Diseño Basado en Componentes

Patrones de diseño

Ingeniería Informática
Universidad Carlos III de Madrid

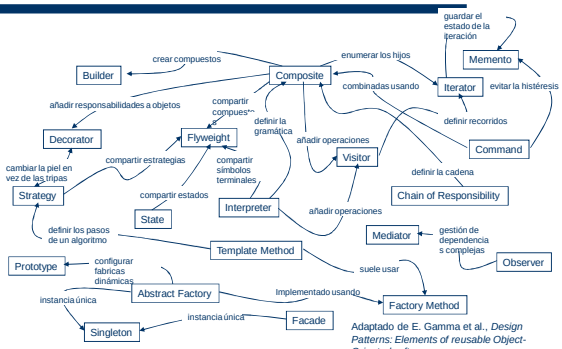
Tabla de contenidos

- Patrones de diseño
  - Relaciones entre patrones de diseño
  - Catálogo de patrones de diseño
- Modelo-Vista-Controlador
- Abstract factory
- Factory Method
- Facade
- Observer
- Command

Patrones de diseño

- Un patrón es un conjunto:
  - Problema
  - Solución
  - Entorno
- Un patrón es una solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparece con frecuencia
- Se intenta reutilizar diseños que hayan demostrado su eficacia en previas aplicaciones
- Esqueleto de aplicación básica que el diseñador ha de adaptar a sus necesidades

Relaciones entre patrones de diseño



Adaptado de E. Gamma et al., *Design Patterns: Elements of reusable Object-Oriented software*

Catálogo de patrones de diseño

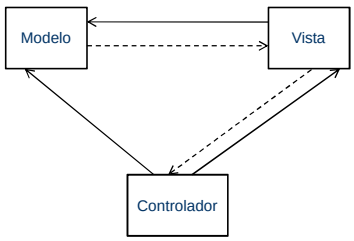
		Propósito		
Arquitectura		*Creación*	*Estructura*	*Comportamiento*
Presentation-abstraction-control Three-tier Pipeline Implicit invocation Blackboard system Peer-to-peer Service-oriented architecture Naked objects Model-View-Controller	Ámbito: Clase (Compilación)	Factory Method	Adapter	Interpreter Template Method
	Ámbito: Objeto (Ejecución)	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Patrón Modelo-Vista-Controlador

OBJETIVO

- Desacoplar los objetos del modelo y los objetos de la vista.
- El patrón de arquitectura MVC ayuda a reducir la complejidad en el diseño arquitectural e incrementar la flexibilidad y el reuso.

Estructura MVC



Modelo

- Representación especifica del dominio donde la aplicación opera.
- La lógica del dominio añade significado a los datos (p.e. calcular si hoy es el cumpleaños del usuario, totales, impuestos, etc.)
- Muchas aplicaciones utilizan un mecanismo persistente de almacenamiento para almacenar datos.
- MVC no menciona específicamente la capa de acceso a datos porque se entiende que está por debajo o bien se encuentra encapsulada por el modelo

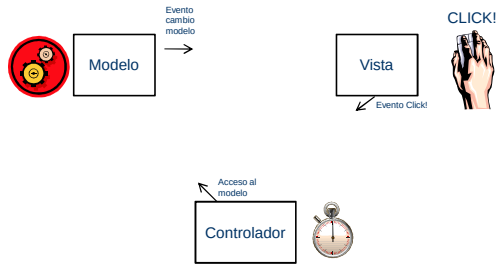
Vista

- La vista renderiza o muestra de una forma apropiada para la interacción
- Típicamente es un elemento de la interfaz de usuario
- Pueden existir múltiples vistas para un mismo modelo.
  - Dichas vistas tendrán un propósito distinto.

Controlador

- Procesa y responde a eventos
  - Normalmente suelen ser estímulos que proceden de las acciones del usuario
  - Pero pueden ser estímulos que proceden del otros sistemas, o incluso del propio software
- Estos estímulos deben ser tratados por el controlador y si procede realizará los cambios necesarios en el modelo.

Funcionamiento MVC (i)



Funcionamiento MVC (ii)

- El usuario interactúa con la interfaz de usuario
- El controlador recibe la notificación de la acción solicitada por el usuario.
- El controlador accede al modelo, actualizándolo o modificándolo
- Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.

Funcionamiento MVC (iii)

- El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario.
- La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se refleja los cambios en el modelo
- El modelo no debe tener conocimiento directo sobre la vista. Sin embargo, el patrón *observer* puede ser utilizado para proveer cierta indirección entre el modelo y la vista, permitiendo al modelo notificar a los interesados de cualquier cambio.

Patrón Abstract Factory

OBJETIVO

- Proporcionar un interfaz para la creación de familias de objetos relacionados o dependientes sin necesidad de especificar las clases concretas.

Caso de estudio (i) Abstract Factory

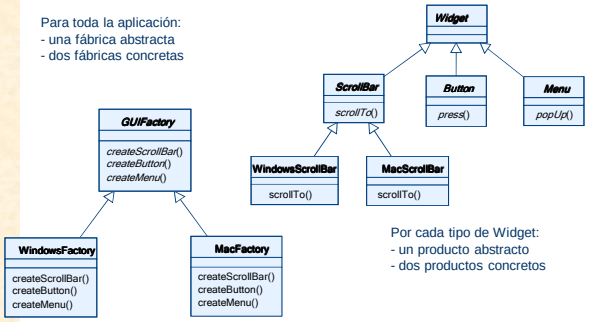
- Una situación real:
  - Se desea desarrollar una aplicación con interfaz gráfica de usuario (GUI), que sea fácil de portar a dos sistemas operativos: Windows y Macintosh.
  - Las GUI manejan una serie de objetos, generalmente conocidos como Widgets, tales como: ScrollBar, List, Button, etc.
- Problema de diseño:
  - La GUI presenta una apariencia diferente (*look and feel*) para cada uno de estos objetos en cada sistema operativo.
  - La aplicación debe poder crear los objetos con la apariencia apropiada para cada sistema operativo.

Caso de estudio (ii) Problema mantenimiento

- La **creación de Widgets** es diferente en cada sistema:
  - `ScrollBar sb = new WindowsScrollBar();`
  - `ScrollBar sb = new MacScrollBar();`
- Si se crean muchos *Widgets* (como suele ocurrir en aplicaciones de cierta complejidad) será muy difícil cambiar de una apariencia a otra, dado que hay que modificar muchas instrucciones que están diseminadas por diferentes partes del programa (**problema de mantenimiento**).
- **Solución**: patrón de diseño *Abstract Factory* (fábrica abstracta).
  - Una "fábrica" es un objeto que se encarga de crear otros objetos.
- Los elementos de este patrón son:
  - Una fábrica abstracta y varias fábricas concretas.
  - Un producto abstracto y varios productos concretos.

Caso de estudio (iii)
Fábricas y productos

Para toda la aplicación:
- una fábrica abstracta
- dos fábricas concretas

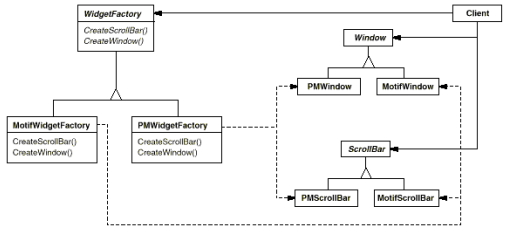


Por cada tipo de Widget:
- un producto abstracto
- dos productos concretos

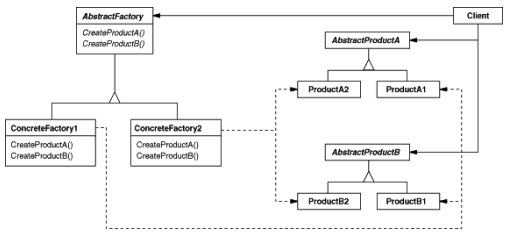
Caso de estudio (iii)
Solución de problemas de mantenimiento

- Creación de Widgets con **constructores concretos**, antes…
  - ScrollBar sb = new WindowsScrollBar();
  - ScrollBar sb = new MacScrollBar();
- …y después, a través de **operaciones abstractas** de la fábrica.
  - ScrollBar sb = guiFactory.createScrollBar();
- La variable global guiFactory debe ser inicializada al comienzo del programa con una de las fábricas concretas:
  - GUIFactory guiFactory = new MacFactory();
- Las operaciones de creación en las fábricas concretas usan los constructores concretos:
  - public ScrollBar createScrollBar() {return new WindowsScrollBar();}
  - public ScrollBar createScrollBar() {return new MacScrollBar();}

Abstract factory: ejemplo



Abstract factory: esquema del patrón



Patrón Factory Method

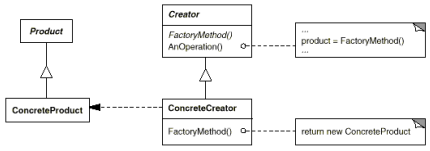
OBJETIVO

- Definir una interfaz para crear un objeto, pero permitir a las subclases decidir que clase instanciar. Este patrón permite a una clase diferir la instanciación a las subclases

Descripción Factory Method

- Se ocupa del problema de la creación de objetos (productos) sin especificar la clase exacta del objeto que será creado.
- Resuelve el problema definiendo un método distinto para la creación de objetos, el cual las subclases pueden sobrescribir para especificar el tipo derivado del producto que se va a crear.
- Frecuentemente se utiliza el termino “factory method” para referirse a cualquier método cuyo propósito principal es la creación de objetos.

Estructura



Participantes

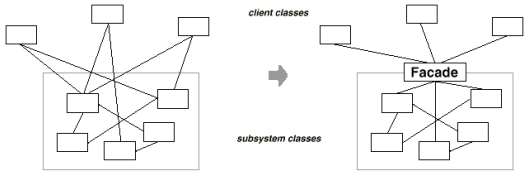
- Producto (Product)
  - Define la interfaz de objetos que crea el “factory method”
- Producto concreto (ConcreteProduct)
  - Implementa la interfaz producto
- Creador (Creator)
  - Declara el “factory method”, el cual decuelve un objeto de tipo Producto
- Creador concreto (ConcreteCreator)
  - Sobrescribe el “factory method” para devolver una instancia de un producto

Patrón Facade

OBJETIVO

- Ofrece una interfaz unificada a un conjunto de interfaces en un subsistema.
- Define una interfaz a alto nivel que hace al subsistema más fácil de usar.

Estructura



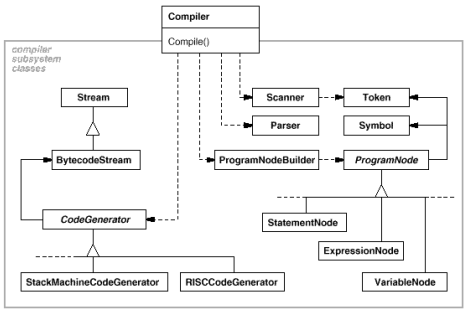
Caso de estudio (i)

- Una situación real:
  - Entorno de programación que ofrece a las aplicaciones acceso al subsistema de compilación
  - El subsistema ofrece clases como “Scanner”, “Parser”, “ProgramNode”, “BytecodeStream” etc.
  - Quizá aplicaciones especializadas necesiten tener un acceso a esas clases directamente
  - Pero la mayor parte de los códigos clientes no les interesan entrar en ese detalle, simplemente quieren compilar cierto código.
  - Las interfaces a bajo nivel dificultan el manejo simple del sistema.

Caso de estudio (ii)

- Para proveer una interfaz a alto nivel para encapsular las clases de los clientes, el subsistema debe incluir una clase “Compilador”
- Esta clase define una interfaz unificada a la funcionalidad del compilador
- La clase compilador actúa como una fachada (Facade). Ofrece al cliente una interfaz única y simple del subsistema del compilador.
- La fachada “Compilador” hace la vida más fácil para la mayoría de los programadores, sin ocultar la funcionalidad a bajo nivel si la necesitáramos.

Caso de estudio (iii)



Participantes

- Facade (compilador)
  - Sabe que clases del subsistema son responsables de una petición.
  - Delega al cliente peticiones más complejas a los objetos del subsistema.
- Clases del subsistema (“Scanner”,”Parser” etc.)
  - Implementan la funcionalidad del subsistema.
  - Manejan las peticiones asignadas al objeto “Fachada” (Facade)
  - No tienen ningún conocimiento del objeto “Fachada”. Es decir no tiene ninguna referencia a él.

Patrón Observer

OBJETIVO

- Definir una dependencia uno a muchos entre objetos tal que cuando un objeto cambia de estado, todos los objetos dependientes (observadores) son notificados automáticamente.

Efecto de la POO

- Un efecto colateral de partir un sistema en colecciones de clases cooperantes es la necesidad de mantener la consistencia entre los objetos relacionados.
- Por otro lado no se quiere mantener la consistencia haciendo que las clases estén estrechamente acopladas, porque esto reduciría la reusabilidad.

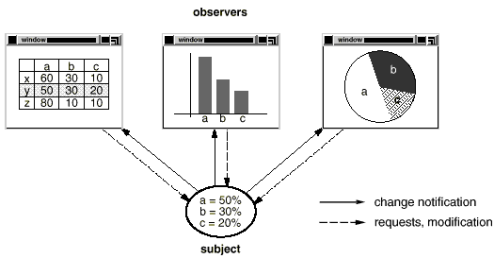
Caso de estudio (i)

- En muchas aplicaciones están separados el aspecto de presentación de los datos de la aplicación
- Las clases que se refieren a los datos de la aplicación y clases dedicadas a ola representación pueden ser reutilizadas e intercambiadas.
- Por ejemplo, un objeto “Hoja de cálculo” y un objeto “Gráfico de barras” pueden representar el mismo objeto de datos usando destintas representaciones.

Caso de estudio (ii)

- La hoja de cálculo y el gráfico de barras no saben el uno del otro, de tal modo que permiten su reutilización.
- Pero se comportan como si lo hicieran.
- Cuando el usuario cambian la información en el objeto “hoja de cálculo”, el objeto “gráfico de barra” refleja los cambios inmediatamente y viceversa.

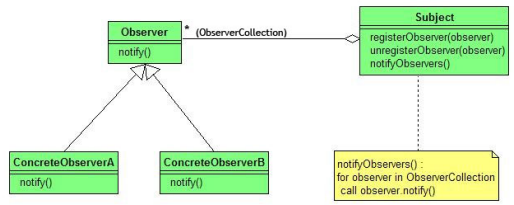
Ejemplo Observer



Funcionamiento

- Los objetos clave son los “sujetos” y los “observadores”.
- Un “sujeto” debe tener cualquier número de “observadores” dependientes.
- Los “observadores” son independientes entre sí.
- Todos los “observadores” son notificados siempre que el “sujeto” experimenta un cambio de estado.
- En respuesta, cada “observador” pedirá al “sujeto” que se sincronice con su estado.

Estructura Observer



Patrón Command

OBJETIVO

- Encapsular una petición como un objeto, permitir la parametrización de clientes, encolar o guardar en el log peticiones, y dar soporte a operaciones de tipo “deshacer”

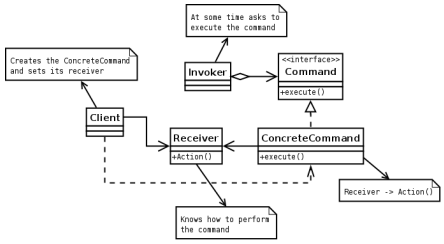
Estructura

- Permite hacer peticiones y convertirlas en objetos.
- Estos objetos “comandos” pueden ser almacenados o enviados como otros objetos.
- La clave en este patrón es la clase abstracta (o interface) “Command”, la cual declara una interfaz para la ejecución de operaciones.
- Esta clase incluye una operación “execute” abstracta.
- Los “comandos” concretos definen un par receptor-acción, almacenando el receptor e implementando la operación “execute”
- El receptor sabe como actuar ante la recepción del mensaje

Consecuencias

- Desacopla el objeto que invoca la operación del que sabe cómo realizarla.
- Los comandos son objetos manipulables y extensibles como cualquier otro.
- Permite combinar comandos en un comando compuesto: patrón Composite.
- Es fácil añadir nuevos comandos sin cambiar las clases existentes
  - Comandos reversibles: añadir unexecute( ), state, history list…
  - Comandos registrados: añadir store( ), load( )…

Estructura.



Bibliografía

- E. Gamma et al.,
  Design Patterns: Elements of reusable Object-Oriented software
- El Lenguaje Unificado de Modelado
  Grady Booch, James Rumbaugh, Ivar Jacobson
  Addison Wesley, 1999

Diseño Basado en Componentes

Introducción

Ingeniería Informática
Universidad Carlos III de Madrid