Resumen de Objetos

Resumen para el examen de promoción de Objetos 2

Resumen de Objetos **1**
- ¿Qué es un Patrón? 4
  - Elementos de un patrón 5
  - Clasificaciones 5
- Patrones Estructurales 6
  - Adapter (Wrapper) 6
    - Propósito 6
    - Estructura 6
    - Cuándo debería ser usado 6
    - Consecuencias de usarlo 6
  - Composite 7
    - Propósito 7
    - Estructura 7
    - Cuándo debería ser usado 7
    - Consecuencias de usarlo 7
  - Decorator (Wrapper) 8
    - Propósito 8
    - Estructura 8
    - Cuándo debería ser usado 8
    - Consecuencias de usarlo 8
  - Proxy (Surrogate) 9
    - Propósito 9
    - Estructura 9
    - Cuándo debería usarlo 9
    - Consecuencias de usarlo 10
  - Null Object (Stub) 10
    - Propósito 10
    - Estructura 10
    - Cuándo debería usarlo 10
    - Consecuencias de usarlo 10
  - Type Object (Power Type, Item Descriptor, Metaobject) 11
    - Propósito 11
    - Estructura 11
    - Cuándo debería usarlo 11
    - Consecuencias de usarlo 12
- Patrones de Comportamiento 12

State (Objects for States)..... 12
Propósito..... 12
Estructura..... 13
Cuándo debería usarlo..... 13
Consecuencias de usarlo..... 13
Strategy (Policy)..... 13
Propósito..... 13
Estructura..... 14
Cuándo debería usarlo..... 14
Consecuencias de usarlo..... 14
Template Method..... 15
Propósito..... 15
Estructura..... 15
Cuándo debería usarlo..... 15
Consecuencias de usarlo..... 15
Patrones Creacionales..... 16
Factory Method (Virtual Constructor)..... 16
Propósito..... 16
Estructura..... 16
Cuándo debería usarlo..... 16
Consecuencias de usarlo..... 16
Builder..... 16
Propósito..... 16
Estructura..... 17
Cuándo debería usarlo..... 17
Consecuencias de usarlo..... 17
Test Double..... 17
SUT y DOC..... 17
Propósito..... 18
Test Stub..... 18
Propósito..... 18
Estructura..... 18
Cuándo debería usarlo..... 18
Test Spy..... 19
Propósito..... 19
Estructura..... 19
Cuándo debería usarlo..... 19
Mock Object..... 19
Propósito..... 19
Estructura..... 20
Cuándo debería usarlo..... 20
Fake Object..... 20
Propósito..... 20

Estructura..... 21
Cuándo debería usarlo.....21
Refactoring..... 21
Leyes de Lehman..... 21
Refactoring (Sustantivo).....22
Refactor (Verbo).....22
Características..... 22
¿Cómo ayuda?.....22
Automatización del Refactoring.....22
Deuda Técnica.....22
Refactorings.....23
Composición de Métodos..... 23
Extract Method.....23
Replace Temp with Query.....23
Mover aspectos entre Objetos..... 23
Move Method..... 23
Manipulación de la generalización.....24
Pull Up Method.....24
Organización de Datos.....24
Simplificación de Expresiones Condicionales..... 24
Replace Conditional with Polymorphism..... 24
Decompose Conditional..... 24
Simplificación de Invocación de Métodos.....24
Rename Method.....25
Bad Smells - Code Smells..... 25
Large Class - Bloaters.....25
Long Method - Bloaters..... 25
Long Parameter List - Bloaters.....25
Feature Envy - Couplers..... 25
Data Class - Dispensables..... 26
Duplicated Code - Dispensables..... 26
Conditionals - Object Orientation Abusers..... 26
Refactoring to Patterns.....26
Sobre-Ingeniería..... 26
Form Template Method..... 27
Replace Conditional Logic with Strategy.....27
Replace State-Altering Conditionals with State..... 27
Move Embellishment to Decorator.....28
Introduce Null Object..... 28
Test Driven Development (TDD)..... 28
Combina.....28
Objetivo.....28
Filosofía..... 28

Reglas de TDD.....28
Diseñar incrementalmente.....28
Los programadores escriben sus propios tests.....28
El diseño debe consistir de componentes altamente cohesivos y desacoplados entre sí.....28
Automatización de TDD.....29
Consecuencias de aplicar TDD correctamente.....29
Dificultades a tener en cuenta a la hora de aplicar TDD.....29
¿Por qué no dejar testing para el final?.....29
Granularidad.....29
Test de Aceptación.....29
Test de Unidad.....29
Otros refactoring to patterns.....30
Fragile Test - Test Smell.....30
Refactorings para Fragile Test.....30
Test Utility Method.....30
Creation Method.....30
Patrones de UI web.....30
Refactorings de UX.....30
Automatización CSWR.....30
Frameworks.....31
Reúso.....31
¿Qué es lo que reusamos?.....31
¿ Por qué reusamos?.....31
¿Qué dificultades tenemos?.....31
Librería de Clases.....31
Framework.....31
Frameworks de Infraestructura.....32
Frameworks de Integración.....32
Frameworks de Aplicación.....32
Frozenspots.....32
Hotspots.....32
Caja blanca vs Caja negra.....33
Inversión de Control.....33
Plantillas y Ganchos.....33
Hook Methods.....33
Template Method.....33

¿Qué es un Patrón?

- **Según Alexander**

    - *“Un patrón describe un problema que ocurre una y otra vez en un contexto, y además, describe el núcleo de la solución a ese problema de tal forma que pueda ser usada millones de veces sin tener que hacer lo mismo dos veces.”*
- **Según GangOfFour**
    - *"Un patrón de diseño nombra, abstrae e identifica los aspectos clave de una estructura de diseño común que lo hace útil para crear un diseño orientado a objetos reutilizable. El patrón de diseño identifica las clases e instancias participantes, sus roles y colaboraciones, y la distribución de responsabilidades. Cada patrón de diseño se centra en un problema o cuestión particular del diseño orientado a objetos. Describe cuándo se aplica, si puede aplicarse en vista de otras restricciones de diseño y las consecuencias y compensaciones de su uso."*
- 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

- Nombre.
- Propósito.
- Motivación (Problema).
- Aplicabilidad.
- Estructura.
- Participantes.
- Colaboraciones.
- Consecuencias.
- Implementación.
- Código.
- Usos conocidos.
- Patrones relacionados.

Clasificaciones

- Según la actividad
    - Patrones de Software.
    - Patrones de Testing.
    - Patrones de Proceso.
    - Patrones pedagógicos.
- **Según el nivel de abstracción**
    - Patrones de Análisis.
    - Patrones de Arquitectura.
    - Patrones de Diseño.
    - Idioms.
- **Según el dominio de aplicación**
    - Financiero.
    - Comercio electrónico.

    - Salud.
    - Tiempo Real.
- **Según el ambiente/paradigma**
    - Patrones Web.
    - Patrones de Modelos de datos.
    - XML.
    - Interacción.
- **De acuerdo al propósito**
    - **Creacionales.**
    - **Estructurales.**
    - **De comportamiento**

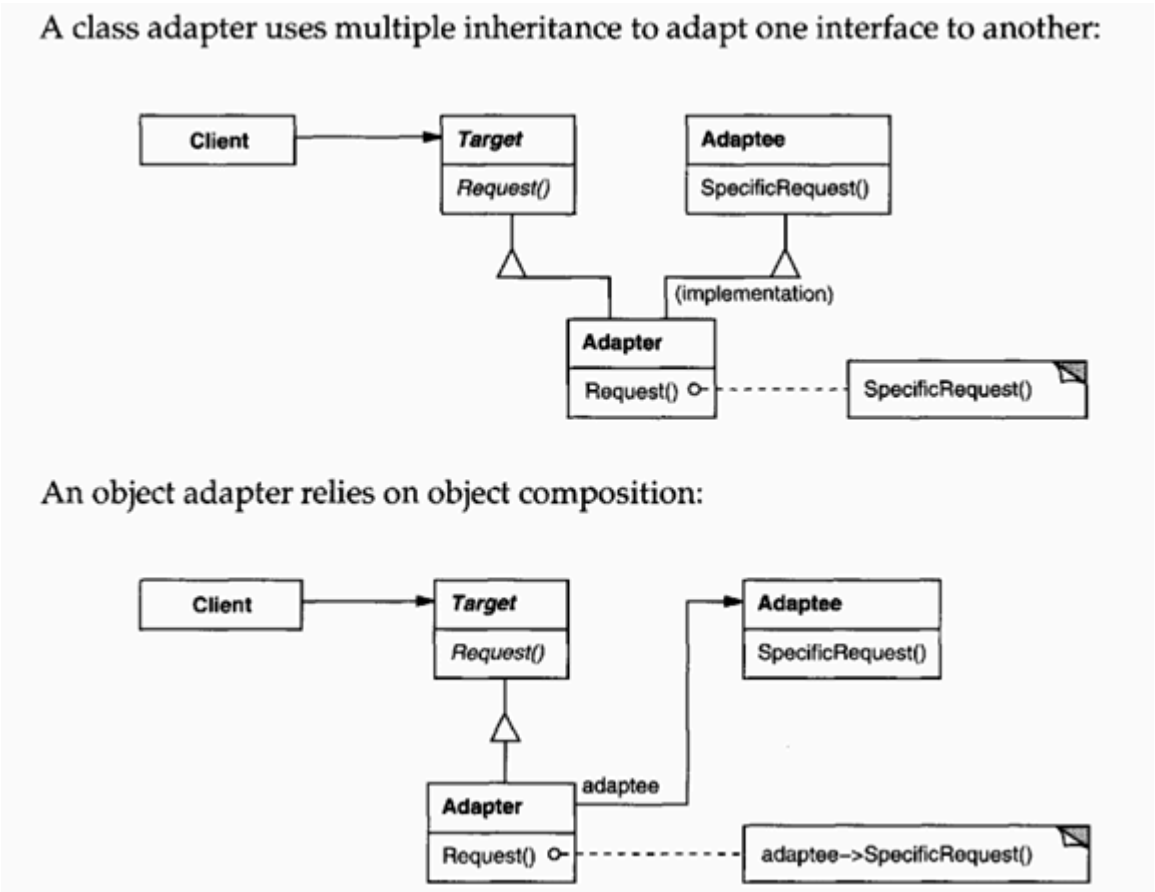
Patrones Estructurales

Adapter (Wrapper)

Propósito

- Convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes.
- Permite que cooperen clases que de otra forma no podrían hacerlo (ya que poseen interfaces incompatibles).

Estructura



Cuándo debería ser usado

- Se quiere usar una clase existente y su interfaz no concuerda con la que se necesita.
- Se quiere crear una clase reutilizable que coopere con clases no tienen porqué tener interfaces compatibles.

Consecuencias de usarlo

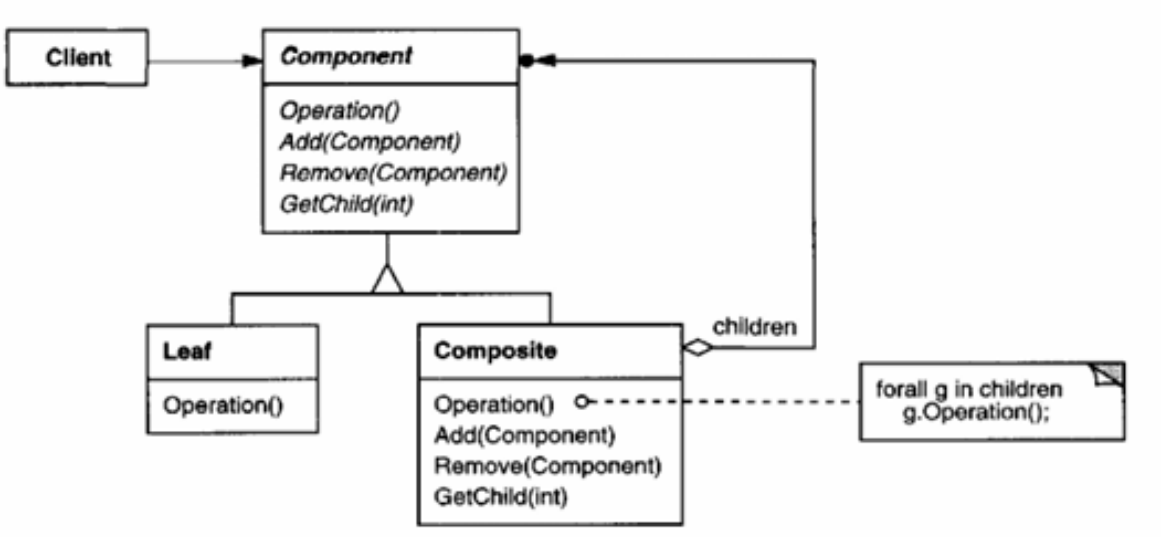
- Un Adaptador de Clases no nos servirá cuando lo que queremos es adaptar una clase y todas sus subclases.
- Permite que el Adaptador redefina comportamiento del Adaptado.
- Se introduce un solo objeto Adaptador y no se necesita ningún puntero de indirección adicional para obtener el objeto adaptado.

Composite

Propósito

- Compone objetos en estructuras de árbol para representar jerarquías de parte-todo.
- Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

Estructura



Cuándo debería ser usado

- Se quiere representar jerarquías de objetos parte-todo.
- Se quiere hacer que los Clientes no diferencien entre un objeto compuesto y uno individual. De esa forma, el Cliente puede tratar a cualquier objeto de la estructura de manera uniforme.

Consecuencias de usarlo

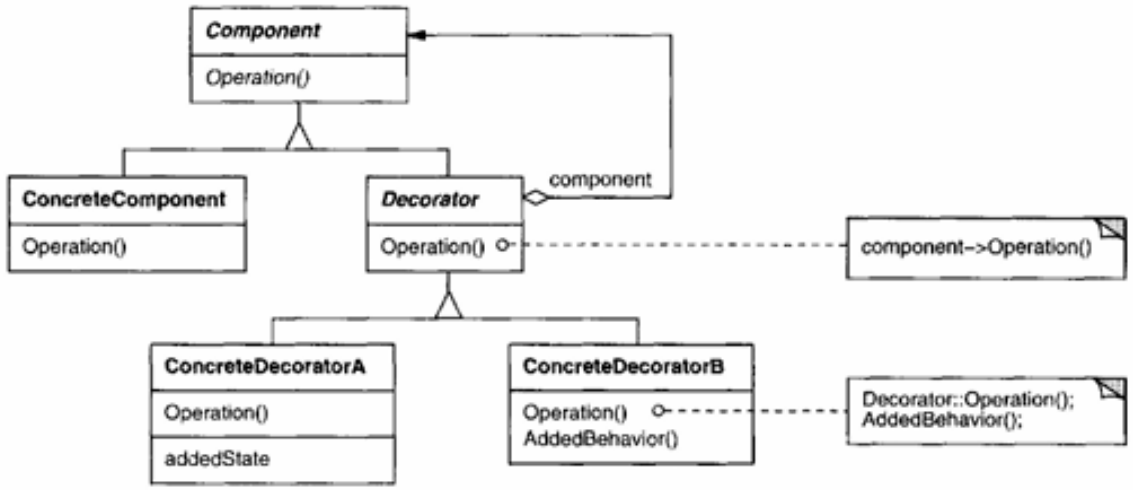
- El patrón define jerarquías de clases formadas por objetos primitivos y compuestos. Los primitivos pueden componerse para generar uno compuesto que sea más complejo, esto hace que si el código en alguna parte espera un primitivo, también pueda recibir uno compuesto.
- Simplifica el código del cliente al no tener que diferenciar si está tratando con un objeto primitivo o uno compuesto.
- Facilita añadir nuevos tipos de objetos ya sean compuestos o primitivos ya que se adaptan automáticamente con la estructura del patrón y el código que conlleva el mismo.
- Puede generar un diseño muy general.
- Hace difícil restringir los componentes de un compuesto, es decir, quizás se quiere que un compuesto tenga ciertos primitivos y esto es difícil de controlar. Por eso, con este patrón no se puede confiar en el sistema de tipos, para esos chequeos, estos se deben de realizar por nosotros en tiempo de ejecución.

Decorator (Wrapper)

Propósito

- Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender funcionalidades.

Estructura



Cuándo debería ser usado

- Se quiere añadir objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos.
- Existen responsabilidades que pueden ser retiradas y añadidas dinámicamente.
- La extensión mediante la herencia no es viable. O puede ser que una definición de una clase esté oculta o que no esté disponible para ser heredada.

Consecuencias de usarlo

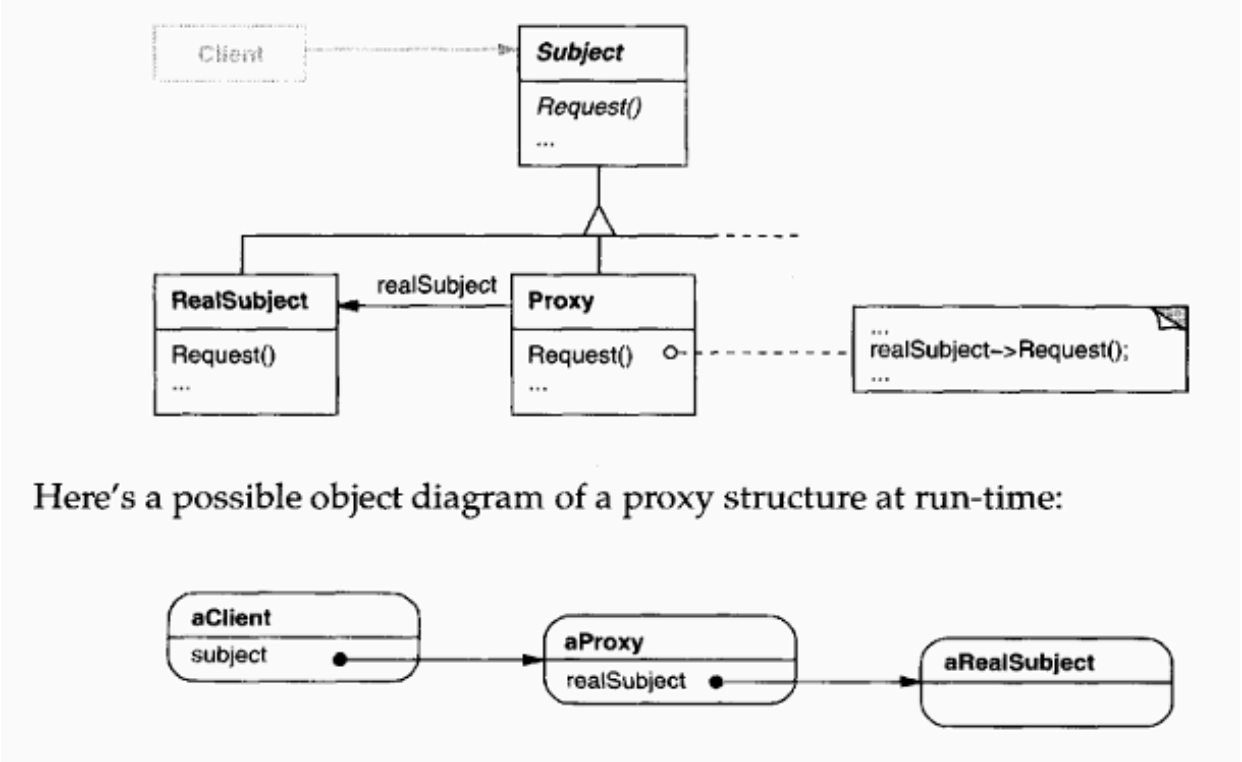
- Da más flexibilidad que la herencia estática. Esto se ve a la hora de añadir y eliminar responsabilidades.
- Evita la generación de clases cargadas de funciones en la parte superior de la jerarquía ya que este patrón posee un enfoque de pagar sólo por aquello que se necesita.
- No existe una identidad de Objetos, es decir, un Decorador y su Componente no son idénticos. Un Decorador se comporta como un envoltorio transparente para el Componente.
- Un diseño que use este patrón normalmente va a ser un diseño formado por muchos objetos pequeños parecidos. Esto hace que los diseños sean más difíciles de entender y depurar.

Proxy (Surrogate)

Propósito

- Proporciona un representante o sustituto de otro objeto para controlar el acceso a éste.

Estructura



Cuándo debería usarlo

- **Proxy Remoto**
    - Proporciona un representante local de un objeto situado en otro espacio de direcciones.

- **Proxy Virtual**
    - Crea objetos costosos por encargo.
- **Proxy de Protección**
    - Controla el acceso al objeto original. Estos proxys son útiles cuando los objetos deberían tener distintos permisos de acceso.
- **Referencia Inteligente**
    - Sustituto de un simple puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto.

Consecuencias de usarlo

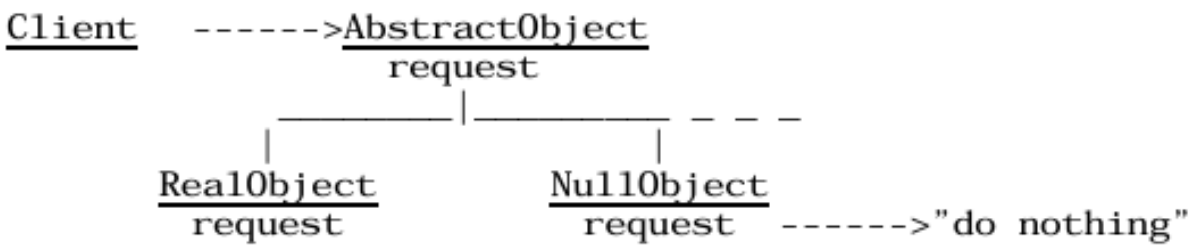
- El patrón introduce un nivel de indirección al acceder a un objeto. Esta indirección adicional tiene diferentes usos según el tipo de Proxy que se use:
    - **Proxy Remoto**
        - Puede ocultar el hecho de que un objeto reside en un espacio de direcciones diferente.
    - **Proxy Virtual**
        - Puede llevar a cabo optimizaciones tales como crear un objeto por encargo.
    - **Proxys de Protección y Referencias Inteligentes**
        - Permiten realizar tareas de mantenimiento.
- Permite ocultar al Cliente una optimización que se denomina copia-de-escritura, y que está relacionada con la creación por encargo. Copiar un objeto grande y complejo puede ser muy costoso. Si ese objeto no se modifica, no hace falta hacer ese gasto de copia. El Proxy permite posponer ese proceso de copia para hacerlo únicamente cuando sea modificado.

Null Object (Stub)

Propósito

- Proporciona un sustituto para otro objeto que comparta la misma interfaz pero que no haga nada. El objeto nulo encapsula las decisiones de implementación de cómo "no hacer nada" y oculta esos detalles de sus colaboradores.

Estructura



Cuándo debería usarlo

- Un objeto requiere un colaborador. El patrón de Objeto Nulo no introduce esta colaboración, sino que hace uso de una colaboración que ya existe.

- Algunas instancias de colaboradores deben hacer nada.
- Desea que los clientes puedan ignorar la diferencia entre un colaborador que proporciona un comportamiento real y uno que no hace nada.
- Desea poder reutilizar el comportamiento de no hacer nada para que varios clientes que necesiten este comportamiento funcionen de manera consistente.

Consecuencias de usarlo

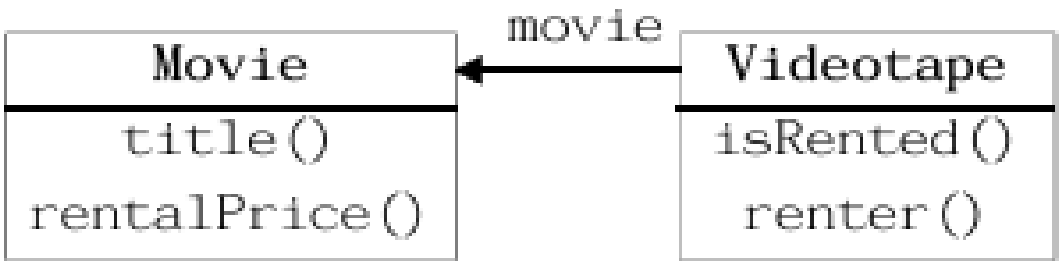
- Define jerarquías de clases que consisten en objetos reales y objetos nulos.
- Hace que el código del cliente sea simple. Los clientes pueden tratar a los colaboradores reales y a los colaboradores nulos de manera uniforme.
- Encapsula el código de no hacer nada en el objeto nulo. El código de no hacer nada es fácil de encontrar.
- Hace que el código de no hacer nada en el objeto nulo sea fácil de reutilizar.
- Hace que el comportamiento de no hacer nada sea difícil de distribuir o mezclar en el comportamiento real de varios objetos colaboradores.
- Siempre actúa como un objeto de no hacer nada. El Objeto Nulo no se transforma en un Objeto Real.

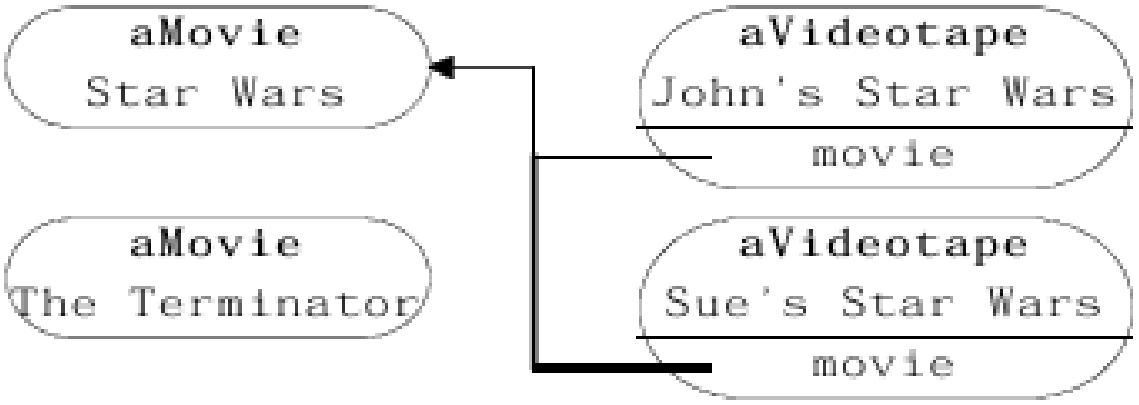
Type Object (Power Type, Item Descriptor, Metaobject)

Propósito

- Desacoplar instancias de sus clases para que esas clases puedan implementarse como instancias de una clase.
- Permite que se creen nuevas "clases" dinámicamente en tiempo de ejecución.
- Permite que un sistema proporcione sus propias reglas de verificación de tipos y puede llevar a sistemas más simples y pequeños.

Estructura





Cuándo debería usarlo

- Las instancias de una clase necesitan agruparse según sus atributos y/o comportamientos comunes.
- La clase necesita una subclase para cada grupo a fin de implementar los atributos/comportamientos comunes de ese grupo.
- La clase requiere una gran cantidad de subclases y/o la variedad total de subclases que pueda ser necesaria sea desconocida.
- Desee poder crear nuevos agrupamientos en tiempo de ejecución que no fueron previstos durante el diseño.
- Desee poder cambiar la subclase de un objeto después de haber sido instanciado sin tener que mutarlo a una nueva clase.
- Desee poder anidar agrupamientos recursivamente, de modo que un grupo sea en sí mismo un elemento de otro grupo.

Consecuencias de usarlo

- Creación de clases en tiempo de ejecución: El patrón permite la creación de nuevas "clases" en tiempo de ejecución. Estas nuevas clases no son realmente clases, sino instancias llamadas TypeObjects.
- Evita la explosión de subclases: El sistema ya no necesita numerosas subclases para representar diferentes tipos de objetos.
- Oculta la separación de instancia y tipo: Los clientes de un objeto no necesitan ser conscientes de la separación entre el objeto y su TypeObject.
- Cambio dinámico de tipo: El patrón permite que el objeto cambie dinámicamente su TypeObject, lo que tiene el efecto de cambiar su clase. Esto es más simple que mutar un objeto a una nueva clase.
- Subclasificación independiente: TypeClass y Class pueden ser subclasificadas independientemente.
- Múltiples TypeObjects: El patrón permite que un objeto tenga múltiples TypeObjects donde cada uno define una parte del tipo del objeto. El objeto debe entonces decidir qué comportamiento de tipo delegar a qué TypeObject.
- Complejidad del diseño: El patrón divide un objeto lógico en dos clases. Su relación, una cosa y su tipo, es difícil de entender.

- Complejidad de la implementación: El patrón mueve las diferencias de implementación fuera de las subclases y dentro del estado de las instancias de TypeObject.
- Gestión de referencias: Cada objeto debe mantener una referencia a su TypeObject. Así como un objeto sabe cuál es su clase, un objeto sabe cuál es su TypeObject.

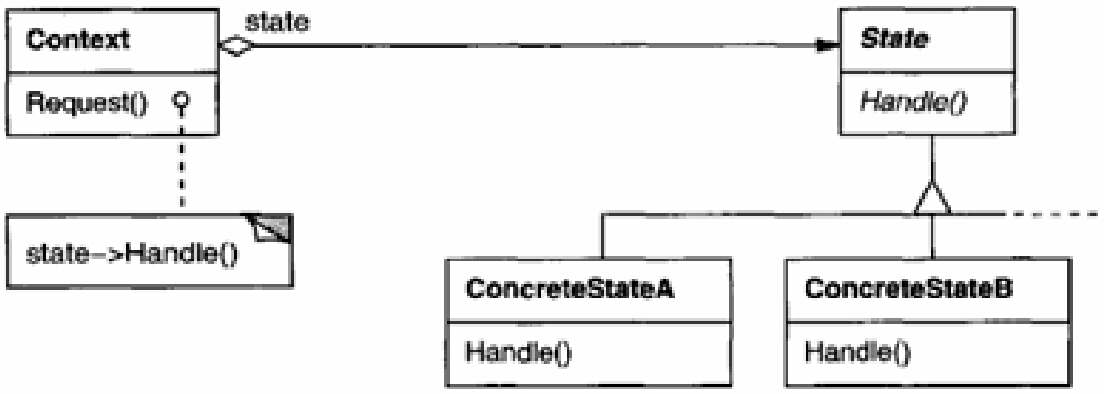
Patrones de Comportamiento

State (Objects for States)

Propósito

- Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.

Estructura



Cuándo debería usarlo

- El comportamiento de un objeto depende de su estado, y debe cambiar en tiempo de ejecución dependiendo de ese estado.
- Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. Muchas veces se ven representados los estados del objeto en esas estructuras condicionales (Descubrimiento de clases).

Consecuencias de usarlo

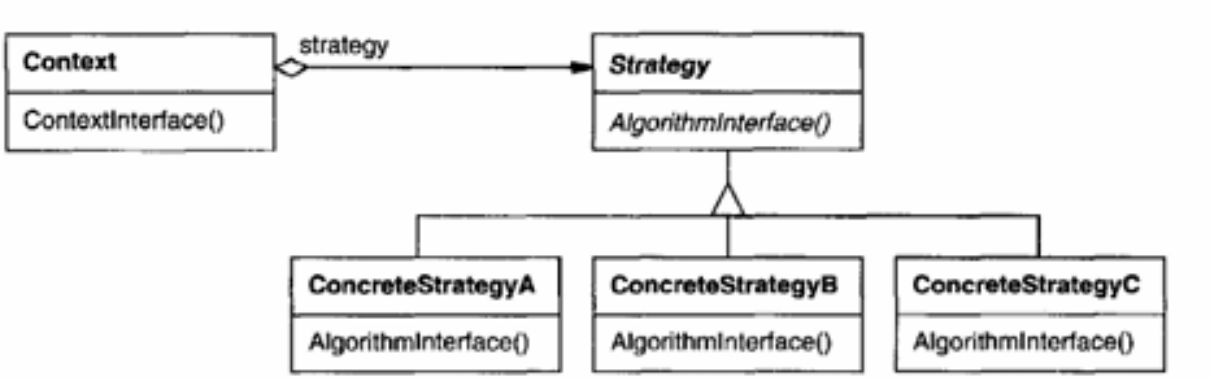
- Localiza el comportamiento dependiente del estado y divide dicho comportamiento en diferentes estados. Esto hace que incremente el número de clases y que el diseño sea menos compacto pero, esta distribución que ofrece el patrón es realmente buena para evitar grandes condicionales que hacen a nuestro código menos legible.
- Hace explícitas las transiciones entre estados que protegen al Contexto de estados internos inconsistentes.
- Los objetos Estado pueden compartirse, generando que no tengan estado intrínseco, sino que solo comportamiento.

Strategy (Policy)

Propósito

- Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables.
- Permite que un algoritmo varíe independientemente de los clientes que lo usan.

Estructura



Cuándo debería usarlo

- Muchas clases relacionadas difieren sólo en su comportamiento. Las estrategias permiten configurar una clase con un determinado comportamiento de entre muchos posibles.
- Se necesitan distintas variantes de un algoritmo.
- Un algoritmo usa datos que los clientes no deberían conocer. Se usa para evitar exponer estructuras de datos complejas y dependientes del algoritmo.
- Una clase define muchos comportamientos, y éstos se representan como múltiples sentencias condicionales en sus operaciones. Evitamos los condicionales representando las ramas como clases Estrategia.

Consecuencias de usarlo

- Las jerarquías de Estrategias definen una familia de algoritmos o comportamientos para ser reutilizados por los contextos. Esta herencia puede ayudar a sacar factor común de la funcionalidad de los algoritmos.
- Es una alternativa a la herencia que permite encapsular el algoritmo en clases Estrategias separadas de modo que podamos variar el algoritmo independientemente de su contexto, facilitando el cambio, la comprensión y la extensión.
- Las Estrategias eliminan las sentencias condicionales en cuánto al comportamiento que representan.
- Brindan al Cliente la capacidad de poder elegir entre Estrategias con diferentes soluciones para un comportamiento específico.

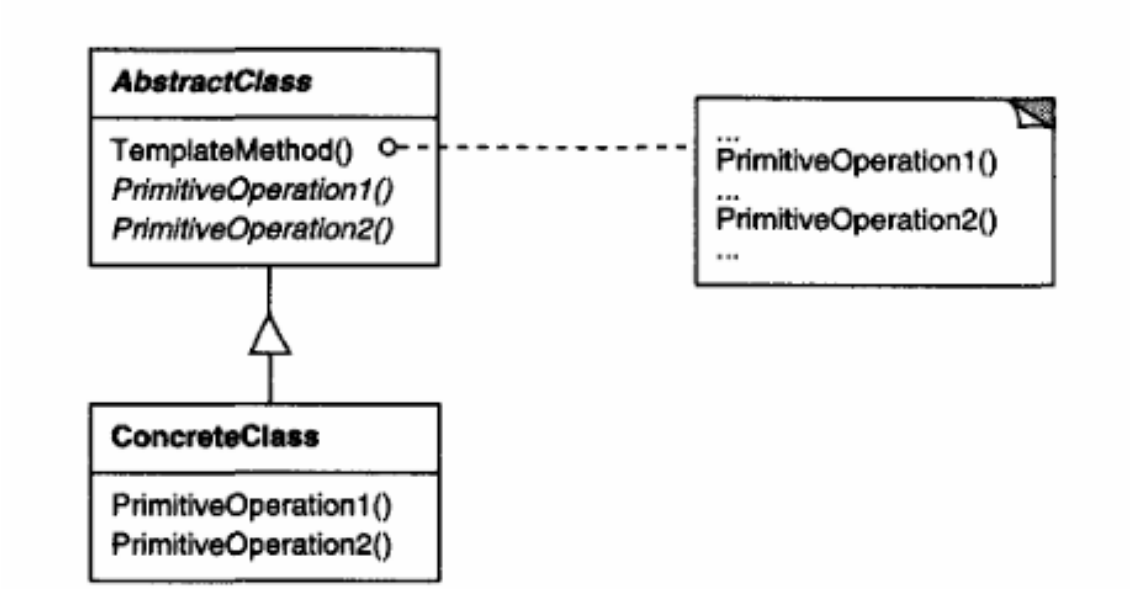
- Como inconveniente se ve necesario que el Cliente conozca las Estrategias y cómo difieren las mismas para poder seleccionar la más adecuada. Esto genera que los Clientes puedan estar expuestos a cuestiones de implementación.
- Se genera un coste de comunicación entre Estrategia y Contexto, esto se debe a que la interfaz de la Estrategia es una sola para todas las concretas, por lo tanto, es probable que algunas Estrategias Concretas no hagan uso de toda la información que reciben por parámetro de parte del Contexto.
- Aumenta el número de Objetos dentro de una aplicación. Se puede solucionar haciendo que los objetos actúen como estrategias compartidas.

Template Method

Propósito

- Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos.
- Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.

Estructura



Cuándo debería usarlo

- Se quiera implementar las partes de un algoritmo que no cambian y dejar que sean las subclases quienes implementen el comportamiento que puede variar.
- Cuando el comportamiento repetido de varias subclases debería factorizar y ser localizado en una clase común para evitar código duplicado.
- Para controlar las extensiones de las subclases.

Consecuencias de usarlo

- Generan reutilización del código y factorización del mismo.

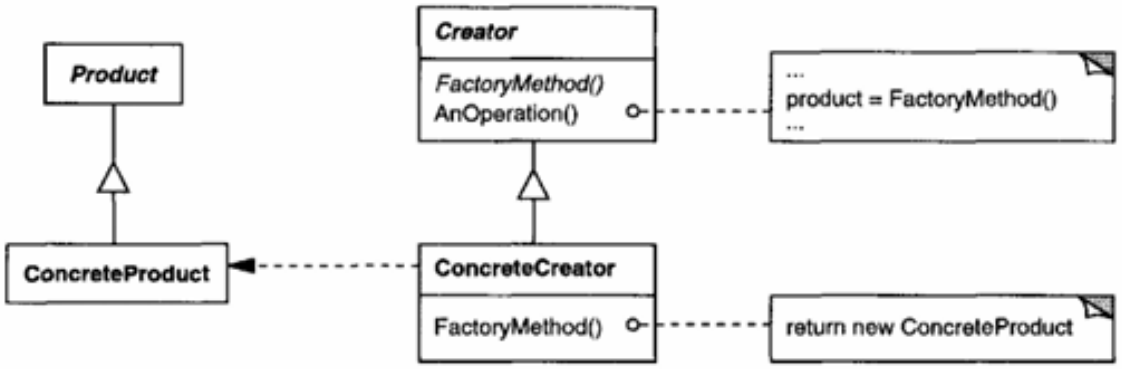
Patrones Creacionales

Factory Method (Virtual Constructor)

Propósito

- Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar.
- Permite que una clase delegue en sus subclases la creación de objetos.

Estructura



Cuándo debería usarlo

- Una clase no puede prever la clase de objetos que debe crear.
- Una clase quiere que sean sus subclases quienes especifiquen los objetos que ésta crea.
- Las clases delegan la responsabilidad en una de entre varias clases auxiliares, y queremos localizar qué subclase de auxiliar concreta es en la que se delega.

Consecuencias de usarlo

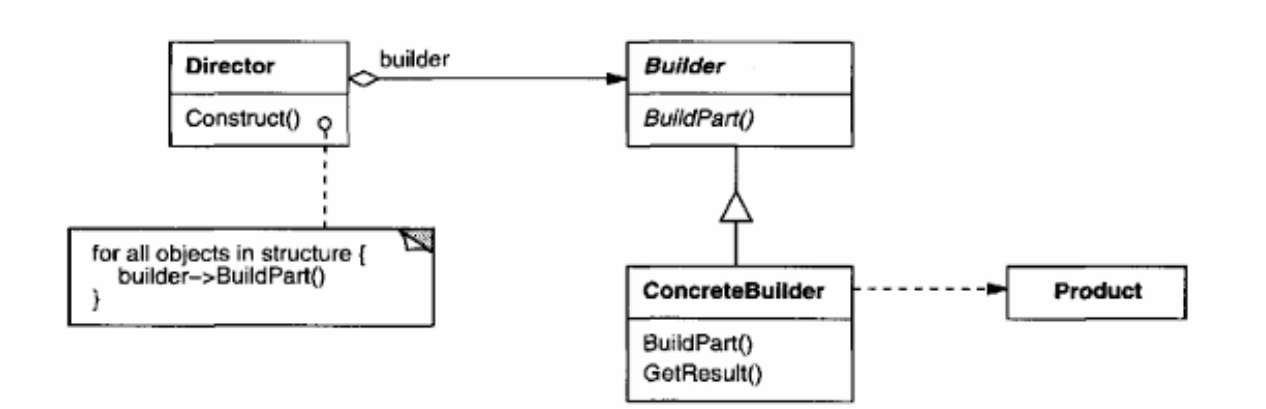
- Proporciona enganches para las subclases. Crear objetos dentro de una clase con un método de fabricación es siempre más flexible que hacerlo directamente.
- Conecta jerarquías de clases paralelas. Esto ocurre cuando una clase delega alguna de sus responsabilidades a una clase separada. Hay veces que el cliente puede encontrar útiles los métodos de construcción.

Builder

Propósito

- Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

Estructura



Cuándo debería usarlo

- El algoritmo para crear un objeto complejo debería ser independiente de las partes de que se compone dicho objeto y de cómo se ensamblan.
- El proceso de construcción debe permitir diferentes representaciones del objeto que está siendo construido.
- La creación de los objetos tiene una secuencia de pasos definida.

Consecuencias de usarlo

- Permite variar la representación interna de un producto. El Constructor proporciona al Director una interfaz abstracta para construir el Producto. Esta interfaz permite que el Constructor oculte la representación y la estructura interna del Producto, a su vez, también oculta el modo en que el Producto está ensamblado.
- Aísla el código de construcción y representación. El patrón aumenta la modularidad al encapsular cómo se construyen y se representan los objetos complejos.
- Proporciona un control más fino sobre el proceso de construcción al hacerlo paso a paso y bajo el control del Director.

Test Double

SUT y DOC

- **SUT: System Under Test - Sistema Bajo Prueba**

    - Es el componente del sistema que se está probando en un determinado caso de prueba. El SUT es la pieza de código que quieres verificar para asegurarse de que funciona correctamente.
- **DOC: Depended-On Component - Componente del que se depende)**
    - Se refiere a cualquier componente que el SUT necesita para funcionar correctamente. Durante una prueba, el SUT interactúa con estos DOCs para llevar a cabo sus operaciones.

Propósito

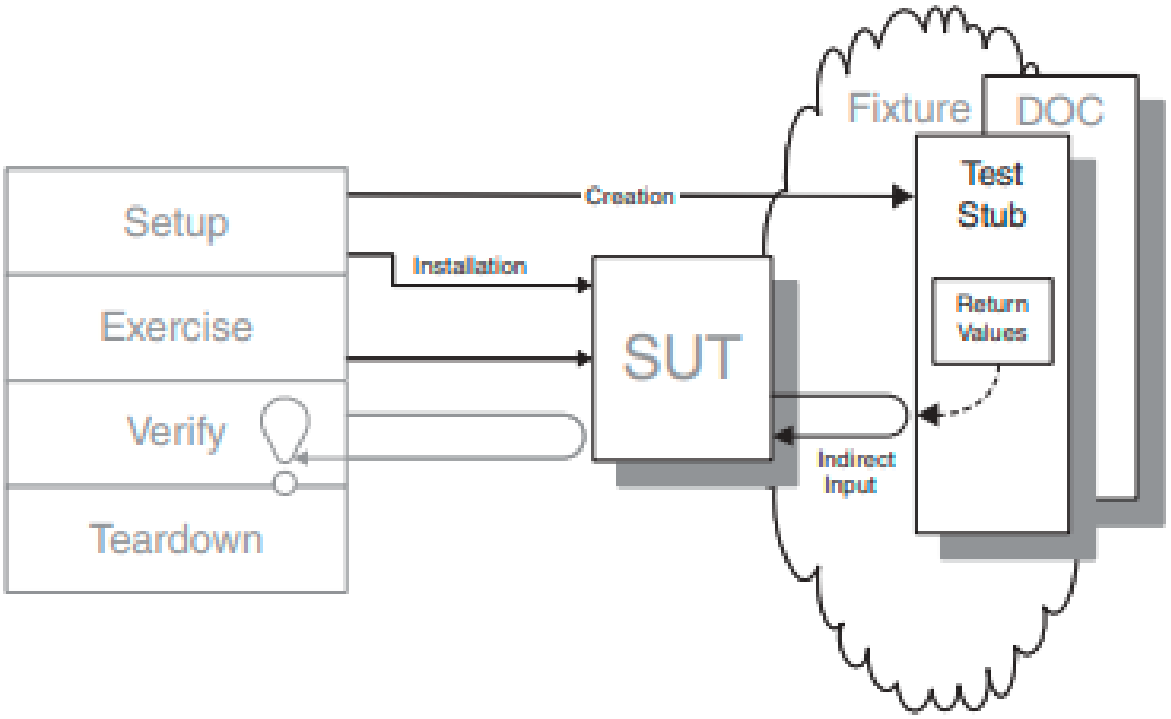
- Reemplazamos un componente del que depende el SUT con un "equivalente específico para pruebas".

Test Stub

Propósito

- Reemplazamos un objeto real con un objeto específico para pruebas que proporciona las entradas indirectas deseadas al sistema bajo prueba.
- Sirve para que el SUT envíe los mensajes esperados.

Estructura



Cuándo debería usarlo

- Una indicación clave para usar un Test Stub es tener código no probado debido a nuestra incapacidad para controlar las entradas indirectas del SUT.

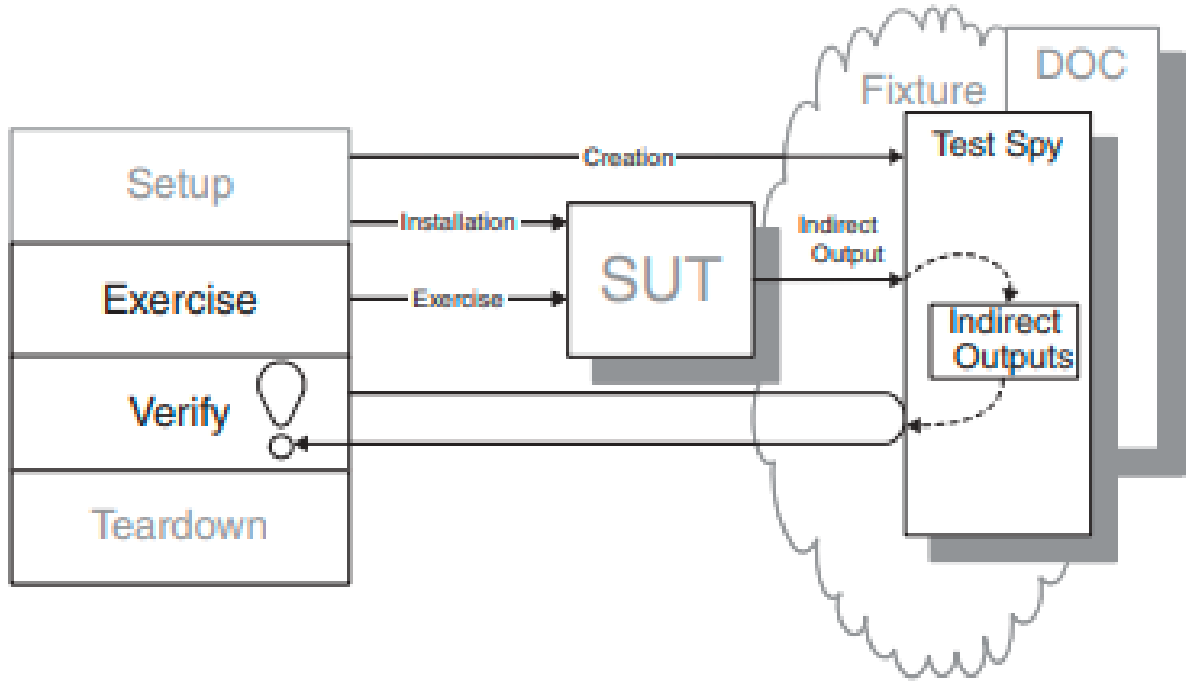
- También podemos usar un Test Stub para inyectar valores que nos permitan superar un punto particular en el software donde el SUT llama a un software que no está disponible en nuestro entorno de pruebas.

Test Spy

Propósito

- Usamos un Test Double para capturar las llamadas de salida indirectas realizadas a otro componente por el SUT para su verificación posterior por la prueba.
- Test Stub + registro de outputs.

Estructura



Cuándo debería usarlo

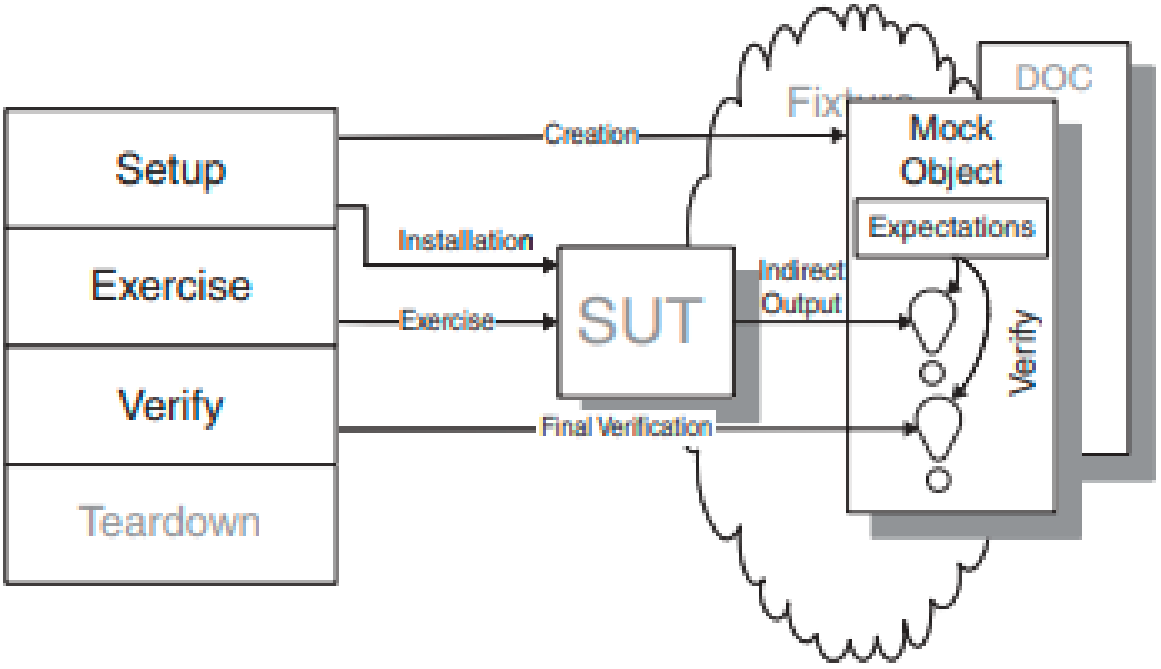
- Estamos verificando las salidas indirectas del SUT y no podemos predecir los valores de todos los atributos de las interacciones con el SUT con anticipación.
- Nos gustaría tener acceso a todas las llamadas salientes del SUT antes de hacer cualquier afirmación sobre ellas.

Mock Object

Propósito

- Reemplazamos un objeto del que depende el SUT con un objeto específico para pruebas que verifica que está siendo utilizado correctamente por el SUT.
- Test Stub + verification of outputs.

Estructura



Cuándo debería usarlo

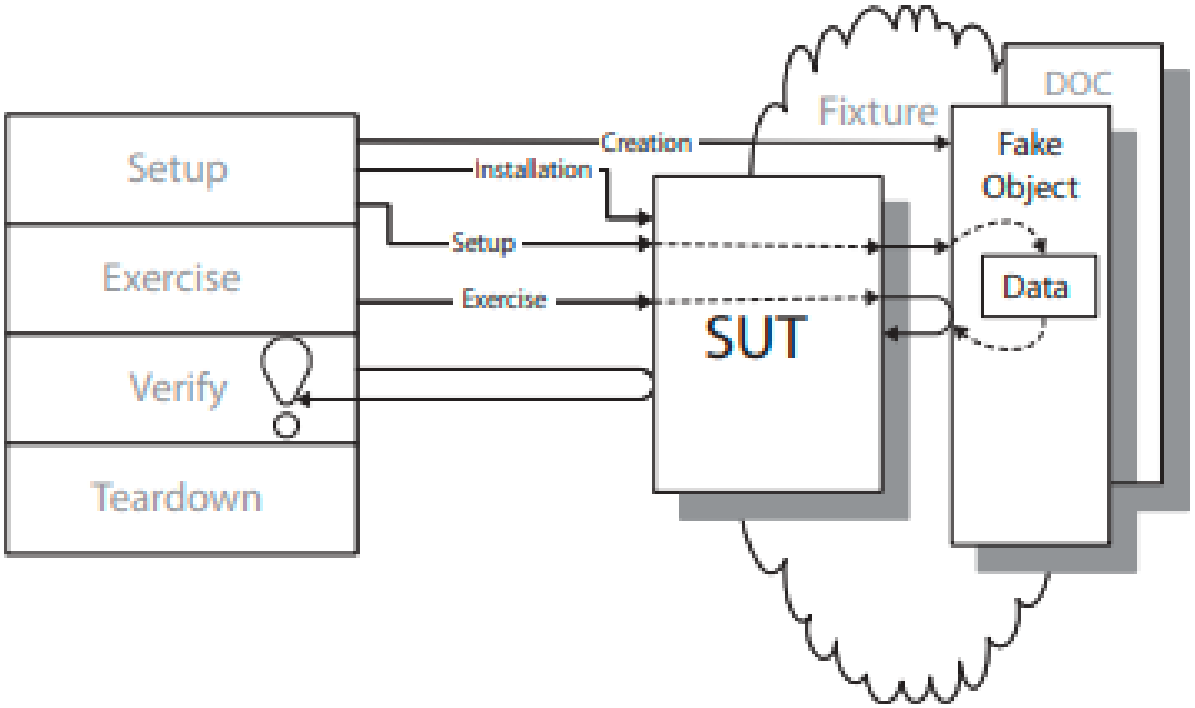
- Podemos usar un Mock Object como un punto de observación cuando necesitamos hacer Verificación de Comportamiento para evitar tener un Requisito No Probado causado por nuestra incapacidad para observar los efectos secundarios de invocar métodos en el SUT.
- Para usar un Mock Object, debemos ser capaces de predecir los valores de la mayoría o todos los argumentos de las llamadas a métodos antes de ejercitar el SUT. No deberíamos usar un Mock Object si una afirmación fallida no puede ser reportada efectivamente al Test Runner.

Fake Object

Propósito

- Reemplazamos un componente del que depende el SUT con una implementación mucho más liviana.
- imitación. Se comporta como el módulo real.

Estructura



Cuándo debería usarlo

- Deberíamos usar un Fake Object siempre que el SUT dependa de otros componentes que no estén disponibles o que dificulten las pruebas y estas requieran secuencias de comportamiento más complejas.

Refactoring

- **Es el proceso a través del cual se cambia un sistema de software**
    - Para mejorar la organización, legibilidad, adaptabilidad y mantenibilidad del código luego que ha sido escrito.
    - Que NO altera el comportamiento externo del sistema pero SI mejora su diseño.

Leyes de Lehman

- **Cambio Continuo**
    - Los sistemas deben adaptarse continuamente o se vuelven progresivamente menos satisfactorios.
- **Crecimiento Continuo**
    - La funcionalidad de un sistema debe ser incrementada continuamente para mantener la satisfacción del cliente.
- **Complejidad Creciente**
    - A medida que un sistema evoluciona su complejidad se incrementa a menos que se trabaje para evitarlo.
- **Calidad Decreciente**

    - La calidad de un sistema va a ir declinando a menos que se haga un mantenimiento riguroso.

Refactoring (Sustantivo)

- Es un cambio hecho a la estructura interna del software para hacerla más fácil de entender y más barata de modificar sin cambiar su comportamiento observable.
- Tienen un **nombre específico** y una **secuencia de pasos ordenados** “mechanics”.
- **Cada uno de los cambios catalogados.**

Refactor (Verbo)

- Proceso de aplicar refactorings para modificar la estructura del software sin cambiar el comportamiento observable.

Características

- **El Refactoring implica**
    - Eliminar duplicaciones.
    - Simplificar lógicas complejas.
    - Clarificar códigos.
- **Debe ser aplicado**
    - Una vez que tengo código que funciona y pasa los tests.
    - A medida que voy desarrollando si me encuentro con código poco legible o necesito reorganizar para hacer un cambio.
- **Testear después de cada cambio.**

¿Cómo ayuda?

- Introduce mecanismos que solucionan problemas de diseño a través de muchos cambios pequeños que son fáciles y seguros de aplicar y que además, ponen en evidencia otros cambios necesarios.

Automatización del Refactoring

- Existen herramientas que hacen menos costoso el proceso del refactoring.
- **Características**
    - Son potentes para realizar refactorings útiles.
    - Restrictivas para preservar el comportamiento del programa.
    - Interactivas, de manera que el chequeo de precondiciones no debe ser extenso.
    - Solo chequean lo que sea posible desde el árbol de sintaxis y la tabla de símbolos.

Deuda Técnica

- Concepto que permite visualizar las consecuencias de un diseño rápido y sucio.
- **Capital de la deuda**

    - Costo de remediar los problemas de diseño, es decir, costo del refactoring.
- **Interés de la deuda**
    - Costo adicional en el futuro por preservar la deuda técnica en el software.

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 - Code Smells

- Indicios de problemas que requieren la aplicación de refactorings.

Large Class - Bloaters

- Una clase posee una lista larga de variables de instancia y/o métodos y puede estar intentando realizar demasiado trabajo.
- **Problemas**
    - Indica un problema de diseño (baja cohesión).
    - Algunos métodos pueden pertenecer a otra clase.
    - Generalmente tiene código duplicado.
- **Usar**
    - Extract Class.
    - Extract Subclass.

Long Method - Bloaters

- Un método tiene muchas líneas de código (como límite 20 pero depende del lenguaje).
- **Problemas**
    - Cuanto más largo es un método, más difícil es entenderlo, cambiarlo y reusarlo.
- **Usar**
    - Extract Method.
    - Decompose Conditional.
    - Replace Temp with Query.

Long Parameter List - Bloaters

- Si un método posee una larga lista de parámetros es más difícil de entender y de rehusar. La excepción es cuando no quiero crear una dependencia entre el objeto llamador y el llamado.
- **Usar**
    - Replace Parameter with Method.
    - Preserve Whole Object.
    - Introduce Parameter Object.

Feature Envy - Couplers

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

- **Problemas**
    - Indica un problema de diseño.
    - Idealmente se prefiere que los datos y las acciones sobre los datos vivan en la misma clase.
    - Indica que el método fue ubicado en la clase incorrecta.
- **Usar**
    - Move Method.

Data Class - Dispensables

- Una clase que actúa como contenedor de datos que solo tiene variables y getters/setters para esas variables.
- Problemas
    - En general sucede que otras clases tienen métodos con “Feature Envy”.
    - Esto indica que esos métodos deberían estar en la “Data Class”.
    - Suele indicar que el diseño es procedural.
- **Usar**
    - Move Method.

Duplicated Code - Dispensables

- El mismo código, o código muy similar, aparece en muchos lugares.
- **Problemas**
    - Hace el código más largo de lo que necesita ser.
    - Es difícil de cambiar, difícil de mantener.
    - Un bug fix en un clone no es fácilmente propagado a los demás clones.
- **Usar**
    - Extract Method.
    - Pull Up Method.
    - Form Template Method.

Conditionals - Object Orientation Abusers

- 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.
- **Problema**
    - El mismo condicional aparece en muchos lugares.
- **Usar**
    - Replace Conditional with Polymorphism.

Refactoring to Patterns

Sobre-Ingeniería

- **Proceso que se realiza para**
    - **Acomodar futuros cambios** → no podemos predecir el futuro.

    - **No quedar inmerso y acarrear un mal diseño** → a la larga el encanto de los patrones puede hacer que no usemos soluciones más simples.
- **Consecuencias**
    - Se genera código complejo que perdura y complica el mantenimiento, este código nadie lo entiende ni lo quiere modificar lo que conlleva a que se generen copias del mismo causando Duplicated Code.

Form Template Method

- **Motivación**
    - Dos o más métodos en subclases realizan pasos similares en el mismo orden, pero los pasos son distintos.
- **Solución**
    - Generalizar los métodos extrayendo sus pasos en métodos de la misma signatura, y luego subir a la superclase común el método generalizado para formar un Template Method.
- **Pros**
    - Elimina código duplicado en las subclases al mover el comportamiento invariable a la superclase.
    - Simplifica y comunica efectivamente los pasos de un algoritmo genérico.
    - Permite que las subclases adapten fácilmente un algoritmo.
- **Contra**
    - Complica el diseño cuando las subclases deben implementar muchos métodos para sustanciar el algoritmo.

Replace Conditional Logic with Strategy

- **Motivación**
    - Existe lógica condicional en un método que controla qué variante ejecutar entre distintas posibles.
- **Solución**
    - Crear un Strategy para cada variante y hacer que el método original delegue el cálculo a la instancia de Strategy.

Replace State-Altering Conditionals with State

- **Motivación**
    - Las expresiones condicionales que controlan las transiciones de estado de un objeto son complejas.
    - Obtener una mejor visualización de los cambios de estados.
    - La lógica condicional entre estados ya no es fácil de extender.
    - La aplicación de refactorings más simples no alcanza.
- **Solución**
    - Reemplazar los condicionales con States que manejen estados específicos y transiciones entre ellos.

Move Embellishment to Decorator

Introduce Null Object

- **Motivación**
    - La lógica para manejarse con un valor nulo en una variable está duplicado por todo el código.
- **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 que haga pasar el test.
- Refactoring.

Objetivo

- Pensar el diseño y qué se espera de cada requerimiento antes de escribir código.
- Escribir código limpio que funcione.

Filosofía

- Se escriben primero los test y luego el código.
- Se escriben test funcionales que capturen casos de uso.
- Se escriben test de unidad para enfocarse en pequeñas partes y aislar errores.
- No agregar funcionalidad hasta que no haya un test que no pasa porque esa funcionalidad no existe.
- Una vez escrito el test, se codifica lo necesario para que todo el test pase.
- Pequeños pasos → un test, un poco de código.
- Una vez que los test pasan se aplica refactoring.

Reglas de TDD

Diseñar incrementalmente

- Teniendo código que funciona como feedback para ayudar en las decisiones entre iteraciones.

Los programadores escriben sus propios tests

- No es efectivo tener que esperar a otro que los escriba por ellos.

El diseño debe consistir de componentes altamente cohesivos y desacoplados entre sí

- Mejora 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

Fragile Test - Test Smell

- Este Smell proviene de un test que falla en compilar o ejecutar cuando el SUT cambia en formas que no afectan la parte que el test ejercita.
- Una de las causas de este smell es la “sensibilidad a la interface” o “sensibilidad al protocolo” del SUT.

Refactorings para Fragile Test

Test Utility Method

- Permite encapsular la dependencia innecesaria con la API del SUT en un método, por ejemplo, de creación.
- Esto es innecesario cuando se refiere a un método del SUT que no es el que se está testeando.

Creation Method

- Es un tipo de Test Utility Method que oculta la mecánica de crear objetos ready-yo-use atrás de un método con nombre adecuado.

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

Reúso

¿Qué es lo que reusamos?

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

¿ Por qué reusamos?

- Para aumentar la productividad, reduciendo costos de desarrollo y mantenimientos, tiempos de entrega y puesta en el mercado. Agilizando el mercado y la customización en masa.
- Para aumentar la calidad, aprovechando mejoras, correcciones y el conocimiento de los especialistas.

¿Qué dificultades tenemos?

- Problemas de mantenimiento si no tenemos control de los componentes que reusamos.
- Algunos programadores prefieren hacer todo ellos mismos.
- Hacer 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 en la librería resuelve un problema concreto, es independiente del contexto de uso, no espera nada de nuestro código y generalmente independiente de otras clases en la librería.
- Nuestro código controla a los objetos de la librería.

Framework

- Aplicación semi completa, reusable, que puede ser especializada para producir aplicaciones a medida.
- También se puede ver como un conjunto de clases concretas y abstractas, relacionadas para proveer una arquitectura reusable para una familia de aplicaciones relacionadas.
- Los desarrolladores incorporan el framework en sus aplicaciones y lo especializan para cubrir sus necesidades.
- El Framework define el esqueleto, y el desarrollador define sus propias características para completar el esqueleto.

Frameworks de Infraestructura

- Estos frameworks ofrecen una infraestructura portable y eficiente sobre la cual construir una gran variedad de aplicaciones.
- Algunos de los focos de este tipo de frameworks son: interfaces de usuario (desktop, web, móviles), seguridad, contenedores de aplicación, procesamiento de imágenes, procesamiento de lenguaje, comunicaciones.
- Atacan problemas generales del desarrollo de software, que por lo general no son percibidos completamente por el usuario de la aplicación (es decir, resuelven problemas de los programadores, no de los usuarios).

Frameworks de Integración

- Estos frameworks se utilizan comúnmente para integrar componentes de aplicación distribuidos (p.e., la base de datos con la aplicación y ésta con su cliente liviano).
- Los frameworks de integración (o frameworks middleware) están diseñados para aumentar la capacidad de los desarrolladores de modularizar, reusar y extender su infraestructura de software para que trabaje transparentemente en un ambiente distribuido.
- El mercado de los frameworks de integración es muy activo y, al igual que ciertos frameworks de infraestructura, se han vuelto commodities (mercancía de uso común).

Frameworks de Aplicación

- Estos frameworks atacan dominios de aplicación amplios que son pilares fundamentales de las actividades de las empresas.
- Ejemplos de frameworks enterprise son aquellos en el dominio de los ERP (Enterprise Resource Planning), CRM (Customer Relationship Management) Gestión de documentos, Cálculos financieros.
- Los problemas que atacan derivan directamente de las necesidades de los usuarios de las aplicaciones y por tanto hacen que el retorno de la inversión en su desarrollo/adquisición sea más evidente y justificado.
- Un framework enterprise puede encapsular el conocimiento y experiencia de muchos años de una empresa, transformándose en la clave de su ventaja competitiva y su más preciado capital.

Frozenspots

- Aspectos del Framework que no se pueden cambiar, si estos se ven modificados será porque se aplicó Hacking.
- Se ven en los requerimientos.

Hotspots

- Puntos de extensión que nos permiten introducir variantes y así construir aplicaciones diferentes.
- Se ven en los requerimientos.

Caja blanca vs Caja negra

- Los puntos de extensión pueden implementarse en base a herencia o en base a composición.
- A los frameworks que utilizan herencia en sus puntos de extensión, les llamamos de Caja Blanca. Estos son más fáciles de desarrollar pero más difíciles de usar.
- A los que utilizan composición les llamamos de Caja Negra. Estos son más fáciles de usar pero más difíciles de desarrollar.
- La mayoría de los frameworks están en algún lugar en el medio.

Inversión de Control

- La inversión de control se da cuando el código del framework utiliza el código que nosotros escribimos, por lo tanto ahora el control lo posee el Framework en vez de mi programa.
- Esta característica 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

- Las plantillas y los ganchos son una estrategia de programación comúnmente utilizada para introducir puntos de variabilidad (hotspots) en los frameworks orientados a objetos.
- Las plantillas implementan lo que se mantiene constante, los ganchos lo que varía.
- Se puede utilizar herencia o composición para separar la plantilla de los ganchos.
- Si uso herencia, la plantilla se implementa en una clase abstracta y los ganchos en sus subclases. Esto sirve cuando hay pocas alternativas y/o combinaciones, al implementar los métodos ganchos puedo utilizar las variables de instancia y todo el comportamiento heredado y si hay muchas variantes y/o combinaciones, aumenta la cantidad de clases y duplicación de código.
- Si uso composición, un objeto implementa la plantilla y delega la implementación de los ganchos en sus partes. Evita la duplicación de código y el creciente número de clases cuando hay muchas alternativas y/o combinaciones, requiere el pasaje por parámetro de todo lo necesario para implementar los métodos ganchos ya que no hay acceso a las variables de instancia y permite cambiar el comportamiento en tiempo de ejecución sin mayor dificultad.
- En un framework, las plantillas suelen estar bajo control del desarrollador del framework (y las usa para implementar el frozenspot) que delega en el usuario la implementación/instanciación de 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.