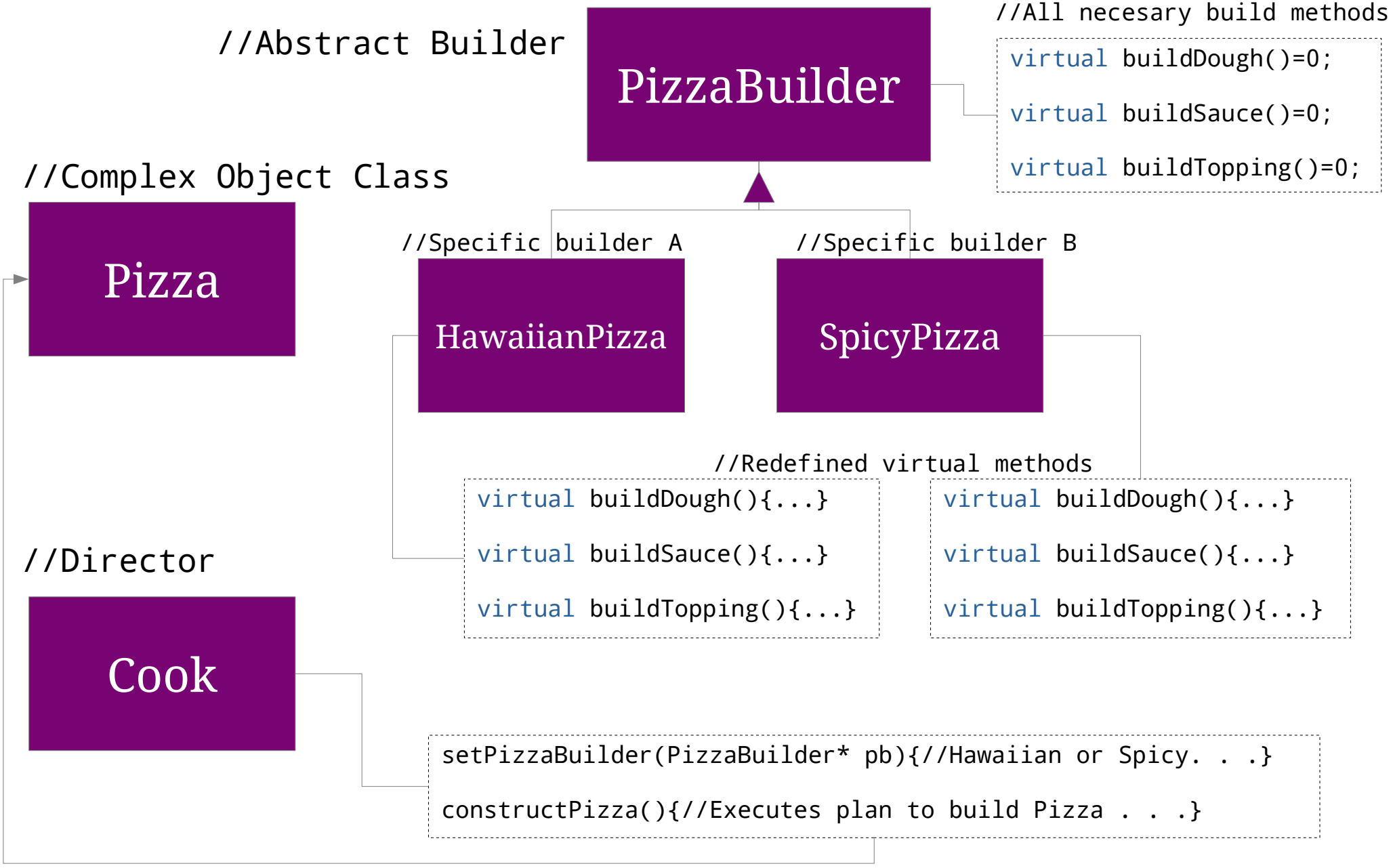
(los parámetros podrían además tener diversas restricciones y dependencias entre ellos, complicando aún más el proceso)

b) Secuencia de pasos para la construcción:

```
A.obj();
A.setParam1(...)
A.setParam2(...)
. . .
A.setParamN(...)
A.configura1(...)
A.configura2(...)
. . .
A.configuraN(...)
```

El proceso podría incluso requerir ser experto/a

builder.cc


//Abstract Builder
PizzaBuilder
//All necesary build methods
virtual buildDough()=0;
virtual buildSauce()=0;
virtual buildTopping()=0;
//Complex Object Class
Pizza
//Specific builder A
HawaiianPizza
//Specific builder B
SpicyPizza
//Redefined virtual methods
virtual buildDough(){...}
virtual buildSauce(){...}
virtual buildTopping(){...}
virtual buildDough(){...}
virtual buildSauce(){...}
virtual buildTopping(){...}
//Director
Cook
setPizzaBuilder(PizzaBuilder* pb){//Hawaiian or Spicy. . .}
constructPizza(){//Executes plan to build Pizza . . .}

builder.cc

El *Abstract Builder* (clase PizzaBuilder) va a ayudar en la construcción de objetos complejos:

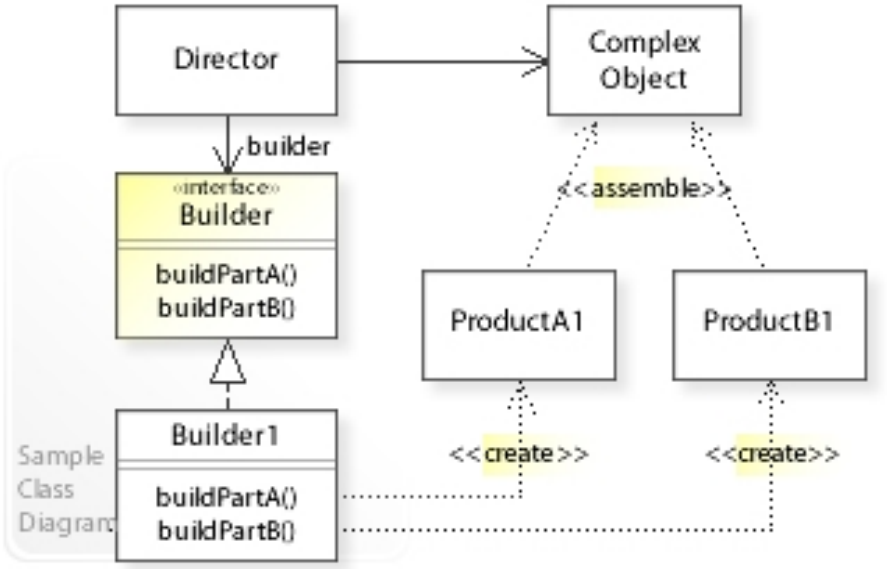
- Estableciendo la interfaz mediante funciones virtuales puras (en clase base Builder)
- Derivando de ella los *builders* concretos (HawaiianPizzaBuilder, SpicyPizzaBuilder) que serán los que se utilicen para crear fácilmente los objetos

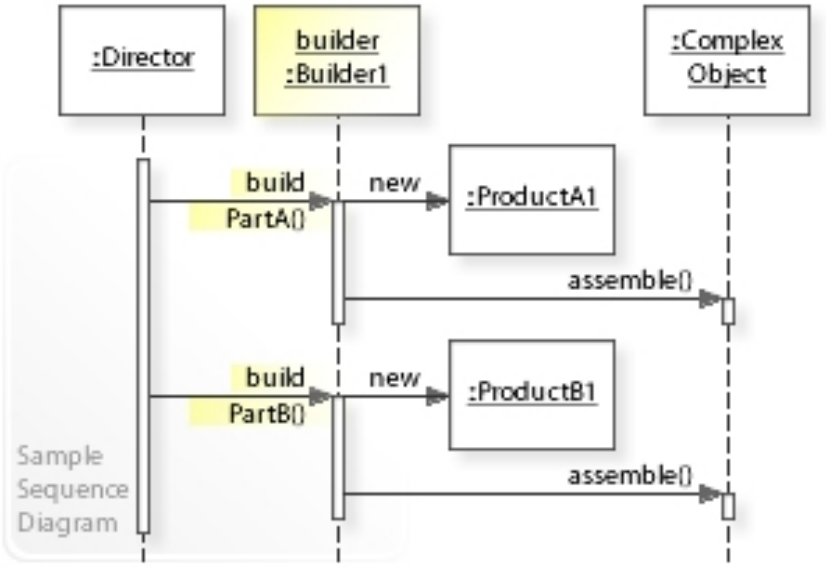
```
Cook cook;
PizzaBuilder* hawaiianPizzaBuilder = new HawaiianPizzaBuilder;
PizzaBuilder* spicyPizzaBuilder   = new SpicyPizzaBuilder;

cook.setPizzaBuilder(hawaiianPizzaBuilder);
cook.constructPizza(); // se encarga del proceso complejo
                       // de creación
Cook.getPizza();
```

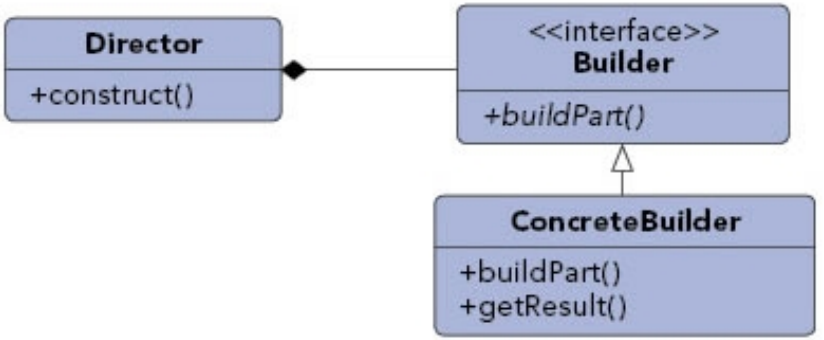
Builder

- Encapsula el algoritmo de creación de un objeto complejo.
- Hay unos pasos, un plan, que debe seguirse para crear el objeto complejo.
- Una clase abstracta “Builder” establece los pasos (funciones virtuales) de ese plan.
- De la que derivan los Builders concretos que redefinen las funciones virtuales.
- Una clase “Director” que despliega el patrón ejecutando el plan para crear el objeto concreto.

Builder







En este caso el objeto complejo necesita ProductA1 y ProductB1 de lo que se encarga el Builder1. Podrá haber tantos BuilderN como se quiera

Builder

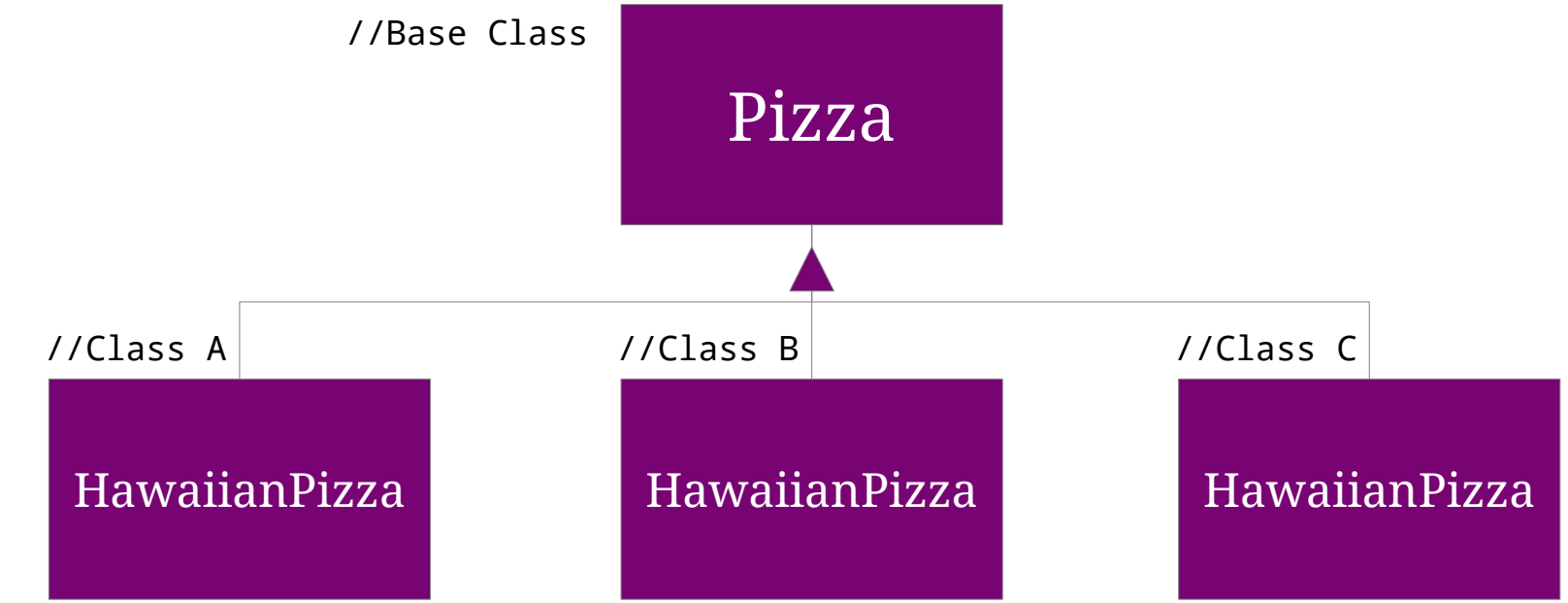
Otro ejemplo:

https://gist.github.com/pazdera/1121152

Abstract Factory

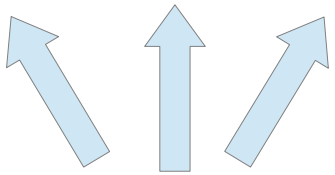
- Descripción:
  - Selecciona fácilmente objetos de una misma familia pero de distinta clase.
  - Son objetos de clases diferentes aunque derivan de la misma clase base.
  - Factory simplemente devuelve el nuevo objeto seleccionado.
- Solución:
  - Definir una clase que ayuda a la selección del objeto de una familia
  - Ejemplo: `factory.cc`

factory.cc



//Factory

PizzaFactory



```
Pizza * pizza;

pizza = PizzaFactory::createPizza(PizzaFactory::HamMushroom);
```

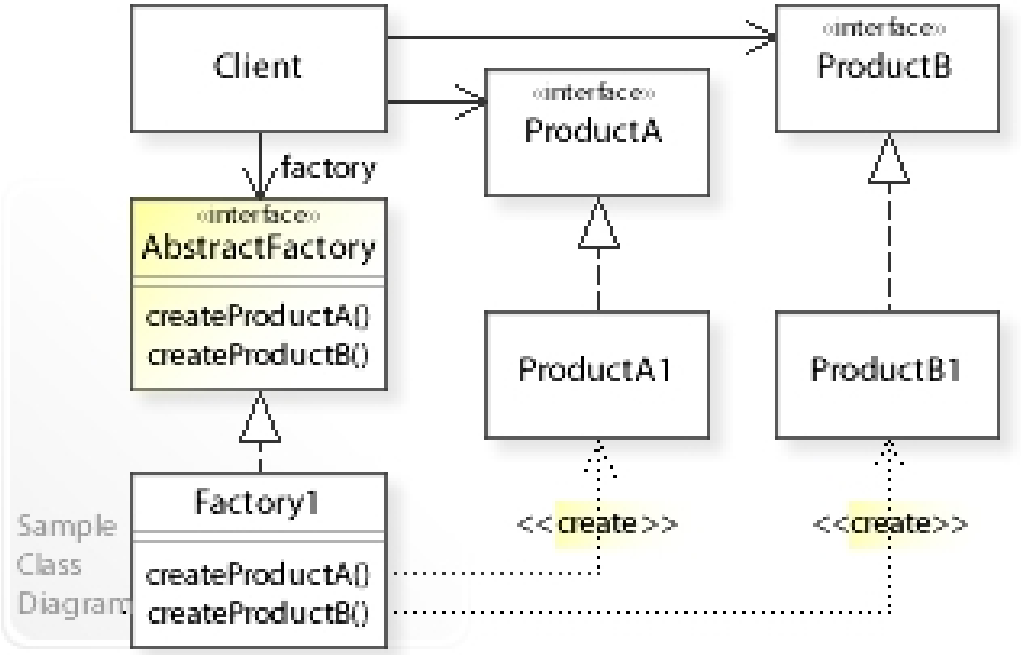
Abstract Factory

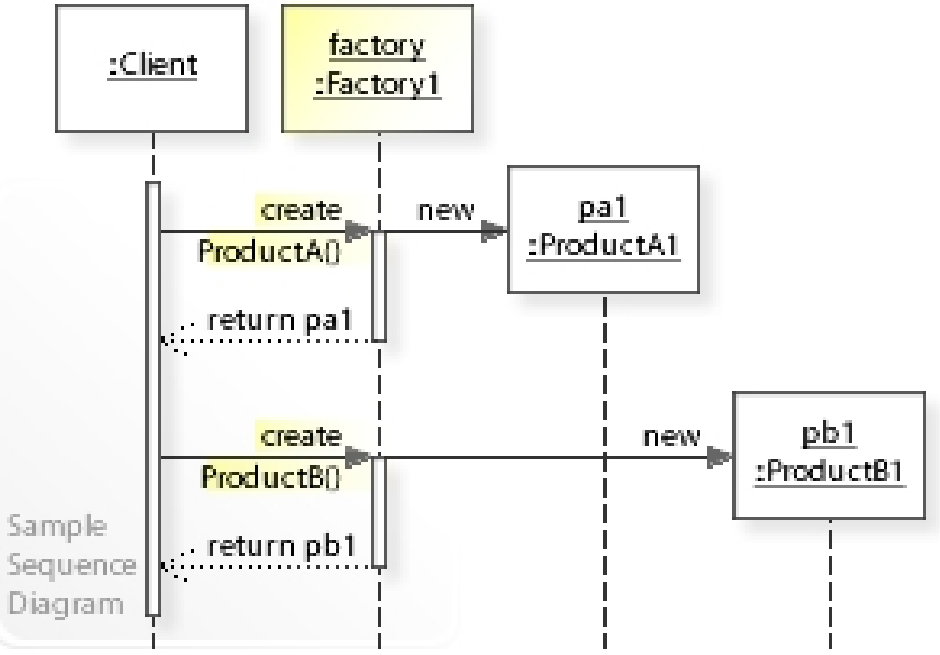
Resumen ejemplo: factory.cc

- La clase PizzaFactory es la que ayuda en la construcción de objetos de diferente tipo derivados de la misma clase.
- PizzaFactory hace la selección, en este caso en base a un parámetro.
- Las pizzas concretas (HamAndMushroomPizza, DeluxePizza, HawaiianPizza) se crean de la misma forma:

```
Pizza* pizza;

pizza = PizzaFactory::createPizza(PizzaFactory::HamMushroom);
pizza->getPrice()
. . .
pizza = PizzaFactory::createPizza(PizzaFactory::Deluxe);
pizza->getPrice()
. . .
pizza = PizzaFactory::createPizza(PizzaFactory::Hawaiian);
pizza->getPrice()
```

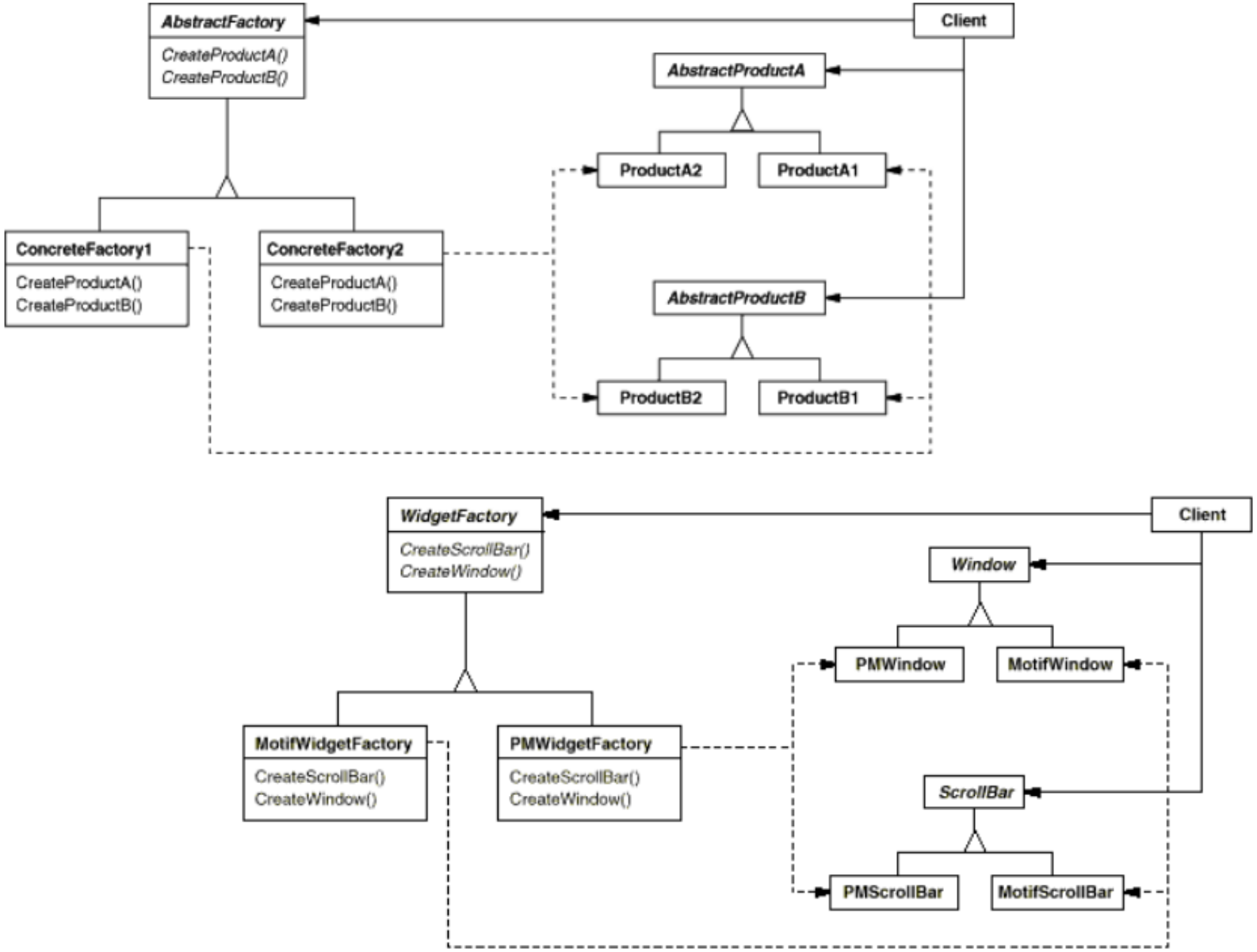
Abstract Factory





Abstract Factory

Structure



Client obtiene distintos productos según el *Factory Method* utilizado.
Ej. *ProductA1* (creado por ConcretaFactory1) y *ProductA2* (creado por ConcreteFactory2)

Abstract Factory

- Para crear objetos diferentes de diferentes clases
- Se crean en una sola y sencilla llamada
- Aunque las clases están relacionadas. Normalmente derivan de una misma clase base (misma familia de productos).
- Creating Object by using multiple factory method
- Crea objetos deferentes pero intercambiables al ser de la misma clase base
- Es parecido a builder (puede verse como una simplificación de builder...).

Singleton

- Nombre: singleton
- Descripción/estructura: Creational
  - En matemáticas, un “singleton set” o un “unit set” es un conjunto con exactamente un elemento.
  - Nos asegura que solo exista una única instancia de una clase.
  - This is useful when exactly one object is needed to coordinate actions across the system.
  - Encapsula un recurso del que solo hay una instancia y lo pone a disposición de toda la aplicación. Puede ser hardware, un servicio, un dato global, etc.
- Solución:
  - Sea cual sea su forma de creación y uso, solo debe existir una única instancia del objeto accesible fácilmente desde cualquier punto del sistema.
  - Definir la clase con métodos y datos estáticos y constructores privados.
  - Ejemplo: `singleton.cc`

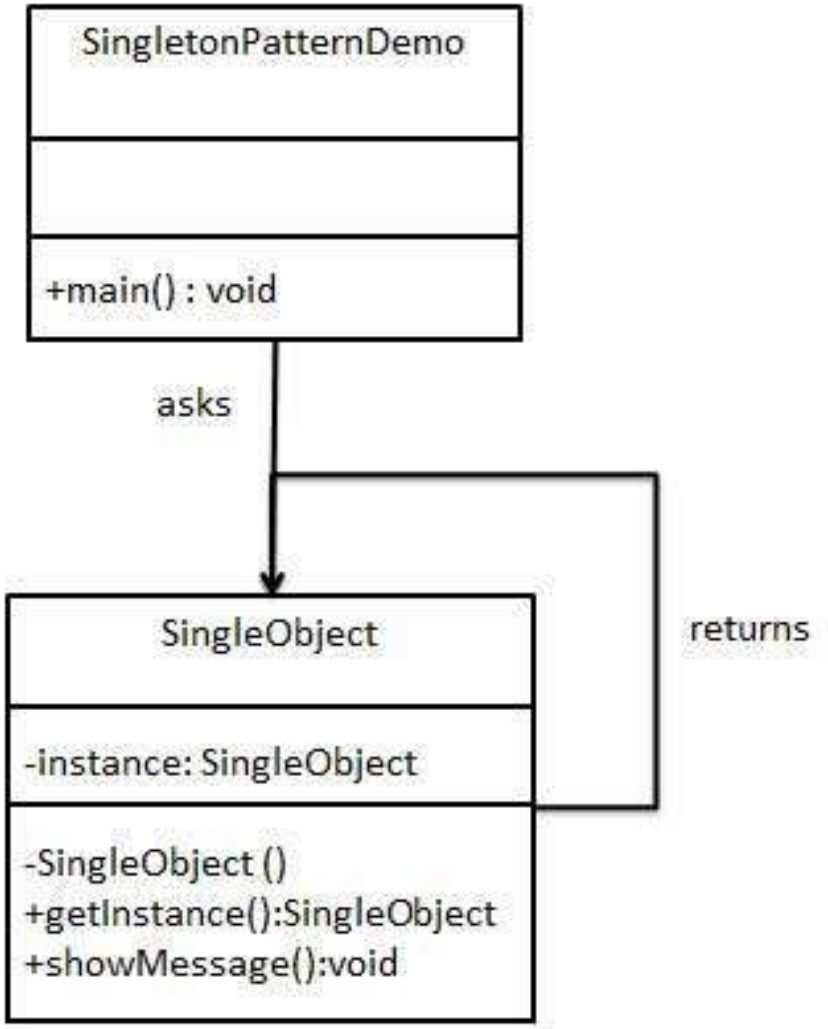
Singleton

- Ejemplos:
  - Cuando hay un único recurso (porque es un recurso escaso y solo hay uno, o porque solo puede haber uno) con el que tener interface desde distinsos lugares de la aplicación.
  - En un smartphone solo hay una touch screen para todos los objetos que la usan a la vez.
  - Configuración global del sistema.
  - Variables globales se administran mejor en una clase con una única instancia.
  - Etc.

Singleton

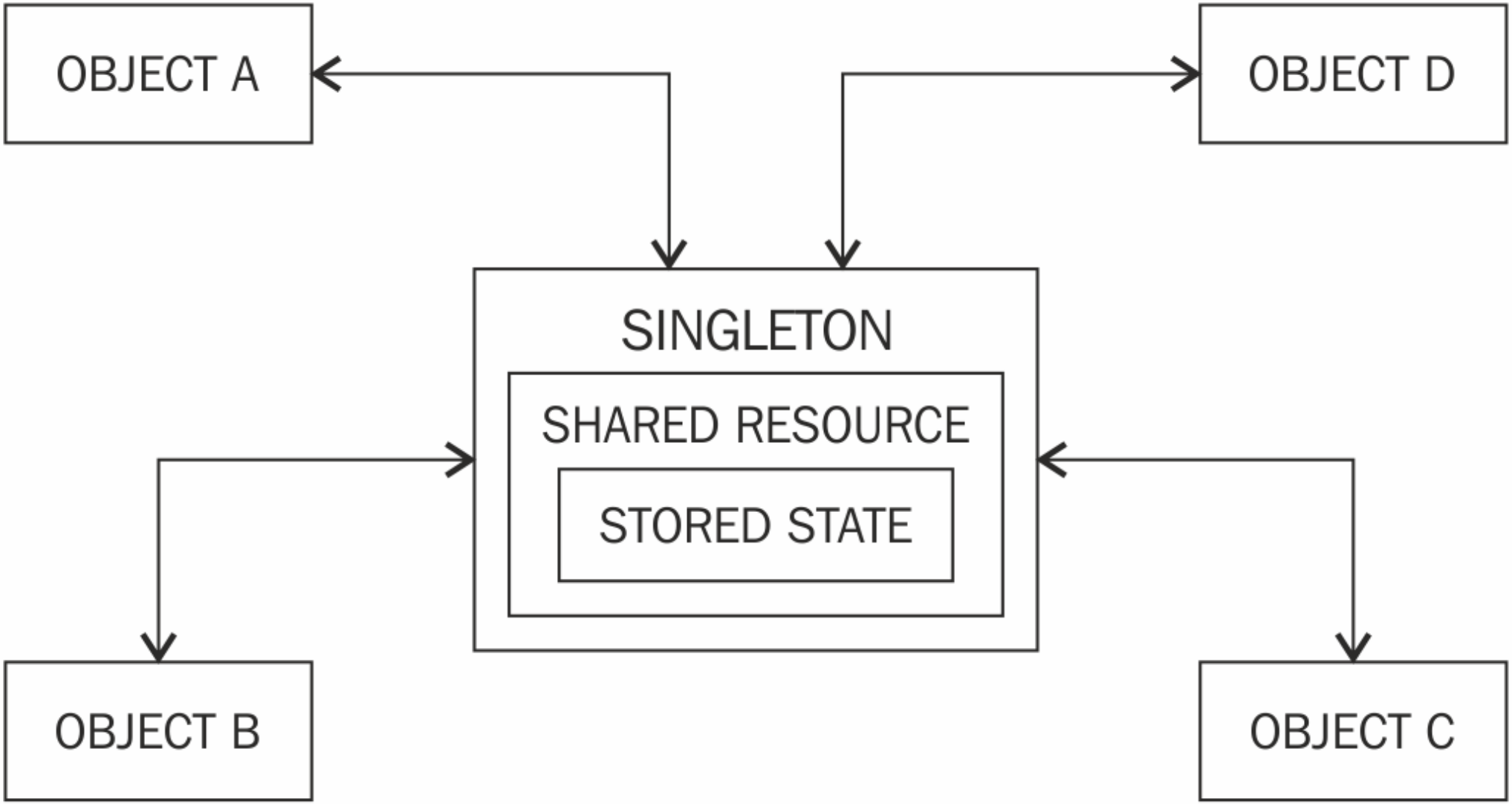
- Aplicación:
  - For application configuration
  - Configurar una aplicación o monitorizarla puede requerir un proceso complejo que con el tiempo quizá se amplie.
  - Cuando se usan variables globales pueden meterse en una struct global, pero la clase es más flexible.
- Consecuencias:
  - Garantiza una única instancia
  - Simplifica el uso de datos globales

Singleton



Cuando main(), en cualquier momento, desde cualquier otro objeto, pregunta por SingleObject, se devuelve siempre la misma instancia.

Singleton


OBJECT A
OBJECT D
SINGLETON
SHARED RESOURCE
STORED STATE
OBJECT B
OBJECT C

Resumen

- Reutilización de diseños
- Patrones de diseño:
  - Creational (**3**): Builder, Factory
  - Structural (**2**): Composite, MVC
  - Behavioral (**3**): Iterator, Observer, Strategy
- Son cada vez más usados y la tendencia actual es hacia su mayor uso cada día

References

- *Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides*. ***Patrones de diseño***. Pearson Eduicación, S.A. Núñez de Balboa 120. Madrid 2003.
- *Russ Olsen*. ***Design Patterns in Ruby.*** Addison-Wesley 2008.
- ***C++ Programming/Code/Design Patterns***. Wikibooks, Open books for an open world. http://en.wikibooks.org.
- *Code Proyect. http://www.codeproject.com/Articles/386982/Two-Ways-to-Realise-the-Composite-Pattern-in-Cplus*
- *Ejemplos de código C++: composite.cc, biulder.cc, factory.cc y singleton.cc (trabajar estos ejemplos)*
- *Design Patterns. Building Maintainable and Scalable Software. Quick Reference to the original 23 GoF design patterns. Written by Jason McDonald https://dzone.com/refcardz/design-patterns*
- *https://en.wikipedia.org/wiki/Abstract_factory_pattern*