Objetos 2: Teoria

Patrones

Según alexander

Un patrón describe un problema que se repite en un contexto, describe el núcleo de la solución para que se pueda replicar sin hacer lo mismo dos veces.

Según GOF

Un patrón de diseño nombre, abstrae e identifica los aspectos clave de una estructura de diseño común, útil para crear un diseño OO.
Se centran en un problema o cuestión particular del diseño OO, describiendo cuando se aplica, qué

restricciones hay y las consecuencias y compensaciones de usarlo.

En general

- Son pares problema-solución que tratan con problemas recurrentes buscan soluciones para los mismos.
- Deben incluir una regla: ¿Cuándo aplico el patrón?

Elementos de un patrón (NPM APE CCIC UP)

1. **N** - Nombre
2. **P** - Propósito
3. **M** - Motivación (Problema)
4. **A** - Aplicabilidad
5. **P** - Participantes
6. **E** - Estructura
7. **C** - Colaboraciones
8. **C** - Consecuencias
9. **I** - Implementación
10. **C** - Código
11. **U** - Usos conocidos
12. **P** - Patrones relacionados

Clasificaciones (A A D A P)

Según la actividad (PA PT PP PPe)

- **P**atrones de **S**oftware.
- **P**atrones de **T**esting.
- **P**atrones de **P**roceso.
- **P**atrones **P**edagógicos.

Según el nivel de abstracción (PA PA PD I)

- **P**atrones de **A**nálisis.
- **P**atrones de **A**rquitectura.
- **P**atrones de **D**iseño.
- **I**dioms.

Según el dominio de aplicación (F CE S TR)

- **F**inanciero.
- **C**omercio **E**lectrónico.

- **S**alud.
- **T**iempo **R**eal.

Según el ambiente/paradigma (PW PMD X I)

- **P**atrones **W**eb.
- **P**atrones de **M**odelos de **D**atos.
- **X**ML.
- **I**nteracción.

De acuerdo al propósito (C E B)

- **C**reacionales.
- **E**structurales.
- **B**e**h**aviorales (de comportamiento).

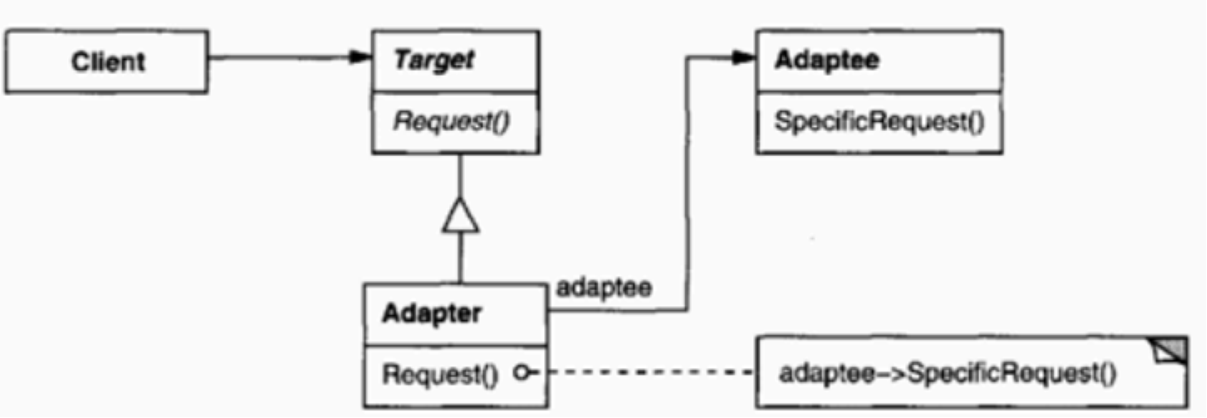
Patrones Estructurales

Nombre: Adapter

Propósito

- Convierte la interfaz de una clase en otra interfaz esperada por los clientes
- Permite cooperación entre clases con interfaces incompatibles.

Estructura



Motivación (problema)

- Quiere usar una clase existente pero la interfaz es incompatible
- Se quiere crear una clase reutilizable que coopere con otras que no tienen interfaces compatibles.

Consecuencias

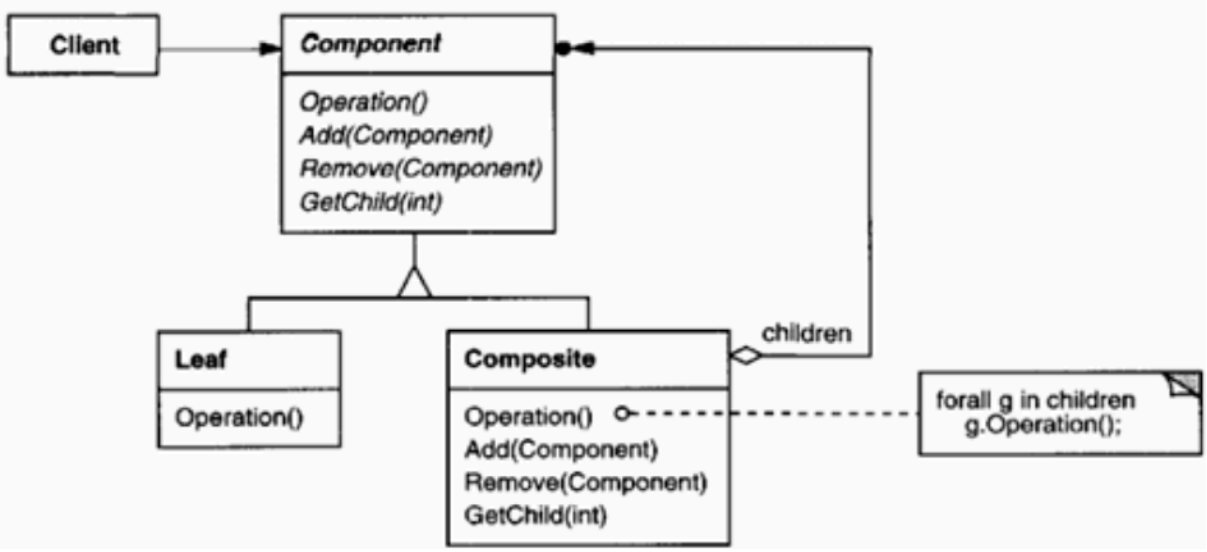
- No sirve para adaptar toda una jerarquía.
- Permite que el adapter redefina el comportamiento del adaptee.
- Un solo objeto adapter, no se necesita ningún puntero de indirección adicional para llegar al adaptee.

Nombre: Composite

Propósito

- Compone objetos en estructuras de árbol representando jerarquías parte-todo.
- Permite que los clientes traten de manera uniforme a todos los objetos de la jerarquía.

Estructura



Motivación (problema)

- Se requiere representar jerarquías parte-todo.
- Se quiere que los clientes no diferencian un compuesto de un simple y los trate de forma uniforme.

Consecuencias

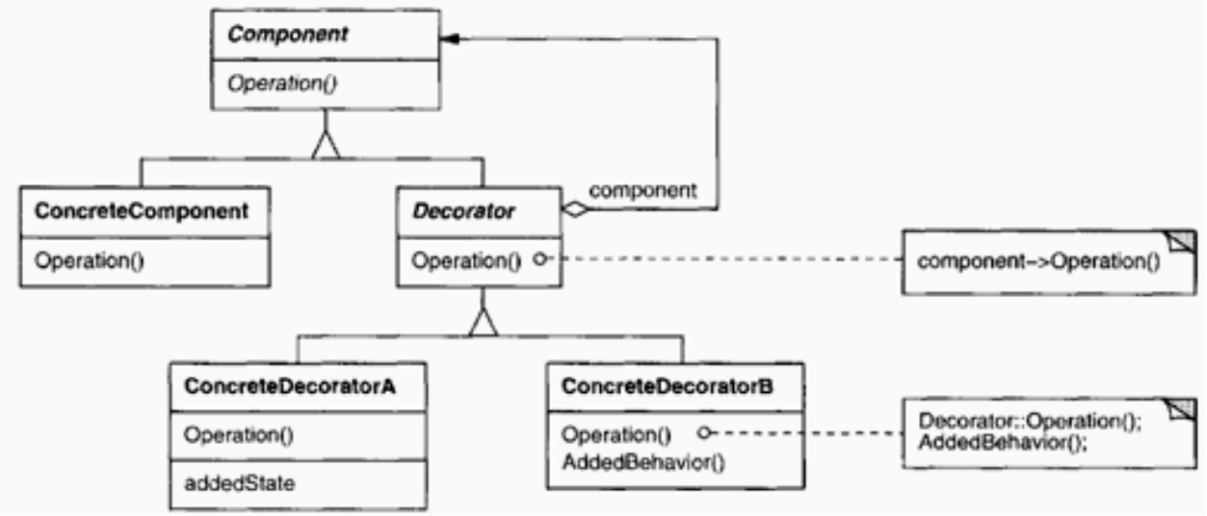
- Se definen jerarquías conformadas por objetos compuestos y primitivos. Donde se espere un primitivo puede recibir un compuesto.
- Simplifica el código al no tener que diferenciar compuesto y primitivo.
- Añadir nuevos tipos es fácil porque se adaptan a la estructura del patrón automáticamente.
- El diseño sale muy general, el compuesto puede tener limitaciones de solo contener ciertos primitivos y eso se debe chequear de forma dinámica.

Nombre: Decorator

Propósito

- Asigna responsabilidades adicionales de forma dinámica a un objeto.

Estructura



Motivacion (problema)

- Se requiere añadir objetos individuales de forma dinámica sin afectar a los otros.
- Las responsabilidades se quitan y sacan dinámicamente.
- No es viable extender por herencia.

Consecuencias

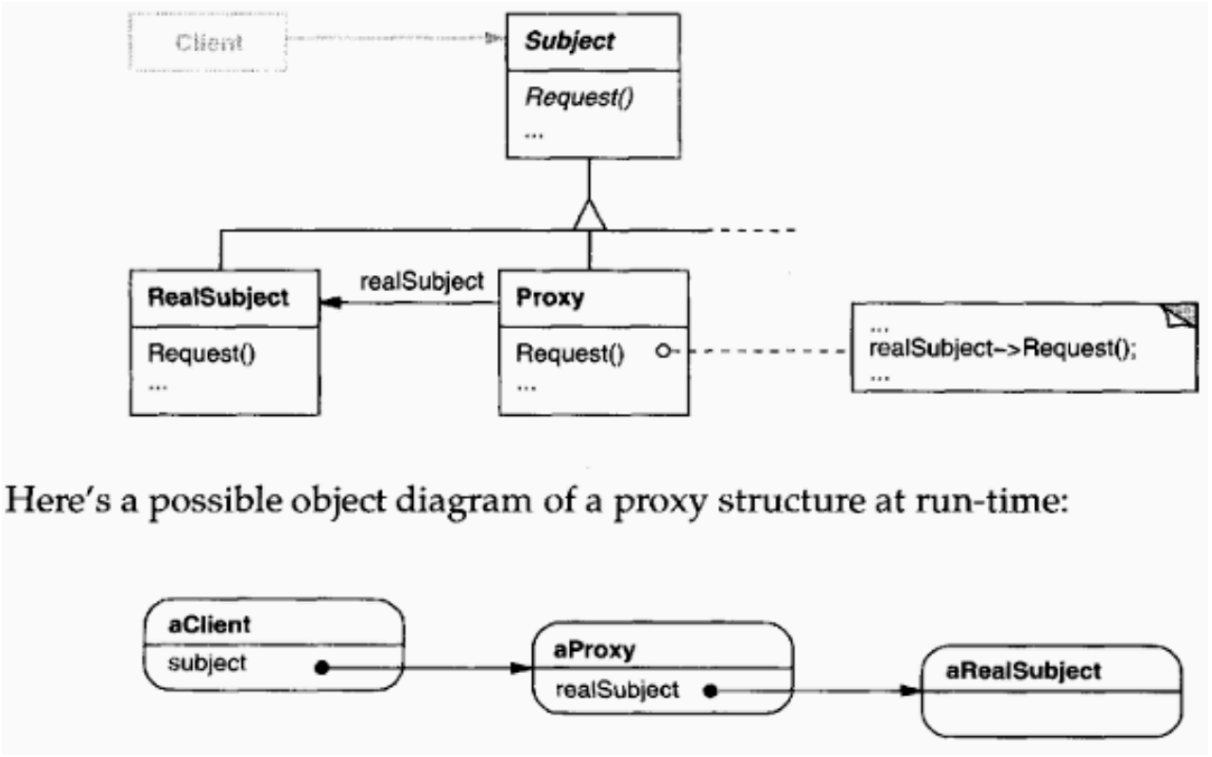
- Más flexibles que la herencia estática.
- Las superclases están menos cargadas, se enfoca en pagar solo lo que se necesita.
- No existe una identidad de objetos, un decorador se comporta como wrapper al componente.
- Muchos objetos pequeños y similares, diseño difícil de entender y depurar.

Nombre: Proxy

Propósito

- Intermediario de otro objeto para controlar su acceso.

Estructura



Motivación

- **Remoto**
  - Representante local de un objeto en otro espacio de direcciones.
- **Virtual**
  - Crea objetos costosos por encargo. (Conseguir info de db).
- **Protección**
  - Controla el acceso al objeto adicional. Tema permisos.
- **Referencia inteligente**
  - Lleva a cabo operaciones adicionales cuando se accede a un objeto.

Consecuencias

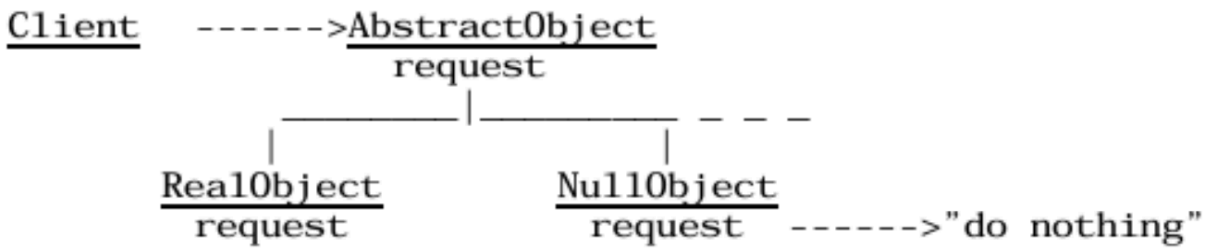
- Hay un nivel de indirección adicional, en el **remoto** oculta que el objeto está en un espacio de direcciones diferente, en el **virtual** optimiza cosas como crear objetos por encargo y en el de **protección** y **referencias inteligentes** permite realizar tareas de mantenimiento.
- Permite optimizar la copia-de-escritura, que se da cuando se copia un objeto grande, se deja hasta al final traerlo cuando se modifica efectivamente.

Nombre: null object

Propósito

- Actúa como sustituto de un objeto, compartiendo su interfaz pero centraliza el “no hacer nada” y oculta esos detalles al colaborador.

Estructura



Motivacion (problema)

- Debe haber una colaboración preexistente.
- Algunos colaboradores no deben hacer nada.
- Se quiere que los clientes ignoran la diferencia entre uno real y uno nulo.
- Se quiere reutilizar el comportamiento de no hacer nada en varios clientes de forma consistente.

Consecuencias

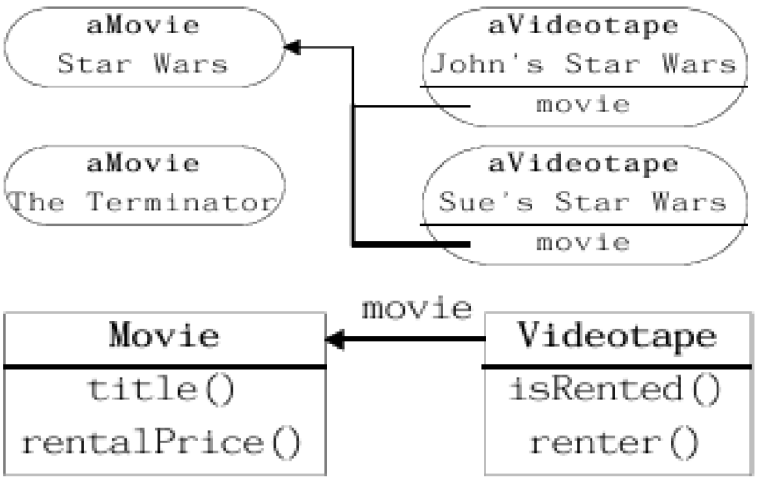
- Las jerarquías consisten de objetos reales y nulos.
- El código del cliente se simplifica al poder tratar reales y nulos uniformemente.
- Se encapsula en lo hacer nada y es fácil de encontrar y reutilizar.
- El comportamiento de no hacer nada es difícil de distribuir o mezclar comportamiento real de varios colaboradores.
- El nulo no se convierte en uno real nunca.

Nombre: Type object

Propósito

- Encontrar los atributos/métodos que se repiten en las clases y extraerlos a una clase.
- Crear nuevas “clases” de forma dinámica.
- Permite que un sistema proporcione reglas de verificación de tipos.

Estructura



Motivacion (problema)

- Hay que agrupar instancias de una clase según sus atributos y comportamientos comunes.
- Se requieren muchas subclases o no se saben cuantas.
- Se quieren crear agrupamientos no previstos durante el diseño de forma dinámica o cambiar la subclase de un objeto en ejecución.

Consecuencias

- Permite crear clases nuevas en tiempo de ejecución llamadas TypeObjects.
- Evita la explosion de subclases.
- Cambio dinámico de tipo. Cada objeto tiene la referencia a su TypeObject.
- Se complejiza al diseño al separar la cosa y el tipo, difícil de entender.

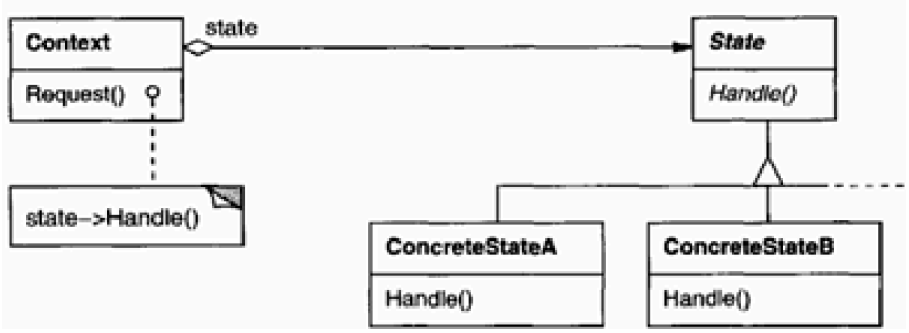
Patrones de comportamiento

Nombre: State

Propósito

- Permitir que un objeto modifique su comportamiento cada vez que cambie el estado interno, parecerá que el objeto cambia de clase.

Estructura



Motivacion (problema)

- Si el objeto depende de su estado para su comportamiento y debe cambiar dinámicamente.
- Si tiene condicionales con ramas que dependen del estado, muchas veces ahí se ven los estados del objeto (descubrimiento de clases).

Consecuencias

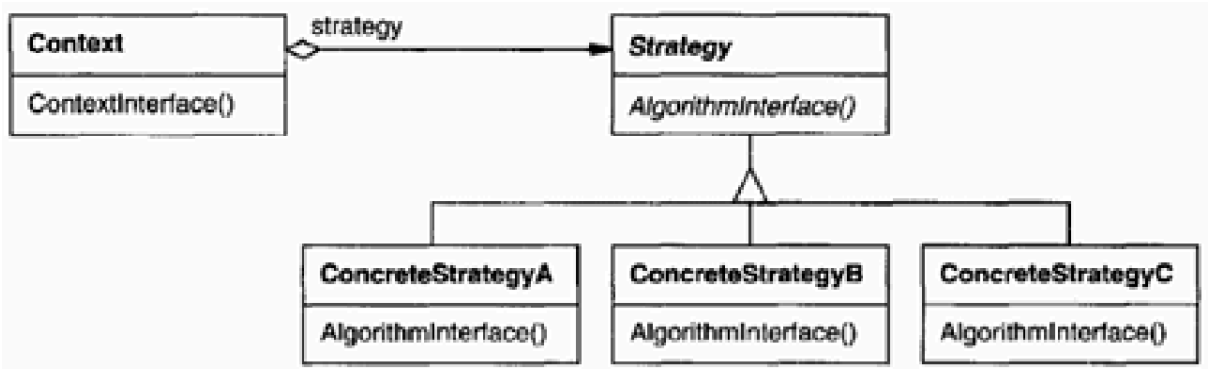
- Localiza el comportamiento dependiente del estado y lo divide en clases, haciendo un diseño menos compacto pero evita condicionales.
- Se hacen explícitas las transiciones entre estados protegiendo el contexto de estados internos inconsistentes.
- Se puede usar estados singleton, que se comparten.

Nombre: Strategy

Propósito

- Definir una familia de algoritmos, encapsularlos y hacerlos intercambiables,
- El algoritmo varía independiente del cliente que lo usa.

Estructura



Motivacion (problema)

- Muchas clases que solo difieren en comportamiento.
- Se necesita variar el algoritmo.
- El algoritmo requiere datos que el cliente no debe saber.
- La clase define varios comportamientos y elige mediante condicionales.

Consecuencias de usarlo

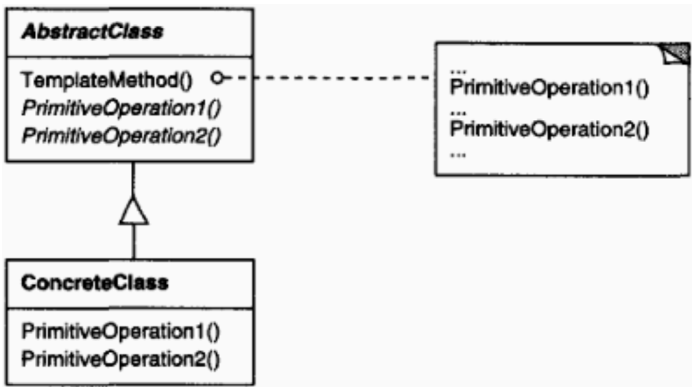
- Se define una jerarquía de estrategias con algoritmos y comportamientos para ser reutilizados.
- Se encapsula el algoritmo y permite variar el algoritmo independiente de contexto, facilitando el cambio, compresión y extensión.
- Se eliminan las sentencias condicionales sobre qué comportamiento presentan.
- El cliente conoce las estrategias y cómo difiere (queda expuesto a cuestiones de implementación), pero permite que elija entre estrategias.
- A la estrategia se le pasa todo el contexto pero puede no necesitar toda la info.
- Aumento de objetos en la aplicación.

Nombre: template method

Propósito

- Definir en un método el esqueleto de un algoritmo y que las clases redefinan los pasos sin cambiar la estructura.

Estructura



Motivacion (problema)

- Se quiere implementar parte de un algoritmo que no cambian, y dejar a las subclases implementar el resto.
- Para controlar las extensiones de la subclases.

Consecuencias

- Reutilización y factorización de código.

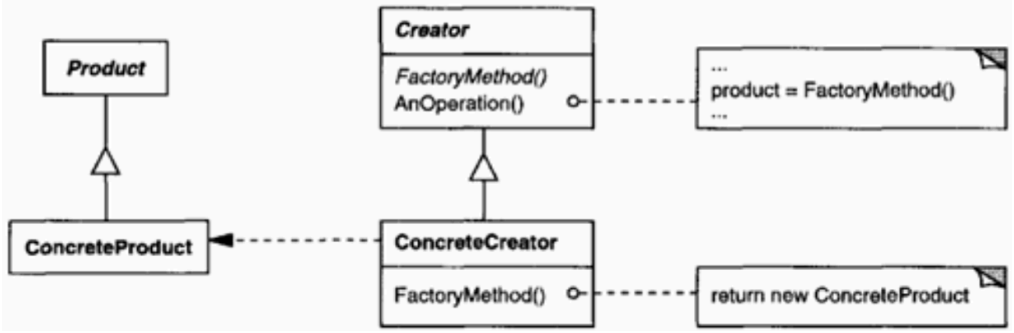
Patrones creacionales

Nombre: Factory Method

Propósito

- Definir una interfaz para crear un objeto y se deja que las subclases decidan que instanciar.

Estructura



Motivacion (problema)

- Una clase no puede prever qué objetos debe crear.
- Una clase quiere que sus subclases especifiquen que crean.
- Queremos saber en qué clase se delega la responsabilidad de crear algo.

Consecuencias

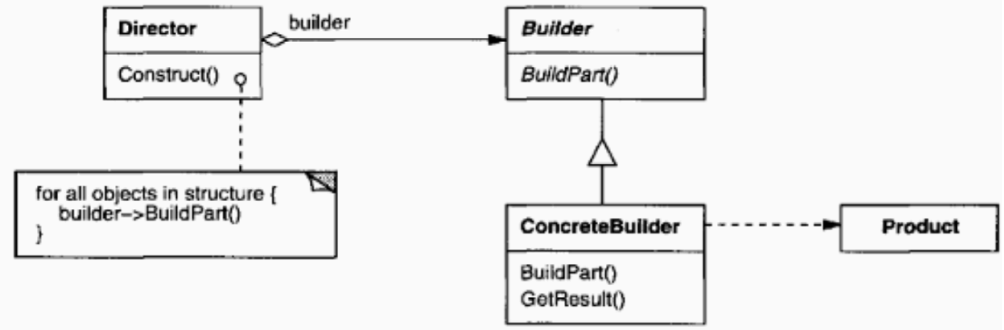
- Proporciona enganches para las subclases.
- Conecta jerarquías de clases paralelas.

Nombre: Builder

Propósito

- Separar la construcción de un objeto complejo de su representación, el mismo proceso de construcción puede crear diferentes representaciones.

Estructura



Motivacion (problema)

- El algoritmo de creación de un objeto debe ser independiente del objeto.
- El proceso de construcción debe permitir diferentes representaciones del objeto.
- Hay pasos.

Consecuencias

- El constructor proporciona al director una interfaz abstracta para armar el producto, pero oculta la representación y estructura interna, a su vez como está ensamblado.
- Aísla el código de construcción y representación.
- Aumenta el control sobre el proceso de construcción.

Nombre: Test double

SUT: System under test (lo que testeamos)
Doc: Dependent on component (el objeto que falta)

Propósito

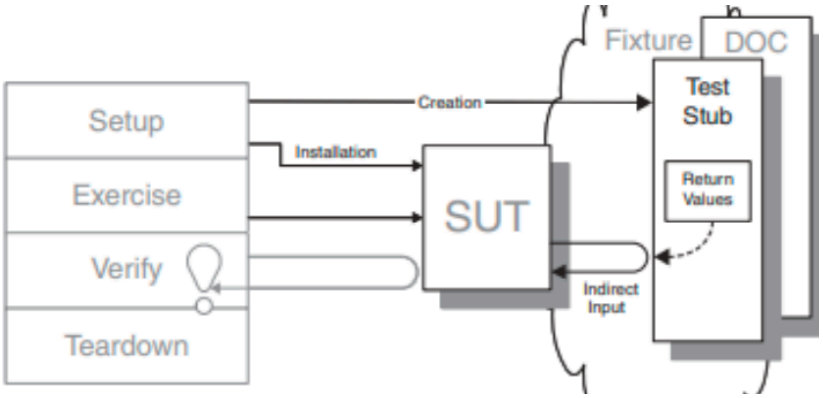
- Regresamos el DOC con un equivalente de pruebas

Test stub

Propósito

- Reemplazamos al objeto real con otro que proporciona entradas indirectas deseadas al SUT. (Recibe los mensajes que requiere el SUT enviar y retorna siempre lo mismo).

Estructura



Motivación

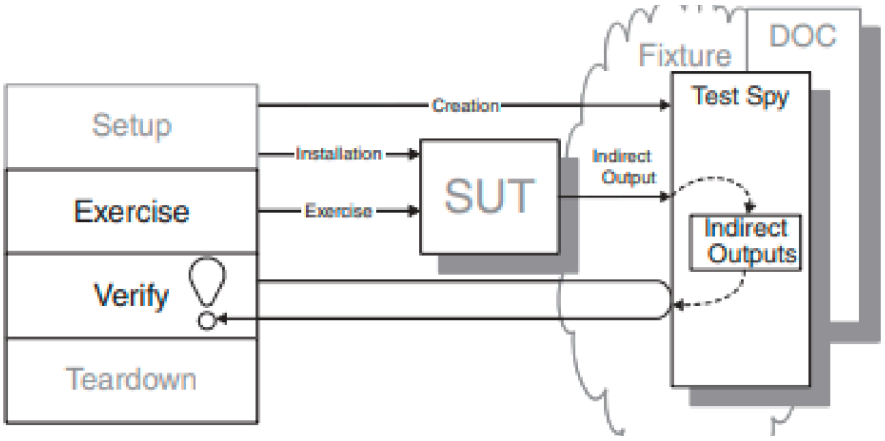
- Si el SUT tiene que recibir entradas de algún lugar y no podemos controlarlas y tenemos que predefinirlas
- O para inyectar valores para pasar un test que requiera una llamada a otro software no disponible.

Test spy

Propósito

- Test stub + registros de outputs. Capturamos las llamadas realizadas al componente por el SUT.

Estructura



Motivación (propósito)

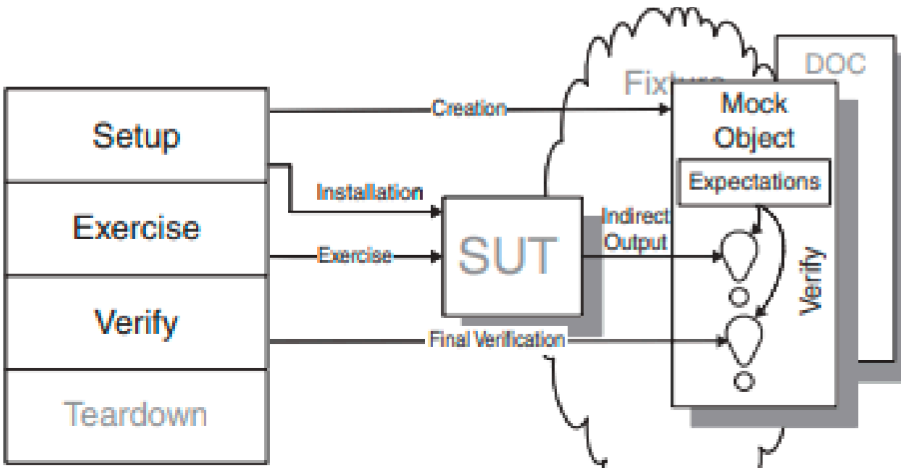
- Hay que verificar las salidas indirectas del SUT.
- Queremos tener acceso a todas las llamadas salientes del SUT antes de afirmar algo.

Mock Object

Propósito

- Reemplazamos un objeto del que depende el SUT para hacer pruebas que verifica que está siendo utilizado correctamente.
- Test stub + verification of outputs

Estructura



Motivación (problemas)

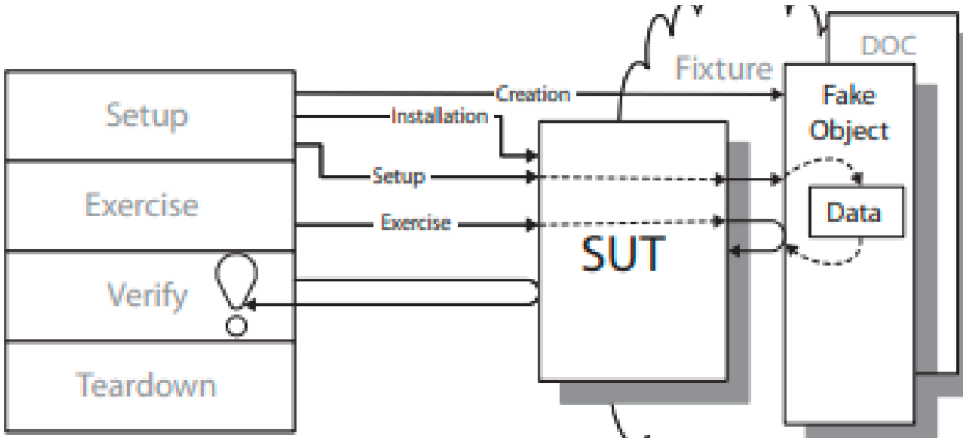
- Usamos el Mock para observar cuando necesitamos verificar el comportamiento para evitar tener requisitos no probados debido a no poder observar efectos secundarios del SUT.
- Debemos poder predecir los valores de la mayoría o todos los argumentos de las llamadas a métodos antes de ejercitar el SUT. No usar si una afirmación fallida no puede ser reportada al testrunner.

Fake object

Propósito

- Literalmente imitamos al real con un reemplazo pero que su implementación no es suficiente para producción.

Estructura



Motivación (problemas)

- Lo usamos cuando el DOC no ta' y se requieren secuencias de comportamiento más complejas.

Refactoring

Proceso de cambio de un sistema de software que mejora la organización, flexibilidad, adaptabilidad y mantenibilidad sin alterar el comportamiento externo del sistema.

Leyes de LEHMAN (Ca Cr Co Ca)

- **Cambio continuo**
  - Los sistemas se deben adaptar o se harán menos satisfactorios.
- **Crecimiento continuo**
  - La funcionalidad se debe incrementar para mantener la satisfacción del cliente.
- **Complejidad creciente**
  - Al evolucionar un sistema se debe trabajar para evitar la complejidad.
- **Calidad decreciente**
  - La calidad declina si no se mantiene rigurosamente.

Refactoring (sustantivo)

Cambio hecho a la estructura interna del software para aumentar la legibilidad y modificabilidad, tienen nombre y pasos. Cada cambio es catalogado.

Refactor (verbo)

Es el aplicar los refactorings.

Características (I D T)

Implica

Eliminar duplicaciones, simplificar lógicas complejas, y clarificar códigos

Debe ser aplicado

Cuando tengo el código y pasó los test, o a medida que voy desarrollando y me encuentro código ilegible o necesito reorganizar.

Testear después de cada cambio

Como ayuda

La introduce mecanismos que solucionen problemas mediante cambios pequeños, fáciles y seguros de aplicar y evidencian otros cambios.

Automatización del Refactoring

Herramientas de automatización del refactoring son potentes, restrictivas para preservar el comportamiento del programa, interactivas y chequean solo lo posible desde el árbol de sintaxis y la tabla de símbolos.

Deuda técnica

Este concepto representa las consecuencias de un diseño rápido y sucio.

- **Capital de la deuda**
    - Costo de remediar los problemas de diseño (costo de refactoring).
- **Interés de la deuda**
    - Costo adicional por preservar deuda técnica.

Refactorings

Composición de Métodos

Permiten “distribuir” el código adecuadamente ya que los métodos largos son problemáticos.

Extract Method

Motivación

- Métodos largos.
- Métodos muy comentados.
- Incrementar el reuso.
- Incrementar la legibilidad.

Solución

Mueve el código a un nuevo método (o función) separado y reemplaza el código antiguo con una llamada al método.

Replace Temp with Query

Motivación

- Evitar métodos largos, las temporales, al ser locales, fomentan métodos
- largos.
- Para poder usar una expresión desde otros métodos.
- Antes de un Extract Method, para evitar parámetros innecesarios.

Solución

- Extraer la expresión en un método.
- Reemplazar todas las referencias a la var. temporal por la expresión.
- El nuevo método luego puede ser usado en otros métodos.

Mover aspectos entre Objetos

Ayudan a mejorar la asignación de responsabilidades.

Move Method

Motivación

Un método está usando o usará muchos servicios que están definidos en una clase diferente a la suya.

Solución

- Mover el método a la clase donde están los servicios que usa.
- Convertir el método original en un simple delegación o eliminarlo.

Manipulación de la generalización

Ayudan a mejorar las jerarquías de clases.

Pull Up Method

Motivación

Existen subclases que realizan trabajos similares.

Solución

Hacer que los métodos sean idénticos y luego moverlos a la superclase correspondiente.

Organización de Datos

Facilitan la organización de atributos.
Algunos ejemplos de Refactorings

- Self Encapsulate Field.
- Encapsulate Field/Collection.
- Replace Data Values with Object.
- Replace Magic Number with Symbolic Constant.

Simplificación de Expresiones Condicionales

Ayudan a simplificar los condicionales.

Replace Conditional with Polymorphism

Motivación

Existe un condicional que realiza varias acciones dependiendo del tipo de objeto o propiedades.

Solución

Crear subclases que coincidan con las ramas de la condicional. En ellas, crear un método compartido y mover el código de la rama correspondiente de la condicional a ésta. Luego reemplazar la condicional con la llamada al método relevante. El resultado es que la implementación adecuada se logrará a través del polimorfismo dependiendo de la clase del objeto.

Decompose Conditional

Motivación

Se tiene una expresión condicional compleja.

Solución

Extraer métodos de la condición, la parte “then” y la parte “else”.

Simplificación de Invocación de Métodos

● Sirven para mejorar la interfaz de una clase.

Rename Method

Motivación

Todo buen código debería comunicar con claridad lo que hace, sin necesidad de agregar comentarios.

Bad smells

Indicios de problemas que requieren la aplicación de refactorings.

Bloaters: Large Class

Una clase posee muchas variables de instancias o métodos y pueden ir en otras clases.

Problemas

- Baja cohesión
- Algunos métodos pueden ser de otra clase
- Código duplicado

Usar

- Extract class
- Extract subclass

Bloaters: Long Method

Muchas líneas de código (20 max pero depende el lenguaje).

Problemas

- Un método muy largo es difícil de entender, cambiar o reusar.

Usar

- Extract method
- Decompose conditional
- Replace temp with query

Bloaters: Long Parameter List

Problemas

- Un método con muchos parámetros es más difícil de entender y reusar. La excepción es para no crear dependencia entre el llamador y llamado.

Usar

- Replace parameter with method
- Preserve whole object
- Introduce parameter object

Couplers: Feature envy

Un método en una clase usa los datos y métodos de otra clase para realizar su trabajo.

Problemas

- Los datos y acciones sobre los mismos se deben tener en la misma clase, por tanto el método está mal ubicado e indica un problema en el diseño.

Usar

- Move method

Dispensables: Data Class

Clase contenedor de datos que solo tiene variables y getters/setters para ellos.

Problemas

- Otras clases tienen feature envy.

Usar

- Move method

Dispensables: Duplicated Code

El mismo código, o código muy similar, aparece en muchos lugares.

Problemas

- Alargan el código, y los cambios no son tan fáciles porque hay que replicarlos en cada duplicación.

Usar

- Extract Method.
- Pull Up Method.
- Form Template Method.

Object Orientation Abusers: Conditionals

Existen sentencias condicionales que contienen lógica para diferentes tipos de objetos. Si estos son instancias de una misma clase, eso indica que es necesario subclasificar.

Problemas

- El mismo condicional aparece en muchos lugares.

Usar

- Replace Conditional with Polymorphism.

Refactoring to Patterns

Sobre-Ingeniería

Se hace para

- Predecir futuros cambios (no se puede).
- No quedar inmerso y acarrear un mal diseño (los patrones pueden hacer que no usemos soluciones más simples).

Consecuencia

El código complejo perdura y complica el mantenimiento, nadie lo entiende ni quiere cambiar y llevar a código duplicado.

Form template method

Motivación

Dos o más métodos en subclases realizan pasos similares en el mismo orden, pero los pasos son distintos.

Solución

Generalizamos los métodos, extract method y pull up method del método con los pasos.

Pros

- Eliminar código duplicado.
- Simplifica y comunica efectivamente los pasos de un algoritmo genérico
- Permite que las subclases adapten un algoritmo fácilmente.

Contras

Complica el diseño si las subclases tienen muchos métodos para el algoritmo.

Replace Conditional Logic with Strategy

Motivación

La lógica condicional de un método controla qué variante va a ejecutar.

Solución

Crear un strategy para cada variante, y hacer el qué método original delegue al strategy.

Replace State-Altering Conditionals with State

Motivación

- Las condiciones para controlar las transiciones de estado son complejas.
- Saber qué cambios hacen los estados.
- La lógica condicional se vuelve difícil de extender.
- Los refactorings más simples no alcanzan.

Solución

Reemplazar los condicionales con States que manejen estados específicos y transiciones entre ellos.

Move Embellishment to Decorator

Introduce Null Object

Motivación

Código duplicado relacionado con cómo manejar el valor nulo de una variable.

Solución

Reemplazar la lógica de testeo por null con un Null Object.

Test Driven Development (TDD)

Combina

Test First Development (escribir el test antes del código) y Refactoring.

Objetivo

Pensar en el diseño y que se espera de cada requerimiento antes de escribir el código, escribir código limpio funcional.

Filosofía

- Primero escribir el test, luego el código.
- Test funcionales para los casos de uso, test de unidad para las pequeñas partes y aislar errores.
- No agregar funcionalidad hasta que no ande el test.
- Pequeños pasos, un test, un poco de código.
- Si pasan los test refactorizar.

Reglas de TDD

- **Diseño incremental**
    - Si el código funciona lo usamos como feedback para tomar decisiones entre iteraciones.
- **Los programadores hacen sus propios test**
    - No es eficiente de otra forma.

- **Diseño con componentes cohesivos y desacoplados**
    - Para mejor evolución y mantenimiento del sistema.

Automatización de TDD

Es necesario para aplicar TDD el uso de herramientas de automatización en el testing.

Consecuencias de aplicar TDD correctamente

- Menor chance de sobrediseño
- Proceso de elicitación del dominio (análisis).
- Arquitectura surge incrementalmente.
- Refactoring mantiene código mínimo y limpio.
- Requerimientos se convierten en test cases (casos de prueba).
- Último “Release” como entregable.
- Adaptabilidad al cambio.

Dificultades a tener en cuenta a la hora de aplicar TDD

- Refactoring mantiene código mínimo y limpio.
- Mantener objetivos a largo plazo.
- Cambios a DB pueden ser costosos.
- Cambios a API 's son difíciles de realizar sin romper el código de los clientes.
- Interacción expertos del dominio.

¿Por qué no dejar testing para el final?

- Para conocer cuál es el final.
- Para mantener bajo control un proyecto con restricciones de tiempo ajustadas.
- Para poder refactorizar rápido y seguro.
- Para darle confianza al desarrollador de que va por buen camino.
- Como una medida de progreso.

Granularidad

Test de Aceptación

- Por cada funcionalidad esperada.
- Escritos desde la perspectiva del cliente.

Test de Unidad

- Aislar cada unidad de un programa y mostrar que funciona correctamente.
- Escritos desde la perspectiva del programador.

Otros refactoring to patterns

Test Utility Method

Encapsulamos la dependencia innecesaria (del método que no testeamos) un método, por ejemplo la creación.

Creation Method

Tipo de Test Utility Method que oculta como se crean objetos ready-to-use atrás de un método.

Patrones de UI web

- Debemos poder adaptarnos a los cambios (crecimiento del sitio, agregado de mayor interacción).
- Necesitamos poder innovar.
- Queremos poder
    - Aprender del feedback.
    - Fallar rápido.
    - Adaptarnos más rápido.

Refactorings de UX

Propósito

Mejorar la calidad externa de una aplicación web (usabilidad, accesibilidad, UX). preservando la funcionalidad.

Alcance

Navegación, Presentación e Interacción del usuario.

Automatización CSWR

Los Client-Side Web Refactorings son refactorings que solucionan los malos olores de la UX aplicando cambios del lado del cliente sobre el DOM (Document Object Model) de las páginas web.

Frameworks

Re-usamos

- Conceptos e ideas
- Diseños o estrategias de diseño
- Funciones y estructuras de datos
- Componentes y servicios
- Aplicaciones completas que adaptamos e integramos

Porque re-usar

- Más productividad, reduciendo costos de desarrollo y mantenimiento, tiempos de entrega y puesta en el mercado.
- Aumenta la calidad aprovechando mejoras, correcciones y conocimiento de especialistas externos.

Dificultades

- Problemas de mantenimiento si no controlamos lo que re-usamos.
- Algunos programadores quieren hacer todo ellos mismos.
- Hacer software reusable es más difícil y costoso.
- Software reusable es más difícil y costoso.
- Encontrar, aprender a usar, y adaptar componentes reusables requiere esfuerzo adicional.

Librería de clases

- Resuelven problemas comunes a la mayoría de las aplicaciones.
- Cada clase resuelve un problema concreto, es independiente y no espera nada de nosotros.
- Nuestro código controla al código de la librería.

Framework

- Aplicación semicompleta, re-usable, Devs incorporan el framework en las apps y los especializan.
- Conjunto de clases concretas y abstractas, relacionadas para proveer una arquitectura re-usable para un familia de aplicaciones relaciones.
- Define el esqueleto, el dev define sus propias características y lo completa.

Frameworks de Infraestructura

- **Descripción**: Proporcionan una infraestructura portátil y eficiente para construir una gran variedad de aplicaciones.
- **Focos Principales**: Interfaces de usuario (desktop, web, móviles), seguridad, contenedores de aplicación, procesamiento de imágenes, procesamiento de lenguaje, comunicaciones.
- **Propósito**: Resuelven problemas generales del desarrollo de software, dirigidos principalmente a los programadores y no a los usuarios finales.
- **Ejemplo**: Qt, un framework para el desarrollo de interfaces de usuario en múltiples plataformas.

Frameworks de Integración

- **Descripción**: Utilizados para integrar componentes de aplicaciones distribuidas, como la base de datos con la aplicación y esta con su cliente liviano.
- **Focos Principales**: Modularización, reutilización y extensión de la infraestructura de software en un entorno distribuido.
- **Propósito**: Mejoran la capacidad de los desarrolladores para construir aplicaciones modulares y extensibles que funcionen de manera transparente en ambientes distribuidos.
- **Ejemplo**: Spring Framework, un framework de integración popular en el desarrollo de aplicaciones empresariales Java.

Frameworks de Aplicación

- **Descripción**: Abordan dominios de aplicación amplios que son fundamentales para las actividades empresariales.
- **Focos Principales**: ERP (Enterprise Resource Planning), CRM (Customer Relationship Management), gestión de documentos, cálculos financieros.
- **Propósito**: Resuelven problemas derivados de las necesidades de los usuarios de las aplicaciones, ofreciendo un retorno de inversión claro y justificado.
- **Ejemplo**: SAP ERP, un framework en el dominio de ERP que ayuda a las empresas a gestionar procesos de negocio clave y optimizar sus operaciones.

Frozenspots

- **Definición**: Aspectos del framework que no se pueden cambiar. Si estos se ven modificados, es porque se aplicó hacking.
- **Identificación**: Se ven en los requerimientos.

Hotspots

- **Definición**: Puntos de extensión que permiten introducir variantes y construir aplicaciones diferentes.
- **Identificación**: Se ven en los requerimientos.

Caja blanca	Caja negra
Implementación de puntos extensión con herencia	Implementación de puntos extensión con composición
Más fácil de desarrollar.	Más difícil de desarrollar.
Más difícil de usar.	Más fácil de usar.
Los frameworks se hallan en un lugar en el medio.	

Inversión de control

- Es el proceso mediante el cual el código de framework llama al nuestro en vez de que nosotros lo llamemos como si fuera un librera
- Permite que los pasos canónicos de procesamiento de la aplicación, comunes a todas las instancias sean especializados por **objetos tipo manejadores de eventos a los que invoca el framework como parte de su mecanismo reactivo de despacho.**

Plantillas y Ganchos

- **Plantillas**: implementan lo constante.
- **Ganchos**: implementan lo variable.

Usando herencia:

- Plantilla implementada en clase abstracta.
- Ganchos implementada en subclases.
- Útil con pocas variantes/combinaciones.
- Acceso a variables de instancia.
- Puede llevar a muchas clases y duplicación de código.

Usando composición:

- Objeto implementa la plantilla y delega los ganchos.
- Evita duplicación de código y muchas clases.
- Necesita parámetros para implementar ganchos.
- Permite cambiar comportamiento en tiempo de ejecución.

En frameworks:

- Desarrollador del framework controla las plantillas.

- Usuario implementa/instancia los ganchos.

Hook Methods

Herramienta que nos permite definir Hotspots. No es un Hotspot.

Template Method

Herramienta que nos permite definir Frozenspots. No es un Frozenspot.