

LPS: Introducción a los Patrones de Diseño Software



Federico Peinado
www.federicopeinado.es

Depto. de Ingeniería del Software e
Inteligencia Artificial
disia.fdi.ucm.es

Facultad de Informática
www.fdi.ucm.es

Universidad Complutense de Madrid
www.ucm.es

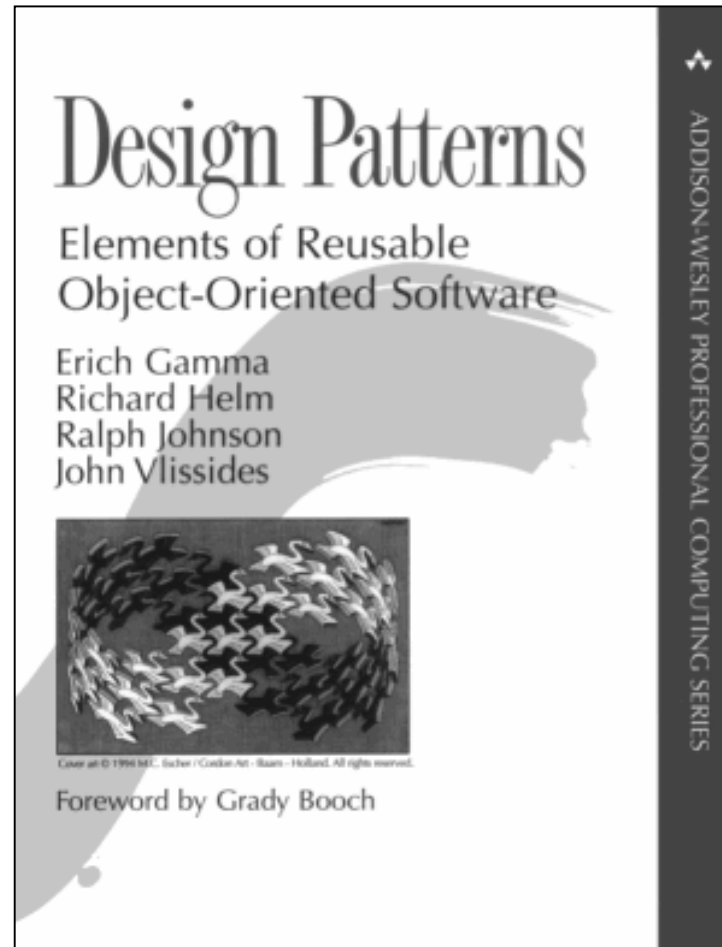
Reutilización

- ◉ Conviene diseñar e implementar elementos software que puedan usarse para construir muchos programas diferentes
 - Ahorra *esfuerzo* (evitamos “reinventar la rueda”)
 - Permite concentrarse siempre en *funcionalidad nueva*
 - Mejora indirectamente la calidad (hay más *revisiones*)
- ◉ ¡Ojo! Desarrollar software reutilizable es más difícil y costoso que *no reutilizable*
 - Implica resolver una *familia de problemas* en lugar de resolver un único problema particular
 - Implicar saber programar software *flexible* y prever los *ejes de cambio*

Niveles de reutilización

- ◎ Análisis
 - Requisitos de la aplicación
 - Especificaciones funcionales
- ◎ Diseño de alto nivel
 - Arquitecturas software
 - Armazones software (*frameworks*)
- ◎ Diseño de bajo nivel
 - Patrones de diseño software
- ◎ Código
 - Componentes software
 - Bibliotecas de código
 - PE en funciones / POO en métodos
 - “Cortar y pegar” ☹

Patrones de Diseño Software



Patrones de Diseño Software

- ◉ El DOO es difícil, y el DOO reutilizable más difícil todavía
- ◉ Los desafíos de diseño se repiten y los diseñadores con experiencia suelen acabar conociendo *las mejores soluciones*
 - Esas buenas ideas de diseño se conocen como patrones de diseño
- ◉ Definiciones de patrones de diseño
 - Originalmente, en Arquitectura (Alexander, 1977)

“Cada patrón describe *un problema* que ocurre una y otra vez en nuestro entorno, y luego describe el núcleo de *la solución a dicho problema*, de manera que puedas usar esa solución un millón de veces sin que en realidad lo estés resolviendo dos veces de la misma forma”
 - Posteriormente, en Ingeniería del Software (Gamma, Helm, Johnson y Vlissides, 1995) alias Gang of Four (GoF)

“Un patrón de diseño nombra, abstrae e identifica los *aspectos fundamentales de una estructura común de diseño* que la hacen útil para la creación de un *diseño orientado a objetos reutilizable*”

Tabla de patrones (GoF 1995)

<i>Nivel/Tipo</i>	Creación	Estructura	Comportamiento
Clase	Factory method/ Método de fabricación	Adapter/Clase adaptadora	Interpreter/Intérprete Template method/Método plantilla
Objeto	Abstract factory/ Fábrica abstracta Builder/Constructor virtual Prototype/Prototipo Singleton/Ejemplar único	Adapter/Objeto adaptador Bridge/Puente Composite/Objeto compuesto Decorator/Decorador Facade/Fachada Flyweight/Peso ligero Proxy/Intermediario	Chain of responsibility/ Cadena de responsabilidad Command/Orden Iterator/Iterador Mediator/Mediador Memento/Recuerdo Observer/Observador State/Estado Strategy/Estrategia Visitor/Visitante

Estructura de un patrón

- ◉ Nombre
 - Suele usarse el original en inglés del libro de GoF
 - Se han convertido en vocabulario común en DOO
- ◉ Propósito
 - Descripción de la situación problemática o compleja en la que parece conveniente aplicar el patrón
- ◉ Solución
 - Estructuras de diseño involucradas
 - Elementos
 - Relaciones entre estos elementos
 - Responsabilidades y colaboraciones que mantienen estos elementos
- ◉ Consecuencias
 - Discusión acerca de las ventajas y los inconvenientes del patrón
 - Coste asociado, flexibilidad conseguida, alternativas que hay o no hay...

Uso de los patrones

- ◉ Anticípate a las situaciones que te puedan obligar a modificar el diseño
- ◉ Decide qué cambios te gustaría poder hacer en el futuro sin necesidad de modificar el diseño
- ◉ Flexibiliza aquello que haya sido frecuente de cambios en las últimas iteraciones del desarrollo
- ◉ Recuerda: La reutilización siempre tiene un coste
 - No afrontes un proyecto usando patrones “a priori”
 - Conoce a fondo las consecuencias del patrón antes de usarlo
 - Los patrones deberían surgir “naturalmente” al diseñar

Utilidades de los patrones

- ◉ Controlar explícitamente la creación explícita de objetos (Fábrica abstracta, Método de fabricación y Prototipo)
- ◉ Gestionar dependencias con operaciones concretas (Cadena de responsabilidad y Orden)
- ◉ Gestionar dependencias con la plataforma (Fábrica abstracta y Puente)
- ◉ Gestionar dependencias con la implementación interna de los objetos (Fábrica abstracta, Puente, Recuerdo e Intermediario)
- ◉ Gestionar dependencias con los algoritmos (Constructor virtual, Iterador, Estrategia, Método plantilla, y Visitante)
- ◉ Minimizar el acoplamiento (Fábrica abstracta, Puente, Cadena de responsabilidad, Orden, Fachada, Mediador y Observador)
- ◉ Controlar la extensión de funcionalidad mediante subclases (Puente, Cadena de responsabilidad, Objeto compuesto, Decorador, Observador y Estrategia)
- ◉ Resolver la imposibilidad de modificar ciertas clases libremente (Clase adaptadora, Decorador y Visitante)

Antipatrones de diseño software

- ◉ Lo contrario de los patrones: malas soluciones de DOO que debemos evitar
 - Clases con demasiada funcionalidad (haciendo DE, en realidad)
 - Clases que sólo funcionan si se llama a sus métodos en orden
 - Clases que heredan funcionalidad “auxiliar” de otras cuando deberían delegar responsabilidades por composición
 - Objetos innecesarios, que realmente sólo sirven para pasar información a otros (típico al abusar de los patrones)
 - Objetos que comparten con cualquiera su información interna
 - ...
- ◉ Merece la pena conocer más antipatrones de DOO y también de POO, gestión de proyectos, etc.
<http://en.wikipedia.org/wiki/Anti-pattern>

Referencias

- ◉ Gamma, E., Helm, R., Johnson, R., y Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
- ◉ Shalloway, A., Trott, J.R.: Design Patterns Explained: A New Perspective on Object-Oriented Design. Addison-Wesley (2001)
- ◉ Grand, M.: Patterns in Java. John Wiley & Sons (1998)
- ◉ Freeman, E., Freeman, E., Bates, B., Sierra, K.: Head First Design Patterns. O'Really Media (2004) (disponible en Safari UCM)
- ◉ OODesign
<http://www.oodesign.com/>

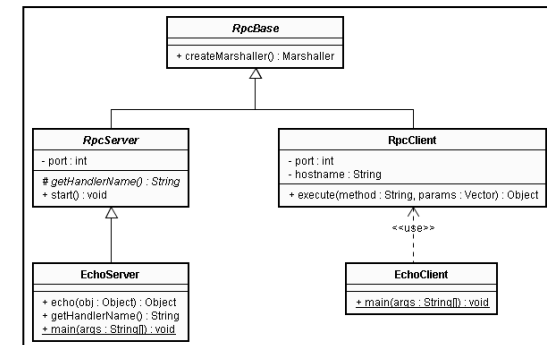
Previo: UML



Previo: UML

◉ UML significa Lenguaje Unificado de Modelado

- Lenguaje gráfico para visualizar, especificar, construir y documentar sistemas software
- UML 2.0 es ampliamente utilizado y estandarizado por el OMG (Object Management Group)

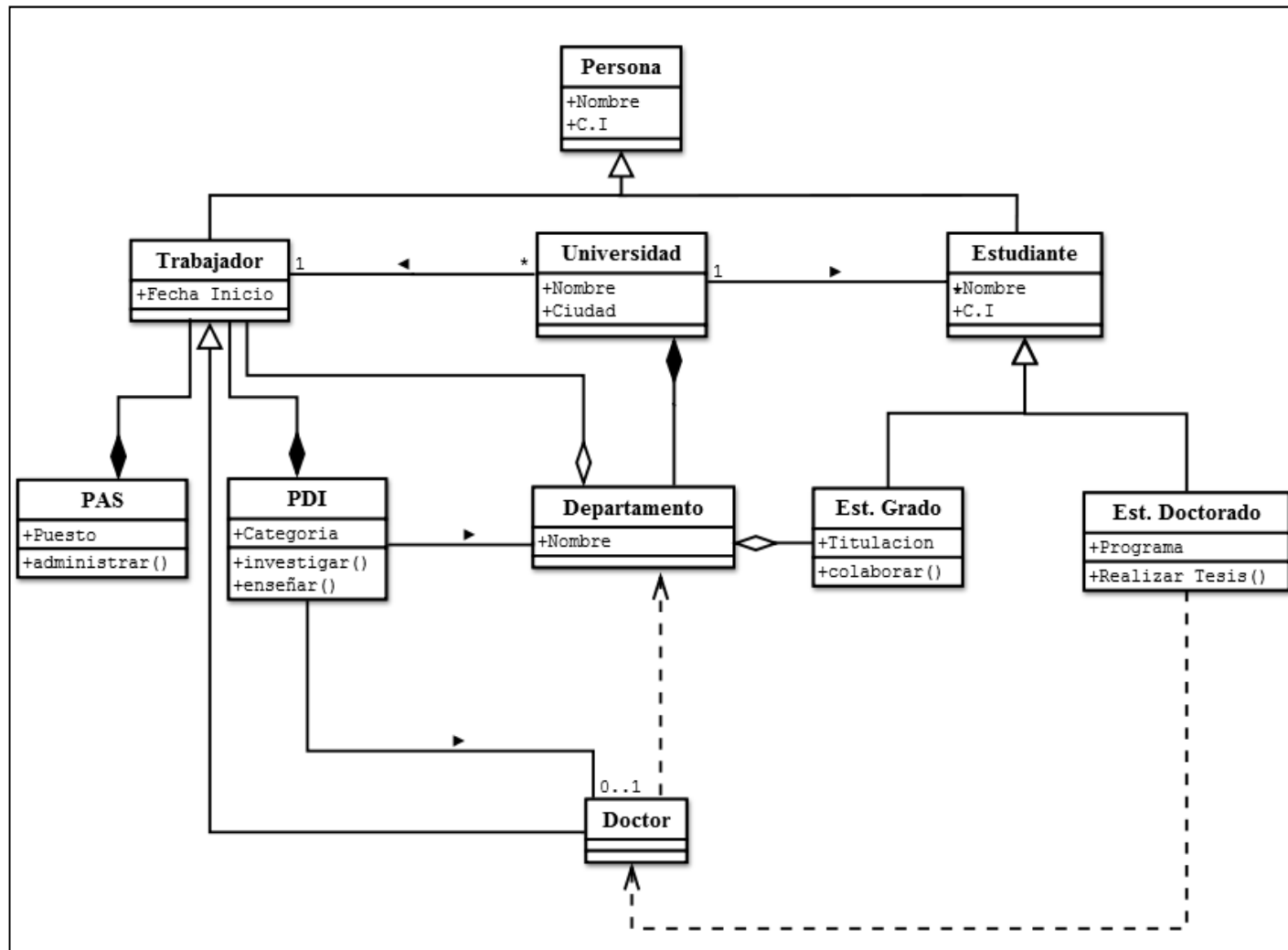


◉ En clase usaremos Diagramas de Clase UML

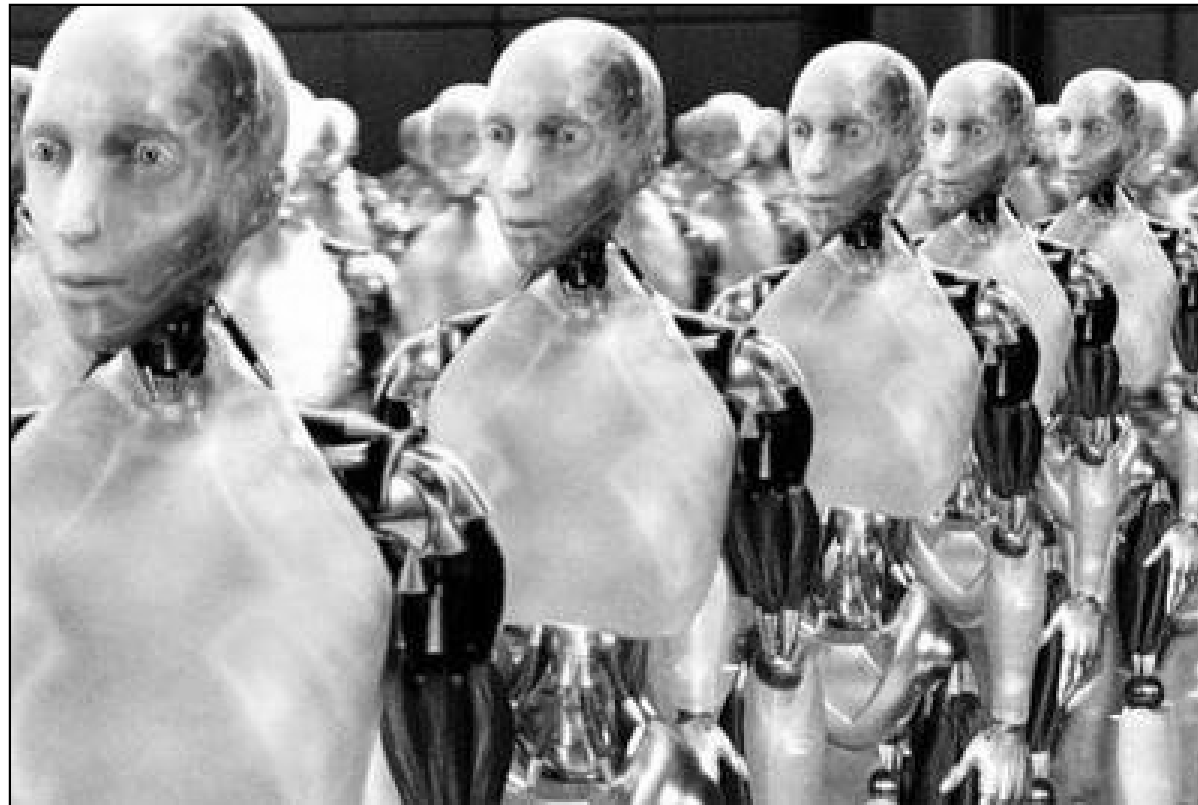
- Representación de la estructura estática de un programa orientado a objetos, mostrando sus clases -con atributos y métodos- y las relaciones que hay entre ellas
- ## ◉ BlueJ, interesante IDE educativo para aprender Java y UML al mismo tiempo
- <http://www.bluej.org/>



Previo: UML

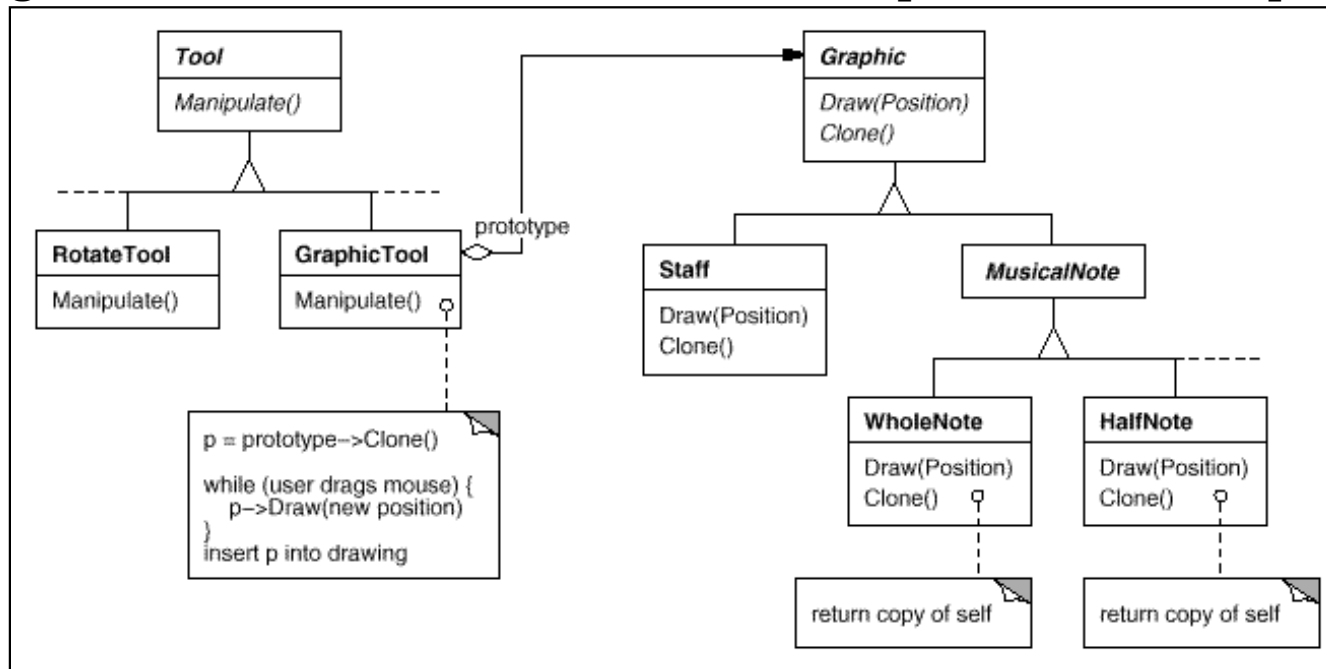


Prototipo



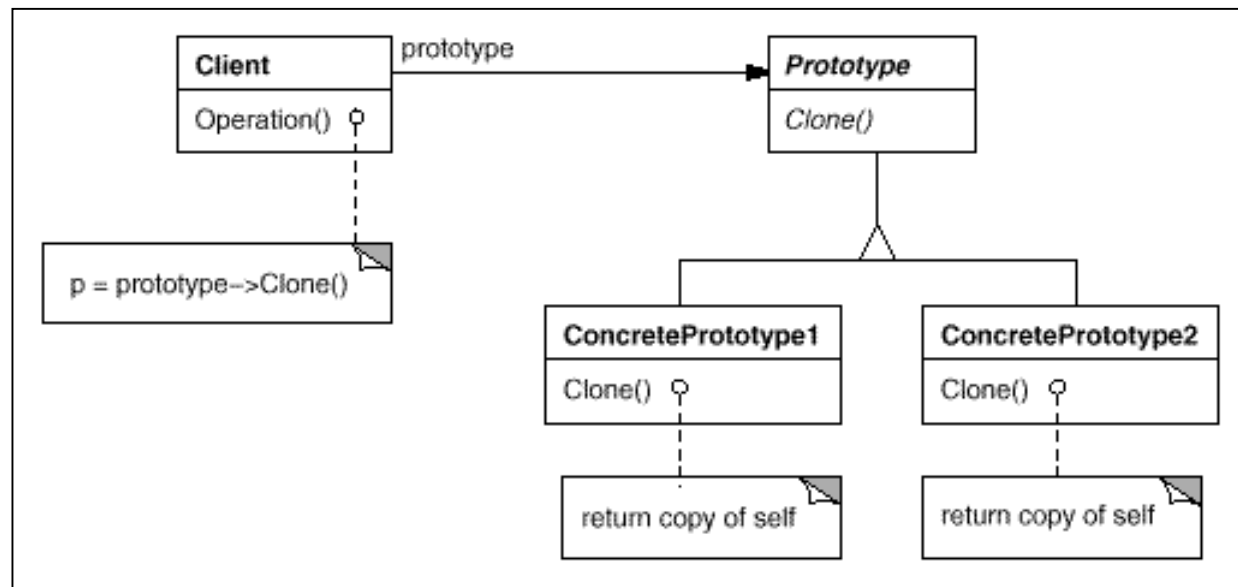
Prototipo

- ◉ **Nombre original:** Prototype
- ◉ **Propósito:** Permite representar un cierto “tipo” de objetos mediante un ejemplar “prototípico” (ya creado), que se *copia* cada vez que se ha de crear un objeto nuevo de este “tipo”
 - Ejemplo: ¿Cómo construir las herramientas de creación de un editor gráfico cuando todavía no conocemos el tipo de elementos que tendrán?



Solución

- ◉ *Prototype* declara una interfaz de copia
- ◉ *ConcretePrototype* implementa una operación para copiarse a sí mismo
- ◉ *Client* crea objetos como copias de un *ConcretePrototype*



Consecuencias

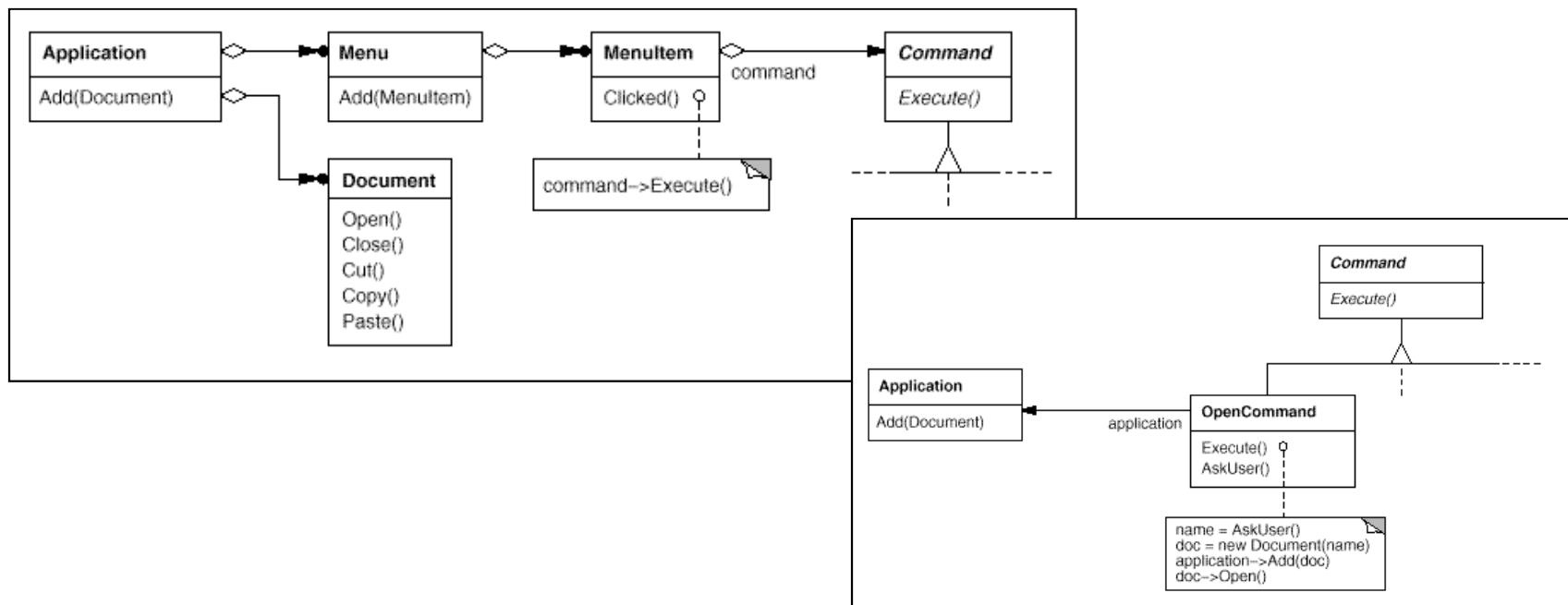
- ◉ Desacopla al cliente de la creación de objetos
- ◉ Permite añadir nuevos “tipos” de elementos en tiempo de ejecución
- ◉ Permite modificar el comportamiento dinámicamente a través de la composición
 - El cliente delega en un prototipo cuyo “tipo” podemos cambiar
- ◉ Facilita la construcción de objetos complejos a partir de partes predefinidas (los prototipos)
- ◉ Reduce la necesidad de definir subclases que introducen otros patrones de creación (jerarquía de factorías paralela a la jerarquía de productos)
- ◉ ¡Ojo! La operación de copia (*clone*) no siempre es trivial de implementar

Orden



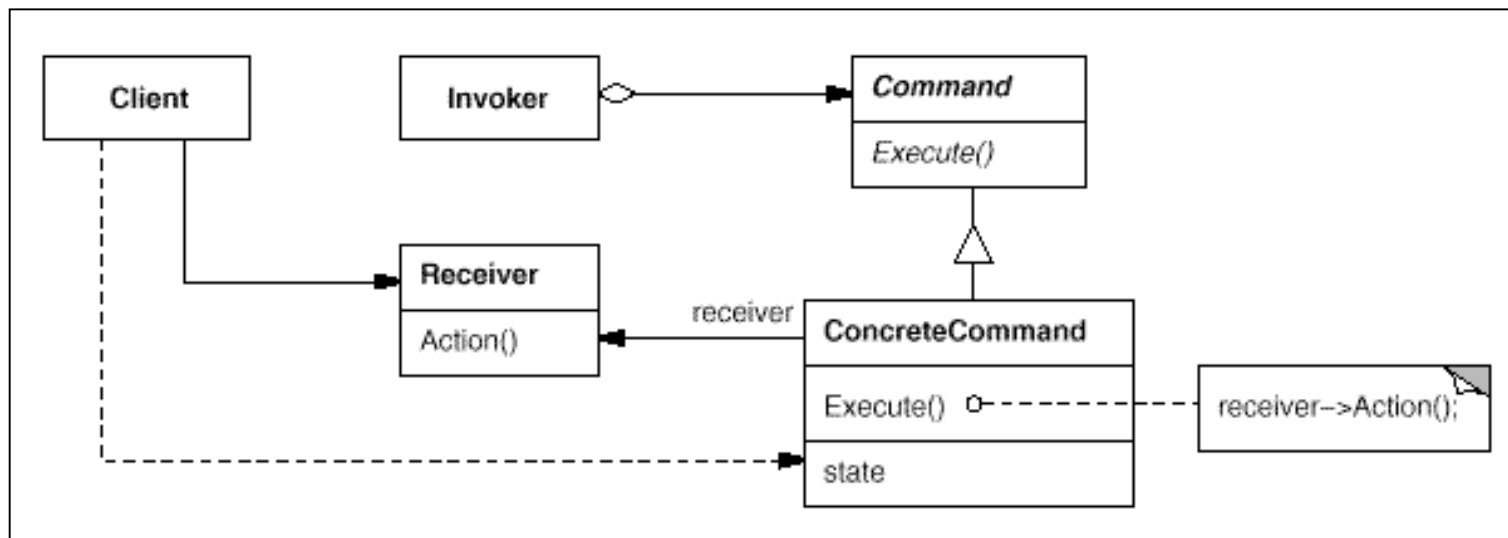
Orden

- ◉ **Nombre original:** Command
- ◉ **Propósito:** Encapsula un mensaje en forma de objeto, con lo que se puede “parametrizar” a los clientes, gestionar colas de mensajes, tener un registro de mensajes y hasta “deshacer” operaciones
 - Ejemplo: ¿Cómo hacer que los *widgets* (botones, menús, etc.) de una GUI en un armazón para crear GUIs invoquen operaciones específicas de aplicaciones que aún no se hemos escrito?



Solución

- ◉ *Command* declara la interfaz para ejecutar una operación
- ◉ *ConcreteCommand* vincula un objeto *Receiver* y una operación, implementando el método *execute* que la invoca
- ◉ *Client* crea un *ConcreteCommand* y establece su receptor
- ◉ *Invoker* solicita que se ejecute la orden. *Receiver* es el que se encarga de llevar a cabo las operaciones (cualquier clase puede jugar este papel)



Consecuencias

- ◉ *Command* desacopla el objeto que invoca la operación del que sabe cómo realizarla
- ◉ Los *Command* son objetos normales y como tales se pueden manipular y entender como cualquier otro objeto
- ◉ Se pueden crear “macro-órdenes” componiendo varios comandos
- ◉ Resulta sencillo añadir nuevos tipos de *Command*
- ◉ ¿Cuánta información tiene un *Command*?
 - Simplemente vincula un receptor con las operaciones a las que ha de responder
 - Es capaz de encontrar el receptor dinámicamente
 - Se hace cargo por sí mismo de atender a las solicitudes
- ◉ Operaciones de deshacer/rehacer
 - Siempre debe ser “lógicamente” posible deshacer la operación
 - Se mantiene una lista de objetos *Command*
 - Puede ser necesario que el objeto *Command* guarde información adicional: receptor concreto y su estado, argumentos de la invocación...

Críticas, dudas, sugerencias...



Federico Peinado
www.federicopeinado.es