Ingeniería de Software II

Patrones de Diseño

Facultad de Informática
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Universidad Pontificia de Salamanca en Madrid

Historia

- El arquitecto Christopher Alexander escribió varios libros sobre planificación urbana y arquitectura de edificios e introdujo el concepto de patrón.
- 1987 - Ward Cunningham y Kent Beck usaron varias de las ideas de Alexander para desarrollar un pequeño lenguaje de patrones para programadores nóveles de Smalltalk.
- 1991 - Jim Coplien publica *Advanced C++ Programming Styles and Idioms* que compila un catálogo de *idioms* para C++.
- 1994 - Se publica el libro *Design Patterns : Elements of Reusable ObjectOriented Software* por GoF.

Ventajas

- Ayudan a los desarrolladores de software a resolver problemas comunes encontrados durante todo el proceso de la Ingeniería del Software.
- Ayudan a crear un vocabulario y un lenguaje compartidos para comunicar comprensión profunda (*insight*) y experiencia sobre dichos problemas y sus soluciones.
- Comunican las razones de determinadas decisiones de diseño y no únicamente las soluciones.
- Lo hacen dando un nombre y codificando formalmente dichos problemas y sus soluciones.

¿Qué es un Patrón?

- Un patrón es una información con nombre que captura la estructura esencial de soluciones comprobadas a problemas recurrentes en un cierto contexto y sistema de fuerzas.
- Un patrón es a un tiempo una cosa que ocurre en el mundo real y la regla que nos dice cómo crear esa cosa, la descripción del proceso que da lugar a esa cosa (patrón generativo).
- Un anti-patrón nos dice lo que no hay que hacer; describen malas soluciones.

Tipos de Patrones

- **El foco inicial, sobre todo a raíz del libro de GoF, fue sobre patrones de diseño.**
- **Hay otros tipos:**
  - patrones de análisis
  - patrones organizativos
  - organización del desarrollo
  - proceso software
  - planificación de proyectos
  - ingeniería de requerimientos
  - . . .

Patrones: Clasificación

- **Según Frank Buschman los patrones se pueden clasificar en:**
  - **ARQUITECTURA**: Expresan una estructura fundamental de organización de los sistemas SW. Proveen un conjunto de subsistemas predefinidos, especificando sus responsabilidades y relaciones.
  - **DISEÑO**: Proporcionan un esquema para refinar subsistemas o componentes. Resuelven problemas específicos de diseño.
  - **IDIOMS**:Son específicos de un lenguaje de programación. Describen cómo implementar ciertos aspectos de un problema utilizando las características de un lenguaje de programación.

Patrones de Diseño: Clasificación (II)

- **Teniendo en cuenta el nivel de detalle:**

+ ↑
- **ARQUITECTURA**: Afectan a la estructura global del sistema.
- **DISEÑO**: Definen microestructuras de subsistemas y componentes.
- ↓ -
- **IDIOMS**: Se centran en detalles de la estructura y comportamiento de un componente.

Patrones de Arquitectura

- **La mejor referencia:**

  F. Buschmann et al: *Pattern-OrientedSoftware Architecture: A System of Patterns*. Ed. Wiley
- **Los Patrones de Arquitectura propuestos:**
  - **Layers**: Permite estructurar aplicaciones que se pueden descomponer en grupos de subtareas, donde cada grupo está en un determinado nivel de abstracción.
  - **Pipes & Filters:**Provee una estructura para sistemas que procesan un flujo de datos. Cada etapa del proceso es encapsulada como un filtro. Los datos se pasan entre filtros adyacentes mediante Pipes. Recombinando filtros obtenemos familias de sistemas relacionados.

Patrones de Arquitectura (II)

- **Blackboard:** Útil para sistemas en que no se conoce una solución o estrategia determinista. Varios subsistemas especializados ensamblan su conocimiento para construir una posible solución parcial.
- **Broker:** Permite estructurar sistemas distribuidos desacoplados que interaccionan mediante invocación de servicios remotos. Un componente Broker es responsable de coordinar la comunicación, así como de transmitir resultados y excepciones.
- **Model-View-Controller:** Divide una aplicación interactiva en 3 componentes. El Model contiene la información y funcionalidad principal. Views muestran información al usuario. Controllers gestionan la entrada de usuario. Un mecanismo de propagación de cambios asegura la consistencia entre el modelo y la interfaz de usuario.

Patrones de Arquitectura (III)

- **Presentation-Abstraction-Control:** Estructura una aplicación interactiva como una jerarquía de agentes que cooperan. Cada agente es responsable de un determinado aspecto de la funcionalidad y consta de tres componentes: Presentacion, Abstraccion y Control, que separan la interacción con el usuario de la funcionalidad central y la comunicación con otros agentes.
- **Microkernel:** Separar un mínimo núcleo funcional de funcionalidad extendida y partes específicas del cliente. El Microkernel también sirve como punto donde engarzar estas piezas y coordinar su colaboración.
- **Reflection:** Proporciona un mecanismo para cambiar la estructura y el comportamiento del sistema dinámicamente. La aplicación se divide en dos partes: un meta-nivel y un nivel-base. El meta-nivel hace al software autoconsciente.

© Maria Luisa Díez, Héctor Castán, Paloma Centenera
UNIVERSIDAD PONTIFICIA DE SALAMANCA EN MADRID
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software

Patrones de Diseño

- **Un Patrón de Diseño identifica, abstrae y nombra los aspectos elementales de una estructura de diseño.**
- **Identifica:**
  - participantes y sus instancias
  - colaboraciones
  - distribución de responsabilidades
- **Cada patrón de diseño se centra en un problema de diseño particular y describe cuando es o no aplicable a la luz de otras restricciones de diseño.**

© Maria Luisa Díez, Héctor Castán, Paloma Centenera
UNIVERSIDAD PONTIFICIA DE SALAMANCA EN MADRID
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software

Patrones de Diseño (II)

- **El libro que más ha contribuido a la popularización de los patrones:**

  Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides *Design Patterns: Elements of Reusable Object Oriented Software* Addison Wesley 1995
  - Compila un catálogo de 23 patrones de diseño

© Maria Luisa Díez, Héctor Castán, Paloma Centenera
UNIVERSIDAD PONTIFICIA DE SALAMANCA EN MADRID
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software

Patrones de Diseño (III)

- **La descripción de los patrones se realiza mediante una 'ficha' con la siguiente estructura:**
  - **Nombre y Clasificación:** Un nombre que sucintamente transmita la esencia del patrón. Ha de escogerse bien ya que formará parte del vocabulario de diseño.
  - **Intención:** Una breve descripción de cuál es el problema que el patrón resuelve.
  - **Alias:** Otros nombres, si existen, del patrón.
  - **Motivación:** Un escenario que ilustra el problema de diseño y cómo el patrón lo resuelve. Ayuda a comprender la descripción más abstracta del patrón del resto de apartados.
  - **Aplicabilidad:** Situaciones en que el patrón es aplicable. Cómo reconocerlas.
  - **Estructura:** Representación gráfica de las clases del patrón utilizando algún tipo de notación, diagramas de interacción, etc.

Patrones de Diseño (IV)

- **Participantes:** Clases participantes y sus responsabilidades
- **Colaboraciones:** Cómo colaboran las clases para llevar a cabo sus responsabilidades.
- **Consecuencias:** Cómo el patrón cumple los objetivos. Compromisos y resultados al usar el patrón.
- **Implementación:** Ayudas, trucos, elementos específicos del lenguaje a la hora de implementar el patrón.
- **Código de ejemplo:** Fragmentos que ilustran la implementación en algún(os) lenguaje(s).
- **Usos conocidos:** Ejemplos de uso del patrón en sistemas conocidos.
- **Patrones relacionados:**Patrones relacionados, alternativos para el problema, etc.

Patrones de Diseño (V)

AMBITO
a qué es aplicable el patrón

TIPO

	CREACIONAL Creación de objetos	ESTRUCTURAL Composición de objetos	COMPORTAMIENTO Distribución de responsabilidades
CLASES	Delegan parte de la creación a las subclases	Usan la herencia para componer clases	Usan la herencia para describir algoritmos y flujos de control
OBJETOS	Delegan parte de la creación a otro objeto	Describen formas de ensamblar objetos	Cómo grupos de objetos colaboran para llevar a cabo una tarea

Patrones de Diseño (VI)

	CREACIONAL	ESTRUCTURAL	COMPORTAMIENTO
Clase	Factory Method	Adapter (class)	Interpreter Template Method
Objeto	Abstract Factory Builder Protoype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Patrones creacionales

- **Abstraen el proceso de la instanciación (creación)**
- **Ayuda a que el sistema sea independiente de cómo se crean, componen y representan los objetos.**
- **Un patrón creacional de clase usa la herencia para las distintas clases que instancia, mientras que uno de objeto delega la instanciación a otro objeto.**

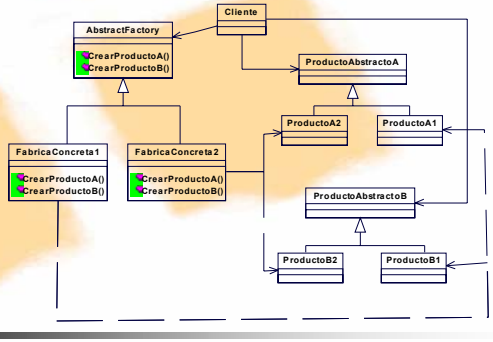
Patrones creacionales: *Abstract Factory*

- **Propósito**

  Proporcionar una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas
- **Alias➔ Kit**
- **Aplicabilidad**
  - Cuando un sistema deba ser independiente de la creación, composición y representación de sus productos
  - Cuando un sistema deba ser configurado con una de las múltiples familias de productos
  - Cuando un conjunto de objetos relacionados se diseña para ser usado conjuntamente
  - Cuando se desea proporcionar una biblioteca de productos de los que sólo se quiere conocer su interfaz

Patr...: *Abstract Factory* (II)

- **Estructura**



Patr...: *Abstract Factory* (II)

- **Participantes**
  - **Abstract Factory**: declara una interfaz para operaciones que crean productos abstractos
  - **Fabrica Concreta**: implementa las operaciones para crear productos concretos
  - **ProductoAbstracto**: declara una interfaz para un tipo de producto
  - **ProductoConcreto**: define un producto para ser creado por la fabrica concreta correspondiente. Implementa la interfaz del producto abstracto
  - **Cliente**: usa solamente las interfaces de las fabricas y los productos abstractos.

Patr...: *Abstract Factory* (III)

- **Colaboraciones**
  - Se suele crear, en tiempo de ejecución, una única instancia de *FabricaConcreta*, que crea productos con una implementación específica. Para crear objetos con distintas implementaciones el *Cliente* debe disponer de varias *FabricaConcreta* distintas.
  - La fábrica abstracta encarga la creación de productos a sus subclases
- **Consecuencias**
  - Aísla las clases concretas.
  - Permite intercambiar fácilmente familias de productos.
  - Proporciona consistencia entre productos.
  - Dificulta la incorporación de nuevas clases de productos

Patr...: *Abstract Factory* (IV)

- **Implementación**
  - Usar clases como singleton
  - La fábrica sólo declara una interfaz para crear objetos (puede implementarse a través de otros patrones).
  - Se pueden definir fábricas extensibles que puedan facilitar la creación de nuevas clases de productos

Patr...: *Abstract Factory* (V)

- **Ejemplo de código:**

```
class FabricaLaberintos {
  public:
    FabricaLaberintos();
    virtual Laberinto* HacerLaberinto() const {
        return new Laberinto; }
    virtual Pared* HacerPared() const { return new Pared; }
    virtual Habitacion* HacerHab(int n) const {
        return new Habitacion(n); }
    virtual Puerta* HacerPuerta(Habitacion* r1, Habitacion* r2)
    const { return new Puerta(r1, r2); }
};
```


Patr...: *Abstract Factory* (VI)

```
Laberinto* JuegoLaberinto::CrearLaberinto (FabricaLaberintos & fabrica) {
    Laberinto* UnLaberinto = fabrica.HacerLaberinto();
    Habitacion* r1 = fabrica.HacerHabitacion(1);
    Habitacion* r2 = fabrica.HacerHabitacion(2);
    Puerta* unaPuerta = fabrica.HacerPuerta(r1, r2);
    UnLaberinto->Ad_Habitacion(r1);
    UnLaberinto-> Ad_Habitacion(r2);
    r1->FijarLado(Norte, fabrica.HacerPared());
    r1->FijarLado(Este, unaPuerta);
    r1->FijarLado(Sur, fabrica.HacerPared());
    r1->FijarLado(Oeste, fabrica.HacerPared());
    r2->FijarLado(Norte, fabrica.HacerPared());
    r2->FijarLado(Este, fabrica.HacerPared());
    r2->FijarLado(Sur, fabrica.HacerPared());
    r2->FijarLado(Oeste, unaPuerta);
    return UnLaberinto;
}
```

Patr...: *Abstract Factory* (VII)

❑ **Creamos *FrabricaLaberintoHechizado***

```
class FabricaLaberintoHechizado : public FabricaLaberintos {
   public:
        FabricaLaberintoHechizado();
        virtual Habitacion* HacerHabitacion(int n) const{
            return new HabitacionHechizada(n, LanzarHechizo()); }
        virtual Puerta* HacerPuerta(Habitacion* r1, Habitacion* r2) const {
            return new PuertaHechizada(r1, r2); }
   protected:
        Hechizo* LanzarHechizo() const;
};
```


Patr...: *Abstract Factory* (VIII)

```
Pared* FabricaLaberintoconBombas::HacerPared () const {
    return new ParedconBomba;
}
Habitacion* FabricaLaberintoconBombas ::HacerHabitacion(int n) const {
   return new HabitacionconBomba(n);
 }
JuegoLaberinto juego;
FabricaLaberintoconBombas fabrica;
juego.CrearLaberinto(fabrica);
```


Patrones creacionales: *Singleton*

❑ **Propósito**

Asegurar que una clase tiene una instancia única y proporcionar un punto de acceso global a ella

❑ **Motivación**

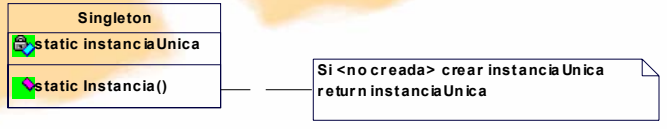
Algunas clases tienen una única instancia. Una solución sería utilizar una variable global pero ello no evita que podamos instanciar múltiples objetos. Una solución mejor es hacer que la propia clase gestione su única instancia.

Patrones creacionales: *Singleton* (II)

❑ **Aplicabilidad**

- Queremos que una clase tenga una instancia única globalmente accesible.
- La instancia única podría ser extensible por una subclase, y los clientes podrían usar una instancia extendida sin modificar su código.

❑ **Estructura**



Patrones creacionales: *Singleton* (III)

- **Participantes**
  - **Singleton**: define una operación Instancia que deja que los clientes accedan a su única instancia. Instancia es una operación de clase (en C++ una función miembro estática). Puede ser la responsable de crear su única instancia
- **Colaboraciones**
  - Los clientes acceden a la instancia de Singleton sólo a través de la operación Instancia.

Patrones creacionales: *Singleton* (IV)

- **Consecuencias**
  - Controla el acceso a la única instancia
  - Reduce el espacio de nombre
  - Permite el refinamiento de operaciones y representaciones
  - Permite un numero variable de instancias
- **Implementación**
  - Para garantizar una única instancia se debe ocultar la operación que crea las instancias

Patrones creacionales: *Singleton* (V)

```
class Singleton {
   public:
      static Singleton* Instance();
   protected:
      Singleton();
   Singleton;
   private:
      static Singleton* _instance;
};
```

```
Singleton* Singleton::_instance=0;

Singleton* Singleton::Instance () {
      if (_instance == 0) {
            _instance = new

      }
      return _instance;
}
```


Patrones creacionales: *Singleton* (VI)

- **Uso del patrón para gestionar varias aperturas de un archivo**

```
class Archivo{
    //Parte estática, control de singleton:
    static private Archivo instancia = null;
    static int nAperturas = 0;
    static private String nombre;
```

```
//método para obtener las instancias de "Archivo"
static public Archivo Abrir(String Nombre){
      if (instancia == null){ //es la primera apertura
            instancia = new Archivo(Nombre);
      }
      if (instancia.nombre.equals(Nombre)){
            nAperturas++;
            return instancia;

      }
      else
            return null;
}
```

Patrones creacionales: *Singleton* (VII)

```
static public boolean Cerrar(String Nombre)
{
  if (Nombre.equals(instancia.nombre)){
      nAperturas--;
      if(nAperturas == 0){
            instancia = null;
      }
      return true;
  }
  else
      return false;
}
```

```
//Información añadida a las instancias
//creadas

private Archivo(String Nombre)
{
      this.nombre=new String(Nombre);

}

public String obtenerNombre()
{
      //devolvemos una "copia" del nombre
      return new String(nombre);
}
}
```


Patrones estructurales

- **Definen cómo componer clases y objetos para formar estructuras más grandes**
- **Los patrones estructurales de clases utilizan la herencia para componer interfaces o implementaciones**
- **Los patrones estructurales de objetos describen formas de componer objetos para realizar nuevas funcionalidades, en lugar de componer interfaces o implementaciones**

Patrones estructurales: *Composite*

- **Propósito**

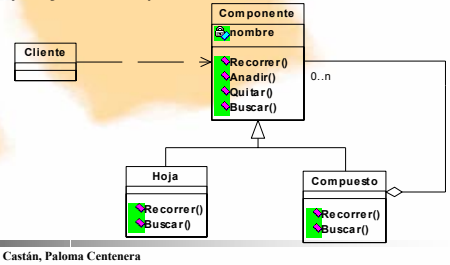
  Componer objetos en una estructura de árbol para representar relaciones todo-parte y que los clientes traten uniformemente objetos simples y compuestos

- **Motivación**

  En muchos casos es necesario definir clases y componentes que almacene sus objetos. Esta aproximación puede tener problemas ya que el código que maneja estas clases debe tratar a los objetos contenedores y a los objetos simples de forma distinta. Este patrón permite representar mediante una clase abstracta tanto a los objetos simples como a los contenedores

Patrones estructurales: *Composite* (II)

- **Aplicabilidad**
  - Cuando queremos representar jerarquías todo-parte
  - Los clientes deben ignorar las diferencias entre objetos individuales y objetos compuestos.
- **Estructura**



Patrones estructurales: *Composite* (III)

- **Participantes**
  - **Componente:**
    - ✓ Declara la interfaz para los objetos de la composición
    - ✓ implementa comportamiento por defecto para dicho interfaz
    - ✓ declara interfaz para acceder y manipular objetos hijos
  - **Compuesto:** Define el comportamiento de los objetos que pueden contener a otros y almacena componentes hijos
  - **Hoja:** Define el comportamiento de los objetos primitivos
  - **Cliente:** Interacciona los objetos de la composición a través del interfaz Componente.

Patrones estructurales: *Composite* (IV)

- **Consecuencias**
  - Hacen que el cliente sea más sencillo al otorgar uniformidad de acceso a los compuestos y a las hojas
  - Facilita la tarea de añadir nuevos tipos de componentes
  - Dificulta la tarea de de restringir los componentes que forman el compuesto
- **Implementación**
  - Maximizar la interfaz de *Componente*
  - Ordenación de los hijos
  - Borrado de los componentes
  - Estructura de almacenamiento

Patrones estructurales: *Composite* (IV)

```
class Lista;
class Elemento{
protected:
      char *Nombre;
      char Hora_Creaccion [9];
      char Hora_Actualizacion [9];
      char Activo;
      Lista *Sig_Lista;
public:
      Elemento (char *);
      virtual void Dir () const =0;
      virtual void DirRecursivo(int n = 0) const =0;
      virtual Elemento* Buscar (char *) =0;
      virtual void Activar() = 0;
      virtual void Desactivar() = 0;
      const char * DarNombre() {return Nombre;};
      virtual Lista* Dar_Sig_Lista() = 0;
      friend class Lista;
};
```

```
class Directorio;
class Lista{
      Elemento *Info;
public:
      Lista () {Info = NULL;};
      ~Lista();
      void Insertar (char *,int);
      void Eliminar (char *);
      Directorio* Cd (char *) const;
      Elemento* DaInfo() {return Info;};
      friend class Elemento;
      friend class Archivo;
      friend class Directorio;
};
```


Patrones estructurales: *Composite* (V)

```
class Archivo:public Elemento{
      int Tamano;
public:
      Archivo (char *nombre, int tam): Elemento
      (nombre), Tamano (tam) {};
      Archivo (const Archivo &a): Elemento(a.Nombre),
      Tamano (a.Tamano){
        strcpy(Nombre,a.Nombre);
        strcpy(Hora_Creaccion,a.Hora_Creaccion);
        strcpy(Hora_Actualizacion,a.Hora_Actualizacion);
      };
      void Dir () const;
      void DirRecursivo(int n = 0) const;
      Elemento* Buscar(char *) ;
      void Activar() {Activo = '&';};
      void Desactivar() {Activo = ' ';};
      Lista* Dar_Sig_Lista() {return Sig_Lista;};
};
```

```
class Directorio:public Elemento{
      Lista *SubDir;
public:
      Directorio (char *nombre): Elemento (nombre)
      {SubDir = new Lista;};
      Directorio (const Directorio &d):
      Elemento(d.Nombre){
        strcpy(Nombre,d.Nombre);
        strcpy(Hora_Creaccion,d.Hora_Creaccion);
        strcpy(Hora_Actualizacion,d.Hora_Actualizacion);
        SubDir = d.SubDir;};
      void Dir () const;
      void DirRecursivo(int n = 0) const;
      void DirRecursivo2() const;
      void Mostrar () const;
      Elemento* Buscar (char *) ;
      void Activar() {Activo = '*';};
      void Desactivar() {Activo = ' ';};
      Lista* Dar_Sig_Lista() {return Sig_Lista;};
      Lista* Dar_SubDir() {return SubDir;};
};
```

Patrones estructurales: *Composite* (VI)

```
#include "Elemento.h"
#include <string.h>
#include <iostream.h>
#include <time.h>
#include <iomanip.h>

Elemento* Directorio::Buscar (char *nom)  {
  if (strcmp(Nombre, nom) == 0)
     return (Elemento *) this;
  else
     if (Sig_Lista->Info != NULL)
            return Sig_Lista->Info->Buscar(nom);
     else{
            Directorio *aux = new Directorio("ERROR");
            return aux;
     };
};
```

```
void Directorio::Mostrar () const{
  Elemento *aux;
  if (SubDir->Info != NULL){
     aux = SubDir->Info;
     while (aux != NULL){
            aux->Dir();
            aux = aux->Dar_Sig_Lista()->Info;
     };
  };
};
void Directorio::Dir () const{
     cout << Nombre;
     cout << setw(25 - strlen(Nombre)) <<
     Hora_Creaccion;
     cout << setw(10) << Hora_Actualizacion;
     cout << setw(10) << "<DIR>" << endl;
};
```

© Maria Luisa Díez, Héctor Castán, Paloma Centenera
UNIVERSIDAD PONTIFICIA DE SALAMANCA EN MADRID
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software

Patrones estructurales: *Composite* (VII)

```
void Directorio::DirRecursivo(int n) const{
  if (SubDir->Info != NULL){
     if (n>0){
        for (int i = 0; i<n;i++) cout << " ";
        cout << "|" ;
        for ( i= 0;i<3;i++) cout << "-";
     };
     cout << Activo <<  Nombre << endl;
     if (SubDir->Info != NULL)
            SubDir->Info->DirRecursivo(n += 3);
     if (Sig_Lista->Info != NULL)
            Sig_Lista->Info->DirRecursivo(n -= 3);
  }
  else{
     for (int i=0;i<n;i++) cout << " ";
     cout << "|" ;
     for (i=0; i<3; i++) cout << "-";
     cout << Activo << Nombre << endl;
     if (Sig_Lista->Info != NULL)
            Sig_Lista->Info->DirRecursivo (n);
  };};
```

```
void Directorio::DirRecursivo2() const{
     Directorio *aux = (Directorio *)SubDir->Info;
     aux->DirRecursivo();
};
```

© Maria Luisa Díez, Héctor Castán, Paloma Centenera
UNIVERSIDAD PONTIFICIA DE SALAMANCA EN MADRID
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software

Patrones estructurales: *Proxy*

- **Propósito**

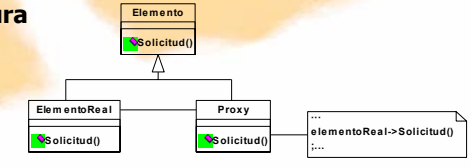
  Proporcionar un sustituto o suplente para otro objeto, de forma que controle el acceso a él
- **Alias➔Sustituto**
- **Motivación**

  La creación de un objeto para controlar el acceso a otro permite hacer transparente al cliente la creación e inicialización de objetos o la ubicación real de los mismos. El uso de este tipo de objetos 'sustitutos' permite aumentar la extensibilidad y la reutilización

© Maria Luisa Díez, Héctor Castán, Paloma Centenera
UNIVERSIDAD PONTIFICIA DE SALAMANCA EN MADRID
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software

Patrones estructurales: *Proxy* (II)

- **Aplicabilidad**
  - Cuando se desea hacer transparente el acceso a un objeto que puede estar en el mismo espacio de direcciones o en otro
  - Cuando se desea controlar la creación de objetos 'costosos'
  - Cuando se desea añadir seguridad o control al acceso sobre un objeto existente
- **Estructura**



© Maria Luisa Díez, Héctor Castán, Paloma Centenera
UNIVERSIDAD PONTIFICIA DE SALAMANCA EN MADRID
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software

Patrones estructurales: *Proxy* (III)

- **Participantes:**
  - **Proxy**:
    - ✓ Mantiene la referencia que le permite el acceso al ElementoReal
    - ✓ Proporciona una interfaz idéntica a la de Elemento, de forma que el Proxy puede ser sustituido por el ElementoReal
    - ✓ Controla el acceso sobre el ElementoReal
    - ✓ Realiza las tareas especificas para proporcionar acceso remoto, carga diferida o protección y seguridad
  - **Elemento**: Define la interfaz comun para el ElementoReal y para el Proxy
  - **ElementoReal**: Define el objeto real al que el Proxy representa

Patrones estructurales: *Proxy* (IV)

- **Consecuencias**
  - Se introduce un nivel de indirección cuando se accede al objeto o elemento real que permite añadir funcionalidades
- **Implementación**
  - Sobrecarga de operadores en C++

Patrones estructurales: *Proxy* (V)

Proxy para control de acceso

```
abstract class DatosPersonalesAbstracto
{
   abstract void Datos(String Dni,
         String Nom,String Ape,String Dir);
   abstract String DNI();
   abstract String Nombre();
   abstract String Apellidos();
   abstract String Direccion();
}
class DatosPersonales extends
    DatosPersonalesAbstracto
{
   private String dni = null;
   private String nombre = null;
   private String apellidos = null;
   private String direccion = null;

   public void Datos(String Dni,String Nom,String Ape,String Dir){
      dni = new String(Dni);
      nombre = new String(Nom);
      apellidos = new String(Ape);
      direccion  = new String(Dir);
    }
    public String DNI(){  return new String(dni);  }

    public String Nombre(){  return new String(nombre);  }

    public String Apellidos(){  return new String(apellidos);  }

    public String Direccion(){ return new String(direccion);  }

  }
```


Patrones estructurales: *Proxy* (VII)

```
class ProxyDatosPersonalesSeg extends
      DatosPersonalesAbstracto{
   private DatosPersonales datos = new DatosPersonales();
   private String id = null;
   private String clave = null;
   public ProxyDatosPersonalesSeg(String ID, String Clave){
     id = new String(ID);
     clave = new String(Clave);
     datos = new DatosPersonales();
   }
   public void Datos(String Dni,String Nom,String Ape,String
     Dir){
     if (Verificar()){ datos.Datos(Dni,Nom,Ape,Dir); }
   }
   public String DNI(){
     if (Verificar()){  return datos.DNI();   }
     return null;
   }
```

```
private boolean Verificar(){
    BufferedReader entrada=new BufferedReader(new
      InputStreamReader(System.in));
    String ID = null;
    String CLAVE = null;
    try{
       System.out.print("Identificacion: ");
       ID=entrada.readLine();
       System.out.print("clave: ");
       CLAVE=entrada.readLine();
      }catch(IOException e){System.out.println("Error");}
    if ((ID.equals(id)) && (CLAVE.equals(clave)))
      return true;
    else
      return false;
  }

public String Nombre(){
     if (Verificar()){  return datos.Nombre();  }
    return null;
  }
```

Patrones estructurales: *Proxy* (VIII)

```
public String Apellidos(){
      if (Verificar()){
         return datos.Apellidos();
      }
     return null;
   }

   public String Direccion(){
      if (Verificar()){
         return datos.Direccion();
      }
     return null;
   }

}
```


Patrones de comportamiento

- **Están relacionados con algoritmos y la asignación de responsabilidades entre objetos**
- **No son solo patrones de clases y objetos, también son patrones de las comunicaciones entre ellos**
- **Los patrones de comportamiento de clase utilizan la herencia para distribuir el comportamiento entre clases**
- **Los patrones de comportamiento de objeto utilizan la composición de objetos en lugar de la herencia**

Patrones de comportamiento: *Iterator*

- **Propósito**

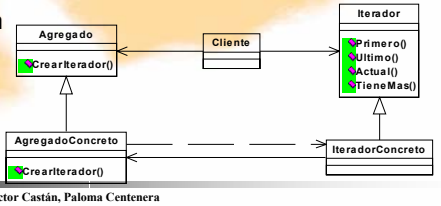
  Proporcionar acceso secuencial a un agregado sin exponer su representación

- **Motivación**

  Deberíamos tener acceso a un agregado como una Lista sin exponer su estructura interna. Podemos requerir distintos tipos de acceso sin cargar la interfaz con métodos para cada uno de ellos. Podemos requerir varios recorridos simultáneos de un mismo agregado.

Patrones de comport...: *Iterator* (II)

- **Aplicabilidad**
  - Para acceder a un agregado sin exponer su estructura interna.
  - Para soportar distintos tipos de recorrido en un mismo agregado.
  - Para proporcionar un interfaz común para recorrer diferentes estructuras de agregados (iteración polimórfica).
- **Estructura**



Patrones de comport...: *Iterator* (III)

- **Participantes:**
  - **Iterador**: define la interfaz para acceder y recorrer los elementos
  - **IteradorConcreto**:
    - ✓ Implementa la interfaz de Iterador
    - ✓ Realiza el seguimiento de la posición actual en el recorrido del agregado
  - **Agregado**: define la interfaz para la creación de un objeto Iterador
  - **AgregadoConcreto**: implementa la interfaz de creación del Iterador devolviendo una instancia del IteradorConcreto Adecuado

Patrones de comport...: *Iterator* (IV)

- **Consecuencias**
  - Permite variaciones en el recorrido de los agregados
- **Implementación**

Patrones de comport...: *Iterator* (V)

```
abstract class AgregadoAbstracto{
      protected int numElem=0;
      protected int maxElem;
      abstract IteradorAbstracto CrearIterador();
      int TamActual(){return numElem;};
}
class Lista extends AgregadoAbstracto{
      private String L[];

      Lista(int maxElem)  {
            L=new String[maxElem];
            this.maxElem=maxElem;
      }
      String Obtener(int n){
            String s=null;
            if (n<numElem)
              s = L[n];
            return s;
      }
```

```
      boolean Insertar(String s){
            boolean Ok=false;
            if (numElem<maxElem)  {
                  L[numElem++]=s; Ok=true;
            }
            return Ok;
      }
      boolean Eliminar(int i){
            boolean Ok=false;
            if((i<numElem) && (i>=0)){
                  for(int j=i;j<numElem-1;j++)
                        L[j]=L[j+1];
                  L[--numElem]=null;
                  Ok=true;
            }
            return Ok;
      }
      IteradorAbstracto CrearIterador(){
            return new IteradorLista(this);
      }
}
```


Patrones de comport...: *Iterator* (VI)

```
abstract class IteradorAbstracto{
      abstract void Primero();
      abstract void Siguiente();
      abstract boolean Lleno();
      abstract String ElementoActual();
}
class IteradorLista extends IteradorAbstracto{
      private Lista L;
      private int actual=0;
      IteradorLista(Lista L){this.L=L;}
      void Primero(){actual=0;};
      void Siguiente(){actual++;};
      boolean Lleno(){return (actual>=L.TamActual());}
      String ElementoActual(){
            String s=null;
            if (! Lleno())
              s=L.Obtener(actual);
            return s;
      }
}
```

Patrones de comportamiento: *State*

- **Propósito**

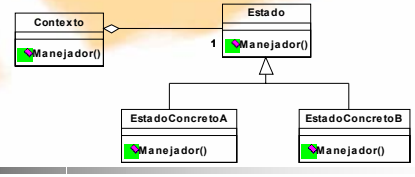
  Permite que un objeto modifique su comportamiento cuando cambie su estado interno. Parecerá que el objeto cambie de clase.

- **Motivación**

  Implementar los distintos estados de una clase dinámica como si fueran clases independedientes

Patrones de comport...: *State* (II)

- **Aplicabilidad**
  - El comportamiento de un objeto depende de su estado y debe cambiar en tiempo de ejecución dependiendo de ese estado.
  - Las operaciones tienen sentencias condicionales múltiples que dependen del estado del objeto, que normalmente se representa con una constante. A menudo varias operaciones tienen la misma estructura.
- **Estructura**



Patrones de comport...: *State* (III)

- **Participantes:**
  - **Contexto**: define el interfaz de interés para los clientes, manteniendo una instancia de la subclase EstadoConcreto que define el estado actual
  - **Estado**: define una interfaz para encapsular el comportamiento asociado a un estado particular de Contexto
  - **EstadoConcretoA, EstadoConcretoB,...**: cada subclase implementa un comportamiento asociado con un estado de Contexto
- **Consecuencias**
  - Pueden añadirse nuevos estados y transiciones con facilidad.
  - Menos compacto que única clase.

Patrones de comport...: *State* (IV)

```
class CFuncionActivacion  {
public:
    virtual double derivada(double x)=0;
    CFuncionActivacion();
    virtual ~CFuncionActivacion();
    virtual double operator()(double d)=0;
};
class TangenteHiperbolica : public
  CFuncionActivacion  {
public:
    double derivada(double x);
    TangenteHiperbolica(double _a=1.716,
                double _b=3.333333);
    virtual ~TangenteHiperbolica();
    double operator()(double d) {
        return a*tanh(d*b);}
private:
    double b;double a;};
```

```
class Sigmoidal : public CFuncionActivacion  {
public:
    double derivada(double x);
    Sigmoidal(double _a=1.716);
    virtual ~Sigmoidal();
    double operator()(double d) {
      return 1/(1+exp(-a*d));}
private:
    double a;
};
class CNeurona  {
public:
    virtual void calculaError()=0;
    virtual void calculaDelta()=0;
    void activa(){
      double suma=0;
      for(int i=1;i<Nestimulos;i++)
        suma+=Tpesos[i]*estimulo[i].get();
    respuesta.set(fact(suma));}
```

Patrones de comport...: *State* (V)

```
    double ajustaPesos();
    CNeurona(int ide,int nume,CSinopsis& e, CEnlace&
      s, CFuncionActivacion& f,float _alfa,float _nu);
protected:
    double error;
    int identif;
    double delta;
    float alfa; //coeficiente de inercia
    float nu;//coeficiente de aprendizaje
    double* Tpesos;//pesos sinópticos actuales
    double* Vpesos;//variación de los pesos sinópticos
                   // en la última epoca
    int Nestimulos; //número de entradas
    CFuncionActivacion& fact;//función de activación
    double ultimarespuesta; //respuesta en el último
                            // periodo
    CEnlace & respuesta;//conexión de salida
    CSinopsis& estimulo;//sinopsis de estrada
};
```

- **Al construir una capa de la red neuronal se construirían neuronas de la forma:**

```
Tneuronas[i]=new
CNeuronaSalida(i,entrada.taman
io(),entrada,salida[i],fa,fch,
_alfa,_nu);
```