Tema 9: Patrones de Diseño

Análisis y Diseño del Software
Grado en Ingeniería Informática

Departamento de Estadística, Informática y Matemáticas
Universidad Pública de Navarra-*Nafarroako Unibertsitate Publikoa*

Curso académico 2023-2024

upna

Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Objetivos del tema

- Qué son los patrones de diseño y para qué se usan
- Conocer los patrones más comunes y sus implementaciones

1. Patrones de Diseño

2. Patrones de Creación
   - Singleton
   - Factory

3. Patrones Estructurales
   - Adapter
   - Decorator

4. Patrones de Comportamiento
   - Iterator
   - Observer

5. Otras clasificaciones

Qué son los Patrones de Diseño

- Hay problemas que ya tienen una solución probada y eficiente
  - En el desarrollo de software pasa lo mismo
- Un patrón de diseño presenta una forma estándar, efectiva y probada de solución para el problema genérico
  - Para aplicar un patrón es necesario adaptarlo al problema especifico que tengamos

Para qué son los Patrones de Diseño

- El uso de patrones ayuda a los programadores no solo a la hora de programar
  - Si otra persona ha programado algo usando patrones de diseño, podremos identificar qué patrones ha usado y qué problema tenía, y de esta manera entender mas fácilmente el código

Críticas a los Patrones de Diseño

- Son parches de diseño
  - Cubren carencias del lenguaje de programación o tecnología usados
  - Las funciones anónimas (*lambdas*) dejan obsoleto el patrón *Strategy*
- Son soluciones inadaptadas
  - La situación no debe forzarse para que se amolde a la solución
  - La aplicación de patrones no debe convertirse en dogma

Clasificados según su Propósito

Patrones de *Gang of Four*

- Patrones creacionales
  - Solucionan problemas de creación de instancias
  - Ayudan a encapsular y abstraer la creación
- Patrones estructurales
  - Solucionan problemas de agregación o composición entre clases
- Patrones de comportamiento
  - Solucionan problemas de comunicación y asignación de responsabilidades

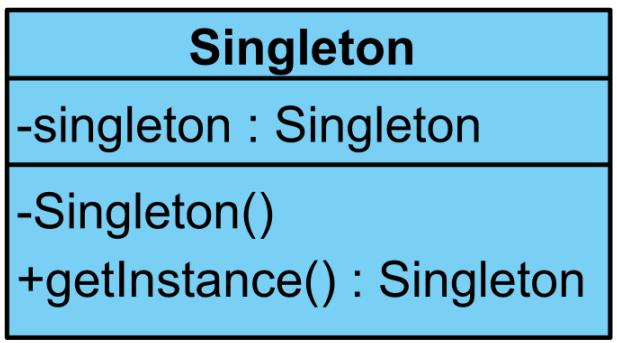
Singleton

¿Para qué?

Asegurar que solo es posible crear una única instancia de una clase

- El patrón no solo garantiza una única instancia, también proporciona un punto de acceso
- Dos usos principales
  - Controlar el acceso a un recurso único
  - Si ciertos tipos de datos deben estar disponibles a todos los objetos

Singleton Implementación



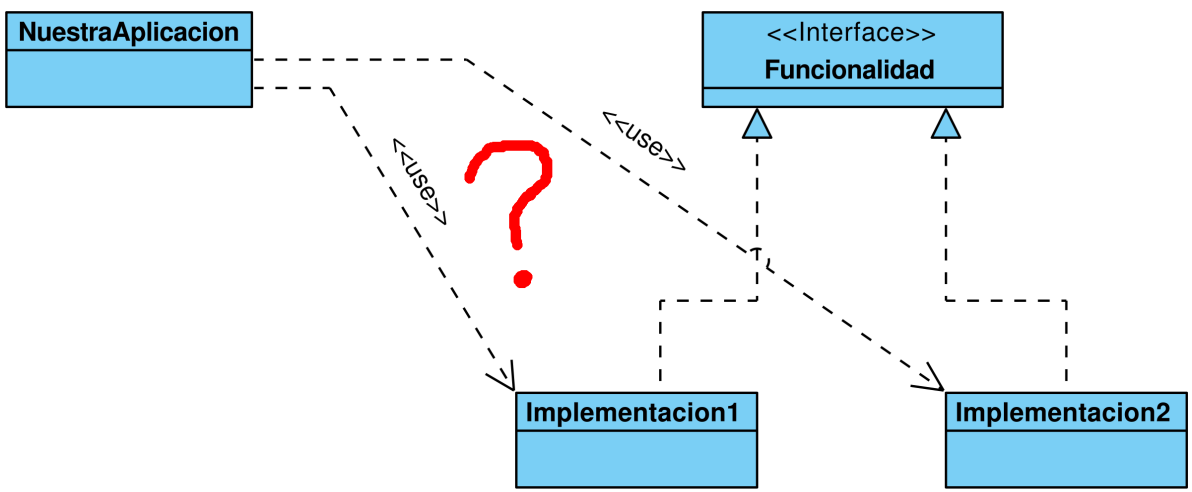

```java
public class Singleton {
    private static Singleton INSTANCE = null;

    private Singleton(){}

    private synchronized static void createInstance() {
        if (INSTANCE == null) {
            INSTANCE = new Singleton();
        }
    }

    public static Singleton getInstance() {
        if (INSTANCE == null) {
          createInstance();
        }
        return INSTANCE;
    }
}
```

Factory

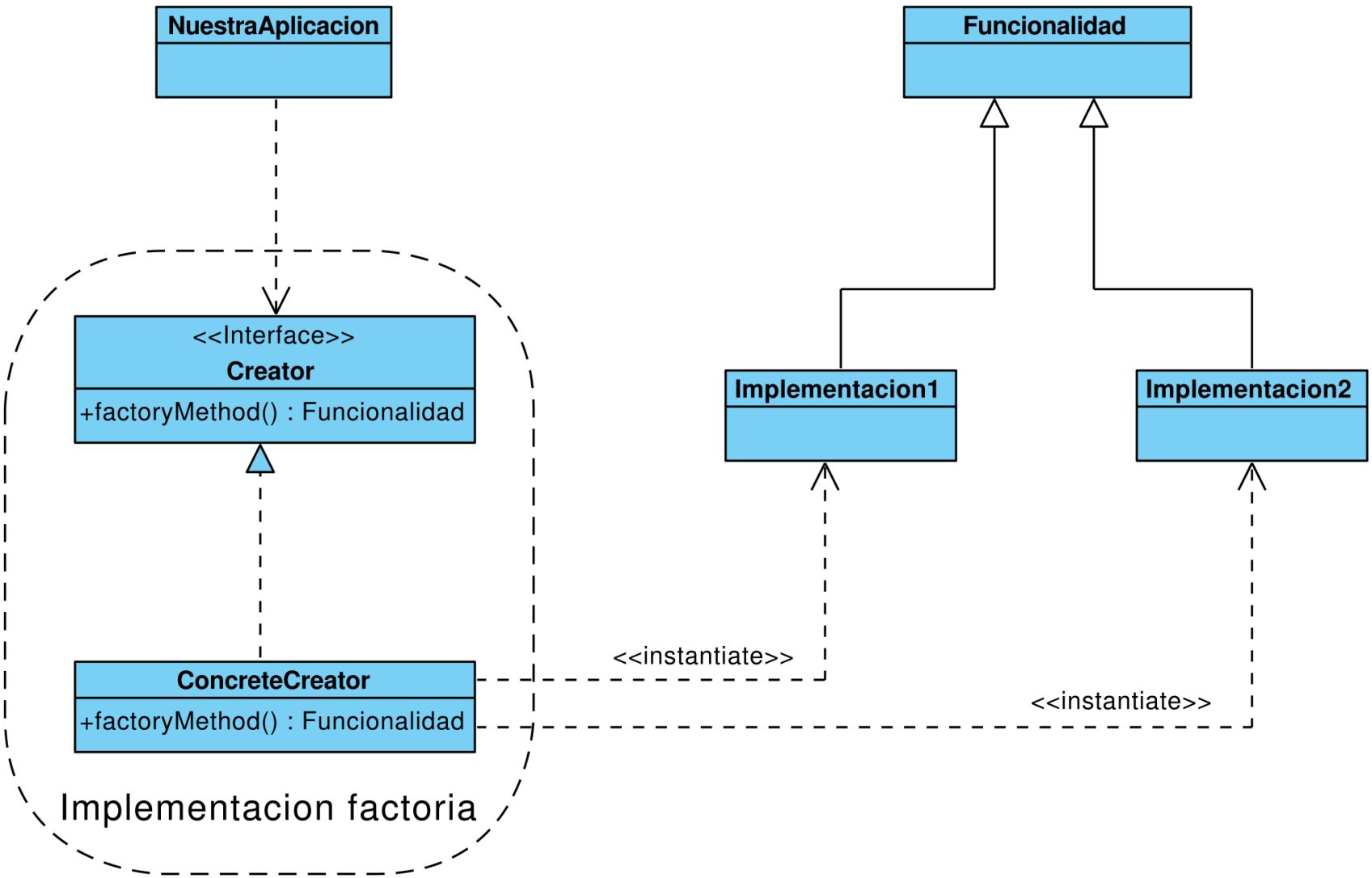
¿Para qué?

Si queremos crear un objeto sin conocer la clase especifica que implementa la funcionalidad que nos interesa

- La funcionalidad está definida en una interfaz
- Tenemos una clase que se encarga de crear la clase que nos interesa



Factory Implementación



Ejemplo *Factory* logs

```
public class LoggerFactory {
    //Se carga un archivo de configuracion
    public boolean isFileLoggingEnabled() {
        Properties p = new Properties();
        try {
            p.load(ClassLoader.getSystemResourceAsStream("Logger.properties"));
            String fileLoggingValue = p.getProperty("FileLogging");
            if (fileLoggingValue.equalsIgnoreCase("ON") == true)
                return true;
            else
                return false;
        } catch (IOException e) {
            return false;
        }
    }
    //Factory Method
    public Logger getLogger() {
        if (isFileLoggingEnabled()) {
            return new FileLogger();
        } else {
            return new ConsoleLogger();
        }
    }
}
```

1. Patrones de Diseño
2. Patrones de Creación
3. Patrones Estructurales
   - Adapter
   - Decorator
4. Patrones de Comportamiento
5. Otras clasificaciones

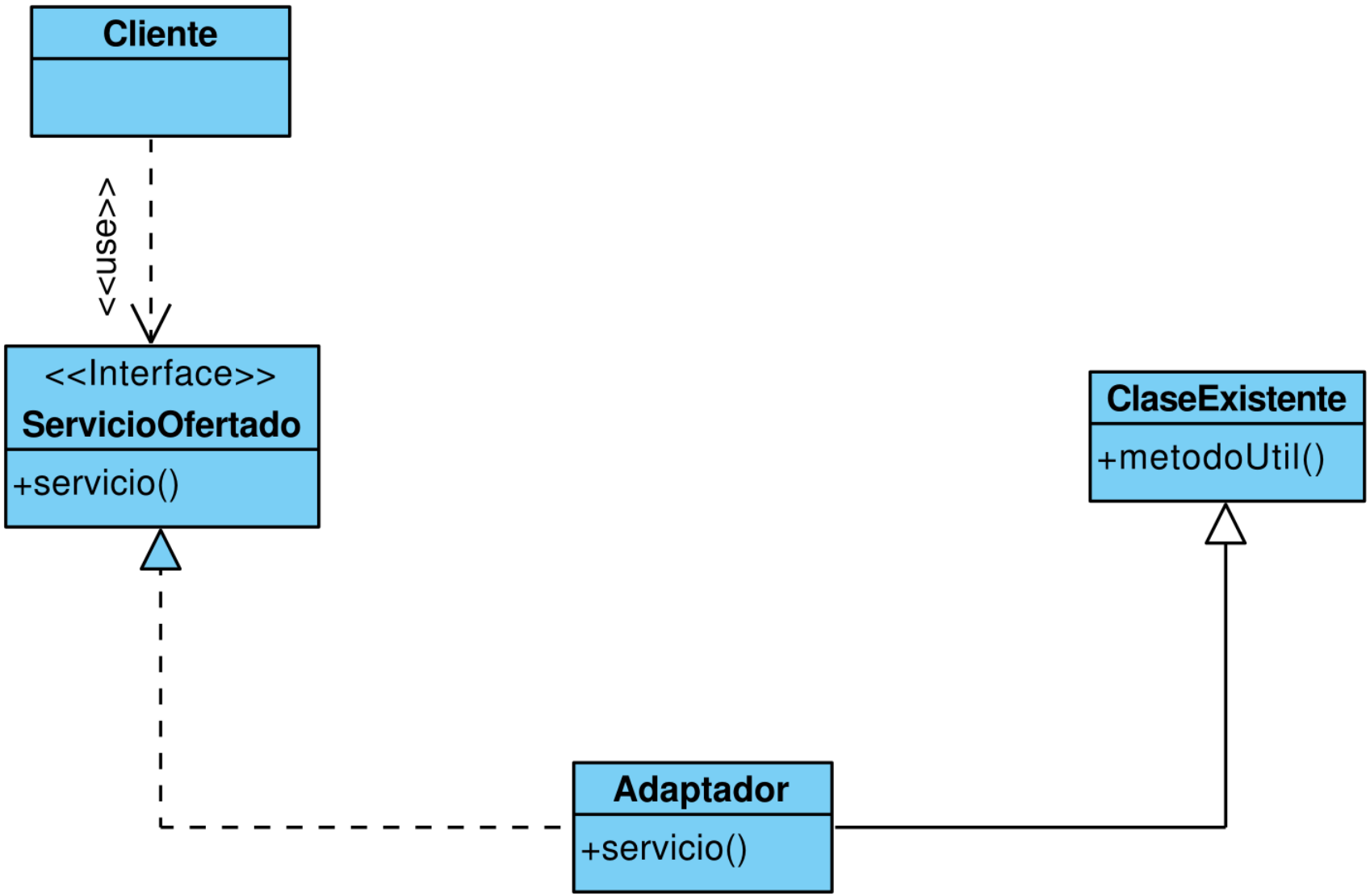
Adapter

¿Para qué?

Hay veces en las que ya tenemos una implementación que nos interesa, pero el servicio no se ofrece de la manera que el cliente espera.

- El cliente quiere hacer uso de un servicio a través de una interfaz
- Creamos una clase que haga de intermediaria entre la interfaz y la funcionalidad implementada
- El cliente no tiene porqué saber cual es la clase especifica que implementa la funcionalidad

Adapter Implementación



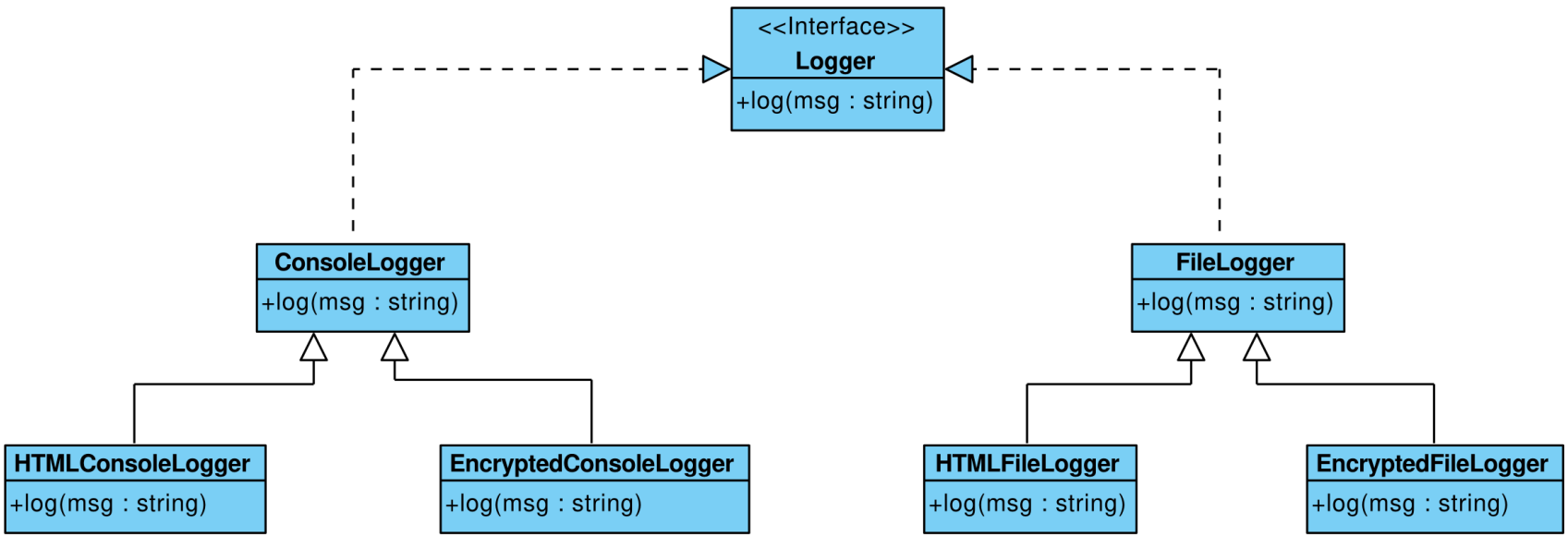
Decorator

¿Para qué?

Añadir extensiones a una clase existente sin modificarla o usar herencia

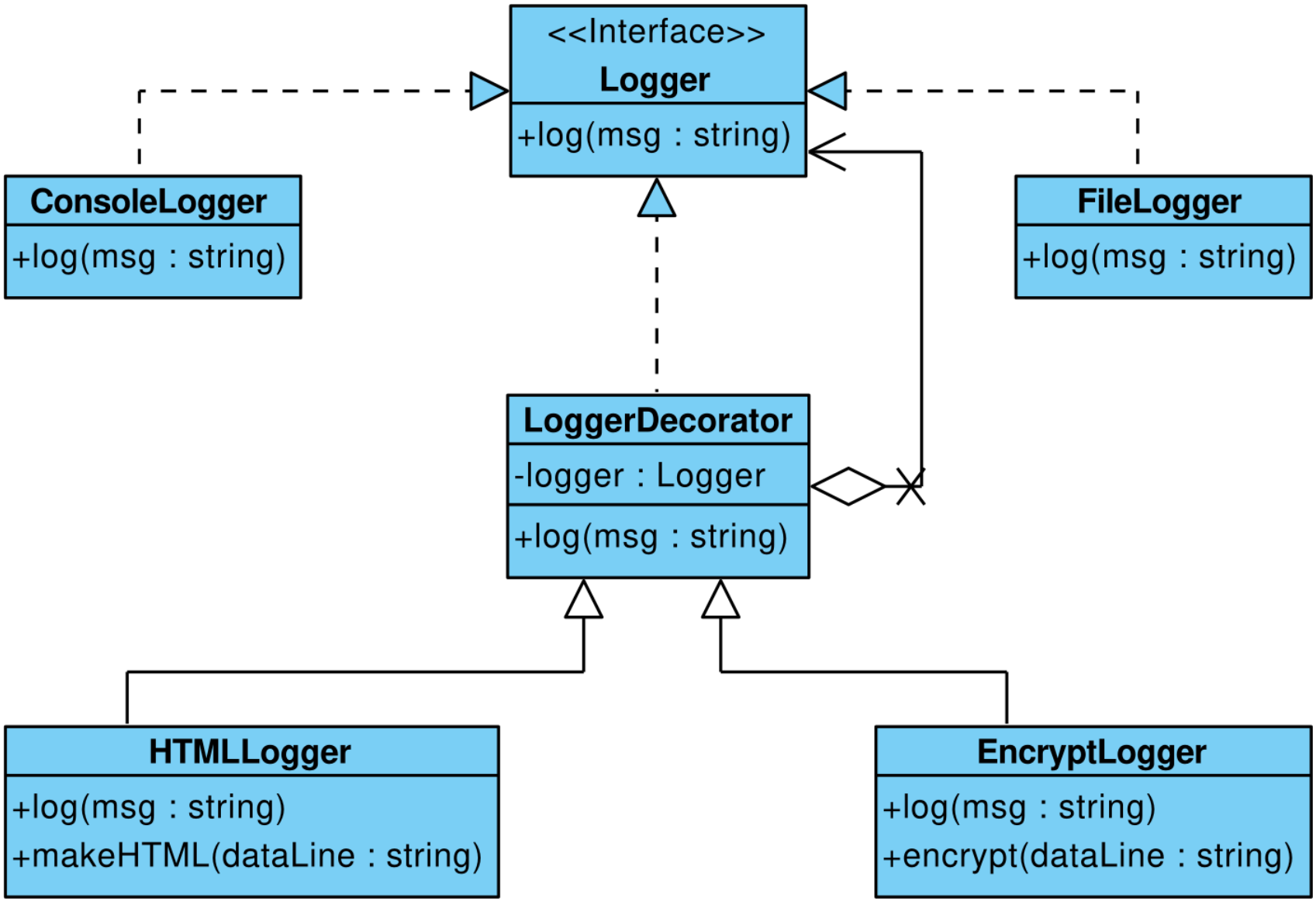
- El objeto decorador tendrá la misma interfaz (métodos) que el decorado
- El decorador contendrá una referencia al objeto decorado
- Hace de intermediario, redirigiendo llamadas de un cliente
  - En este redireccionamiento podemos añadir funcionalidades extra

Problema herencia exponencial

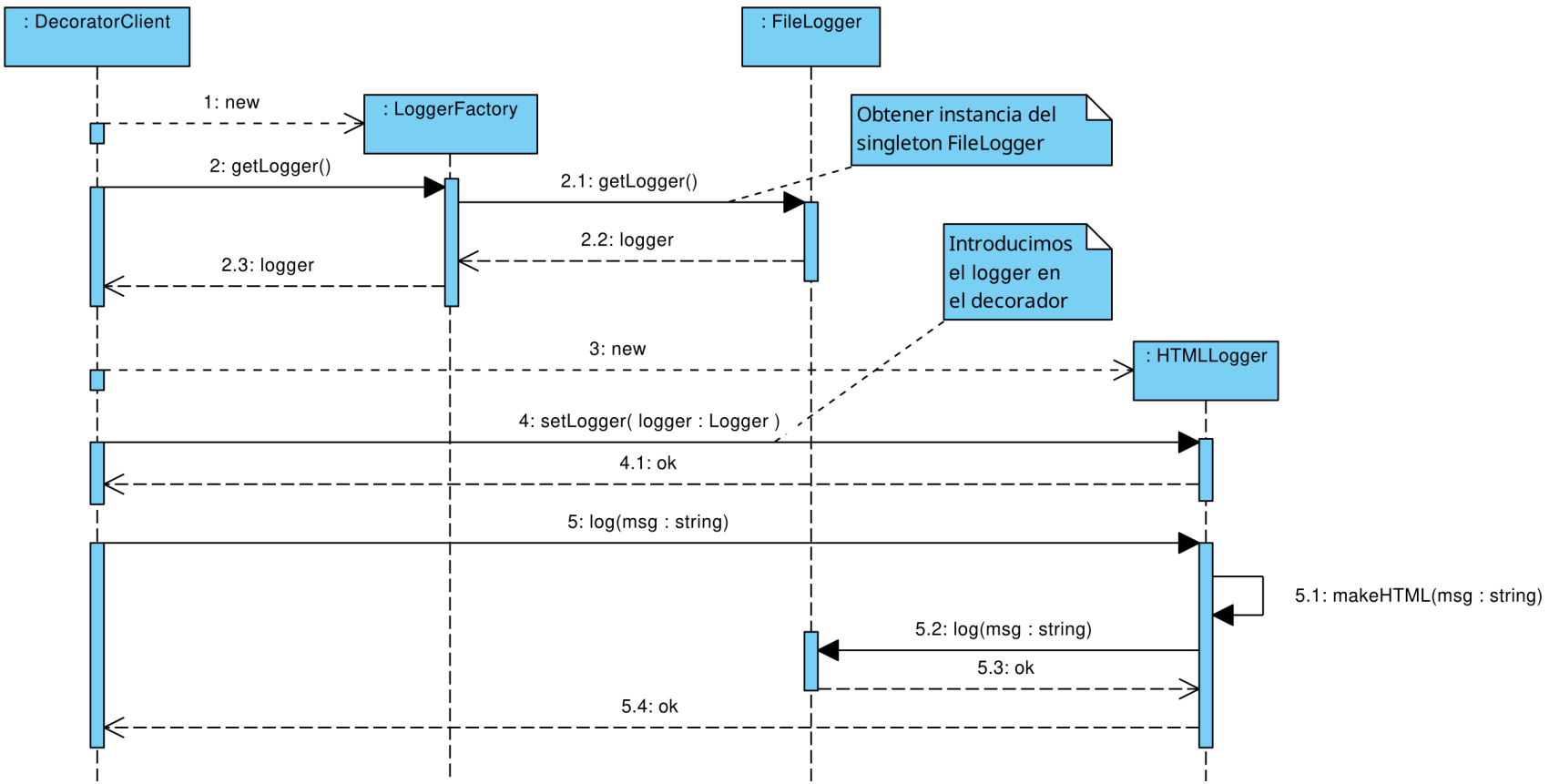


¿Qué ocurre si queremos añadir una clase DBLogger?

Simplificación con *Decorator*



Uso de *Decorator*



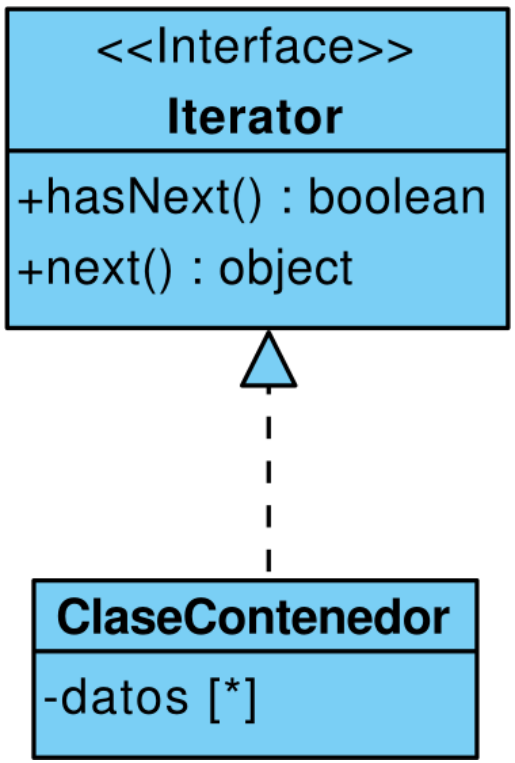
Iterator

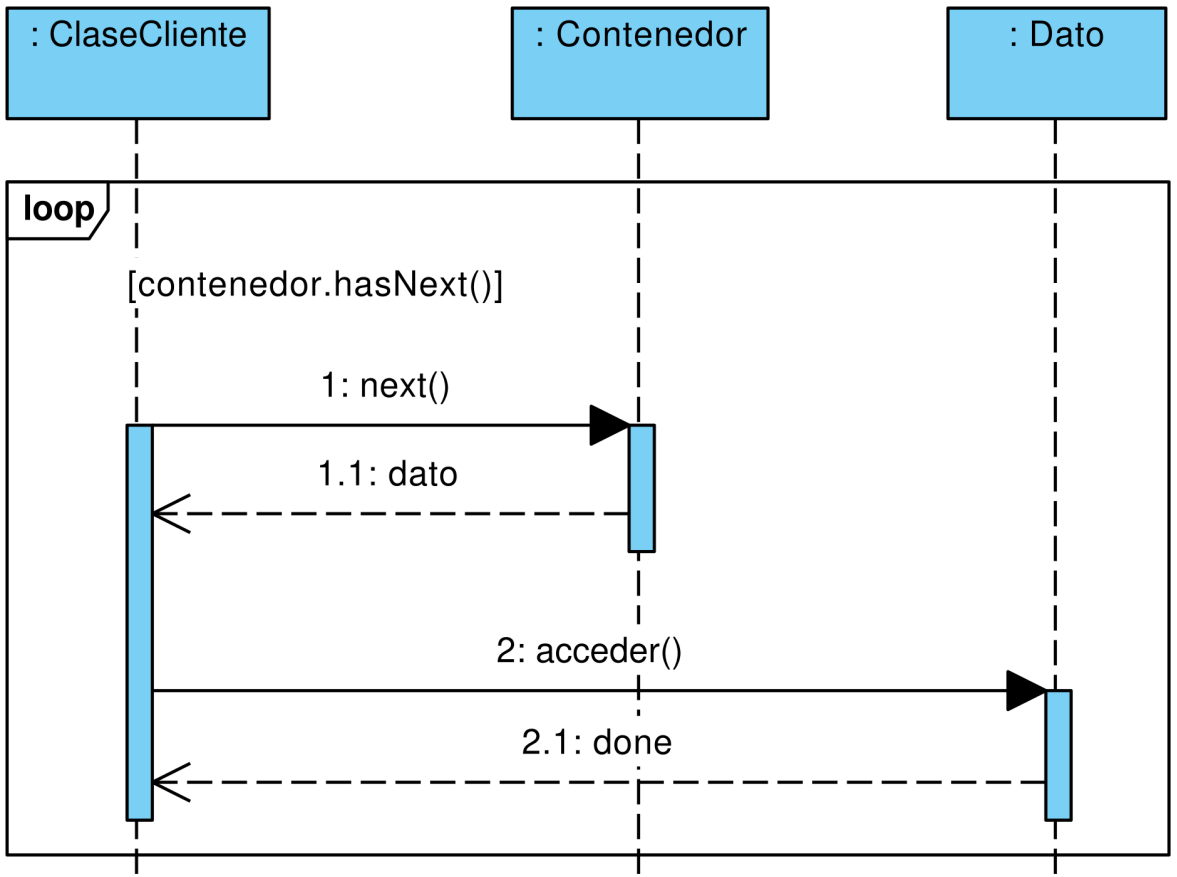
¿Para qué?

Acceder a los contenidos de un contenedor de manera secuencial sin tener conocimiento de la estructura interna de los mismos

- El contenedor puede ser una colección de datos u objetos
- Nos permite ocultar la estructura interna de los datos
- A nivel básico se implementan 2 métodos
  1. `hasNext() : bool`
  2. `next() : object`

Iterator: Implementación y Uso





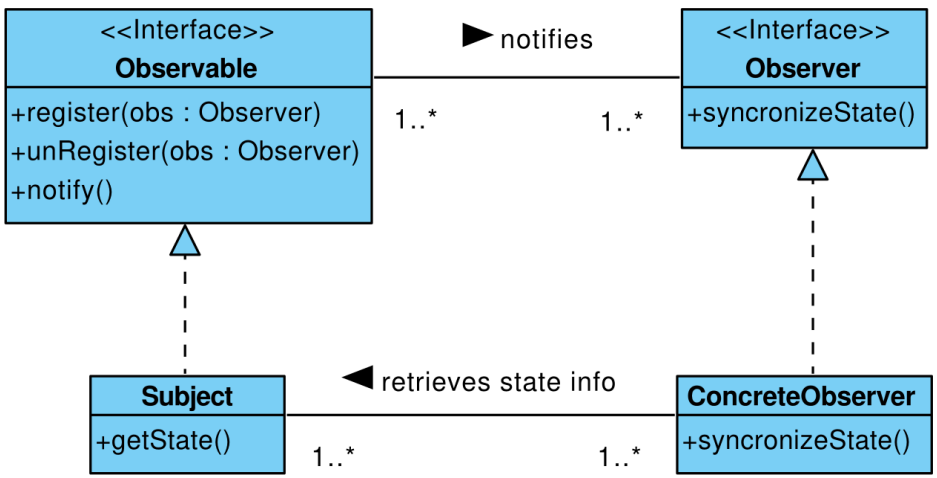
Observer

¿Para qué?

Mantener un mecanismo de comunicación consistente entre un objeto principal y sus dependientes

- Permite sincronizar el estado de un objeto con otros que dependan de él
- El observador es el objeto dependiente
- Los observadores deben registrarse para poder recibir notificaciones de cambio
- Los `Listener` de `Swing` son observadores

Observer Implementación



Dos modos de uso

1. Modelo *pull*
   - El objeto debe implementar una interfaz que permita a los observadores obtener su estado
2. Modelo *push*
   - El objeto enviará directamente información sobre cambios de estado a los observadores

Otros tipos de patrones de diseño

- Patrones de concurrencia
  - Objetos activos
  - *Lock*, exclusión mutua, *mutex*
  - *Monitor*
  - *Thread pool*
- Patrones arquitecturales
  - Modelo vista controlador (MVC)
  - Publicador/Subscriptor
  - *Data access object* (DAO)
- Otros
  - Inyección de dependencias
  - Encadenado de métodos