Departamento de Informática | Universidad de Valladolid

Programación III.I.T.I. de Sistemas

Patrones de diseño

Félix Prieto

Curso 2008/09

Contenidos

- Patrones de diseño
  - Introducción
  - Conceptos básicos
  - Patrones Estructurales
  - Patrones de Comportamiento
  - Patrones Creacionales

Información sobre patrones de diseño

- Jean-Marc Jézéquel, Michel Train, Christine Mingins *Design Patterns and contracts*. Addison Wesley, 1999
- (GoF) Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *Design Patterns. Elements of Reusable Object-Oriented Software* Addison Wesley, 1995
- *Patterns Home Page*
- *Patterns and Software*

Motivación

- Disponemos de las técnicas básicas de la Orientación a Objetos, sin embargo...
  - Encontrar las clases es difícil
  - Estructurar bien las clases es más difícil
  - Hacerlo de forma reutilizable es más difícil aún
- Necesitamos técnicas y estrategias que nos guíen en la construcción de buenos sistemas Orientados a Objetos
  - Ayudando a construir clases
  - Ayudando a estructurar sistemas de clases

Definiciones

- Cada *patrón* describe un problema que ocurre una y otra vez en nuestro entorno y describe también el núcleo de su solución, de forma que puede utilizarse un millón de veces sin hacer dos veces lo mismo (Christoph Alexander, Arquitecto y urbanista)
- Un *patrón de diseño* es una descripción de clases y objetos que se comunican entre si, adaptada para resolver un problema general de diseño en un contexto particular (GoF)
- Un patrón de diseño es una solución a un problema en un contexto

Qué no son los patrones de diseño

- Patrones de diseño no es lo mismo que
  - Bibliotecas de clases
  - Frameworks
  - Assets de grano grueso
  - Técnicas y/o herramientas de refactorización
  - Programación Extrema
- También hay patrones de Análisis, de Arquitectura, de Interfaz de usuario, de diseño Web,... incluso hay Anti-patrones

Ventajas de los patrones de diseño

- Facilitan la *localización* de los objetos que formarán el sistema
- Facilitan la determinación de la *granularidad* adecuada
- Especifican *interfaces* para las clases
- Especifican *implementaciones* (al menos parciales)
- Facilitan el *aprendizaje* y la *comunicación* entre programadores y diseñadores

Clasificación

- Respecto a su propósito
  - *Creacionales*: Resuelven problemas relativos a la creación de objetos
  - *Estructurales*: Resuelven problemas relativos a la composición de objetos
  - *de Comportamiento*: Resuelven problemas relativos a la interacción entre objetos
- Respecto a su ámbito
  - *Clases*: Relaciones estáticas entre clases
  - *Objetos*: Relaciones dinámicas entre objetos

Los patrones del GoF

	Clases	Objetos
Creacional	Método Fábrica	Fábrica Abstracta, Constructor, Prototipo, *Singleton*
Estructural	Adaptador	*Adaptador*, Puente, *Compuesto*, Decorador, Fachada, Peso Mosca, Apoderado
Comportamiento	Intérprete, Método Plantilla	Cadena de Responsabilidad, *Comando*, *Iterador*, Mediador, Memento, *Observador*, Estado, Estrategia y Visitante

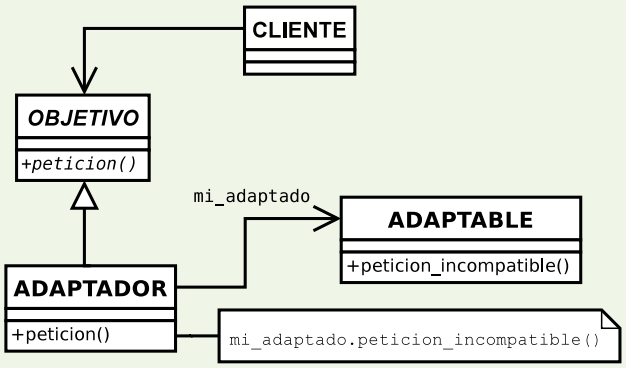
Plantilla general de descripción de un patrón

Nombre	¿Requiere explicación?
Intención	En dos líneas, qué hace, qué problema resuelve
Alias	Otros nombres con que es conocido
Motivación	Escenario de ejemplo
Aplicabilidad	Cuándo debe ser utilizado y cuándo no
Estructura	Diagrama de clases de la solución propuesta
Participantes	Diccionario de las clases que participan en la solución
Colaboraciones	Relaciones que se establecen entre las clases anteriores
Consecuencias	Ventajas e inconvenientes del uso de este patrón
Implementación	Técnicas, trucos que se recomiendan
Patr. relacionados	Utilizados en éste, patrones que lo usan, alternativas,...

Adaptador (Adapter) (GoF p.131)

- *Intención:* Convierte la interfaz de una clase en otra interfaz esperada por los clientes. Permite la cooperación de clases que de otro modo serían incompatibles
- *Motivación:* Necesitamos reutilizar clases ajenas para nuestro sistema, pero aunque la funcionalidad de estas clases es la que deseamos, la interfaz no es compatible. Si no podemos o no queremos modificar las clases a reutilizar, necesitamos hacerlas compatibles con nuestro sistema
- *Observación:* Este patrón tiene dos versiones, una con ámbito en clases y otra con ámbito en objetos. Nosotros nos centraremos en la segunda versión

Adaptador (Adapter) (GoF p.131)



Adaptador (Adapter) (GoF p.131)

- Participantes
  - `OBJETIVO`: Define la interfaz que espera el cliente
  - `ADAPTABLE`: Implementa la interfaz incompatible que necesitamos adaptar
  - `ADAPTADOR`: Implementa la interfaz de `OBJETIVO` mediante llamadas al objeto adaptado
- Colaboraciones
  - Los clientes utilizan la interfaz `OBJETIVO`. Sus peticiones son recogidas por un `ADAPTADOR` que las redirige al objeto adaptable

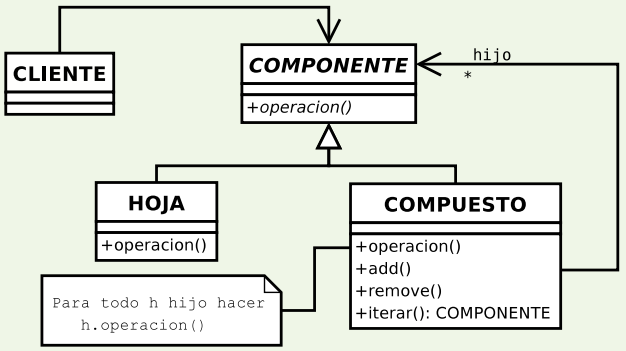
Adaptador (Adapter) (GoF p.131)

- Consecuencias
  - Un adaptador puede funcionar con varios adaptables de forma simultanea, coordinando sus tareas
  - Si se necesita redefinir el comportamiento de `ADAPTABLE` basta construir un heredero y hacer que el adaptador lo adapte
  - Introduce un nivel de indirección extra para satisfacer las peticiones del cliente.

Compuesto (Composite) (GoF p.151)

- *Intención:* Componer objetos en jerarquías todo-parte y permitir a los clientes tratar objetos simples y compuestos de modo uniforme
- *Motivación:* Necesitamos representar un conjunto de elementos de una interfaz gráfica de usuario (GUI). Algunos de estos elementos son simples, mientras que otros están formados por varios elementos más simples. El comportamiento y/o la información que proporciona un elemento complejo está determinada por los elementos que lo componen

Compuesto (Composite) (GoF p.151)



Compuesto (Composite) (GoF p.151)

- Participantes
  - *COMPONENTE:* Declara la intefaz común para todos los objetos de la composición e implementa acciones por defecto cuando sea apropiado. También puede proporcionar la interfaz de acceso a hijos y padres
  - *SIMPLE:* Representa los objetos simples e implementa su comportamiento común
  - *COMPUESTO:* Representa los objetos con hijos, almacenando a éstos e implementando las operaciones de acceso y mantenimiento relacionadas con ellos.
- Colaboraciones
  - Los clientes utilizan la interfaz de *COMPONENTE*
  - Los objetos simples contestan directamente, mientras que los compuestos pueden reenviar la operación a sus componentes
  - Los objetos que construyen la estructura deben ser clientes tanto de *SIMPLE* como de *COMPUESTO*

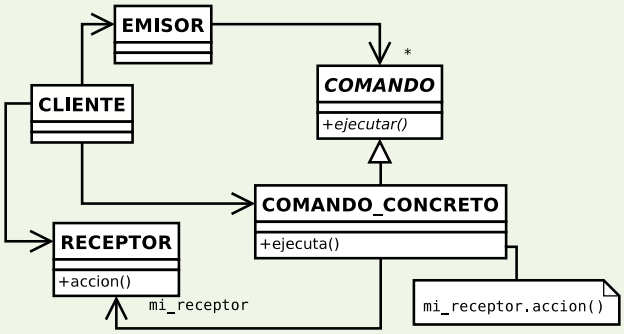
Compuesto (Composite) (GoF p.151)

- Consecuencias
  - Permite el tratamiento uniforme de objetos simples y complejos
  - Simplifica el código de los clientes, que usan una sola interfaz
  - Permite añadir nuevos tipos de componentes sin afectar a los clientes
  - Es difícil restringir los tipos de los hijos
  - Definir las operaciones de gestión de los hijos en *COMPONENTE* crea problemas de consistencia con los hijos
  - La implementación de las operaciones de los compuestos que dependen de los hijos puede ser más fácil utilizando el patrón *iterador*

Comando (Command) (GoF p.215)

- *Intención:* Encapsula una petición en un objeto. Permite con ello parametrizar a los clientes con diferentes peticiones y almacenar peticiones para deshacerlas en caso necesario
- *Motivación:* Parametrización de las acciones a realizar por los objetos GUI de un framework (como hace eGTK)

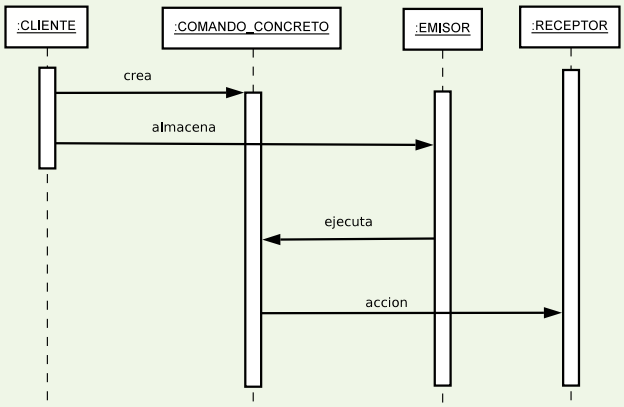
Comando (Command) (GoF p.215)



Comando (Command) (GoF p.215)

- Participantes
  - *COMANDO:* Declara la intefaz para ejecutar una operación
  - *COMANDO_CONCRETO:* Define una relación entre un receptor y una acción, redefiniendo la operación *ejecutar* para que se envíe una petición adecuada al receptor
  - *CLIENTE:* Crea un comando concreto indicándole cuál es su receptor y lo almacena en su emisor
  - *EMISOR:* Pide al comando que lleve a cabo una petición
  - *RECEPTOR:* Sabe cómo realizar la operación relacionada con una petición
- Colaboraciones
  - El cliente crea un comando concreto indicándole cuál es su receptor
  - El emisor almacena uno o varios comandos concretos
  - El emisor solicita al comando que se ejecute
  - El comando pide al receptor que realice las operaciones necesarias

Comando (Command) (GoF p.215)



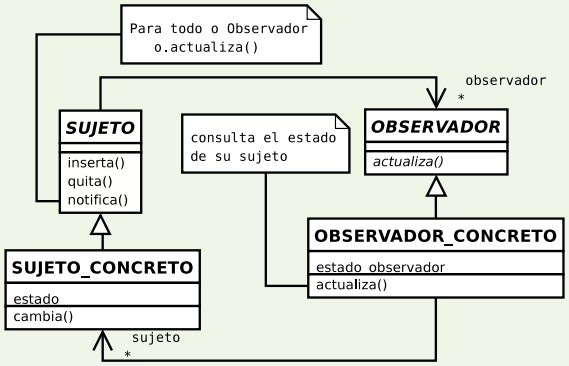
Comando (Command) (GoF p.215)

- Consecuencias
  - La intermediación de comando desacopla emisor y receptor
  - Permite manipular comandos tratados como objetos
  - Permite añadir nuevos comandos sin alterar las otras clases
  - Los comandos pueden ser más o menos inteligentes (solicitar un trabajo al receptor, o realizar tareas más complejas)
  - Aplicando el patrón *compuesto* podemos obtener comandos complejos (macros)
  - Se puede adaptar la infraestructura para la opción «deshacer comando», ya sea de uno o varios niveles (posiblemente eso requiera almacenar el estado previo del receptor)

Observador (Observer) (GoF p.269)

- *Intención:* Definir una dependencia entre un objeto y un conjunto de ellos, de modo que los cambios en el primero se vean reflejados en los otros
- *Motivación:* En un toolkit de GUI necesitamos separar los objetos de presentación de los objetos que modelan los datos interesantes para la aplicación, de forma que se puedan tener varias vistas sincronizadas de los mismos datos

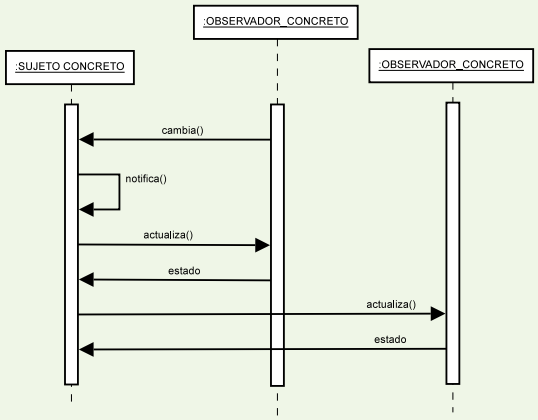
Observador (Observer) (GoF p.269)



Observador (Observer) (GoF p.269)

- Participantes
  - *SUJETO:* Mantiene una lista de observadores y proporciona una interfaz para su gestión
  - *OBSERVADOR:* Define una interfaz para actualizar los objetos que deben reflejar los cambios en el sujeto
  - *SUJETO_CONCRETO:* Envía una notificación a sus observadores cuando cambia su estado
  - *OBSERVADOR_CONCRETO:* Mantiene una referencia a una sujeto concreto, almacenando parte de su estado e implementado la interfaz de *OBSERVADOR*
- Colaboraciones
  - El *SUJETO_CONCRETO* notifica a sus observadores de los cambios que sufre
  - Los observadores concretos solicitan a su sujeto los datos necesarios para mantener la consistencia con su nuevo estado

Observador (Observer) (GoF p.269)



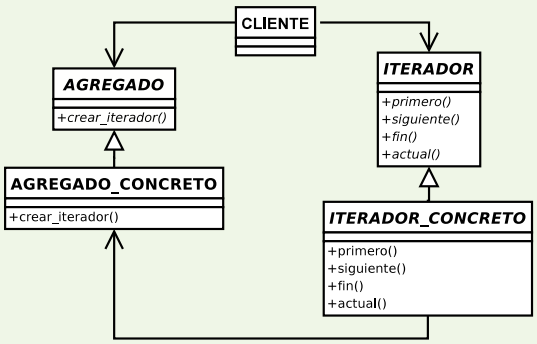
Observador (Observer) (GoF p.269)

- Consecuencias
  - Permite reutilizar sujetos y observadores por separado
  - Permite añadir nuevos observadores sin modificar al sujeto o a otros observadores
  - Que el sujeto no informe a sus observadores de qué cambio ha sufrido permite mantener el acoplamiento en un nivel bajo, puesto que el observador sólo pide los datos del estado del sujeto que le interesan
  - Aunque el observador no esté interesado en ciertos cambios del sujeto será notificado de ellos
  - Se pueden realizar implementaciones con observadores que coordinan información sobre varios sujetos
  - Los cambios en el sujeto pueden ser solicitados por objetos que no son observadores

Iterador (Iterator) (GoF p.237)

- *Intención:* Permite acceder secuencialmente a los elementos de un agregado sin exponer su estructura interna
- *Motivación:* Necesitamos implementar una estructura de datos «compleja». Deseamos proteger a los clientes de la estructura frente a los detalles de implementación de la misma. permitiendo incluso que el cambio de su estructura no los afecte

Iterador (Iterator) (GoF p.237)



Iterador (Iterator) (GoF p.237)

- Participantes
  - *ITERADOR:* Define la interfaz común para recorrer todos los agregados y acceder a ellos
  - *AGREGADO:* Define la interfaz de creación del iterador
  - *ITERADOR_CONCRETO:* Implementa la interfaz de *ITERADOR* y mantiene la posición actual del iterador
  - *AGREGADO_CONCRETO:* Implementa la interfaz de creación de los iteradores
- Colaboraciones
  - El iterador concreto mantiene la información actualizada sobre el recorrido que se está realizando sobre el agregado

Iterador (Iterator) (GoF p.237)

- Consecuencias
  - Podemos variar la forma de recorrer el agregado sin más que cambiar el iterador. Varias formas diferentes de recorrido pueden ser encapsuladas en una jerarquía de iteradores
  - La interfaz del agregado resulta más simple al implementar menos operaciones de recorrido
  - Permite el recorrido simultáneo con varios iteradores, utilizando varios algoritmos para el mismo
  - Hay muchas alternativas de implementación: Iterador interno o externo, cursor, iterador robusto,...
  - La interfaz del iterador puede tener elementos adicionales (anterior, busca)

Singleton (Único) (GoF p.119)

- *Intención:* Asegurarse de que una clase tiene una sola instancia y proporcionar un único punto de acceso a ella
- *Motivación:* Necesitamos definir una única instancia de cierta clase `MODELO` de modo que sea accesible desde múltiples objetos del sistema. Deseamos que el diseño imposibilite la creación accidental de varias instancias de la clase

Singleton (Único) (GoF p.119)

- En Eiffel se han ofrecido diversas implementaciones del patrón
- Todas ellas presentan puntos discutibles, o incluso errores
- Se han llegado a proponer modificaciones del lenguaje para facilitar una implementación adecuada (Karine Arnout, and Éric Bezault. How to get a Singleton in Eiffel?. In Journal of Object-Technology (JOT), Vol.3, No.4, April 2004, Special issue: TOOLS USA 2003, p 75-95.)

Singleton (Único) (GoF p.119)

- Implementación de `std_input` en `ANY`
- Todos los clientes acceden al objeto mediante una función de una sola ejecución
- Los clientes son responsables de no crear nuevas instancias de `STD_INPUT`

Singleton (Único) (GoF p.119)

- Implementación propuesta por Jézéquel
- Participantes
  - `SINGLETON`: Clase que proporciona la funcionalidad deseada del objeto único y «verifica» mediante su contrato que no existen dos instancias suyas
  - `ACCESO`: Punto de acceso que devuelve el objeto singleton. Varias instancias del punto de acceso devuelven la misma instancia del objeto único gracias a una función `once`
- Comentarios a la implementación
  - Los problemas de implementación de singleton están asociados al funcionamiento de `once`
  - No es evidente cómo elaborar versiones abstractas de estas dos clases para simplificar la implementación del patrón