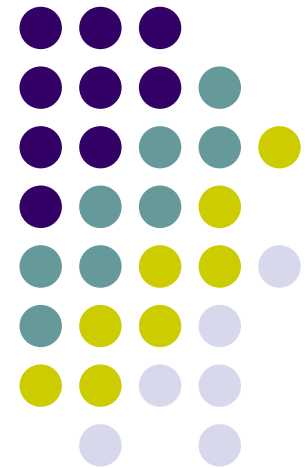


Patrones de diseño

Patrón básico *Handler*



Patrones de diseño

Introducción



- **Objetivos:**
 - Diseño específico para el problema, pero general para poder adecuarse a futuros requisitos
 - Evitar el rediseño en la medida de lo posible
 - Evitar resolver cada problema partiendo de cero
 - Reutilizar soluciones que han sido útiles en el pasado
- **Idea:**
 - Patrones recurrentes de clases y comunicación entre objetos en muchas soluciones de diseño
 - Reutilizar diseños abstractos que no incluyan detalles de la implementación

Patrones de diseño

Introducción



- Qué son:
 - Descripción del problema y la esencia de su solución, que se puede reutilizar en casos distintos
 - Solución adecuada a un problema común
 - Documentar la experiencia en el diseño
- Tipos:
 - De creación: implica el proceso de instanciar objetos
 - Estructurales: composición de objetos
 - De comportamiento: cómo se comunican los objetos, cooperan y distribuyen las responsabilidades para lograr sus objetivos

Patrones de diseño

Estructura de un patrón



- Nombre del patrón
 - Describe el problema de diseño, soluciones y consecuencias
 - Vocabulario de diseño
- Problema
 - Describe cuándo aplicar el patrón (aplicabilidad)
 - Explica el problema y su contexto (motivación)
- Solución
 - Elementos que forman el diseño, relaciones, responsabilidades
 - No un diseño concreto, sino una plantilla que puede aplicarse en muchas situaciones distintas
- Consecuencias
 - Resultados, ventajas e inconvenientes de aplicar el patrón
 - Por ejemplo: relación entre eficiencia en espacio y tiempo, cuestiones de implementación, etc.

Patrón básico *Handler*

Propósito

- Sirve para manejar *identificadores* de objetos de manera independiente a su implementación
- Permite cambiar fácilmente la implementación de un identificador (`int`, `String`, ...) a cualquier tipo básico o clase primitiva, sea sencilla o compuesta

Motivación

- A veces una clase tiene un identificador de cierto tipo, y puede que el tipo cambie en futuras versiones de la clase
- Subclases de una clase común usan identificadores de distinto tipo



Patrón básico *Handler*

Motivación



- Ej: Una aplicación de gestión de alumnos
 - Los alumnos se identifican por un NIA de 10 dígitos

Alumno
-nia: long
+ esMenor (Alumno otro): boolean

```
class Alumno {
    private long nia;
    public Alumno (long nia) {
        this.nia = nia;
    }
    public long getNia () {
        return this.nia;
    }
    public boolean esMenor (Alumno otro) {
        if (this.nia < otro.getNia())
            return true;
        else return false;
    }
    ...
}
```

Patrón básico *Handler*

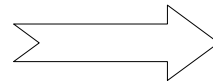
Motivación



- Ej: Una aplicación de gestión de alumnos
 - Los alumnos se identifican por un NIA de 10 dígitos y una letra

Alumno
-nia: long
+ esMenor (Alumno otro): boolean

```
class Alumno {
    private long nia;
    public Alumno (long nia) {
        this.nia = nia;
    }
    public long getNia () {
        return this.nia;
    }
    public boolean esMenor (Alumno otro) {
        if (this.nia < otro.getNia())
            return true;
        else return false;
    }
    ...
}
```



Alumno
-nia: String
+ esMenor (Alumno otro): boolean

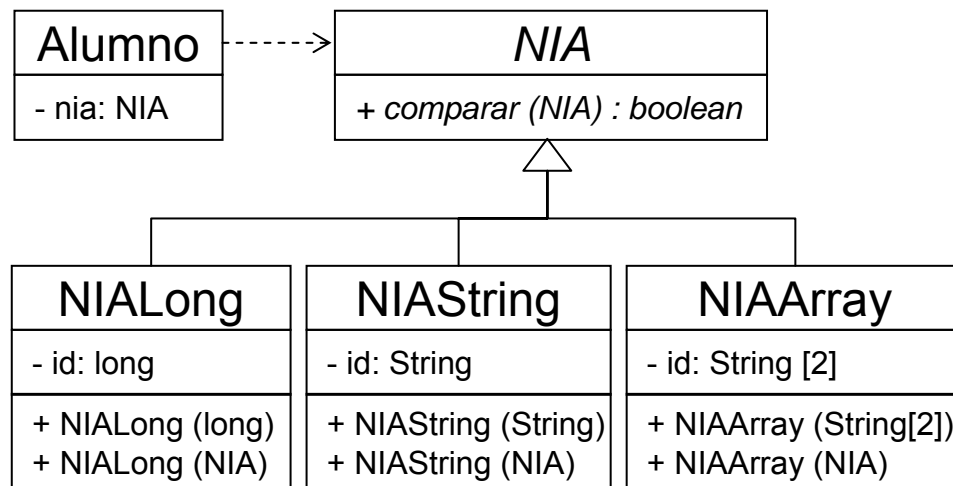
```
class Alumno {
    private String nia;
    public Alumno (String nia) {
        this.nia = nia;
    }
    public String getNia () {
        return this.nia;
    }
    public boolean esMenor (Alumno otro) {
        if (this.nia.compareTo(otro.getNia()))
            return true;
        else return false;
    }
    ...
}
```

Patrón básico *Handler*

Motivación



- Solución:
 - usar atributo del tipo especificado: no es flexible, si el tipo cambia puede requerir muchos cambios en el resto del sistema
 - definir una clase de envoltura para el identificador



```
class Alumno {
    private NIA nia;
    public Alumno (NIA nia) {
        this.nia = nia;
    }
    public NIA getNia () {
        return this.nia;
    }
    public boolean esMenor (Alumno otro) {
        if (this.nia.comparar(otro.getNia()))
            return true;
        else return false;
    }
    ...
}
```

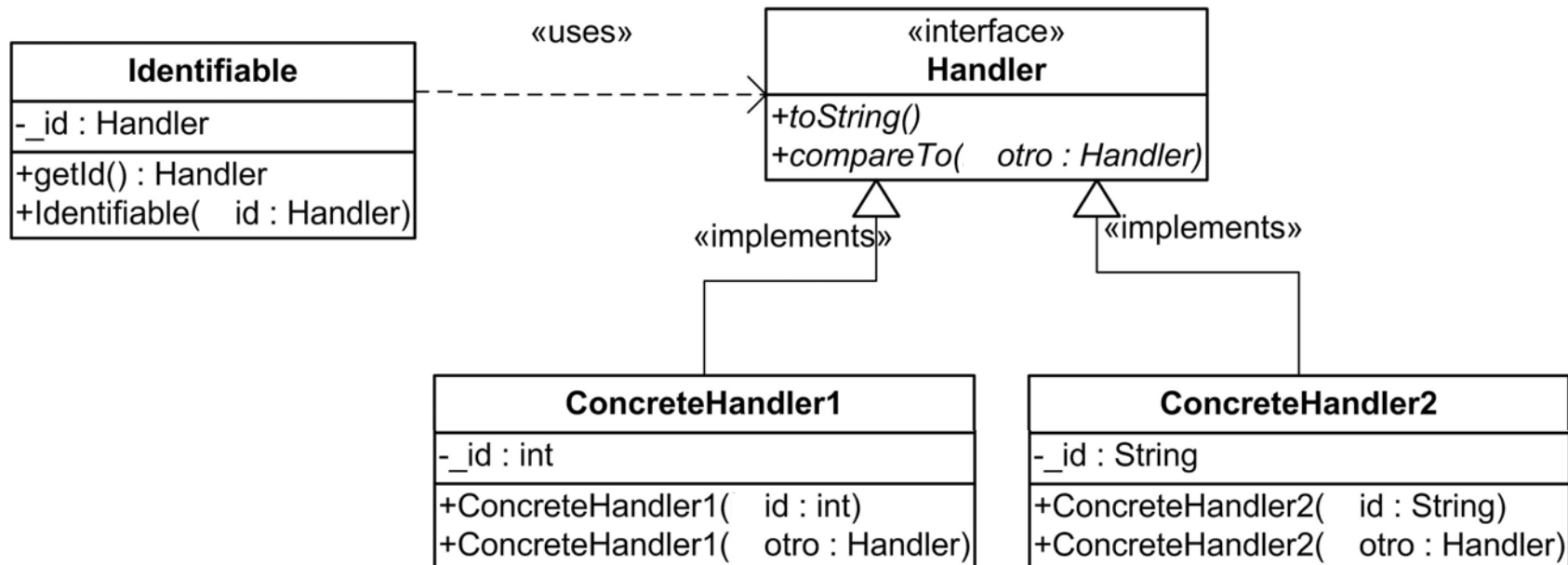

Patrón básico *Handler*

Aplicabilidad



- Usa el patrón *Handler* cuando:
 - tienes alguna clase que usa algún tipo de identificador

Estructura



Patrón básico *Handler*

Participantes



- **Identifiable (*Alumno*)**: clase cliente que necesita identificar a sus objetos a través de un atributo `identificador`
- **Handler (*NIA*)**: interfaz para declarar los identificadores de los objetos de la clase `Identifiable`
- **ConcreteHandler (*NIALong, NIAString, NIAArray*)**: implementación concreta de la interfaz `Handler`

Patrón básico *Handler*

Ejemplo de Implementación



```
interface Handler {
    String toString();
    int compareTo(Handler otro);
}

class ConcreteHandler1 implements Handler {
    private int _id;
    ConcreteHandler1 (String id) throws NumberFormatException {
        _id = new Integer(id).intValue();
    }
    ConcreteHandler1 (Handler otro) throws NumberFormatException {
        _id = new Integer(otro.toString()).intValue();
    }
    public String toString() { return new Integer(_id).toString(); }
    public int compareTo (Handler otro) {
        return toString().compareTo(otro.toString());
    }
}

class Identifiable {
    private Handler _id;
    public Identifiable(String id) { _id = new ConcreteHandler1(id); }
}
```

Patrón básico *Handler*

En java...



- `java.lang.Comparable`
- Implementado por clases de envoltura (`Integer`, `Long`, etc.), `String`, `File`, `Date`, ...
- `public int compareTo (Object o) throws ClassCastException`
 - Invariantes:
 - $\text{sgn}(x.\text{compareTo}(y)) = -\text{sgn}(y.\text{compareTo}(x))$
 - $(x.\text{compareTo}(y) > 0 \text{ and } y.\text{compareTo}(z) > 0) \rightarrow x.\text{compareTo}(z) > 0$
 - $x.\text{compareTo}(y) = 0 \rightarrow \text{sgn}(x.\text{compareTo}(z)) = \text{sgn}(y.\text{compareTo}(z))$ para todo z
 - Consistente con `equals`:
 - $(x.\text{compareTo}(y) = 0) = (x.\text{equals}(y))$