

# Programación Orientada a Objetos

## Patrones de diseño (Design Patterns)

Diciembre 2024



UNIVERSIDAD DE CORDOBA

Juan Antonio Romero del Castillo  
Dpto. Informática y Análisis Numérico

# Design Patterns

- Esta presentación va acompañada de ejemplos en C++ colgados en la web de la asignatura.
- También se proponen algunos ejercicios adicionales opcionales.

# Design Patterns

*Don't keep reinventing the wheel !*

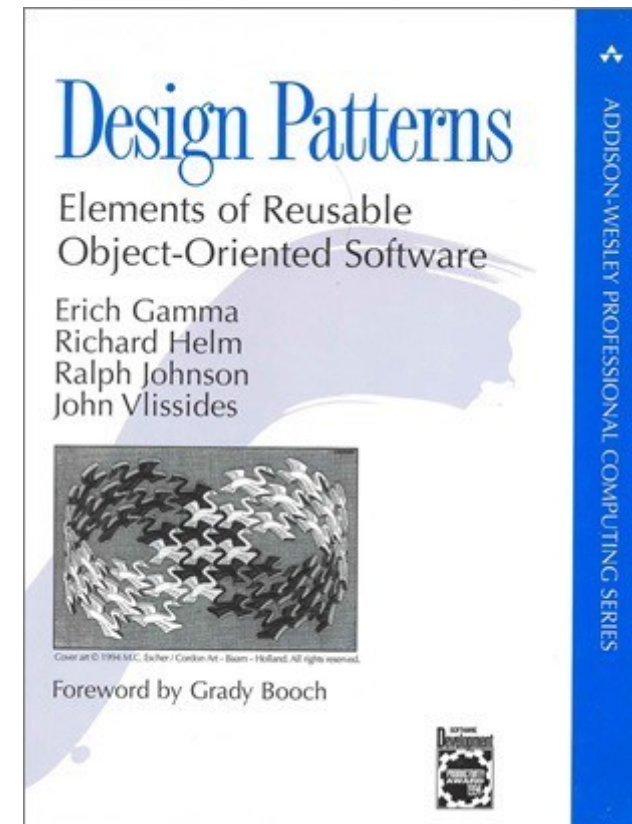
# Design Patterns

*One of the core principles of good programming is:*

*"don't solve the same problem twice".*

# Design Patterns

- Existen patrones en el diseño de software que se repiten una y otra vez
- 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 reconoce genios de la programación y se suelen llamar **Gang of Four (GoF)**



# Patrones de diseño

- Se aprecian “patrones” 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 nuestros problemas de diseño de software
- *Good and clean solution to a difficult problem*

# Patrón de diseño. Definición

- En la **"ETAPA DE DISEÑO DE SOFTWARE"**
  - Es un concepto utilizado en el desarrollo de software
  - Concretamente en la etapa de diseño del software
- “Una **"SOLUCIÓN REUTILIZABLE"** a un problema de diseño de software”
- Soluciones de diseño elegantes y de calidad a problemas complejos
- Son modelos clásicos que debemos adaptar.
- Son un esquema, un modelo, más que un conjunto de instrucciones

# Contenidos de un patrón de diseño

- Un patrón de diseño está compuesto por objetos, clases y las relaciones entre ellos (clases cooperantes).
- 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.
- Nuestros diseños futuros serán:
  - Más sencillos
  - Menos costosos
  - De más calidad

# Tipos de PD - BSC

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

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

# Elementos esenciales de un PD

Elementos esenciales que definen un patrón de diseño concreto:

- **Nombre.** Hará referencia al problema o a la solución que aporta
- **Tipo.** B/S/C
- **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

Study of the most important  
design patterns

# Template Method



Structural  
Pattern

- Principio de diseño fundamental: *Separate out the things that change from those that stay the same.*
- ¿Cómo hacemos que las partes que cambian no afecten a las que no cambian?
- Componentes básicos:
  - Template Method
  - Placeholders.
- Definimos un método en la clase base: **Template Method** que hace uso de **placeholders** que son métodos que se deben **redefinir** en las clases derivadas.
- Para procesos/funciones que cambian según el contexto. (algo muy frecuente).

# Template Method

Structural  
Pattern

- Ejemplo **1**:

- `template-method/report.cc` 
- `template-method/report-extended.cc` 
- `template-method/report-template-method.cc`



- Ejemplo **2**:

- `template-method/template-method.cc`  
con 'placeholders'



# Template Method

Structural  
Pattern

- **Nombre.** Template method
- **Tipo.** Structural pattern
- **Descripción/Estructura.** Un método (el *template method*) describe un comportamiento genérico que se concreta en las clases derivadas con *placeholders*.
- **Aplicaciones.** Que un proceso se concreta de forma diferente en distintas clases es muy habitual
- **Consecuencias.** Modificabilidad

# Template Method

- **Ejercicio adicional opcional:**

- Añadir a template-method.cc una nueva clase para el formato JSON

```
{  
  "title": "Design Pattern" ,  
  "text": "Separate Out the Things That Change from  
          Those That Stay The Same"  
}
```



# Parameterized types

Structural  
Pattern

- También llamado “generics” design pattern
- Templates en C++ (STL, plantillas de función y de clase)
- Se define un nuevo tipo sin especificar los tipos de todos sus componentes

# Parameterized types

Ejemplo 1: parameterized-types/parameterized-types.cc

```
template <class T> class MiClase{
private:
    T x_, y_;
public:
    MiClase (T a, T b){ x_=a; y_=b;};
    T div(){return x_/y_};
};

int main()
{
    MiClase <int> iobj(10,3);
    MiClase <double> dobj(3.3, 5.5);
    cout << "división entera = " << iobj.div() << endl;

    cout << "división real = " << dobj.div() << endl;
}

salida:
$ ./a.out
division entera = 3
division real = 0.6
```

C++ Class  
Template

Codificarlo como ***ejercicio adicional***  
***opcional*** añadiendo otro tipo base  
diferente.

# Parameterized types

Structural  
Pattern

- **Nombre.** Parameterized types
- **Tipo.** Structural pattern
- **Descripción/Estructura.** Se define un nuevo tipo sin especificar los tipos de todos sus componentes.
- **Aplicaciones.** Genericidad (ver STL)
- **Consecuencias.** Genericidad

# Iterator

Behavioural  
Pattern

- Las iteraciones son omnipresentes en el diseño
- Debemos pensar en ellas como patrones
- Son comportamientos muy frecuentes en todos los problemas.
- Diseñarlas bien nos reportará muchas ventajas
- Para iterar fácilmente (de forma abstracta) sin importar la estructura interna.

# Iterator example

iterator.cc

```
std::list<Player>::iterator it;

for (it = listofPlayers.begin(); it != listofPlayers.end(); it++)
{
    // Access the object through iterator
    int id = it->id;
    std::string name = it->name;

    //Print the contents
    std::cout << id << " :: " << name << std::endl;
}
```

# Iterator

Ejemplo en C++:

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

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

# El patrón Iterator facilita la abstracción

Range Based For Loop. Aquí usando punteros

```
Class Observer{...};
```

```
std::vector <Observer> observers_;
```

```
for (Observer* o : observers_) {  
    o->notify(status_);  
}
```

*Como **ejercicio adicional opcional** codificar este tipo de bucle en C++.*

# El patrón Iterator facilita la abstracción

Hay muchas formas de iterar con *Range Based For Loop*. Aquí usando referencias:

```
for (auto const& i : data) {  
    std::cout << i.name;  
}
```



# Iterator

En Python3 (por ejemplo):

```
x=[1,2,True, 4.5, "hola"]
```

```
for i in x:
```

```
    print(i)
```

```
import itertools as it
```

```
counter = it.count(start=1, step=2)
```

```
next(counter) . . .
```

```
counter = it.count(start=0.5, step=0.25)
```

```
next(counter) . . .
```

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

```
next(colors)
```

```
f=open("nombres.txt")
```

```
next(f)
```

```
etc...
```

**nombres.txt**

rita

juan

pedro

maria

...

*Como **ejercicio adicional opcional**  
probar este código en Python3.*

# Iterator

Behavioural  
Pattern

**Description:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**Tipo.** Behavioural pattern

**Applications:** collections, aggregates, containers, lists, vectors, etc.

**Consecuencias:**

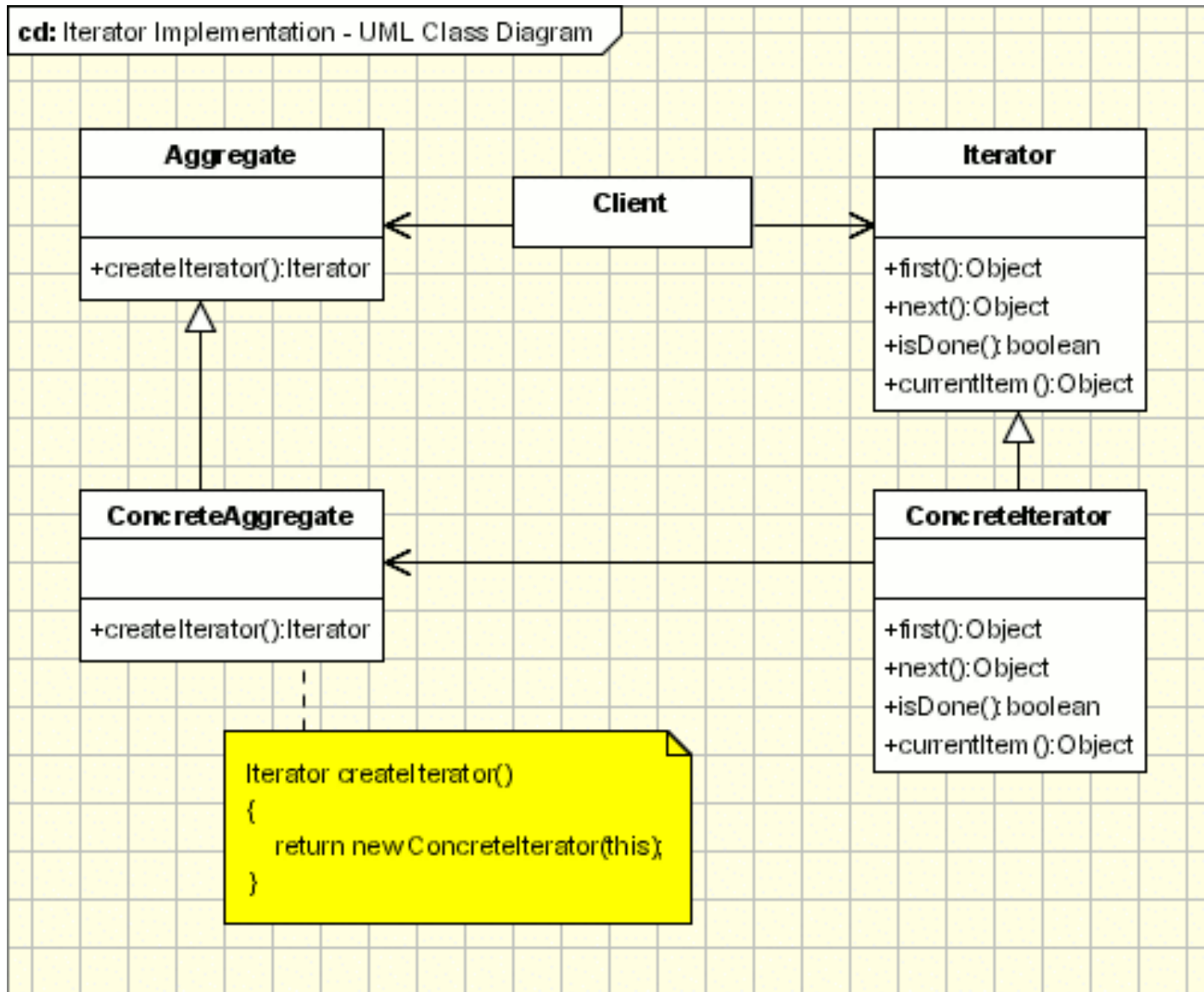
- Sencillez de uso. Facilita y favorece la abstracción
- Acceso uniforme para todas las colecciones
- Independencia de la representación interna

**Ejemplos:** C++: STL iterators. Python: for..in.. etc.

# Iterator

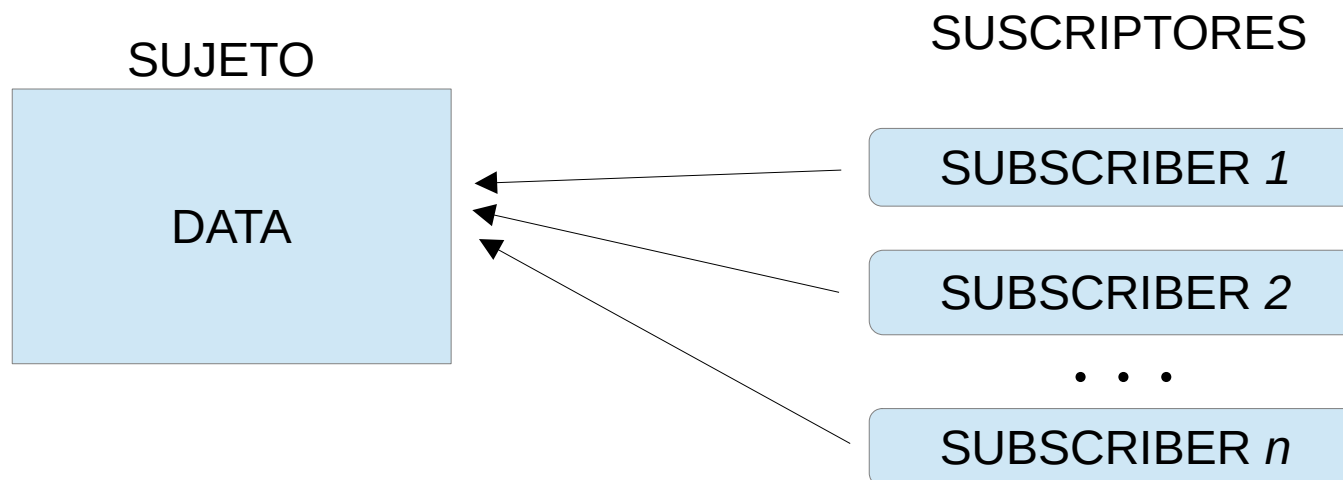
Diagrama UML del patrón de diseño Iterator.

**ConcreteAggregate** tiene un componente **iterator** que podrá recorrer los elementos de **Aggregate**.



# Observer

- Tiene varios componentes en su estructura: un objeto (EL SUJETO) y varios suscriptores (LOS OBSERVADORES).
- Suscripción: uno o varios objetos (suscriptores) dependen/se alimentan de unos datos (el sujeto).
- Esta relación **SUJETO-SUSCRIPTOR** es muy frecuente.
- Este patrón permite la **suscripción** siempre actualizada a los datos.



# Observer

Behavioural  
Pattern

- **Otros nombres:** observador, publish-susbscribe, dependents, etc.
- **Tipo.** Behavioural pattern
- **Descripción/Estructura:**
  - Estructura definida por el par: sujeto-observador
  - Sujeto: tiene los datos que se publican
    - El sujeto registra a los observadores (suscriptores).
    - En su interfaz tiene un método para comunicar cambios a los suscriptores.
  - Observador: objetos que se nutren/suscriben de/a los datos
    - Puede haber varios observadores del mismo sujeto
    - En su interfaz tiene un método para actualizarse

# Observer

Behavioural  
Pattern

- **Aplicaciones**

- Infinidad de aplicaciones tienen el esquema sujeto-observador
- Presente en el patrón de diseño MVC (lo veremos posteriormente)

- **Consecuencias**

- Ambos elementos son independientes
- Solución simple y elegante a un problema complejo
- Los observadores no tienen que ser de la misma clase
- Se pueden añadir observadores nuevos en cualquier momento sin dificultad ni cambiar nada
- Mezclar datos y observador es un ERROR de diseño

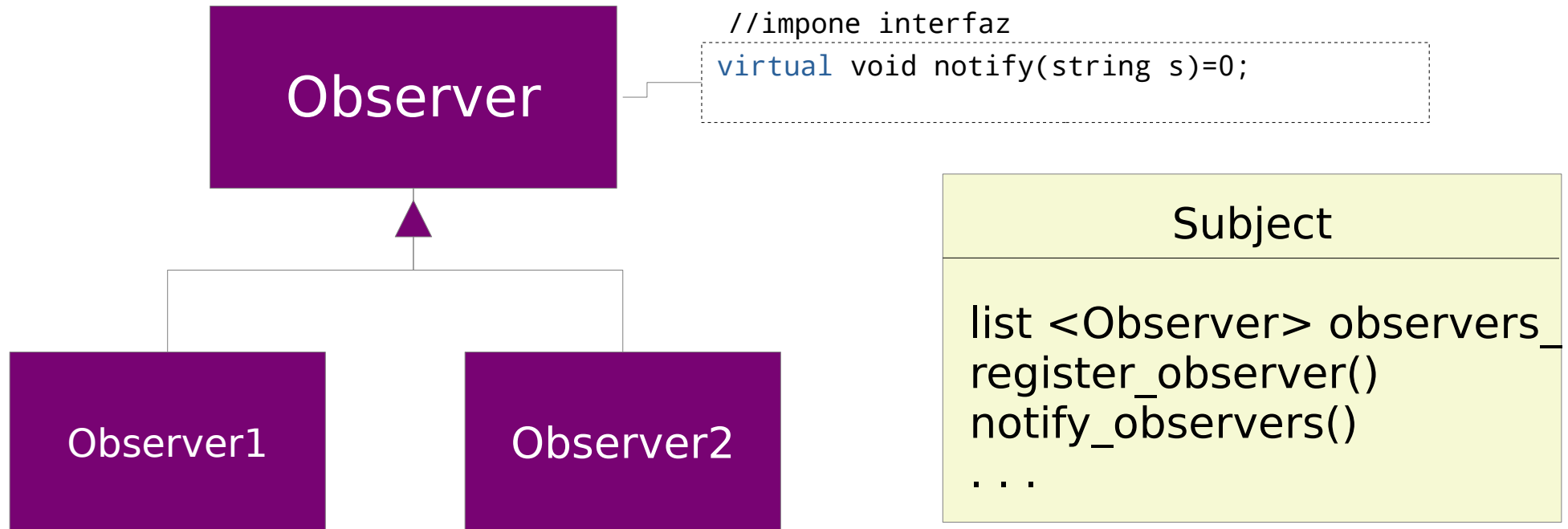
# Observer

- En general datos y visualizaciones de los datos (vistas) deben separarse
  - Datos. Es el sujeto. Una base de datos, noticias, el estado de un objeto, etc.; cualquier dato de interés.
  - Vistas (observadores/suscriptores):
    - Estadística
    - Gráfica
    - Hoja de cálculo
    - Texto
    - Etc.
  - Veremos que las vistas son un componente del patrón de diseño MVC que veremos más adelante.

# Observer. Ejemplo 1/2

observer/observer.cc

- **Sujeto:** datos = string "*status*" en la clase **Subject** que tiene además la lista de suscriptores y el método **notify\_observers()**
- **Suscriptores:** objetos **o1** y **o2**, de la clase **Observer1** y **Observer2** respectivamente que publican el método **notify(string s)** en su interfaz con el sujeto y reciben el dato como parámetro.



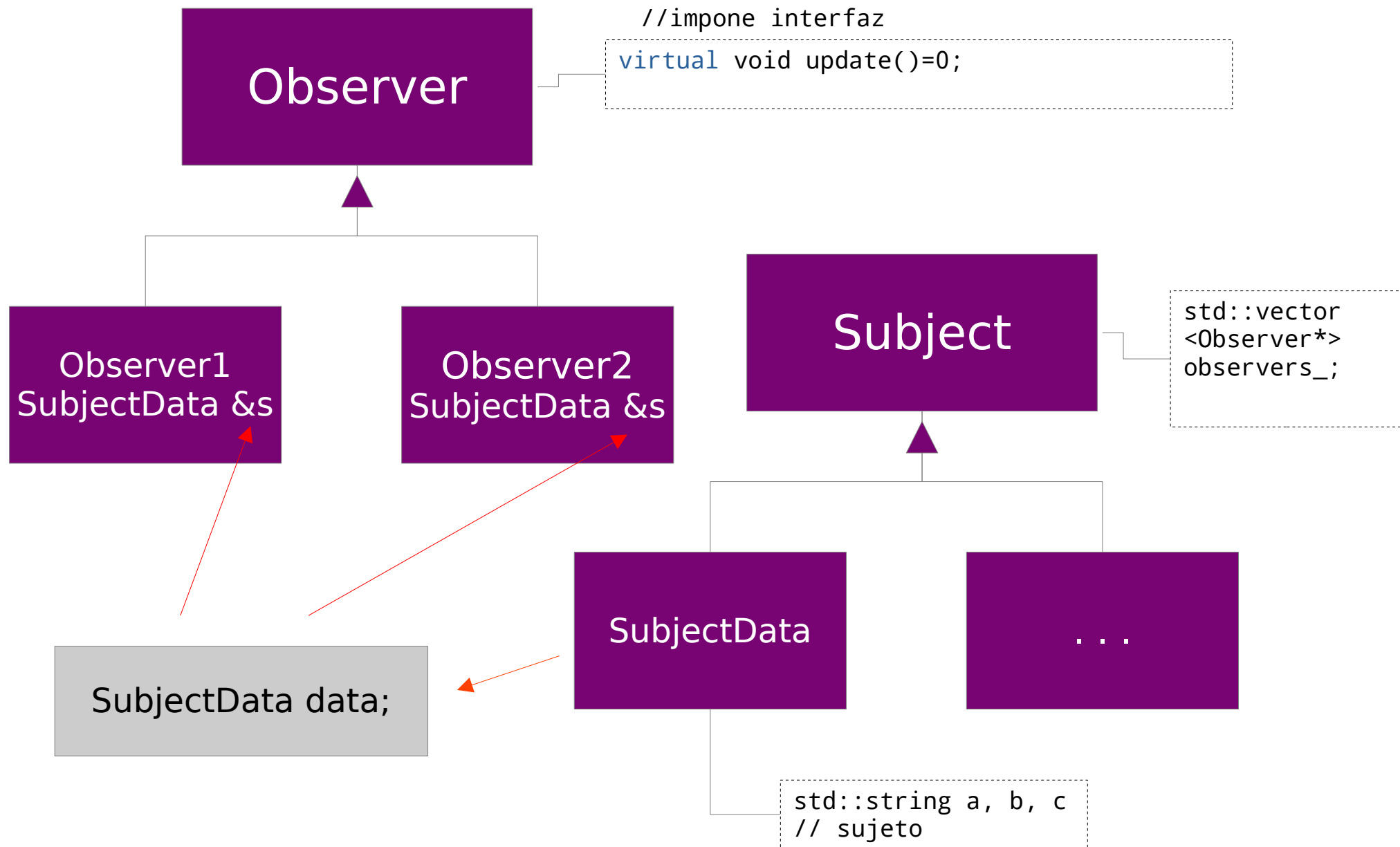


# Observer. Ejemplo 2/2

observer/observer2.cc

- Otra implementación (hay muchas).
- Con dos clases bases que faciliten la adición de de suscriptores y ahora también de sujetos: las clases Observer y Subject.
- **Mejora sustancial**: los suscriptores tienen el subject SIEMPRE actualizado mediante el uso de **una referencia**.

# Observer. Ejemplo 2/2

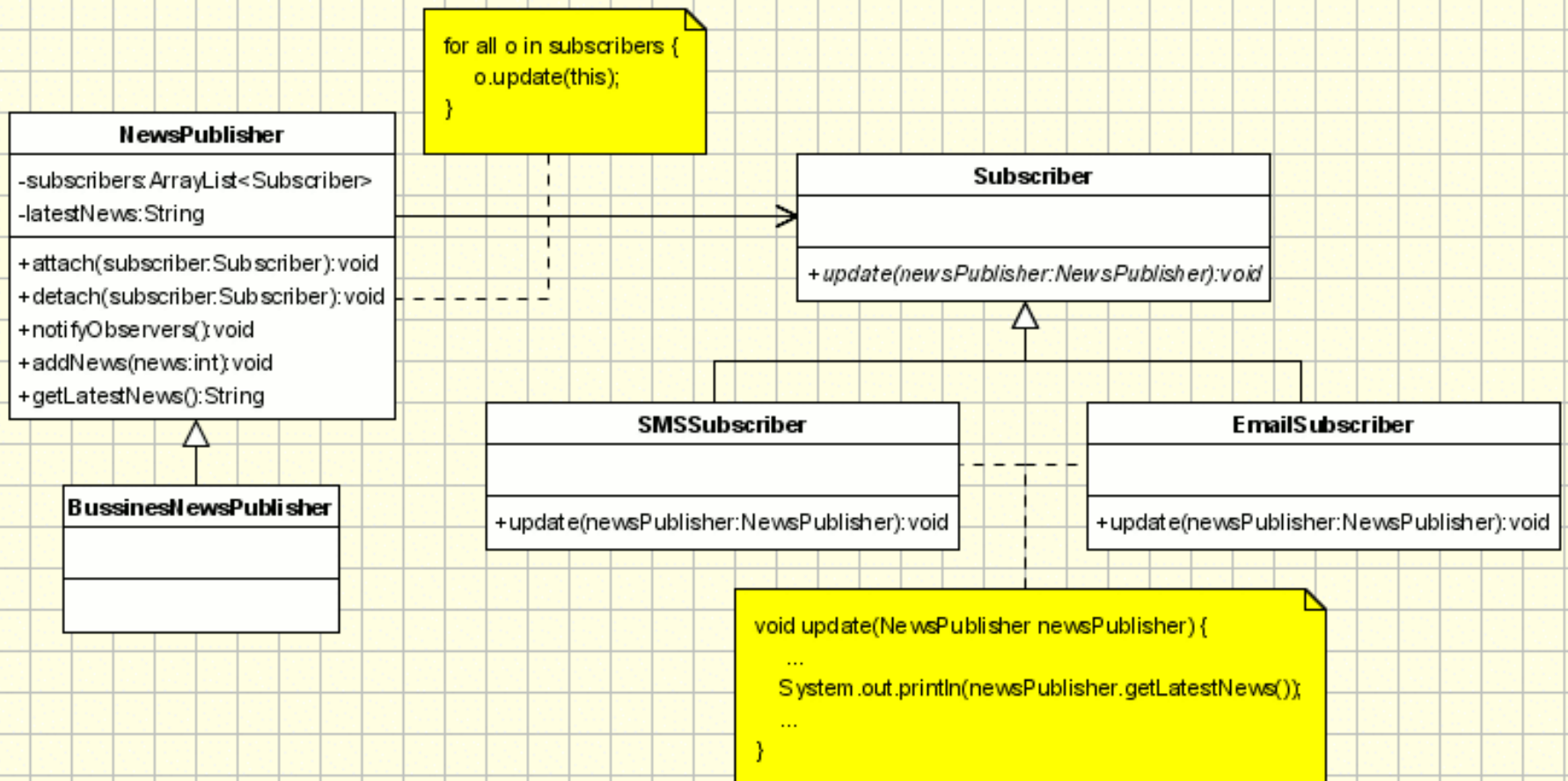


# Observer

Permite la **suscripción** siempre actualizada a los datos.

**NewsPublisher** tiene una lista de **Subscriber** que serán notificados de forma automática con `notifyObservers()`

cd: Observer Newspublisher E xample - UML Class Diagram



# Composite

Structural  
Pattern

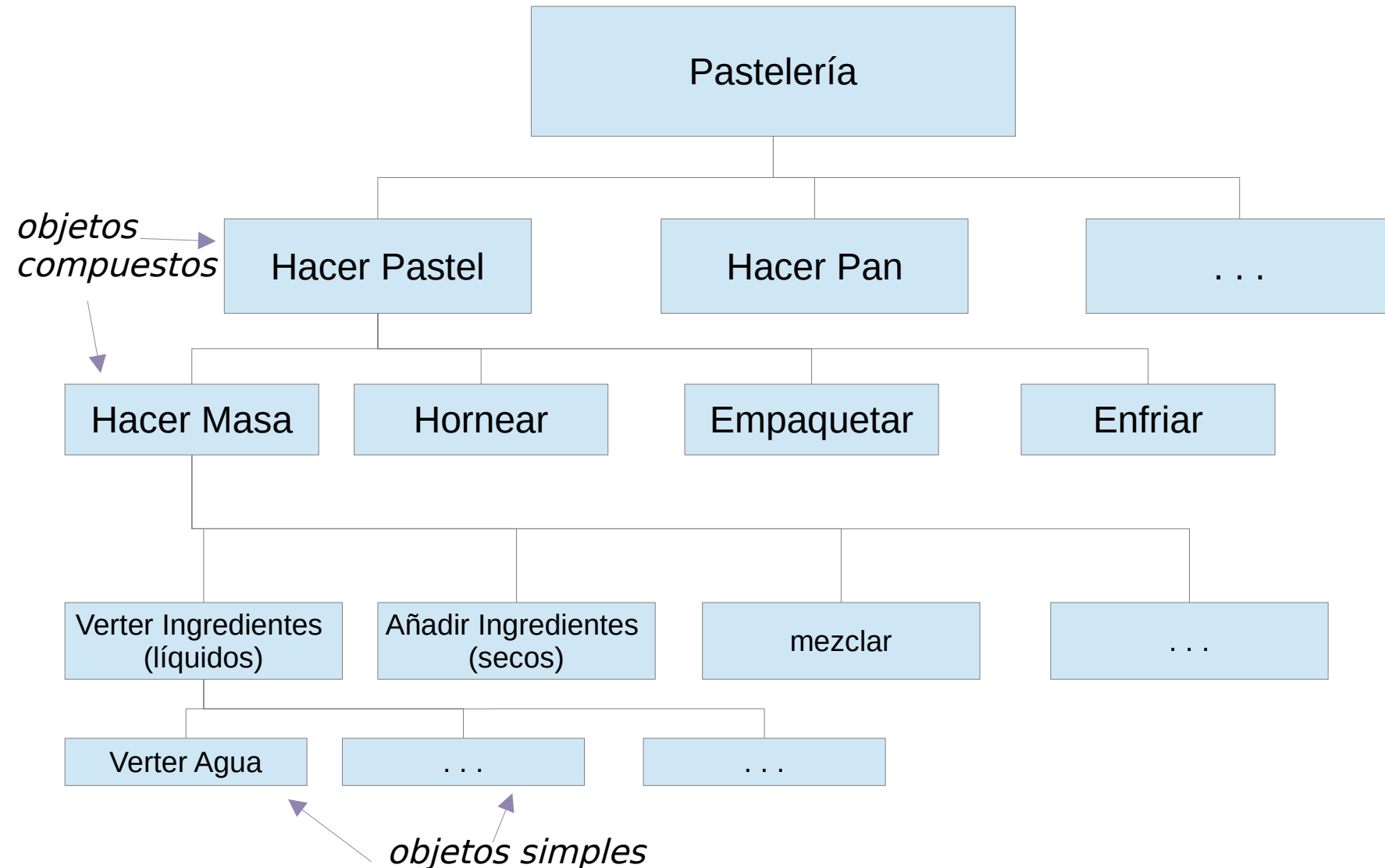
Tratamos uniformemente objetos individuales y composiciones.

Características:

- Observamos similitud entre objetos en distintos niveles del sistema.
- Solo con algunas diferencias leves entre ellas.
- Hay tareas simples, y hay tareas compuestas, pero hay cierto patrón en todas ellas.

Veamos un ejemplo...

# Composite



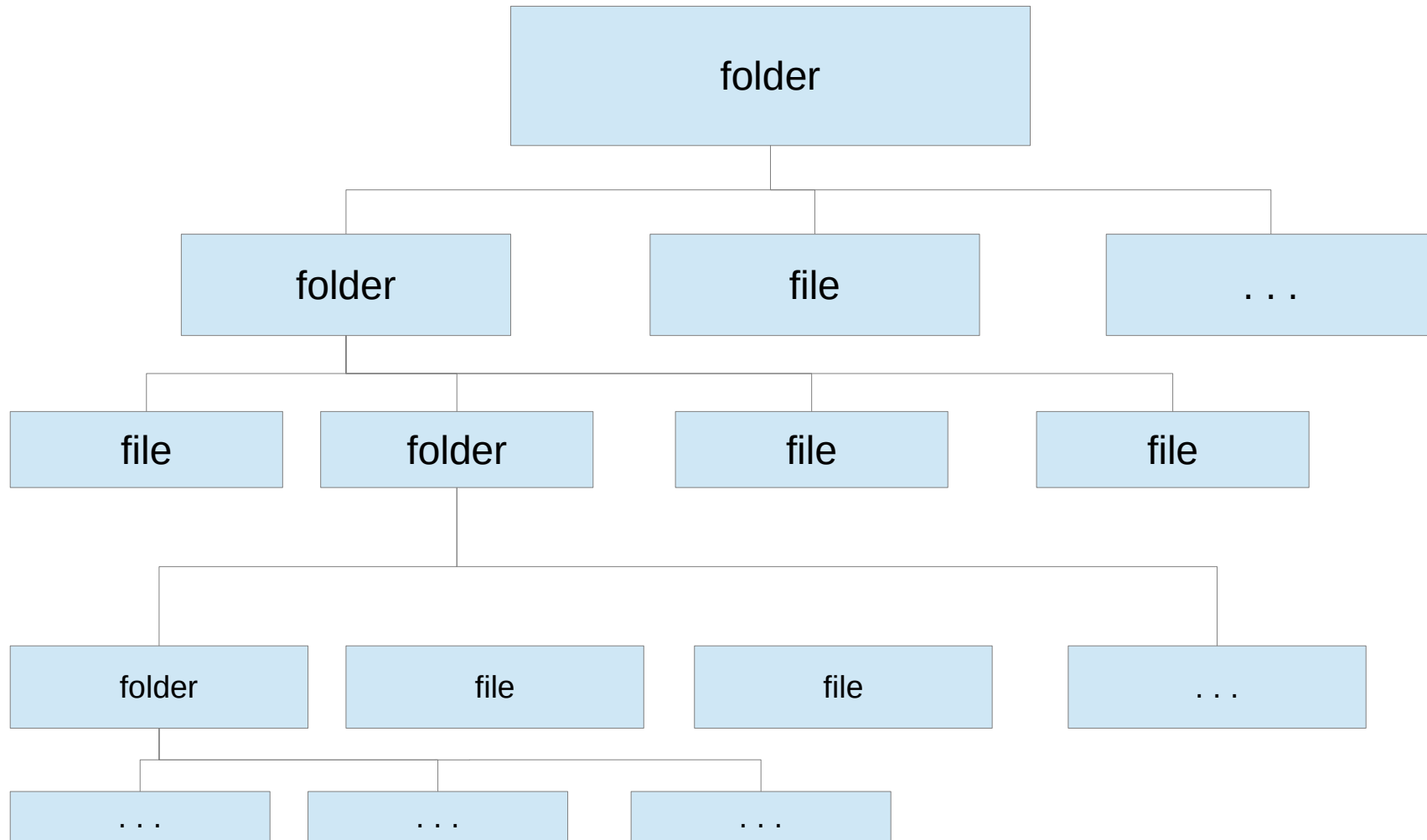
En cada nivel hay una tarea que a su vez puede estar **compuesta (composite)** por subtareas. Si creamos objeto "tarea": todas se manejarán igual y se simplifica mucho todo

# Composite

En este caso es una clara jerarquía de tareas

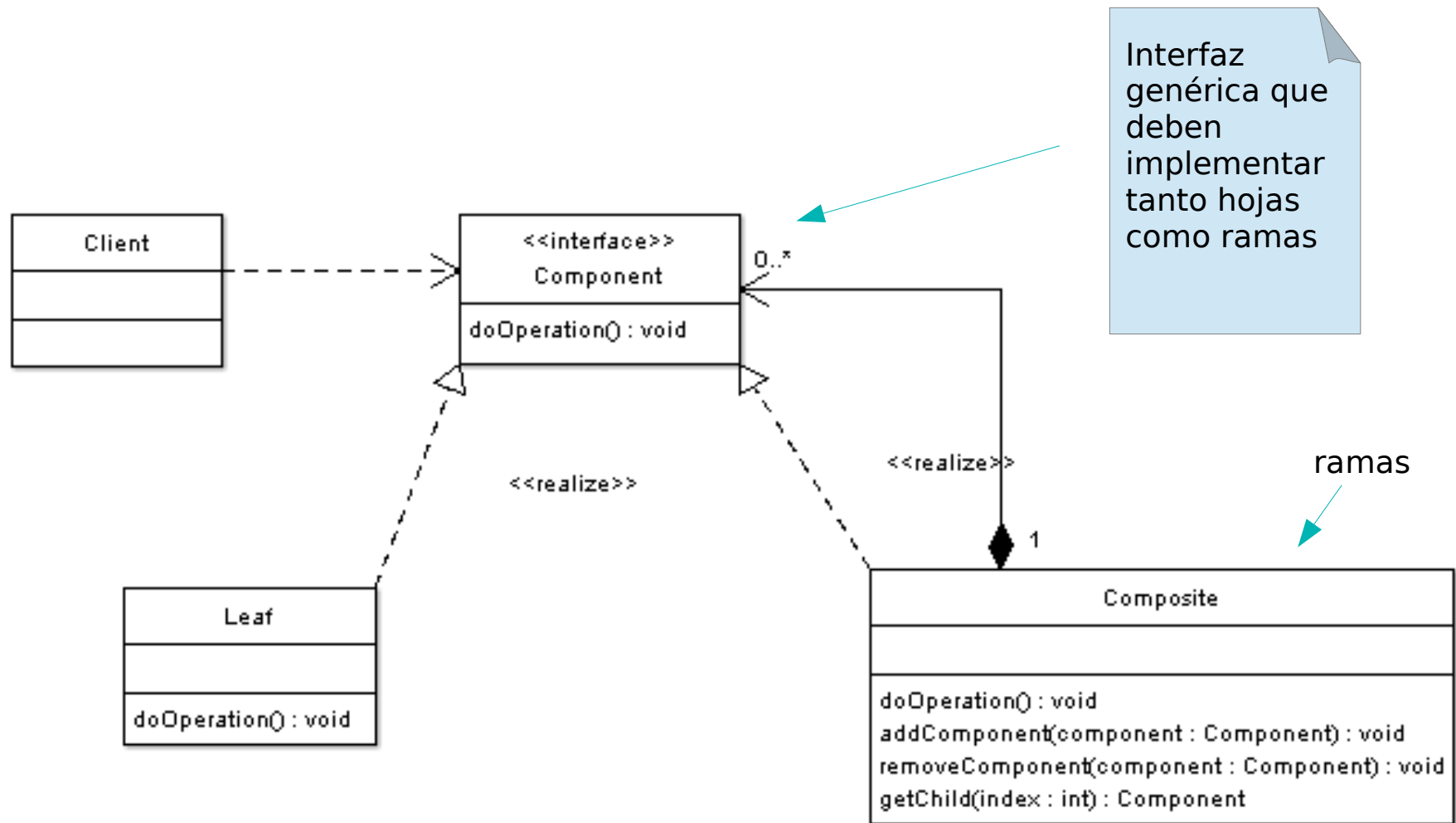
- Desde tareas compuestas: *hacerPan*, *hacerPastel*, etc.
- Hasta llegar a tareas mu simples: *verterAgua*, etc.
- Ambas comparten la misma interfaz: *tiempoTarea*, *ejecutarTarea*, *stop*, *start*, *pause*, etc.
- Es in claro ejemplo de este patrón.

# Composite



*In a file system a tree structure contains **Folders nodes** as well as **Leaf nodes** which are Files. Folder is a complex object where a file is a simple object. Files and folders have many operations and attributes in common, such as moving and copying a file or a folder, file name and size, etc... we have **the composite design pattern**.*

# Composite





# Composite

- 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

- 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.

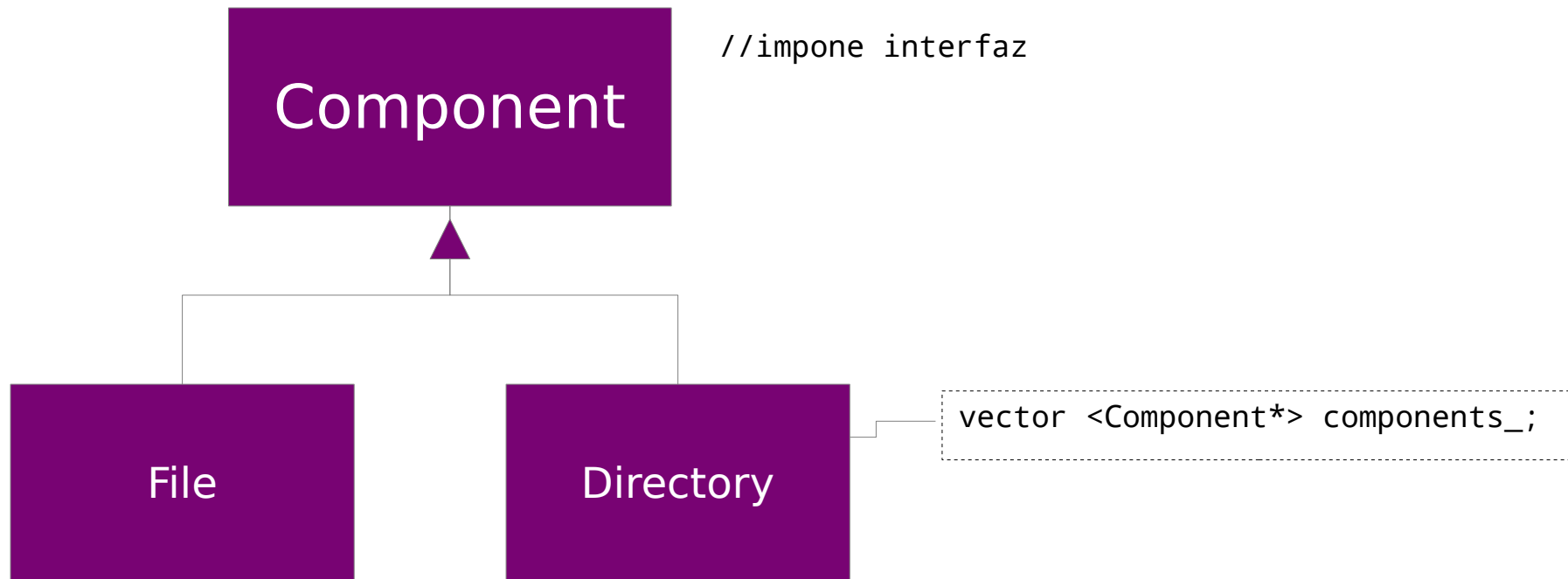
# Composite

Structural  
Pattern

- **Nombre:** objeto compuesto, composite
- **Tipo:** estructural
- **Descripción/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
  - Los objetos se tratan de manera uniforme sean primitivas o grupos
- **Aplicaciones:**
  - Allí donde se quiera simplificar la interfaz sean primitivas, sean grupos complejos de objetos
- **Consecuencias:**
  - Simplificación
  - Interfaz sencilla
  - Acceso uniforme

# Composite

- Ejemplo:
  - `composite/composite-file.cc`



# Strategy

Behavioural  
Pattern

- **Nombre:** estrategia, strategy, algorithm
- **Tipo:** Behavioural pattern
- **Estructura:**
  - Familia de algoritmos intercambiables
  - Se prepara una descripción genérica del algoritmo para que posteriormente se instancie con el que convenga.
  - Se habilita la “llamada” a un algoritmo u otro según el caso.
  - *Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior.*
- **Ejemplo:**
  - Cambiar la estrategia de ordenación según los datos.
  - Cambiar estrategias para: buscar, clasificar, filtrar, mover, etc.

# Strategy

El cliente usa la interfaz. La estrategia concreta le es indiferente:

- `objeto.busca()`
- `objeto.ordena()`
- `objeto.filtra()`
- ...

Tareas genéricas. El algoritmo concreto será el más adecuado en cada caso.

# Strategy

Ejemplo Robot behaviours:

`strategy/strategy.cc`

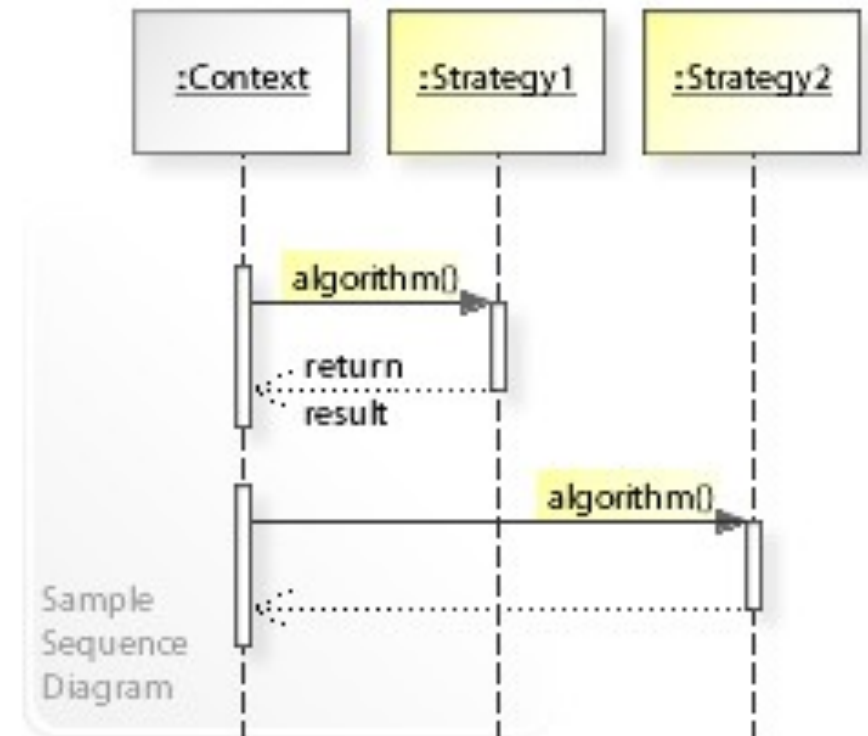
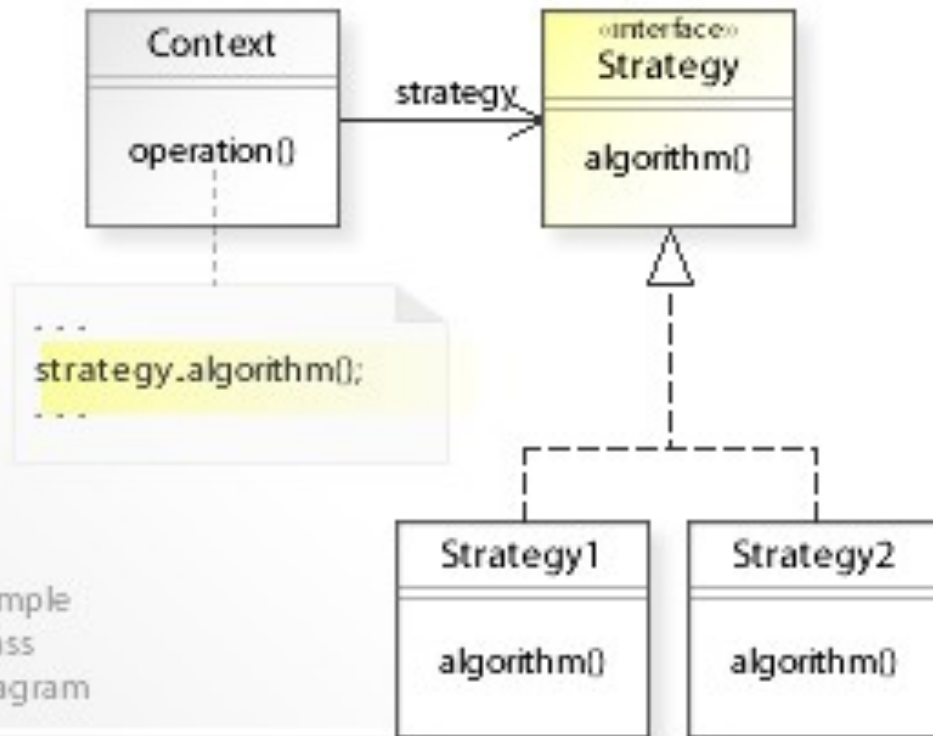
# Strategy vs Template Method

- Hay cierto parecido entre ellos.
- *Strategy* permite que un objeto cambie su comportamiento/función **en tiempo de ejecución** como en el ejemplo `strategy.cc` con la función `setBehaviour()`.
- Template Method permite que una función se desarrolle posteriormente de diferentes maneras con algunos *placeholders*.
- Strategy: intercambia el algoritmo completo.



# Strategy

- UML class and sequence diagram



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

# Strategy

- More UML class diagrams for strategy

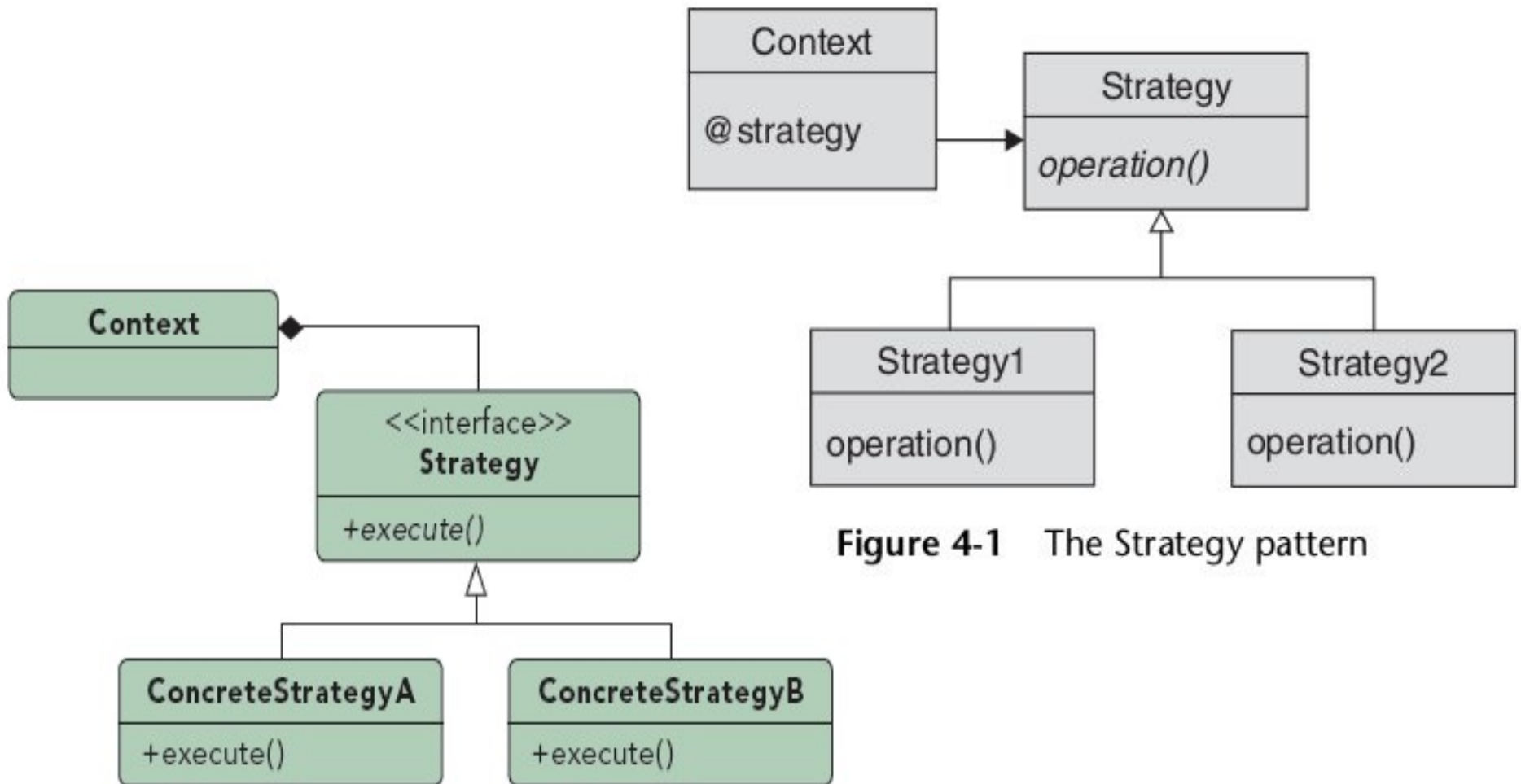


Figure 4-1 The Strategy pattern

# Strategy

- **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)

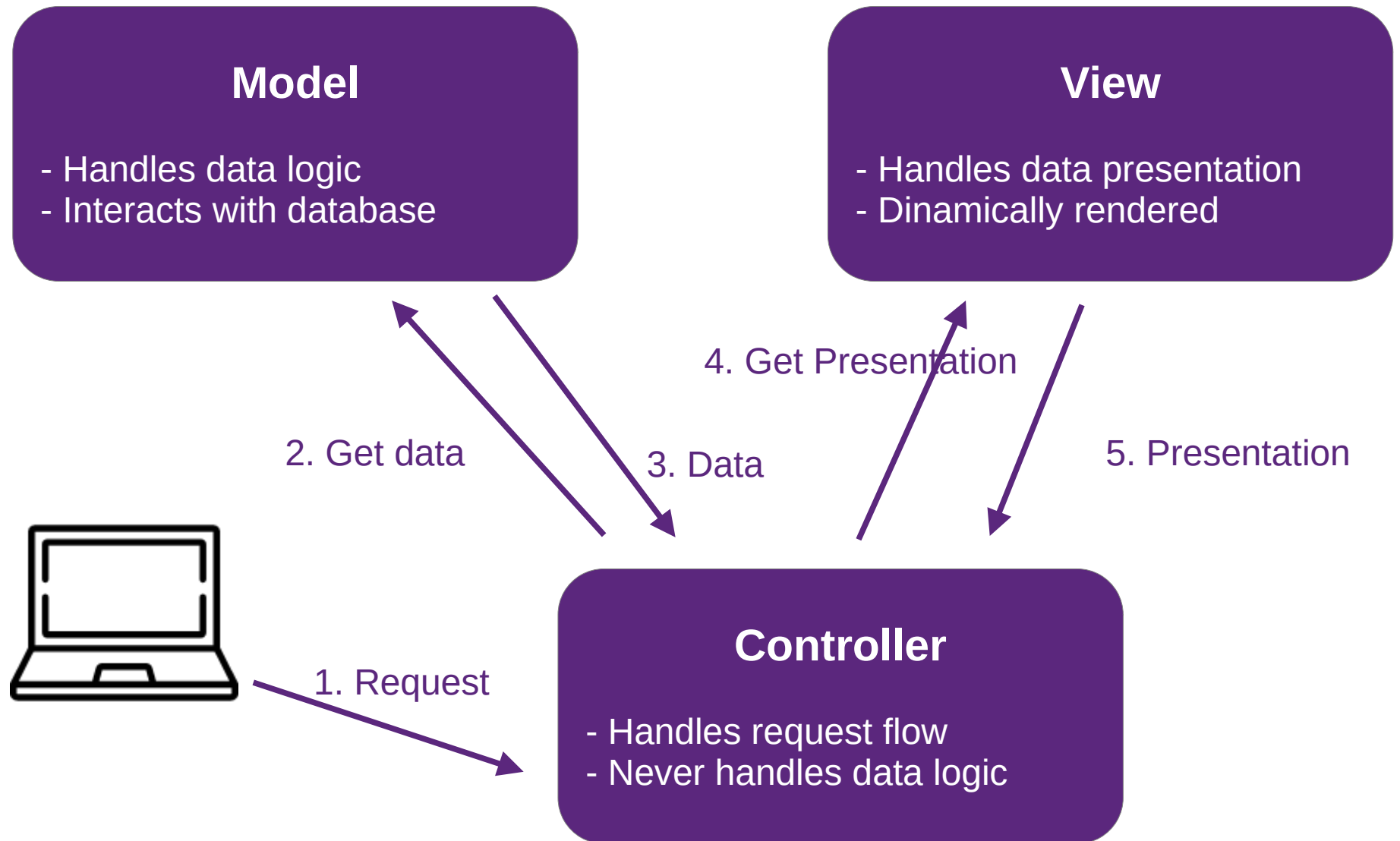
Structural  
Pattern

- **Nombre:** MVC (en realidad es una tríada 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

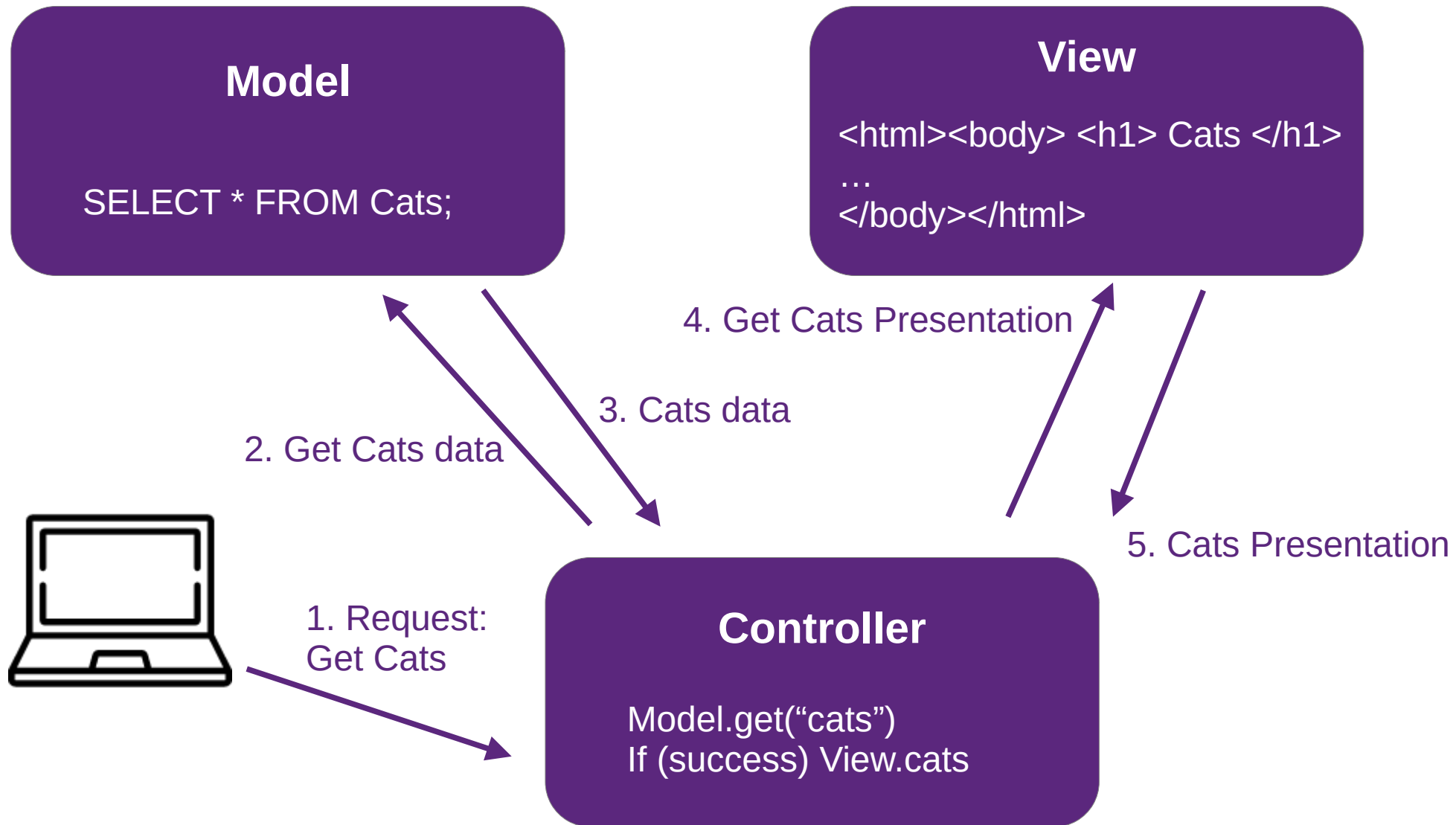
# MVC

- Se emplea siempre en el desarrollo de aplicaciones web
- Aplicaciones web complejas se dividen en 3 secciones:
  - 1. Modelo
  - 2. Vista
  - 3. Controlador

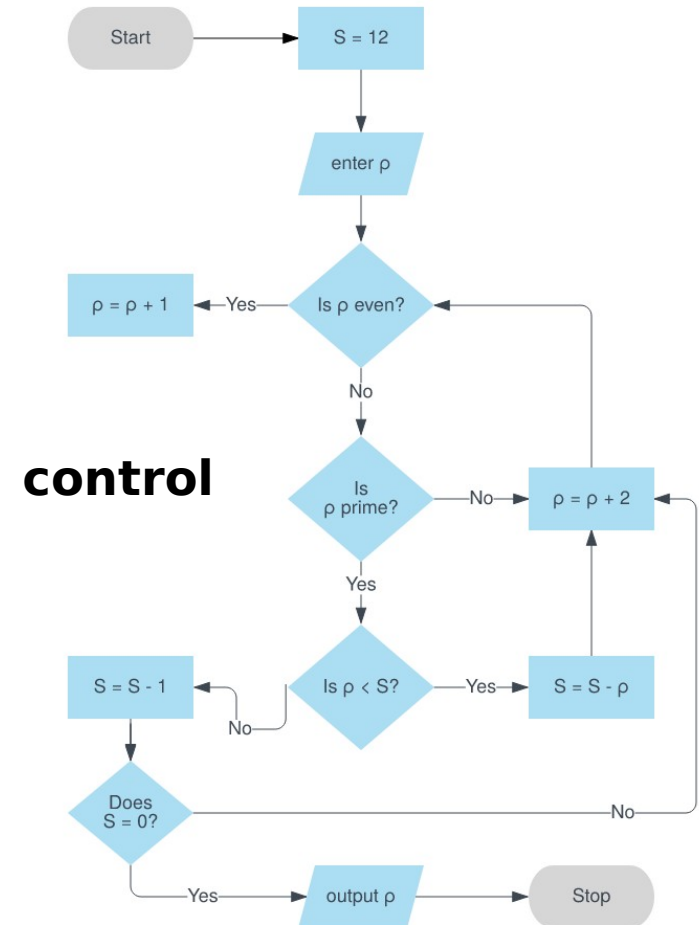
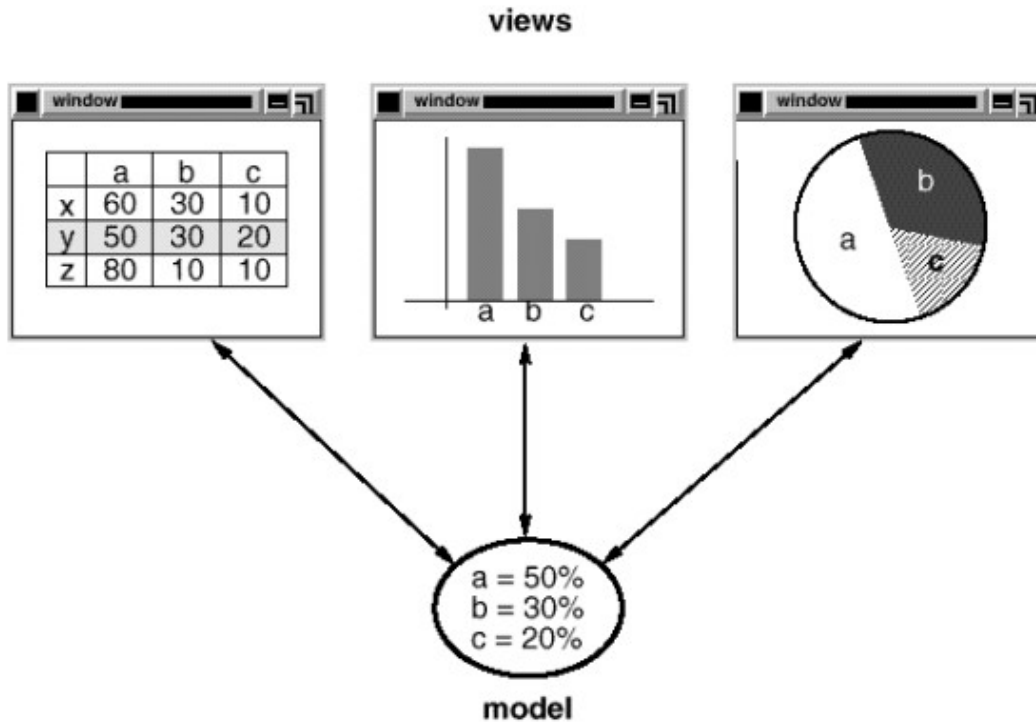
# MVC



# MVC



# MVC





# MVC

- **Aplicaciones:**

- Presente en casi todos los *frameworks* de desarrollo modernos
- Se puede aplicar a cualquier aplicación

- **Consecuencias:**

- Simplificación del desarrollo
- Varias presentaciones para un mismo modelo.
- Desacopla funciones.
- Reparto de roles (frontend, backend).

# Builder

Creational  
Pattern

- **Nombre:** builder, abstract builder
- **Tipo:** Creational
- **Definición/estructura:**
  - Facilita la creación y configuración de objetos complejos.
- **Aplicación:**
  - Cuando manejamos objetos complejos con fuertes restricciones.
  - Cuando un objeto tiene muchos atributos internos y/o parámetros que deben cumplir ciertas condiciones complejas entre ellos.
  - Cuando no se puede construir el objeto en un solo paso.
- **Consecuencias:** facilita la creación de objetos complejos.

# 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) Constructor con muchos 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

En vez de eso, para crear un objeto de esa clase (podríamos llamarla **Complex**), solo tuviéramos que hacer esto:

```
Cook p;    // Clase encargada de fabricar/'cocinar' objetos de  
           // tipo Complex
```

```
Builder1 *b1; // Construye objetos de la clase Complex  
           // con una configuración determinada.
```

```
p.setBuilder(b1);
```

```
p.build();
```

```
p.get(); // Devuelve el objeto de tipo Complex a medida.
```

```
Builder2 *b2; // Otro builder distinto
```

```
p.setBuilder(b2);
```

```
p.build();
```

```
p.get(); // Devuelve otro objeto de tipo Complex a medida.
```

# builder.cc

//Complex Object Class

Pizza

//Abstract Builder

PizzaBuilder

//Director  
(despliega el patrón)

Cook

Los 3 componentes  
del Patrón de  
Diseño Builder

# builder.cc

//Abstract Builder

PizzaBuilder

//All necessary build methods

```
virtual buildDough()=0;  
virtual buildSauce()=0;  
virtual buildTopping()=0;
```

//Complex Object Class

Pizza

//Specific builder A

HawaiianPizzaBuilders

//Specific builder B

SpicyPizzaBuilders

//Redefined virtual methods

```
virtual buildDough(){...}  
virtual buildSauce(){...}  
virtual buildTopping(){...}
```

```
virtual buildDough(){...}  
virtual buildSauce(){...}  
virtual buildTopping(){...}
```

//Director

Cook

```
cook.setPizzaBuilder(PizzaBuilder* pb){//Hawaiian or Spicy. . .}  
cook.constructPizza(){//Executes plan to build Pizza . . .}  
cook.getPizza()
```

builder.cc

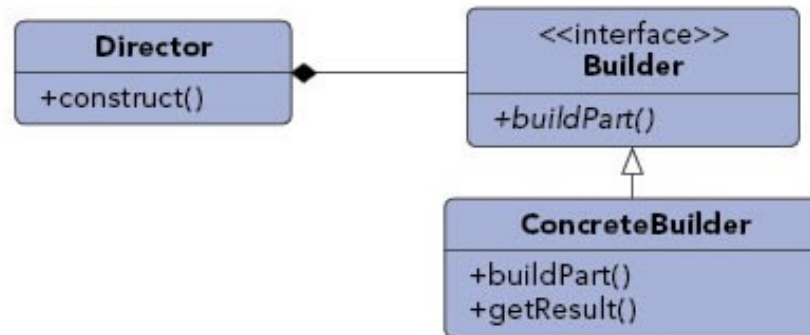
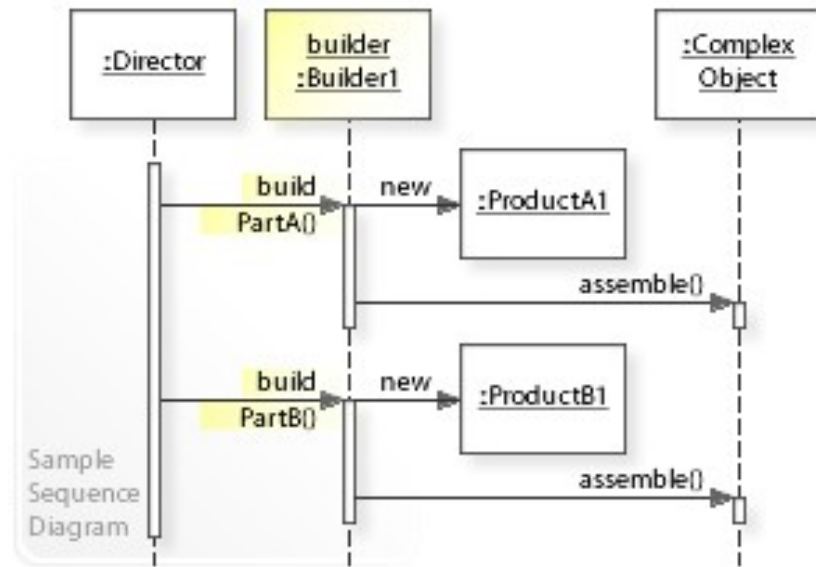
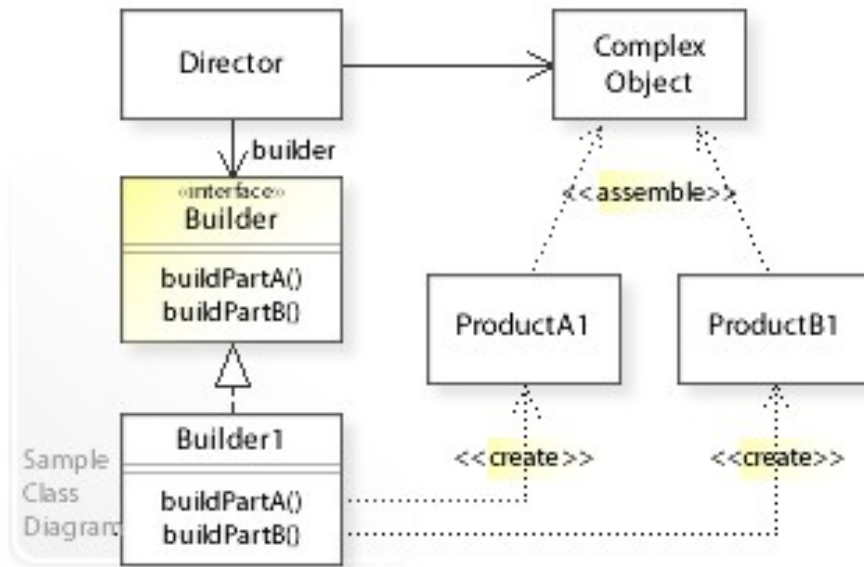
El *Abstract Builder* (clase `PizzaBuilder`) va a ayudar en la construcción de objetos complejos (de tipo `Pizza`):

- Estableciendo la interfaz mediante funciones virtuales puras (en clase base PizzaBuilder)
- 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



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



# Builder

- **Ejercicio adicional opcional:**
  - Paseo por "GitHub Gist" (*code snippets*)
  - Example of 'builder' design pattern in C++ en la URL:

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

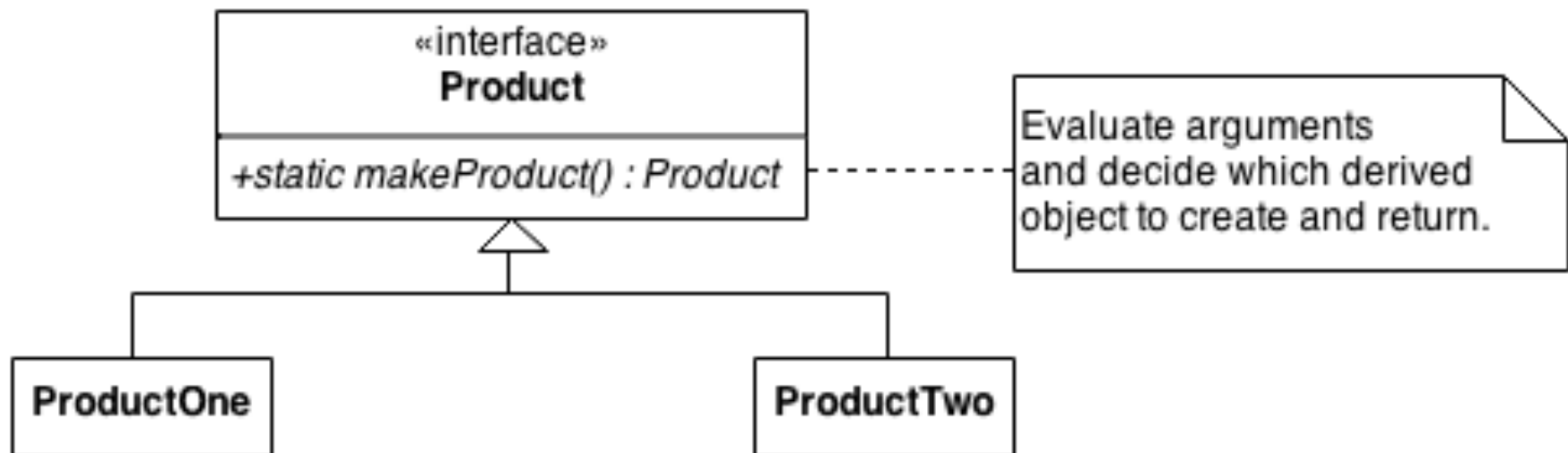
# Factory

Creational  
Pattern

- **Nombre:** Factory, fábrica...
- **Tipo:** creational
- **Descripción/Estructura:**
  - Facilita la creación de instancias de diferentes clases aunque relacionadas (pertenecientes a una misma familia).
- Existen dos implementaciones básicas:
  - 1) También mediante una *utility class* pero con un **método static**, el llamado “**Factory Method**”, que recibe una descripción del objeto deseado y devuelve un puntero de la clase base que apunta al objeto deseado.
  - 2) Mediante una *utility class* (**Abstract Factory**) que define la interfaz de cada sub-factory.  
(Hay más variantes, pero estas son las más conocidas.)
- **Consecuencias:**
  - Facilita la creación y manipulación de objetos de la misma familia.
  - Facilita la ampliación a más clases de la familia.

# Factory 1: static Factory Method

- La denominada “static Factory Method” es una variante de la anterior también muy usada.
- Esta implementación define un **método static** que hace la selección entre uno u otro objeto según el parámetro recibido.



# Factory 1: static Factory Method

```
class Computer
{
public:
    virtual void Run() = 0;
    virtual void Stop() = 0;
};

class Laptop: public Computer
{
public:
    void Run() override {mHibernating = false;};
    void Stop() override {mHibernating = true;};

private:
    bool mHibernating; // Whether or not the machine is hibernating
};

class Desktop: public Computer
{
public:
    void Run() override {mOn = true;};
    void Stop() override {mOn = false;};

private:
    bool mOn; // Whether or not the machine is ON
};

main()
{
    Computer *c;
    switch (choice)
    {
        case "laptop":
            c = new Laptop;
        case "desktop":
            c = new Desktop;
    }
}
```

factory-computer.cc



main() depende de las clases derivadas que haya

# Factory 1: static Factory Method

factory-computer.cc

```
class Computer
{
public:
    virtual void Run() = 0;
    virtual void Stop() = 0;
    virtual ~Computer() {} /* Without this, Laptop or Desktop destructor can't be called with a pointer to Computer */
};

class Laptop: public Computer
{
public:
    void Run() override {mHibernating = false;}
    void Stop() override {mHibernating = true;}
    virtual ~Laptop() {} /* because we have virtual functions, we need virtual destructor */
private:
    bool mHibernating; // Whether or not the machine is hibernating
};

class Desktop: public Computer
{
public:
    void Run() override {mOn = true;}
    void Stop() override {mOn = false;}
    virtual ~Desktop() {}
private:
    bool mOn; // Whether or not the machine has been turned on
};

class ComputerFactory    // Utility Factory Class
{
public:
    static Computer *newComputer(const std::string &description) //Factory Method
    {
        if(description == "laptop")
            return new Laptop;
        if(description == "desktop")
            return new Desktop;
        return nullptr;
    }
};

// static pues es una simple función de selección.
```

```
main() // No depende de las clases derivadas
{
    Computer *c;
    description = . . . ;// "laptop" o "desktop"

    c = ComputerFactory::newComputer(description);
    . . .
}
```



Mejor usando Factory Method

# Factory 1: static Factory Method

AÑADIMOS NUEVA CLASE EN UN  
TIEMPO POSTERIOR

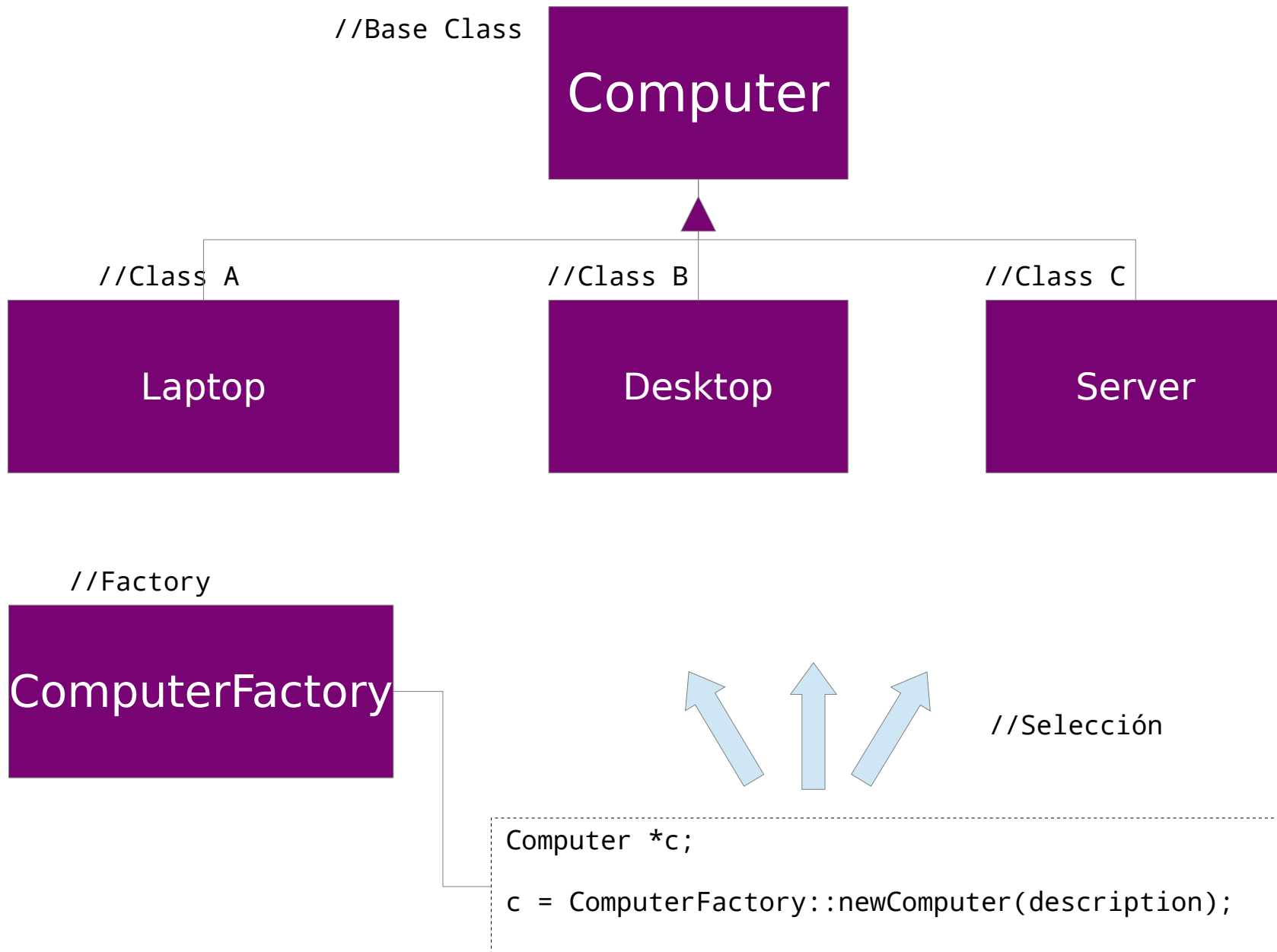
```
class Server: public Computer
{
public:
    void Run() override {. . .};
    void Stop() override {. . .};
    virtual ~Laptop() {};
private:
    . . .
};
```

```
class ComputerFactory
{
public:
    static Computer *newComputer(const std::string &description)
    {
        if(description == "laptop")
            return new Laptop;
        if(description == "desktop")
            return new Desktop;
        if(description == "server")
            return new Server;
        return nullptr;
    }
};
```

```
// Se pueden añadir nuevas clases sin
//     modificar el código de llamada:

main() // No depende de las clases derivadas
{
    Computer *c;
    Description = . . . ; // "laptop" o "desktop"
                        // o "server"
    c= ComputerFactory::newComputer(description)
    . . .
}
```

# Static Factory Method Example: factory-computer.cc



# Ejercicio

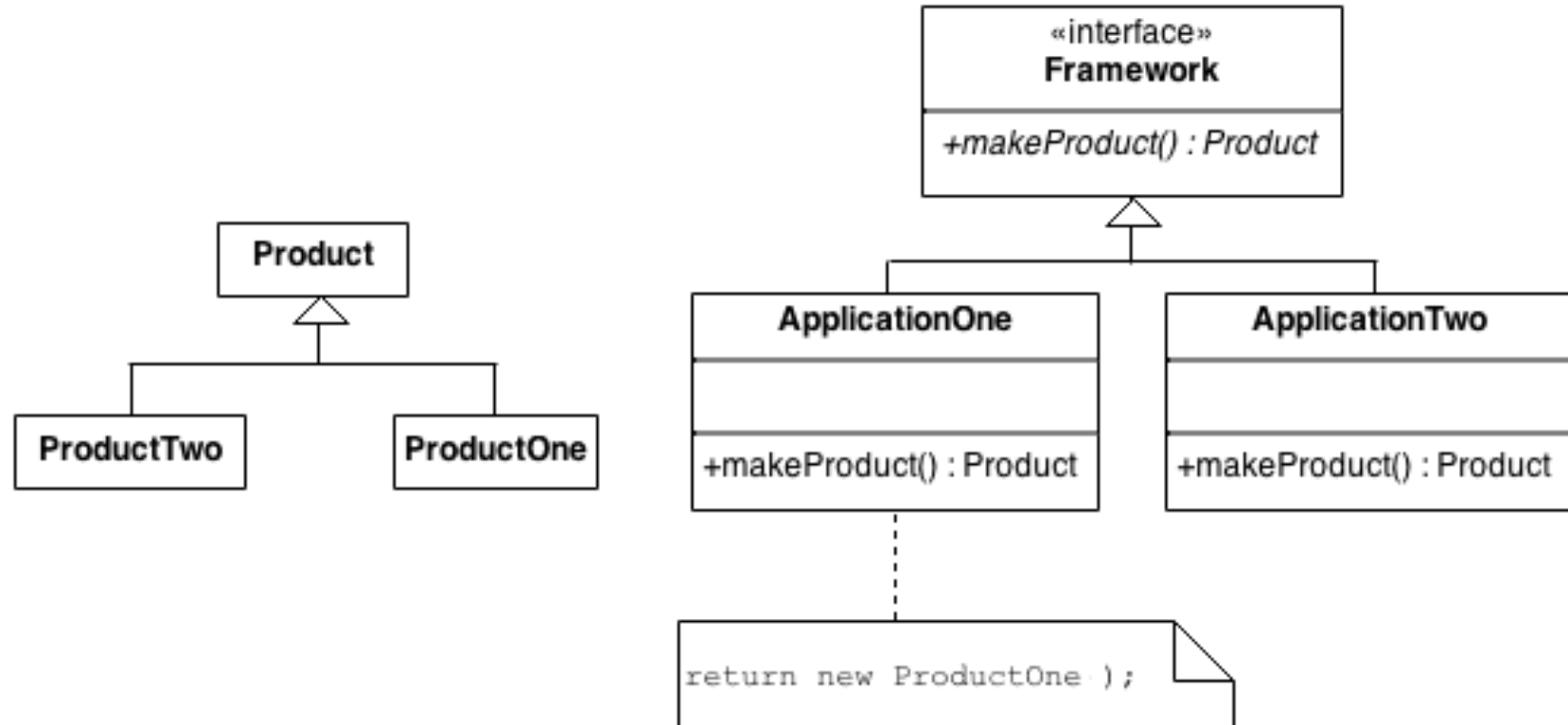
- Ejemplo adicional: `factory-vehicle.cc`



hasta aquí

# Factory 2: Abstract Factory

- “Abstract Factory” es la definición presentada en el libro original de Gang of Four.
- Una clase abstracta (Abstract Factory) define la interfaz que cada “Concrete Factory” tendrá que cumplir.
- ProductOne y ProductTwo son productos diferentes internamente. No como en Builder que las pizzas eran las mismas solo que con diferentes ingredientes/configuración.



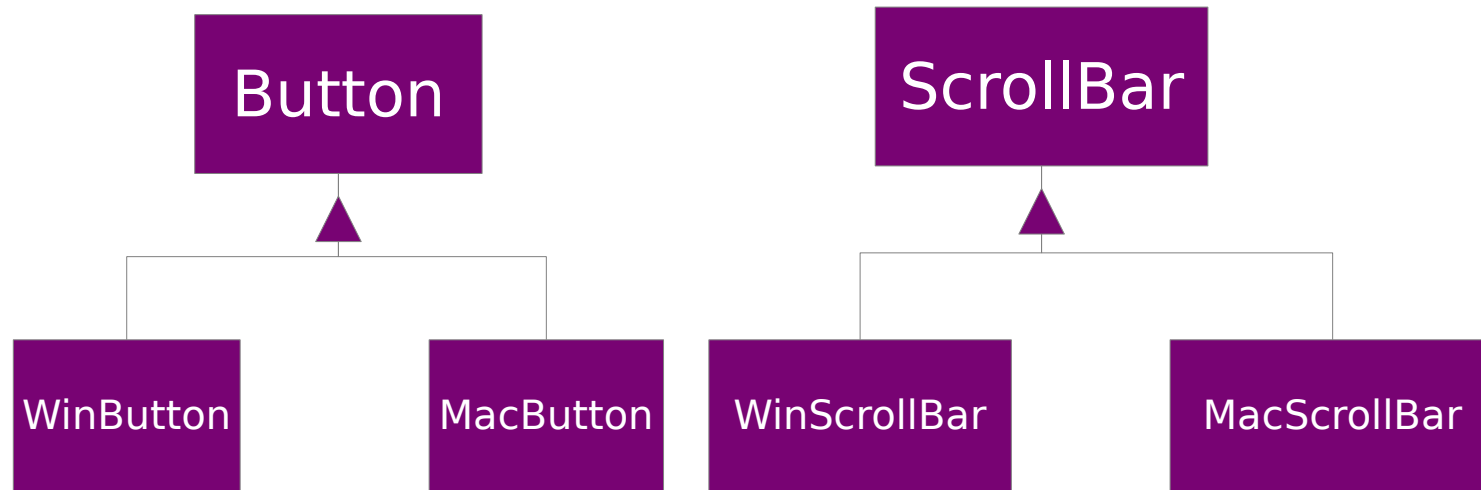
# Factory vs Builder

- **Builder:** para objetos complejos. Un mismo objeto y diferentes configuraciones complejas del mismo. Facilita crear y manipular objetos de la misma clase pero con diferente configuración.
- **Factory:** fabrica un objeto, que puede ser sencillo, pero se puede hacer de formas muy diferentes. Clases derivadas (factorías concretas) derivan del método general (abstract factory).

# Ejemplo: abstractFactory.cc

- La clase GUIFactory es el abstract factory (clase abstracta/interfaz) de las GUIs que impone:
  - createButton()
  - createScrollBar().
- Que las subclases **WinFactory** y **MacFactory** redefinen (las concrete factories)
- Son subclases totalmente diferentes:
  - Winfactory: factory de elementos Win
  - MacFactory: factory de elementos Mac
- Ambas están listas para que **de forma sencilla y ampliable** se creen sus respectivos objetos de la misma familia GUIFactory.

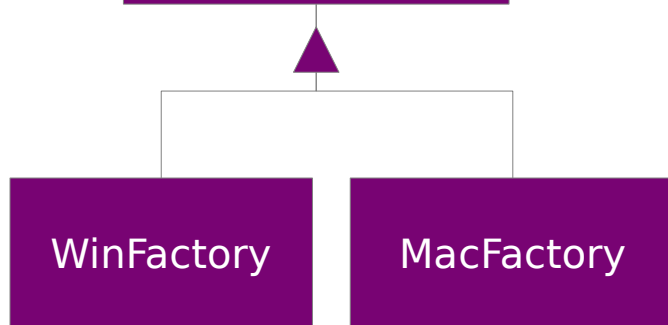
# Ejemplo: abstractFactory.cc



//Abstract Factory

**GUIFactory**

virtual Button\* createButton () = 0;  
virtual ScrollBar\* createScrollBar () = 0;



//Concrete Factory 1    //Concrete Factory 2

# Ejemplo: abstractFactory.cc

- Ejemplo C++: **abstractFactory.cc**
- La clase GUIFactory marca el interfaz para cada “concrete factory”.
- GUIFactory es la utility class “Abstract Factory”
- Cada concrete factory redefine (override) las funciones createButton() y createScrollBar()

```
GUIFactory* guiFactory;  
guiFactory = new MacFactory;  
...  
guiFactory = new WinFactory;
```

# Singleton

Creational  
Pattern

- **Nombre:** singleton
- **Tipo:** Creational pattern
- **Descripción/estructura:**
  - En matemáticas, un “singleton set” o un “unit set” es un conjunto con exactamente un elemento.
  - Nos asegura que solo exista una única instancia/objeto 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.

# Singleton

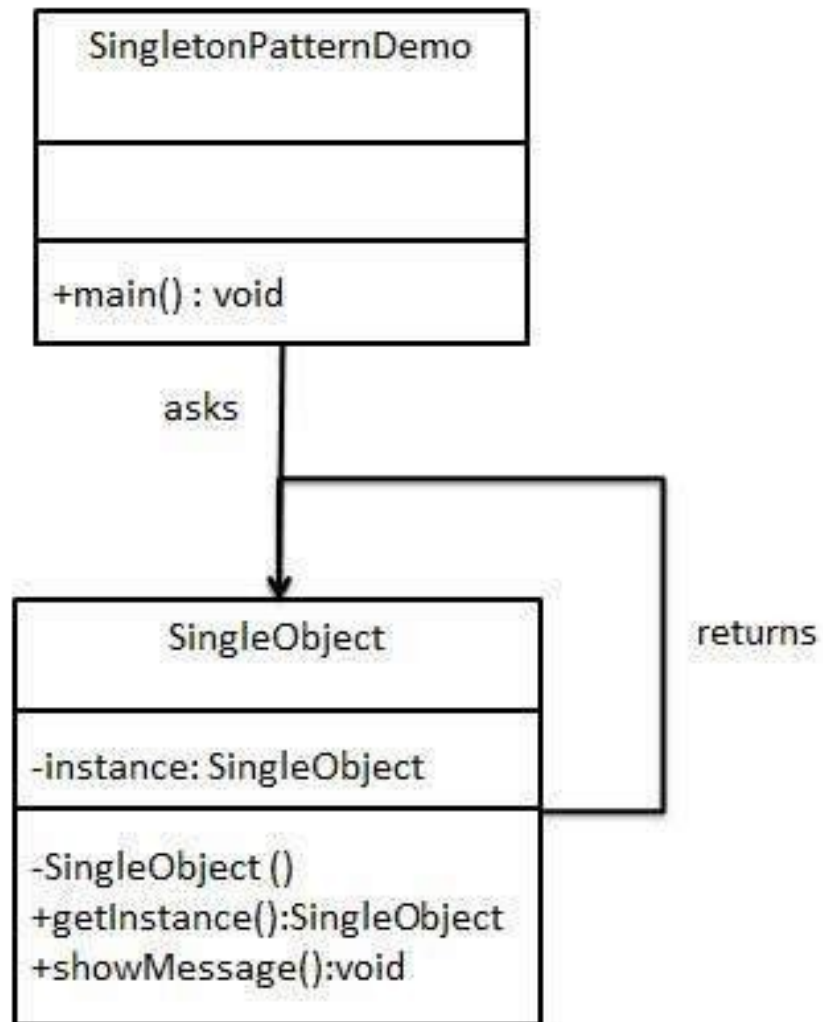
- Ejemplos:
  - Cuando hay un único recurso (porque es un recurso escaso y solo hay uno, o porque solo puede haber uno) con el que tener interface desde distintos lugares de la aplicación.
  - En un smartphone solo hay una touch screen para todos los objetos que la usan a la vez.
  - Configuración global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

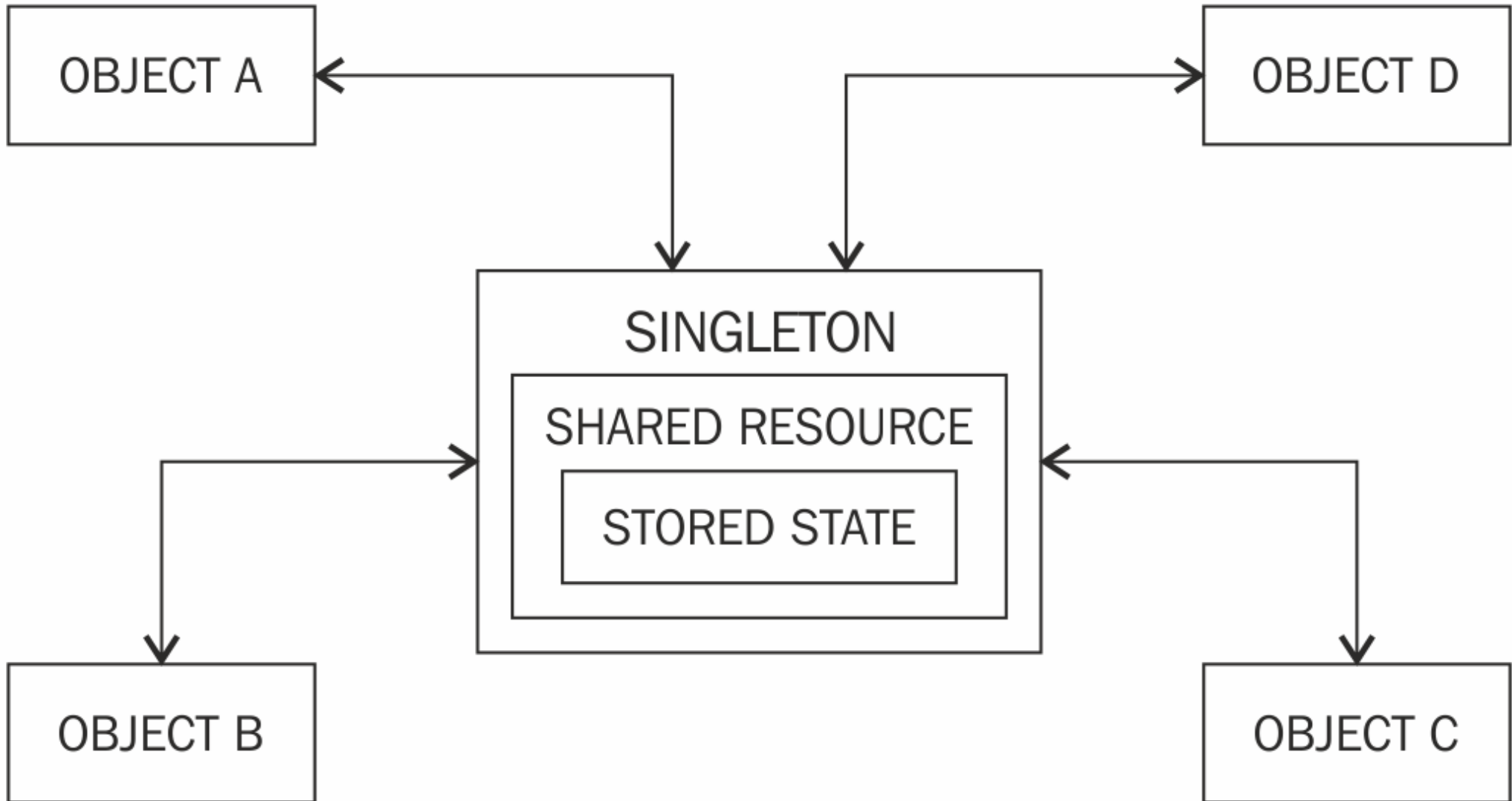
- Aplicación:
  - For application configuration
  - Configurar una aplicación o monitorizarla puede requerir un proceso complejo que con el tiempo quizá se amplie.
  - Cuando se usan variables globales pueden meterse en una struct global, pero la clase es más flexible.
- Consecuencias:
  - Garantiza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



# Resumen

- Reutilización de diseños
- Patrones de diseño:
  - Behavioral (**3**): Iterator, Observer, Strategy
  - Structural (**4**): Composite, MVC, Template Method, Parameterized type
  - Creational (**3**): Builder, Factory, Singleton
- 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 Educación, S.A. Núñez de Balboa 120. Madrid 2003.*
- *Russ Olsen. **Design Patterns in Ruby**. Addison-Wesley 2008.*
- ***C++ Programming/Code/Design Patterns**. Wikibooks.  
[https://en.wikibooks.org/wiki/C%2B%2B\\_Programming/Code/Design\\_Patterns](https://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns)*
- *Code Project. <http://www.codeproject.com/Articles/386982/Two-Ways-to-Realise-the-Composite-Pattern-in-Cplusplus>*
- *Ejemplos de código C++: [composite.cc](#), [builder.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](https://en.wikipedia.org/wiki/Abstract_factory_pattern)*
- *<https://www.geeksforgeeks.org/software-design-patterns/>*



*“The Force is Strong in Design Patterns” - - Yoda*