Contenido

3. Patrones de diseño.

- 3.1 Qué es un patrón de diseño.
- 3.2 Patrones de creación.
- 3.3 Patrones de comportamiento.
- 3.4 Patrones de estructura.

3.1 Qué es un patrón de diseño

En arquitectura

- De acuerdo a Alexander (Alexander et al., 1977*) “Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno y, describe también el núcleo de la solución al problema, de forma que puede usarse un millón de veces sin tener que hacer dos veces lo mismo”

*C. Alexander, S. Ishikawa, M. Silverstein; *A pattern Language: Towns, Buildings, Construction*; Oxford University Press, 1977

3.1 Qué es un patrón de diseño

En general

- Un patrón lo entendemos como una solución general que es reutilizable a un problema repetitivo dentro de un contexto determinado

3.1 Qué es un patrón de diseño

De acuerdo con Gamma et al.

- Un patrón lo entendemos como una solución general que es reutilizable a un problema repetitivo dentro de un contexto determinado

3.1 Qué es un patrón de diseño

Un patrón consta de cuatro elementos

- Nombre. Descripción del patrón.
- Problema. Describe cuando aplicar el patrón. Explica el problema y su contexto.
- Solución. Se describen los elementos, relaciones, responsabilidades y colaboraciones que conforman la solución.
- Consecuencias. Son los resultados esperados tras aplicar el patrón.

3.1 Qué es un patrón de diseño

Formato de Gamma et al. para documentar patrones

- **Nombre y clasificación**
- **Intención.** Qué hace y qué problemas resuelve.
- **Alias.** Otros nombres con los que se le conoce.
- **Motivación.** Escenario de ejemplo.
- **Aplicabilidad.** Cuando aplicarlo.
- **Estructura.** Modelo.

3.1 Qué es un patrón de diseño

Formato de Gamma et al. para documentar patrones (continuación…)

- **Participantes.** Elementos que intervienen.
- **Colaboraciones.** Cómo colaboran los elementos.
- **Consecuencias.** Ventajas e inconvenientes.
- **Implementación.** Errores, técnicas, trucos.
- **Código de ejemplo.**
- **Patrones relacionados.**

3.1 Qué es un patrón de diseño

Organización de los patrones

- Los patrones de diseño pueden variar en su granularidad y nivel de abstracción.
- Gamma et al., usan dos categorías para organizar los patrones, por su propósito y su alcance.

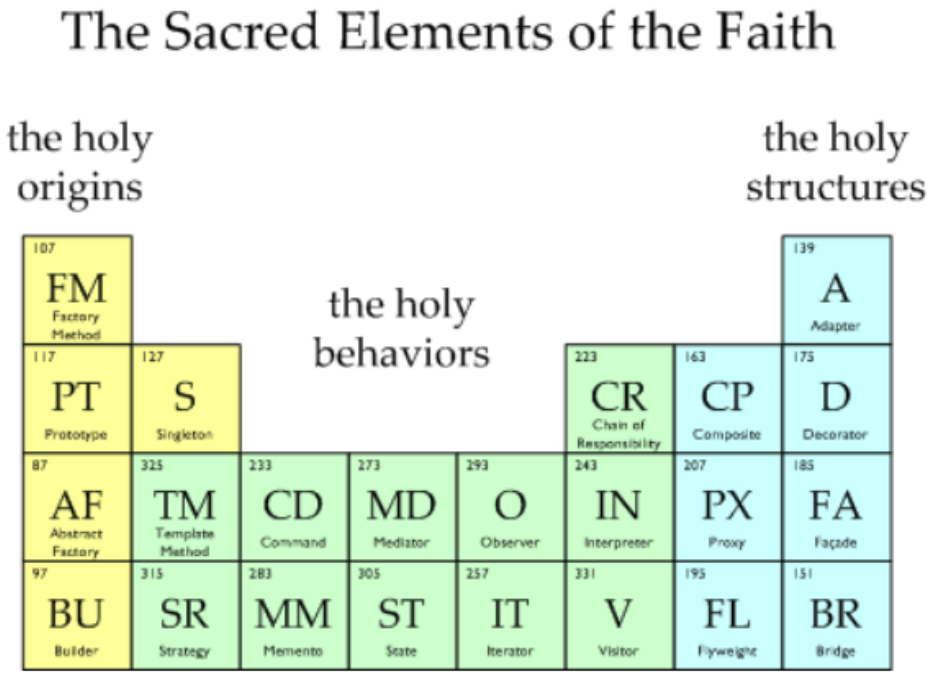
3.1 Qué es un patrón de diseño

Organización por su propósito

- En esta categoría, los patrones se clasifican en los siguientes:
  - **De creación.** Relacionados con el proceso de creación de objetos
  - **De estructura.** Tratan con la composición de clases y objetos
  - **De comportamiento.** Caracterizan diferentes formas en que las clases u objetos interactúan y distribuyen responsabilidades

3.1 Qué es un patrón de diseño

Organización por su propósito



By Purpose		Creational	Structural	Behavioral
By Scope	Class	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

3.1 Qué es un patrón de diseño

Organización por su alcance

- La segunda categoría para organizar los patrones determina si el patrón aplica principalmente a clases o a objetos.
- Los patrones de clase tratan de manera estática las relaciones entre clases y sus subclases.
- Los patrones de objeto tratan de manera dinámica las relaciones entre objetos.

3.2 Patrones de creación

Clasificación

- Factory Method (clase). Define una abstracción para crear objetos, subclases que deciden la clase concreta a instanciar.
- Abstract Factory (objeto). Proporciona una interface para crear familias de objetos relacionadas.
- Builder (objeto). Separa la construcción de objeto complejo de su representación, permitiendo diferentes instrucciones.
- Prototype (objeto). Especifica y crea objetos por medio de un prototipo mediante la clonación.
- Singleton (objeto). Garantiza que una clase solo tenga una instancia y proporciona un acceso global a ella.

Factory Method (clase)

- **Intención.** Definir una interfaz para crear objetos permitiendo a las subclases decidir qué clase concreta instanciar.
- **Alias.** Constructor virtual.

Factory Method (clase)

- Motivación
  - Cuando se instancia una clase a través de `new()` se instancia una clase concreta, esto provoca mayor acoplamiento en el código, lo que se traduce en una menor flexibilidad al momento de añadir nuevos cambios o dar mantenimiento al código.
  - Pensemos en la creación de varios tipos de figuras, por ejemplo, un objeto triángulo:


```
Figura triangulo = new Triangulo();
```

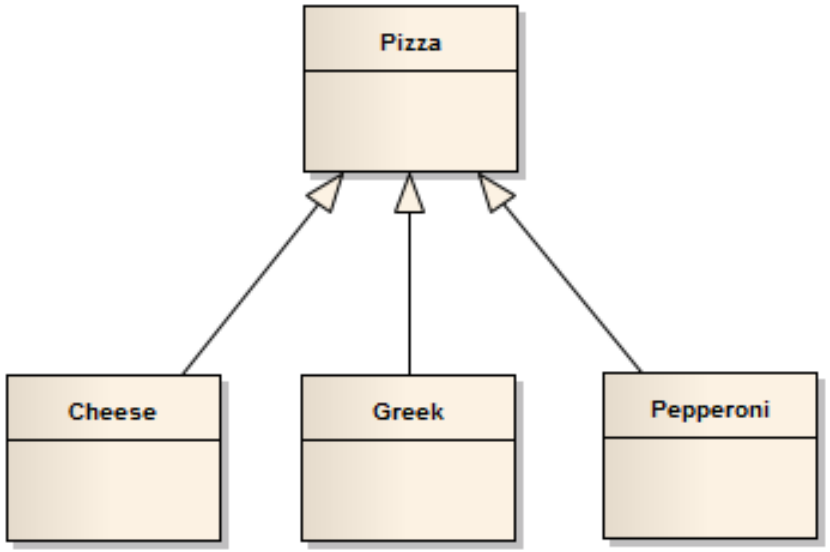
Factory Method (clase)

- Motivación
  - Si requerimos crear otros tipos de figuras dada una determinada opción se tendría algo como:

```
Figura fig;
if(opcion==“triangulo”)
   fig=new Triangulo();
if(opcion==“circulo”)
  fig=new Circulo();
if(opcion=“rectangulo”)
  fig=new Rectangulo();
```

Factory Method (clase)

- Ejemplo



Factory Method (clase)

- Motivación
  - En este ejemplo, la decisión sobre qué clase instanciar depende de ciertas opciones.
  - El inconveniente de este ejemplo se presenta cuando se requiere añadir o borrar tipos de figuras ya que se debe abrir el código y realizar los cambios.
  - A menudo este tipo de código termina en distintas partes del sistema haciéndolo difícil de mantener y aumentando la probabilidad de que contenga defectos.
  - Una estrategia de solución es emplear interfaces.

Factory Method (clase)

- Ejemplo (Pensemos en una pizzería)

```
...
Pizza orderPizza(String type) {
   Pizza pizza = new Pizza();

   pizza.prepare();
   pizza.bake();
   pizza.cut();
   pizza.box();
   return pizza;
}
...
```

*ejemplo tomado del libro Head First Design Patterns (Freeman et al., 2004)

Factory Method (clase)

- Ejemplo

```
...
Pizza orderPizza(String type) {
  Pizza pizza;

  if (type.equals("cheese")) {
    pizza = new CheesePizza();
  } else if (type.equals("greek") {
    pizza = new GreekPizza();
  } else if (type.equals("pepperoni") {
    pizza = new PepperoniPizza();
  }
  pizza.prepare();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza;
}
```

¿Qué sucede cuando nos piden añadir más tipos de pizza?

El método `orderPizza()` comienza a salirse fuera de control

Factory Method (clase)

- Ejemplo

  Una estrategia consiste en encapsular la creación de los objetos

```
public class SimplePizzaFactory {
   public Pizza createPizza(String type) {
     Pizza pizza = null;
     if (type.equals("cheese")) {
       pizza = new CheesePizza();
     } else if (type.equals("pepperoni")) {
       pizza = new PepperoniPizza();
     } else if (type.equals("clam")) {
       pizza = new ClamPizza();
     } else if (type.equals("veggie")) {
       pizza = new VeggiePizza();
     }
     return pizza;
   }
}
```

Factory Method (clase)

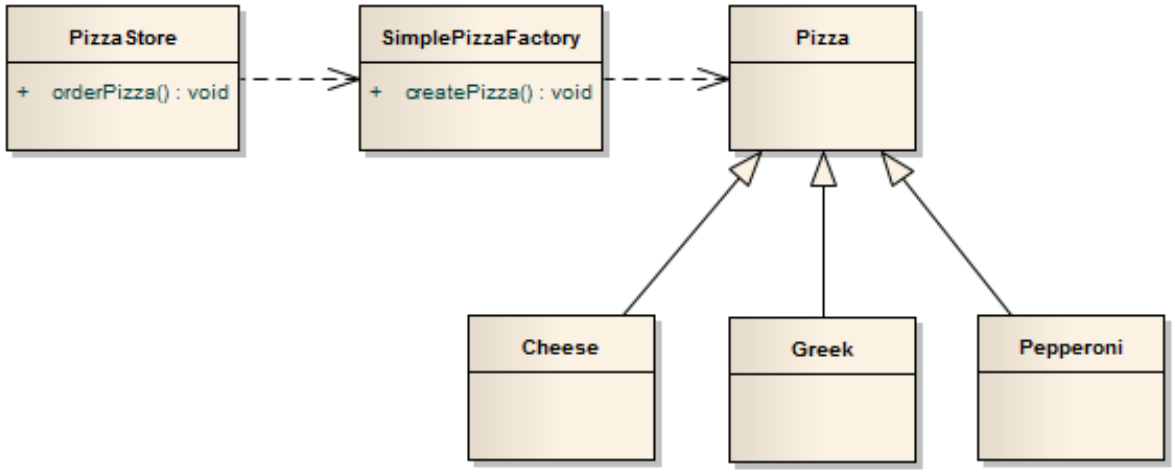
- Ejemplo

```
public class PizzaStore {
  SimplePizzaFactory factory;
  public PizzaStore(SimplePizzaFactory factory) {
    this.factory = factory;
  }

   public Pizza orderPizza(String type) {
   Pizza pizza;
   Pizza = factory.createPizza(type);
   pizza.prepare();
   pizza.bake();
   pizza.cut();
   pizza.box();
   return pizza;
  }
// other methods here
}
```

*Ejemplo Simple Factory

Factory Method (clase)

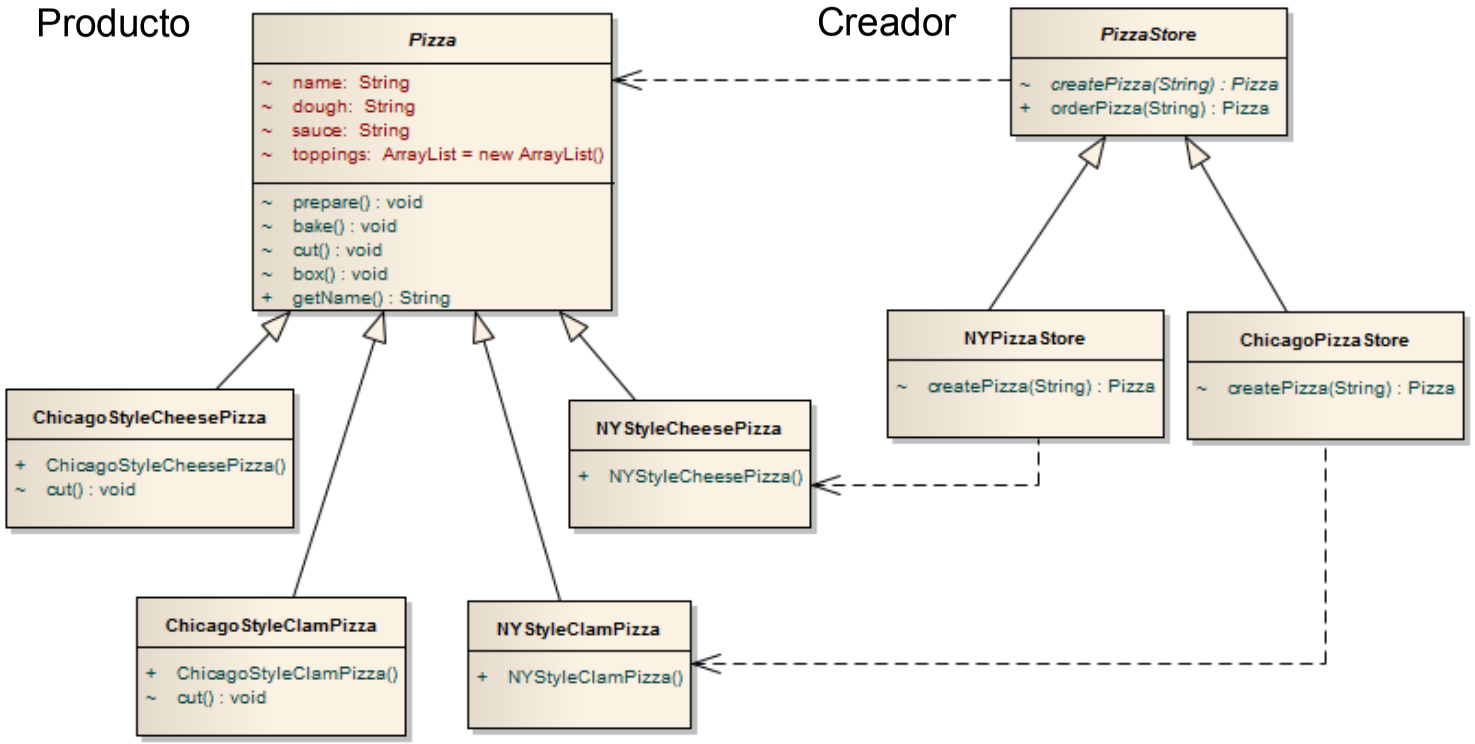
- Ejemplo



¿Cómo podríamos añadir nuevas pizzerías y regionalizar las pizzas existentes?

Factory M

- Ejemplo



Factory Method (clase)

- Ejemplo

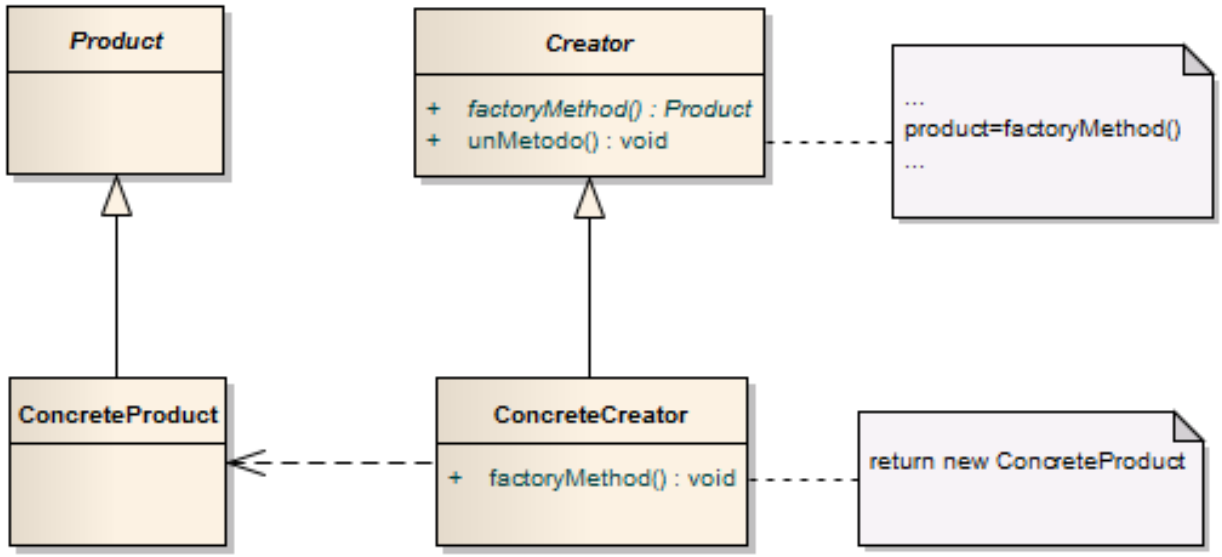
```
PizzaStore nyStore = new NYPizzaStore();
PizzaStore chicagoStore = new ChicagoPizzaStore();

Pizza pizza = nyStore.orderPizza("cheese");
System.out.println("Ethan ordered a " + pizza.getName() + "\n");

pizza = chicagoStore.orderPizza("cheese");
System.out.println("Joel ordered a " + pizza.getName() + "\n");
```

Factory Method (clase)

- **Aplicabilidad.** Usar este patrón cuando:
  - Una clase no pueda anticipar las clases de los objetos que debe crear.
  - Una clase requiere que sus subclases especifiquen los objetos que la propia clase crea.

Factory Method (clase)

- **Estructura**



Factory Method (clase)

- Participantes
  - **Creador.** Declara el método factoría que devuelve un objeto de tipo producto. Suele utilizarse desde alguna de sus propias operaciones.
  - **Creador concreto.** Redefine el método factoría para devolver una instancia de `ProductoConcreto`.
  - **Producto.** Define una clase abstracta o interfaz de los objetos que el método factoría crea.
  - **Producto concreto.** Implementa la interfaz de producto.

Factory Method (clase)

- **Colaboraciones.** El creador confía en que se redefina el método factoría para que devuelva un producto de la clase apropiada.
- **Consecuencias.**
  - Elimina la necesidad de introducir clases específicas en el código creador.
  - Se elimina el acoplamiento entre clases.
- **Implementación** (ver código ejemplo).

3.3 Patrones de estructura

- Como ya mencionamos, los patrones de estructura tratan con la composición de clases y objetos para formar estructuras más grandes.
- Estos patrones facilitan y mejoran las relaciones entre objetos.
- Los patrones de estructura de clases usan la herencia para realizar composiciones de interfaces e implementaciones.
- Por otra parte los patrones de estructura de objetos describen maneras de realizar composiciones entre objetos para añadir nueva funcionalidad, como es el caso del patrón decorator que veremos a continuación.

3.3 Patrones de estructura

Clasificación

- Adapter (class y object). Convierte una interface de una clase en otra clase que esperan los clientes.
- Bridge (objeto). Desacopla una abstracción de su implementación, permitiendo modificaciones independientes de ambas.
- Composite (objeto). Permite crear estructuras en árbol por igual a las hojas que los elementos compuestos.
- Decorator (objeto). Asigna responsabilidades de forma dinámica a objetos, proporcionando una alternativa flexible a la herencia.

3.3 Patrones de estructura

Clasificación (continuación…)

- Facade (objeto). Proporciona una interface unificado para un conjunto de interfaces de un subsistema.
- Flyweight (objeto). Compartir objetos de grado fino de forma eficiente.
- Proxy (objeto). Proporciona un sustituto o representante para controlar el acceso al un objeto.

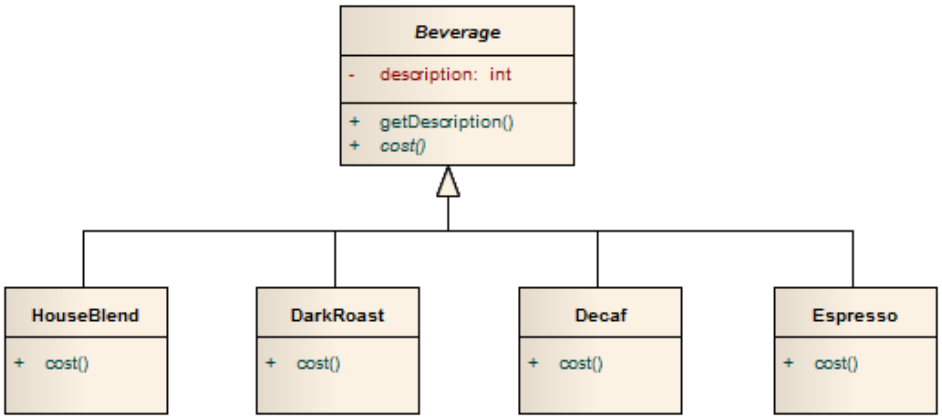
Decorator

- **Intención.**
  - Añadir funcionalidad a un objeto de forma dinámica.
  - Proporcionar una alternativa flexible a la herencia para ampliar la funcionalidad de una clase.
- **Alias.** Conocido también como wrapper (envoltorio).

Decorator

Motivación

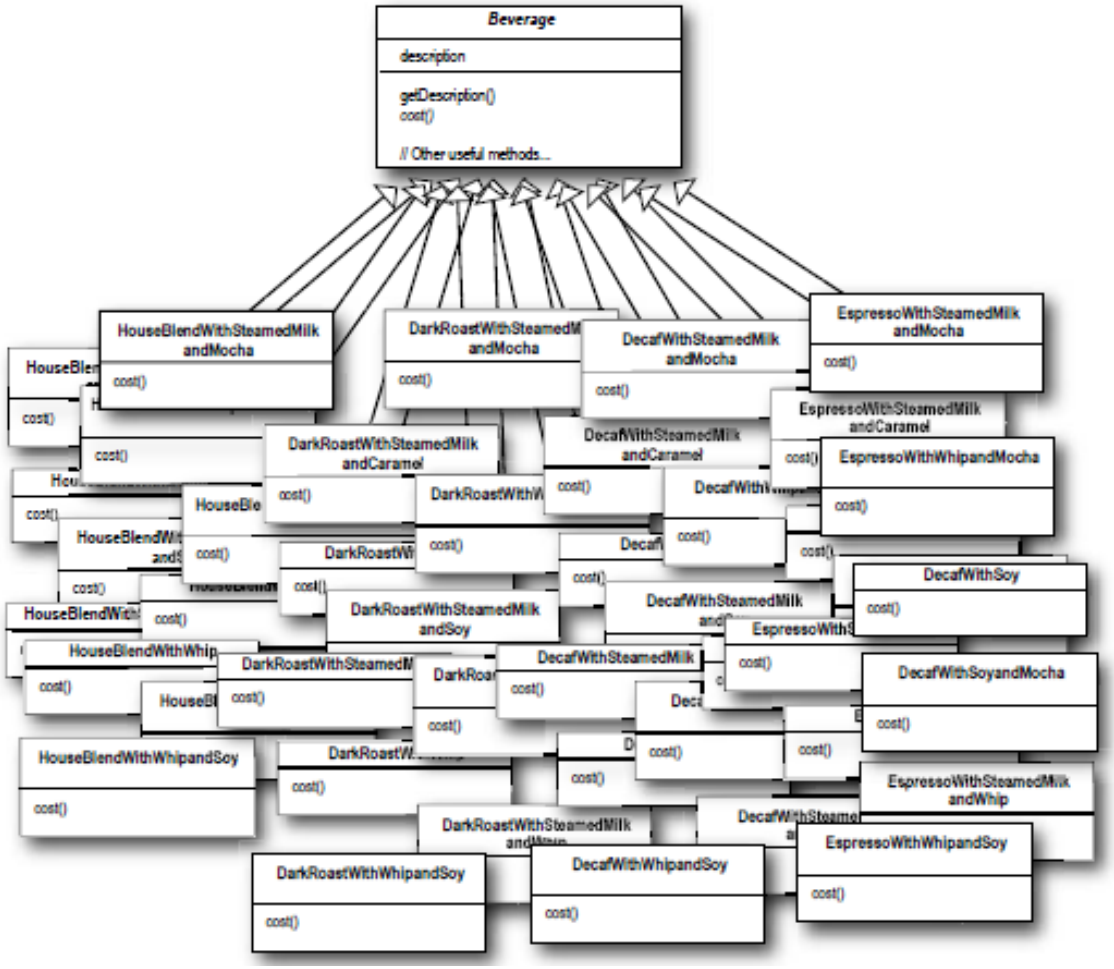
- Pensemos en una cafetería que ofrece varios tipos de cafés.



La cafetería pronto comienza a expandirse por lo que se requiere añadir otros tipos de cafés, así como varios complementos (crema batida, moca, etc.)

3.3 Patrones de estructura

Decorator



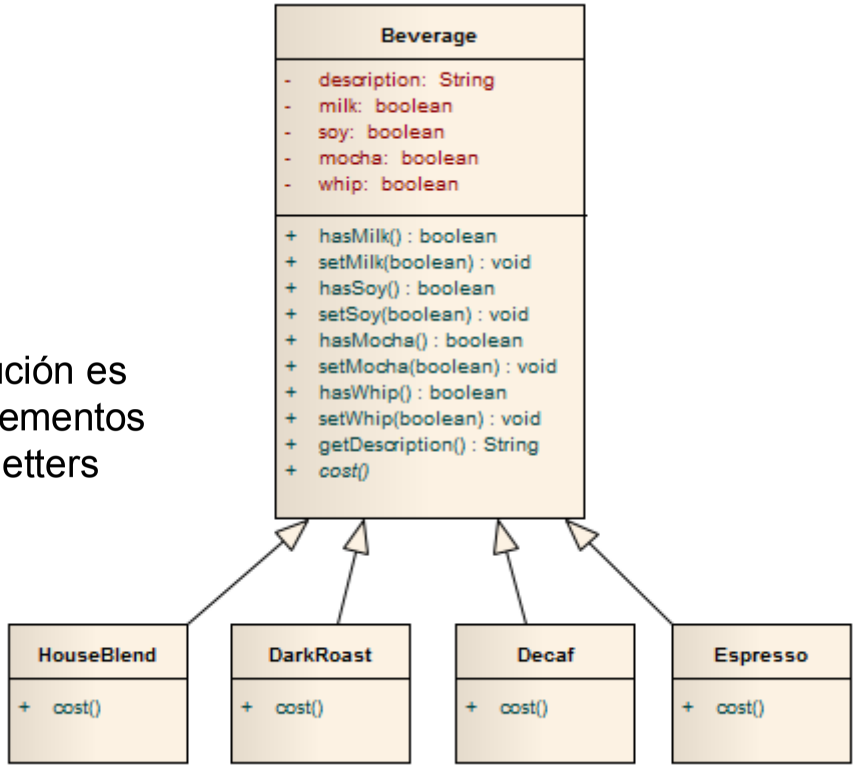
Decorator

- Motivación (continuación…)
  - Dado el modelo anterior, se observa una dificultad para mantener el código.
    - ¿Qué sucede si el precio de uno de los complementos sube?
    - ¿Qué sucede si se desea añadir nuevos complementos?

Decorator

- Motivación (continuación…)

Una posible solución es definir los complementos como setters y getters

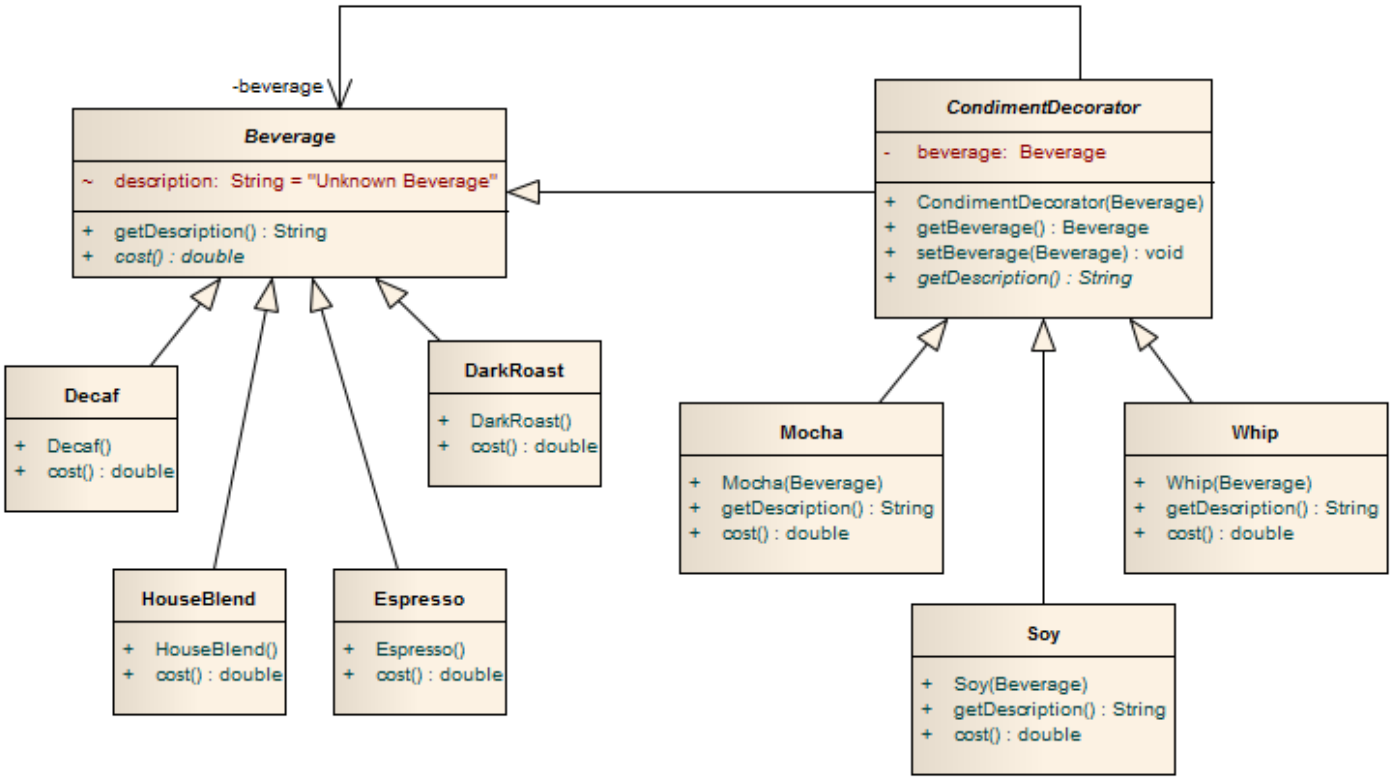


Decorator

- Motivación (continuación…)
  - Una alternativa más flexible es encapsular o “envolver” un objeto dentro de otro objeto que “decore” o añada una nueva funcionalidad, por ejemplo, envolver un tipo de café con algún complemento ya sea crema batida o un sabor extra como moca.
  - Siguiendo este enfoque las clases se mantienen abiertas para extender nueva funcionalidad y al mismo tiempo se mantienen cerradas para modificación. Es decir, el sistema puede crecer sólo añadiendo o extendiendo nueva funcionalidad sin modificar las clases existentes.

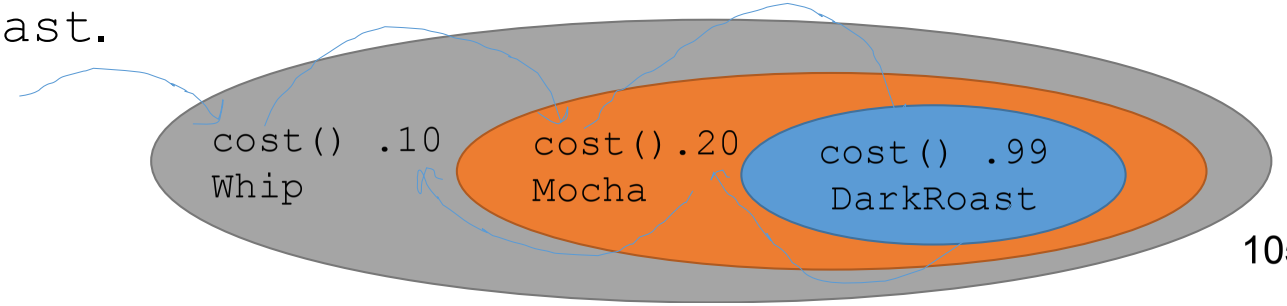
3.3 Patrones de estructura

Decorator



Decorator

- Funcionamiento de varios decorators
  - Se crea un objeto `DarkRoast`.
  - El cliente añade un complemento `Mocha`, y éste se crea envolviendo al objeto `DarkRoast`.
  - El cliente añade `Whip` (crema batida), y éste se crea envolviendo al objeto `Darkroast`.



Decorator

- Código de ejemplo

```
Beverage bev = new DarkRoast();
bev = new Mocha(bev);
bev = new Whip(bev);
System.out.println(bev.getDescription() + " $" + bev.cost());
```

3.3 Patrones de estructura

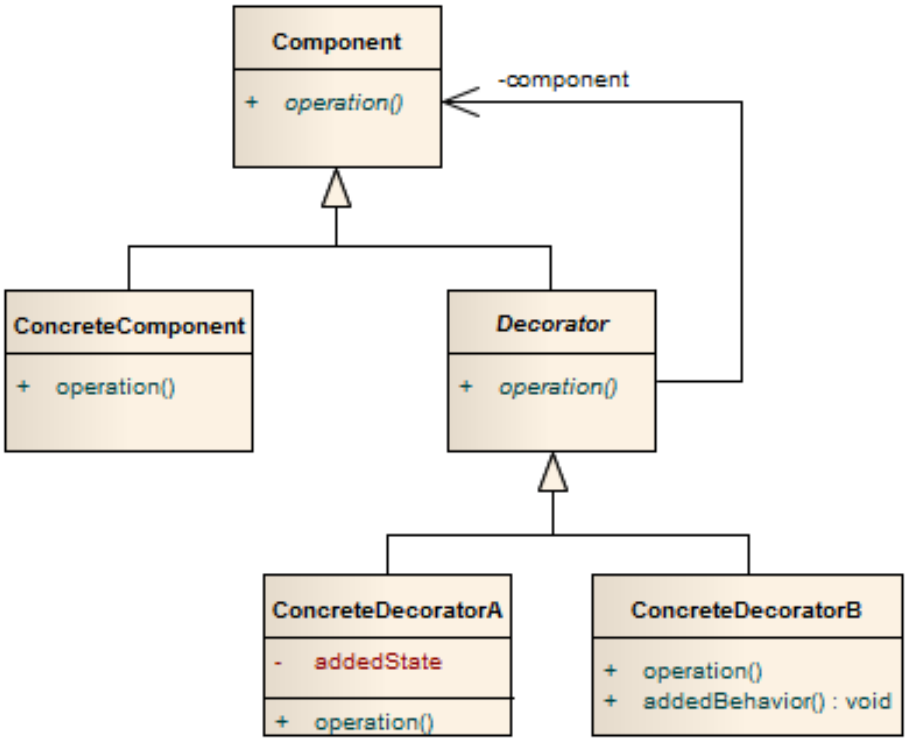
Decorator

- **Aplicabilidad.** Usar este patrón cuando:
  - Se desea agregar responsabilidades a objetos individuales sin afectar a otros objetos.
  - La extensión de funcionalidad por sub-clasificación es impráctica. Este patrón ofrece una alternativa a la explosión de subclases necesarias para tener todas las combinaciones de una serie de funcionalidades independientes.

3.3 Patrones de estructura

Decorator

- **Estructura**



Decorator

- Participantes:
  - **Componente.** Define la interfaz para los objetos a los que se les pueden añadir funcionalidades dinámicas.
  - **Componente concreto.** Define un objeto al que se le pueden añadir funcionalidades adicionales.
  - **Decorador.** Mantiene una referencia a un objeto componente y define una interfaz apegada a la del componente.
  - **Decorador concreto.** Añade funcionalidades al objeto componente.

Decorator

- **Consecuencias**:
  - Mayor flexibilidad que la herencia.
  - Es posible decorar a un objeto más de una vez con el mismo decorador (imposible con herencia).
  - Sólo se usa la funcionalidad que se requiere.
  - No son necesarias clases de grano grueso capaces de hacerlo todo.
  - El decorador y el componente no son el mismo objeto.
  - Como inconveniente, el número de objetos en memoria puede llegar a ser muy grande.

3.3 Patrones de estructura

Decorator

- **Implementación** (ver código ejemplo)

3.4 Patrones de comportamiento

- Como ya mencionamos, los patrones de comportamiento no describen sólo patrones de objetos o clase, sino la comunicación entre objetos.
- Esta categoría de patrones caracterizan flujos de control complejos que son difíciles de identificar en tiempo de ejecución. Estos patrones ayudan a no distraernos en el flujo de control entre objetos y concentrarnos sólo en la manera en cómo se interconectan los objetos.
- Los patrones de comportamiento (a nivel de clases) usan herencia para distribuir el comportamiento entre clases.
- Los patrones de comportamiento (a nivel de objetos) usan la composición, por ejemplo el patrón observer define y mantiene una dependencia entre varios objetos.

3.4 Patrones de comportamiento

Clasificación

- Interpreter (clase). Define una representación de la gramática de un lenguaje intérprete.
- Template Method (clase). Define una operación en el esqueleto de un algoritmo delegando en las subclases los detalles.
- Command (objeto). Se desacopla el objeto que invoca la operación asociada , mediante un objeto. Esto permite realizar órdenes compuestas (patrón composite) o llevar una cola y deshacer operaciones.
- Iterator (objeto). Proporciona un modo de acceder secuencialmente a los elementos de un objeto sin exponer su representación interna.

3.4 Patrones de comportamiento

Clasificación (continuación…)

- Mediator (objeto). Define un objeto que encapsula como interactúan un conjunto de objetos.
- Memento (objeto). Externaliza un el estado interno de un objeto.
- Observer (objeto). Define una dependencia entre uno a muchos, de tal manera, que cuando cambie avise a todos los objetos dependientes.
- State (objeto). Permite que un objeto cambie su comportamiento cada vez que cambie su estado interno.

3.4 Patrones de comportamiento

Clasificación (continuación…)

- Strategy (objeto). Define un conjunto de algoritmo haciéndolos intercambiables dinámicamente.
- Visitor (objeto). Define un conjunto de operaciones sobre una estructura de datos de forma independiente.

3.4 Patrones de comportamiento

Observer

- **Intención.** Definir una dependencia entre objetos de tal manera que cuando cambie el estado de uno de los objetos, sus dependientes sean notificados y actualizados automáticamente.
- **Alias.** Conocido también como publicador-suscriptor.

3.4 Patrones de comportamiento

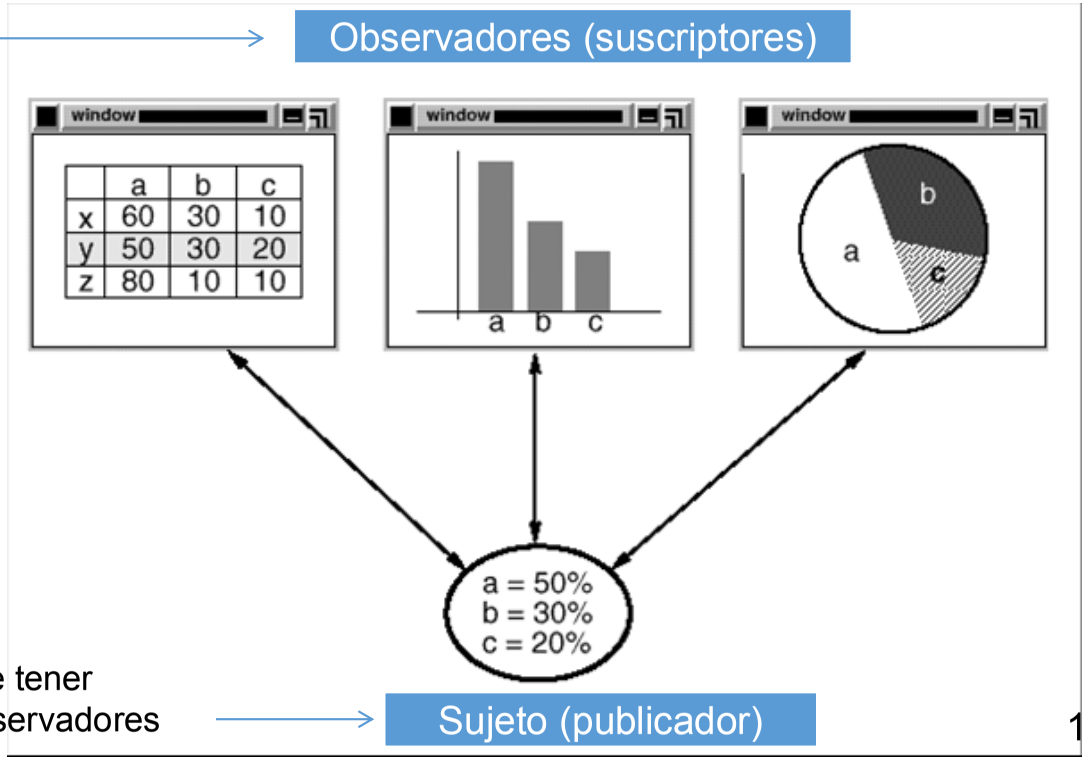
Observer

- Motivación
  - Se parte de la necesidad de mantener la consistencia entre objetos relacionados sin aumentar el acoplamiento entre su clase.
  - Pensemos en una aplicación GUI:
    - Es posible separar los objetos de presentación (vistas) de los objetos de datos.
    - Se pueden tener varias vistas sincronizadas de los mismos datos.

3.4 Patrones de comportamiento

Observer

Los observadores son notificados sobre el cambio de estado del sujeto



Un sujeto puede tener uno o varios observadores

3.4 Patrones de comportamiento

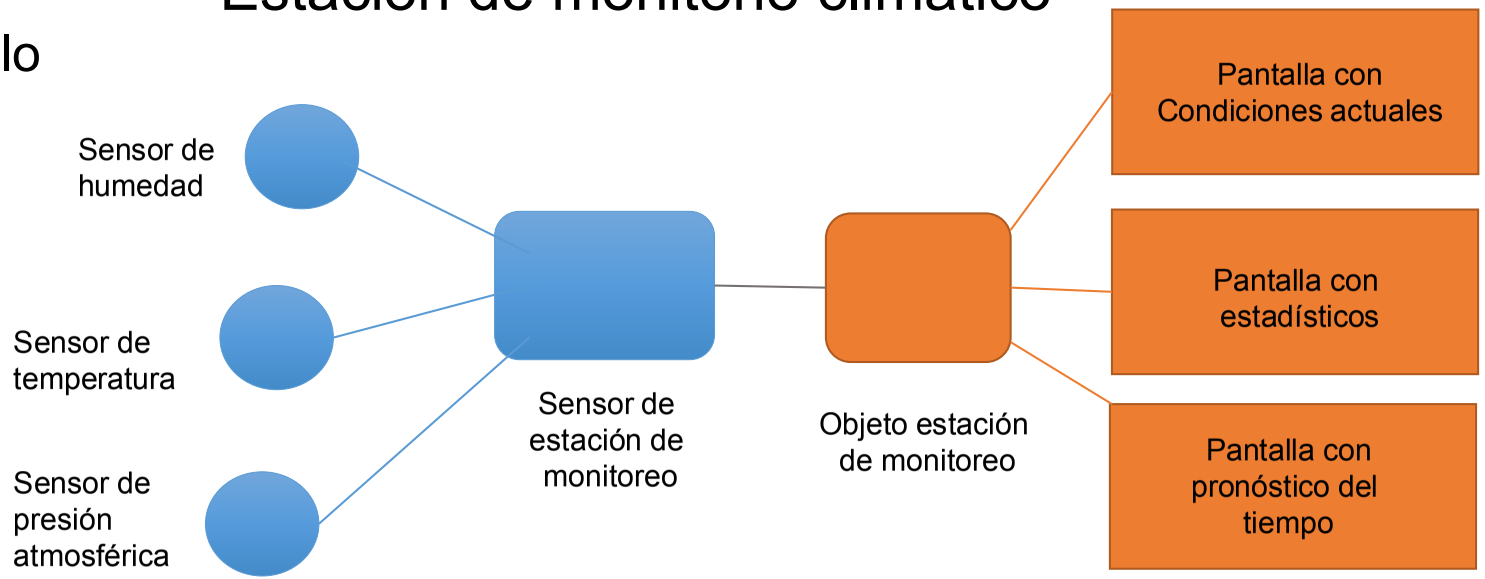
Observer

- **Aplicabilidad**. Usar cuando:
  - El mismo conjunto de datos (estado de un objeto) pueda ser presentado de distintas maneras (vistas) y se desea que las vistas estén sincronizadas con el estado del objeto.
  - El cambio en un objeto requiera cambiar otros objetos sin tener conocimiento de los objetos que requieren cambiar.
  - Un objeto deba ser capaz de notificar a otros sin conocer su clase concreta, evitando así el acoplamiento.

3.4 Patrones de comportamiento

Observer

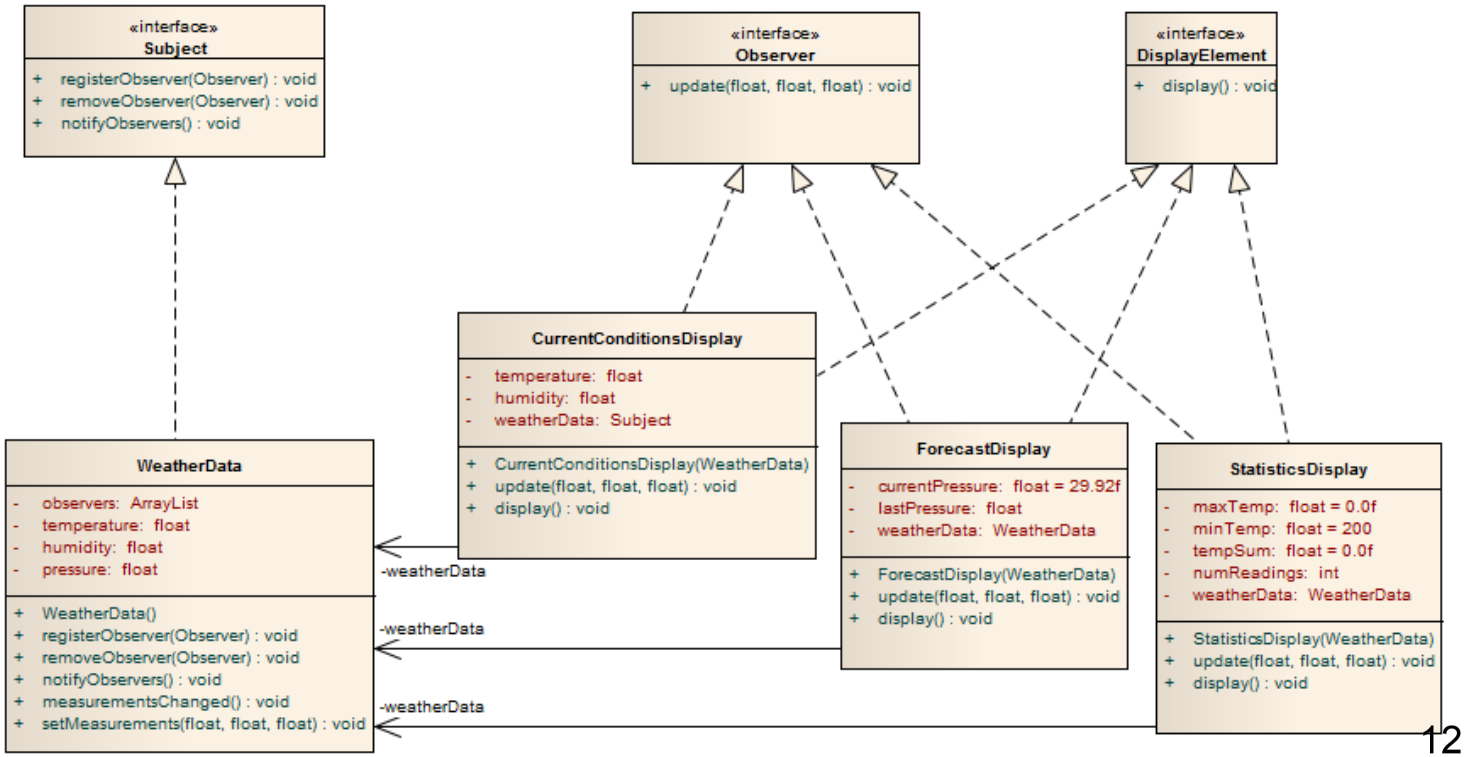
- Estación de monitorio climático
- Ejemplo



3.4 Patrones de comportamiento

Observer

- Ejemplo



Observer

- Ejemplo (continuación…)
  - `WeatherData` implementa la interface `Subject`.
  - `Observer` brinda a `Subject` (publicador) una interface común para actualizar los observadores.
  - `DisplayElement` es una interface usada para implementar los mensajes a mostrar en los observadores.
  - `CurrentConditionsDisplay`, `StatisticsDisplay` y `ForecastDisplay` son los observadores concretos, mantienen una referencia al suscriptor (`Subject`) a través de `WeatherData`.

Observer

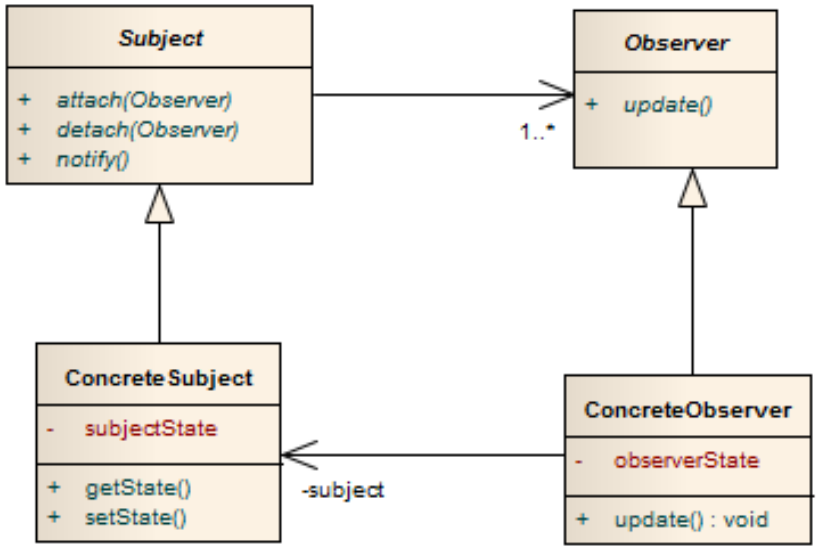
- Ejemplo (continuación…)

```
WeatherData weatherData = new WeatherData();

CurrentConditionsDisplay currentDisplay =
                                new CurrentConditionsDisplay(weatherData);
StatisticsDisplay statisticsDisplay =
                                     new StatisticsDisplay(weatherData);
ForecastDisplay forecastDisplay =  new ForecastDisplay(weatherData);


weatherData.setMeasurements(80,65,30.4f);
weatherData.setMeasurements(82,70,29.2f);
weatherData.setMeasurements(78,90,29.2f);
```

3.4 Patrones de comportamiento

Observer

- **Estructura**



3.4 Patrones de comportamiento

Observer

- Participantes
  - **Sujeto.** Conoce a sus observadores, expone una interface para suscribir y remover la suscripción de observadores.
  - **Sujeto concreto.** Envía una notificación a sus observadores cuando su estado cambia.
  - **Observador.** Define una interface para actualizar los objetos que deban ser notificados de cambios en el sujeto (publicador).
  - **Observador concreto.** Mantiene una referencia a sujeto, almacena parte del estado del sujeto e implementa la interfaz de actualización del observador para mantener sus estado consistente con el del sujeto.

Observer

- **Consecuencias**.
  - Bajo acoplamiento entre el sujeto y el observador.
  - Es posible re-utilizar sujetos sin re-utilizar sus observadores y viceversa.
  - Se pueden agregar observadores sin afectar a los sujetos.
  - Todos los sujetos conocen a sus observadores.
  - Lo sujetos no requieren conocer al observador concreto ya que estos implementan la interface del observador.
  - El sujeto y el observador pueden pertenecer a distintas capas de abstracción.

3.4 Patrones de comportamiento

Observer

- **Implementación** (ver código ejemplo).

Referencias

- E. Gamma, R. Helm, R. Johnson and J. Vlissides; Design Patterns – Elements of Reusable Object-Oriented Software; Addison Wesley, 1995.
- E. Freeman, E. Freeman, K. Sierra and B. Bates; Head First Design Patterns; O'Reilly, 2004.