

Resumen Objetos 2

BAD SMELLS	
Envidia de atributos (Feature Envy)	Soy un objeto que pide cosas a otros objetos para hacer algo yo mismo (por ejemplo un cálculo)
Clase Dios	Una clase que resuelve todo y las demás están todas anémicas. Probablemente también haya envidia de atributos.
Clase larga (Long Class)	Tengo una clase muy grande en comparación al resto.
Código duplicado (Duplicated Code)	Código repetido en varias secciones de mi código (en diferentes métodos).
Método Largo (Long Method)	Tengo un método muy grande en comparación al resto (muchas líneas y muchas responsabilidades).
Objetos que conocen el id de otro	Nunca relacionar objetos por medio de claves o ids.
Obsesión por los primitivos	Cuando se modela como tipo de dato primitivo (ej: String, Integer, Double) cosas que deberían ser objetos.
Switch statements	Cuando uso un case o un switch o ifs anidados para determinar de qué forma se resuelve algo.
Reinventa la rueda	Cuando defino comportamiento que ya está programado en algún lado. Ejemplo: implementar un for cosas que las colecciones, los iteradores y los streams ya saben hacer.
Variables de instancia que deberían ser temporales (Temporary Field)	Cuando tengo una variable de instancia temporal que podría evitarse.
Rompe el encapsulamiento	Cuando se acceden a variables de instancia de otros objetos. Ej: agregar setters y getters para todas las variables de instancia, estamos invitando a otros objetos a que las modifiquen cuanto quieran (como si no existiera un ocultamiento de información). Si modificamos una colección que no es nuestra (es de otro objeto) también atentamos contra el encapsulamiento.
Clase de datos o clase anémica	Una clase que parece un registro de datos (solo tiene datos y no tiene comportamiento).
No quiero mi herencia	Cuando encontramos un método que redefine a uno heredado pero hace algo totalmente diferente.
No es-un	Cuando no se respeta el principio de es-un en una relación de herencia (clase B hereda de clase A: B es un A).
Middle Man	Cuando una clase solo actúa de intermediario entre otras clases.
Nombres poco explicativos	Cuando una clase/método/parámetro/variable tiene un nombre poco explicativo y se necesitan muchos comentarios para que se entienda.

Resumen Objetos 2

En el caso de encontrar Bad Smells, deben aplicarse distintos Refactorings para eliminarlos y mejorar el código

REFACTORING	
Rename Class/Method/Parameter	Se da nombre más explicativo a clase/método/parámetro
Encapsulate Field	Variable de instancia pasa de pública a privada
Remove Middle Man	Se elimina clase intermediaria e innecesaria.
Move Method	Se mueve método a otra clase.
Move Field	Se mueve variable de instancia a otra clase.
Extract Method	Se extra sección de código presente en un método a un nuevo método.
Replace temp with Query	Se elimina variable temporal innecesaria.
Replace Loop with Pipeline	Se reemplaza un bloque loop (ej: for) por iteradores ya existentes (stream, collectors, etc).
Replace Conditional with Polymorphism	Se reemplazan condicionales (if anidados, case, switch) creando una jerarquía de clases y haciendo uso del polimorfismo.
Replace Conditional with Strategy	Se reemplazan condicionales (if anidados, case, switch) creando una jerarquía de clases siguiendo el patrón Strategy.
Replace State-Altering Conditionals with State	Se reemplazan condicionales (if anidados, case, switch) creando una jerarquía de clases siguiendo el patrón State.
Extract Superclass	Se crea clase Padre que servirá para que otras clases la implementen (variables y comportamiento)
Extract Subclass	Se crea Hija que implementará otras clases abstractas/interfaces.
Extract Interface	Se crea clase Interfaz que servirá para que otras clases la implementen.
Pull Up Field	Se mueve variable común a Clase Padre para que la hereden las subclases.
Pull Up Method	Se mueve método común a Clase Padre para que la hereden las subclases.
Replace Data Value with Object	Se reemplaza variable de instancia de tipo primitivo por un Objeto.
Introduce Null Object	Se crea clase Null Object para los casos donde un objeto es null o no tiene comportamiento. Se crea Interfaz con 2 clases que la implementan: RealObject y NullObject Se elimina código repetido en RealObject al no tener que consultar si el objeto != null.

Resumen Objetos 2

Relación entre Bad Smells y Refactorings

BAD SMELLS	REFACTORING
Envidia de atributos (Feature Envy)	Move Method Move Field
Clase Dios	Extract Superclass
Clase larga (Long Class)	Extract Subclass Extract Interface
Código duplicado (Duplicated Code)	Extract Superclass Pull Up field Pull Up Method Introduce Null Object
Método Largo (Long Method)	Extract Method
Objetos que conocen el id de otro	Se utilizan referencias para relacionarse.
Obsesión por los primitivos	Replace Data Value with Object
Switch statements	Replace Loop with Pipeline
Reinventa la rueda	Replace Conditional with Polymorphism Replace Conditional with Strategy Replace State-Altering Conditionals with State
Variables de instancia que deberían ser temporales (Temporary Field)	Replace temp with Query
Rompe el encapsulamiento	Encapsulate Field
Clase de datos o clase anémica	Move Field
No quiero mi herencia	Se reorganiza/elimina jerarquía.
No es-un	
Middle Man	Remove Middle Man
Nombres poco explicativos	Rename Class/Method/Parameter

Resumen Objetos 2

Patrones vistos en la materia.

PATRONES	APLICABILIDAD
Adapter	Cuando se quiere usar una clase existente y su interfaz no es compatible con lo que precisa
Template Method	<ul style="list-style-type: none"> - Para implementar las partes invariantes de un algoritmo una vez y dejar que las subclases implementen los aspectos que varian - Para evitar duplicación de código entre subclases - Para controlar las extensiones que pueden hacer las subclases
Strategy	Cuando se necesitan diferentes algoritmos opcionales para realizar una misma tarea
State	Cuando el comportamiento de un objeto depende del estado en el que se encuentre
Composite	<ul style="list-style-type: none"> - Para representar jerarquías parte-todo de objetos. - Para que los objetos “clientes” puedan ignorar las diferencias entre composiciones y objetos individuales. Los clientes tratarán a los objetos atómicos y compuestos uniformemente
Builder	Para separar la construcción de un objeto complejo de su representación (implementación) de tal manera que el mismo proceso puede construir diferentes representaciones (implementaciones)
Decorator	<ul style="list-style-type: none"> - Para agregar responsabilidades a objetos individualmente y en forma transparente (sin afectar otros objetos) - Para quitar responsabilidades dinámicamente - Para cuando subclásificar es impráctico
Proxy	<p>Virtual proxy: para demorar la construcción de un objeto hasta que sea realmente necesario, cuando sea poco eficiente acceder al objeto real.</p> <p>Protection proxy: para restringir el acceso a un objeto por seguridad.</p> <p>Remote proxy: para representar un objeto remoto en el espacio de memoria local. Es la forma de implementar objetos distribuidos. Se ocupan de la comunicación con el objeto remoto, y de serializar/deserializar los mensajes y resultados.</p>
Null Object	<p>Cuando un objeto tiene un colaborador, que algunas veces no hace nada y:</p> <ul style="list-style-type: none"> - queremos que el objeto cliente pueda ignorar la diferencia del colaborador que hace algo con el que no hace nada - queremos reusar el comportamiento de hacer nada

Resumen Objetos 2

Adapter

Propósito/ Intención	Permite que cooperen clases que de otra forma no podrían por tener interfaces incompatibles Convierte la interfaz de una clase en otra interfaz esperada por el cliente
Aplicabilidad	Cuando se quiere usar una clase y su interfaz no matchea con la interfaz que yo necesito Cuando se quiere crear una clase reusable que coopere con clases no relacionadas o imprevistas (clases que no necesariamente tienen interfaces compatibles)
Estructura	Opción 1: La clase Adapter usa multiples herencias para adaptar una interfaz a otra Opción 2: El objeto Adapter depende de la composición de objetos

Opción 1		Opción 2
A class adapter uses multiple inheritance to adapt one interface to another:		An object adapter relies on object composition:
<pre> classDiagram Client --> Target : Request() Target < -- Adapter Target --> Adaptee : SpecificRequest() Adaptee "Implementation" --> Adapter Adapter --> Client : Request() Adapter --> Adaptee : SpecificRequest() </pre>		<pre> classDiagram Client --> Target : Request() Target < -- Adapter Target --> Adaptee : SpecificRequest() Adaptee "adaptee" --> Adapter Adapter --> Client : Request() Adapter --> Adaptee : SpecificRequest() </pre>
Target		Define la interfaz del dominio que usa el Cliente
Client		Colabora con objetos que se ajustan (implementan) a la interfaz Target
Adaptee		Define una interfaz existente que necesita ser adaptada. Client no puede utilizar directamente esta clase porque tiene una interfaz incompatible
Adapter		Adapta la interfaz de Adaptee a la interfaz Target para que Client la pueda usar

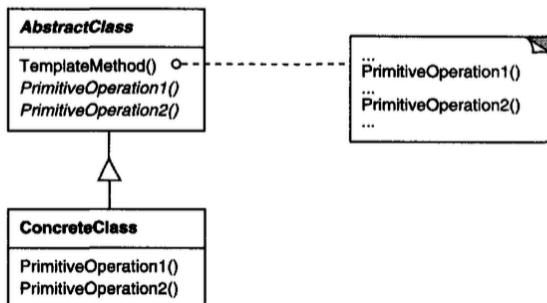
Implementación	
Como Adapter es subclase de Target, Adapter podrá reimplementar los métodos de su clase padre y a su vez la clase Adapter será también “una clase Target” (Adapter extiende a Target)	
Lo importante de esto es que Client podrá usar a los métodos de Target instanciando a Adapter, como se ve a continuación (Editor podrá instanciar a 3dAdapter como si fuera una figura cualquiera)	

Resumen Objetos 2

Template Method

Propósito/ Intención	<ul style="list-style-type: none"> - 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
Aplicabilidad	<ul style="list-style-type: none"> - Para implementar las partes invariantes de un algoritmo una vez y dejar que las subclases implementen los aspectos que varian - Para evitar duplicación de código entre subclases - Para controlar las extensiones que pueden hacer las subclases
Estructura	<p>El template method lleva a una estructura de control invertida, en la cual, la clase padre llama a las operaciones de la subclase y no de la otra manera</p> <p>Operaciones concretas (en la clase concreta o en la clase cliente)</p> <p>Operaciones abstractas concretas (operaciones que deben implementar las subclases)</p> <p>Operaciones hook, las cuales proveen un comportamiento default que las subclases extienden si es necesario. Un hook generalmente no hace nada por default</p>

Structure



Implementación

La clase abstracta que define el esqueleto podría también asignar un comportamiento “default” a las operaciones primitivas

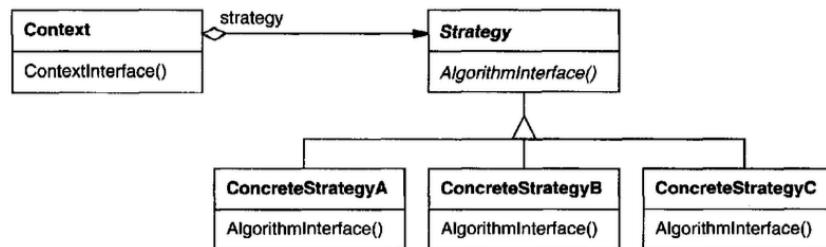
¿Qué pasará cuando a **ConcreteClass** le llegue un mensaje **TemplateMethod**?
 Como **ConcreteClass** no posee dicho método, lo buscara en la superclase (Method lookup)
 Sin embargo, cuando dentro de **TemplateMethod** se llame con **this** a una de las operaciones primitivas, lo que se ejecutará es la operación primitiva de **ConcreteClass** en caso de estar implementada.

Resumen Objetos 2

Strategy

Propósito/ Intención	Permite cambiar el algoritmo que un objeto utiliza en forma dinámica <ul style="list-style-type: none"> - Define una familia de algoritmos, encapsula cada uno y los vuelve intercambiables. - El Strategy permite que el algoritmo varíe independientemente de los clientes que lo usan. - Los clientes pueden cambiar el algoritmo que están utilizando sin necesidad de modificar su propio código
Aplicabilidad	<ul style="list-style-type: none"> - Muchas clases relacionadas difieren solo en el comportamiento. Strategy provee una manera de configurar una clase con uno de muchos comportamientos - Se necesitan distintas variantes de un algoritmo / Existen muchos algoritmos para llevar a cabo una tarea <ul style="list-style-type: none"> - Un algoritmo usa información que el cliente no debería conocer / Cada algoritmo utiliza información propia - Una clase define muchos comportamientos y estos aparecen como múltiples sentencias condicionales en sus operaciones.
Estructura	<p style="text-align: center;">Context Strategy (Abstract Class / Interface) ConcreteStrategy</p>

Structure



Implementación

La clase Strategy puede ser interfaz o clase abstracta

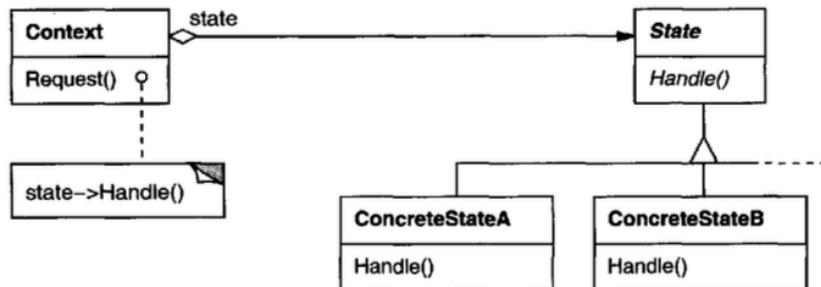
En general, el cliente CREA una estrategia en concreto, la instancia, y se la manda al contexto. Pero cliente no conoce en sí a la estrategia, no hay una relación (Entiendo que no lo tiene como variable de instancia, una cosa así será)

Resumen Objetos 2

State

Propósito/ Intención	<ul style="list-style-type: none"> - Cuando el comportamiento de un objeto depende del estado en el que se encuentre - Genera acoplamiento y puede terminar exponiendo el estado interno de una clase - Modificar el comportamiento de un objeto cuando su estado interno se modifica - Externamente parecería que la clase del objeto ha cambiado
Aplicabilidad	<ul style="list-style-type: none"> - El comportamiento de un objeto depende de su estado y debe cambiar su comportamiento en tiempo de ejecución dependiendo de su estado - Los métodos tienen sentencias condicionales complejas que dependen del estado - Este estado se representa usualmente por constantes enumerativas y en muchas operaciones aparece el mismo condicional <i>Por ejemplo, en varios métodos de una clase tengo algo como if (color == rojo) { ... }</i> - Reemplaza el condicional por clases (polimorfismo). Internamente reemplazo en tiempo de ejecución un objeto por otro (cada objeto representa un estado)
Estructura	<p>Context : Define la interfaz que conocen los clientes. Mantiene una instancia de alguna clase de ConcreteState que define el estado corriente. <u>Quien tiene ESTADOS es el contexto</u></p> <p>State : Define la interfaz para encapsular el comportamiento de los estados de Context</p> <p>ConcreteState : Cada subclase implementa el comportamiento respecto al estado específico</p>

Structure



Implementación

State es una clase Abstracta/Interfaz y debe implementar todos los métodos del contexto

¿Quién define la transición entre estados?

El patrón no especifica qué participante define los criterios para las transiciones entre estados. En general, es más conveniente que sean las propias subclases de **State** quienes especifiquen su estado sucesor y cuándo llevar a cabo la transición. Esto requiere añadir una interfaz al Contexto que permita a los objetos estado asignar explícitamente el estado actual del Contexto (meter un setter al Context, lo cual rompe el encapsulamiento)

Dos opciones

- Crear los objetos Estado sólo cuando se necesitan y destruirlos después

Esta opción es preferible cuando no se conocen los estados en tiempo de ejecución y los contextos cambian de estado con poca frecuencia

- Crearlos al principio y no destruirlos nunca

Este enfoque es mejor cuando los cambios tienen lugar repentinamente, en cuyo caso queremos evitar destruir los estados, ya que pueden volver a necesitarse de nuevo en breve.

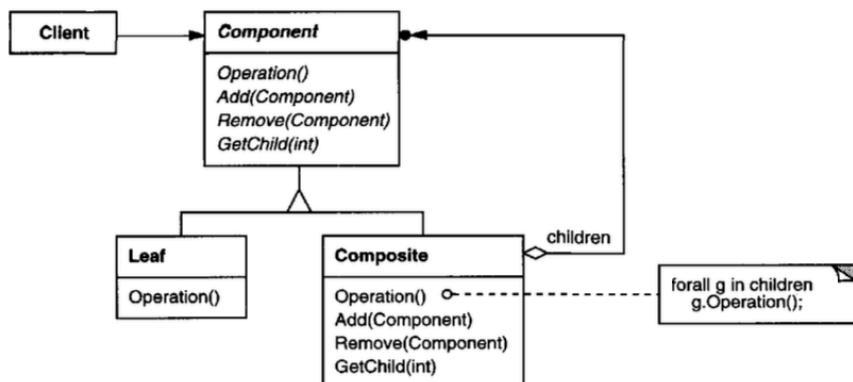
Este enfoque puede no ser apropiado ya que el Contexto debe guardar referencias a todos los estados en los que pudiera entrar

Resumen Objetos 2

Composite

Propósito/ Intención	<ul style="list-style-type: none"> - Componer objetos en estructuras de árbol para representar jerarquías parte-todo - La clave acá está en la composición - Se componen objetos para formar estructuras de árbol que representen jerarquías, y así tratar a los objetos atómicos y composiciones de igual manera - Permite que los clientes traten a los objetos atómicos y a sus composiciones uniformemente
Aplicabilidad	<ul style="list-style-type: none"> - Se quiere que los objetos “clientes” puedan ignorar las diferencias entre composiciones y objetos individuales. Los clientes tratarán a todos los objetos en la estructura compuesta de manera uniforme - Se quiere representar jerarquías parte-todo de objetos
Estructura	<p>Component: Declara la interfaz para los objetos de la composición. Implementa comportamientos default para la interfaz común a todas las clases.</p> <p>Leaf: Representa arboles “hojas” en la composición. Las hojas no tienen hijos. Define el comportamiento de objetos primitivos en la composición</p> <p>Composite: Define el comportamiento para componentes con hijos. Contiene la referencia a los hijos. Implementa operaciones para manejar hijos</p>

Structure



Implementación

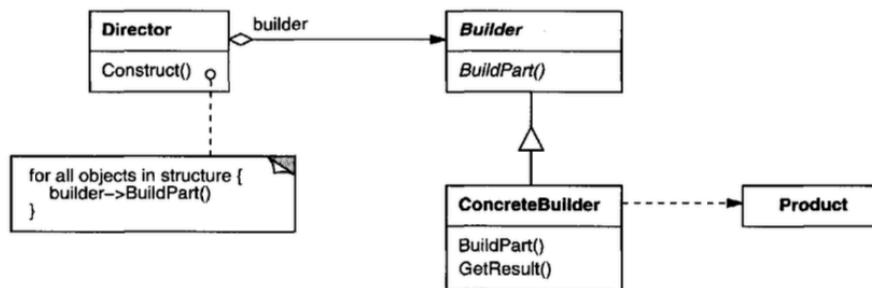
- Tanto las partes simples (leaf) como las composiciones (Composite) responden al mismo protocolo que lo podrá determinar una interfaz o una clase abstracta
- Observar que es medio “recursivo” porque composite (una composición) puede conocer tanto a una leaf como a otra composición. Por eso se habla que el patrón tiene una estructura de árbol
 - El mensaje que recibe composite lo deben ejecutar todos los hijos
- Sería importante redefinir métodos en las hojas para no tener errores. Ya que por ejemplo, leaf no debería devolver nada al recibir el mensaje getChild()
- En general, el rol Composite conoce a muchos Component (caso contrario al Patron Decorator)

Resumen Objetos 2

Builder

Propósito/ Intención	<ul style="list-style-type: none"> - Separa la construcción de un objeto complejo de su representación de manera que el proceso de construcción pueda crear diferentes representaciones - Permite construir un objeto complejo paso a paso. Permite producir diferentes tipos y representaciones de un objeto usando el mismo código
Aplicabilidad	<ul style="list-style-type: none"> - Cuando el algoritmo de creación de un objeto complejo debería ser independiente de las partes que hacen al objeto y cómo se ensamblan - Cuando el proceso de construcción debe permitir diferentes representaciones del objeto que es construido
Estructura	<p>Builder: Especifica una interfaz abstracta para crear partes de un objeto Producto: Declara los pasos de construcción del producto) ConcreteBuilder: Provee una interfaz para recuperar Producto Construye y ensambla partes del Producto implementando la interfaz de Builder Provee diferentes implementaciones de los pasos de construcción. Estos pueden producir productos que no siguen la interfaz común Director: Construye un objeto usando la interfaz Builder</p>

Structure



Implementación

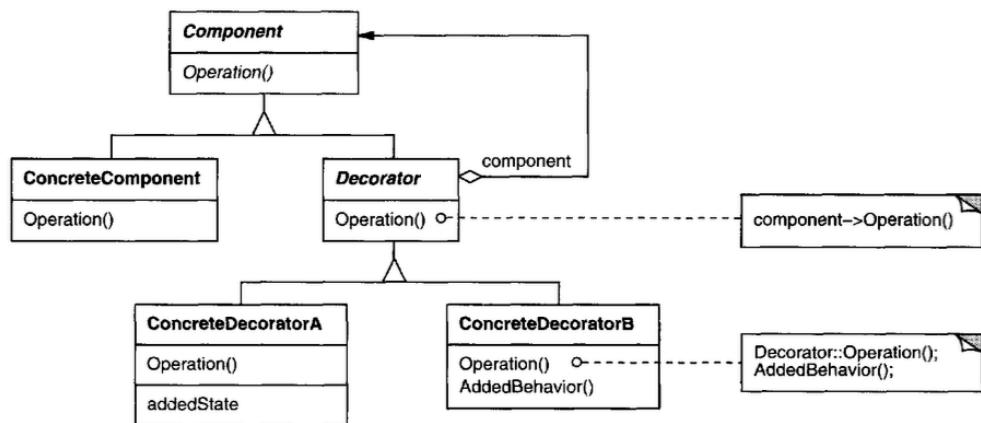
- El patrón Builder permite construir diferentes representaciones de un mismo objeto porque separa el proceso de construcción de la representación final del objeto
- Esto significa que con el mismo Builder podemos crear variaciones del Product, dependiendo de qué datos o configuraciones le demos.

Resumen Objetos 2

Decorator

Propósito/Intención	- Agregar comportamiento a un objeto dinámicamente y en forma transparente - Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender funcionalidad
Aplicabilidad	- Para agregar responsabilidades a objetos individuales dinámicamente y de forma transparente (sin afectar a otros objetos) - Para responsabilidades que pueden ser retiradas - Cuando la extensión mediante la herencia no es viable
Estructura	

Structure



Implementación

- Misma interface entre Component y Decorator
- No hay necesidad de la clase Decorator abstracta

Un Component puede ser 2 cosas : un componenteConcreto o un Decorador

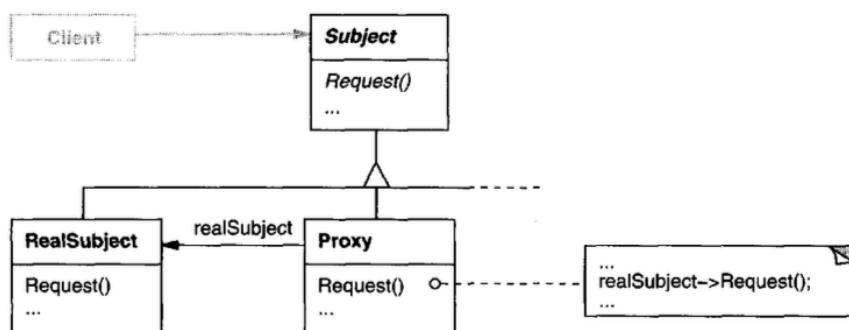
- El Component en si es una interfaz
- Lo que realmente se instancia podrá ser un Decorador o un ConcreteComponent
- La idea es que el Decorador “envuelve” a otro decorador o finalmente a un componente concreto
- Las subclases de Decorador son libres de añadir operaciones para determinadas funcionalidades

Resumen Objetos 2

Proxy

Propósito/ Intención	- Proporcionar un intermediario de un objeto para controlar su acceso - Proporcionar un representante o sustituto de otro objeto para controlar el acceso a este
Aplicabilidad	- Virtual proxy: se usa cuando se tiene un objeto muy pesado que consume muchos recursos, entonces se retrasa su instanciación únicamente hasta que se lo necesita - Protection proxy: se usa cuando se quiere que solo ciertos cliente accedan al servicio real
Estructura	Interfaz Subject Real Subject Proxy Subject

Structure



Implementación

Proxy es un objeto intermedio que respeta el protocolo/mantiene el protocolo del objeto que esta reemplazando

Null Object

Propósito/ Intención	<ul style="list-style-type: none"> - Proporciona un sustituto para otro objeto (similar a proxy) que comparte la misma interfaz pero no hace nada. - El NullObject encapsula las decisiones de implementación de cómo "no hacer nada" y oculta esos detalles de sus colaboradores
Aplicabilidad	<ul style="list-style-type: none"> - Se usa cuando un objeto requiere un colaborador. Esta colaboración ya existía implícitamente, no es que el patrón Null Object "la introduce"
Estructura	<p style="text-align: center;">Client AbstractObject</p> <ul style="list-style-type: none"> - Declara la interfaz que usará Client - Implementa comportamiento por defecto RealObject - Define una subclase concreta de AbstractObject cuya instancia provee comportamiento útil para el cliente <p style="text-align: center;">NullObject</p> <ul style="list-style-type: none"> - Provee una interfaz idéntica a AbstractObject de manera que un null object puede ser sustituido por un real object - Implementa la interfaz para no hacer nada. El hacer nada depende del comportamiento que Client este esperando

Structure

```

classDiagram
    class Client
    class AbstractClass {
        +Operation()
    }
    class RealObject {
        +Operation()
    }
    class NullObject {
        +Operation()
    }

    Client "Uses" --> AbstractClass
    AbstractClass --> RealObject
    AbstractClass --> NullObject
  
```

Implementación

- Elimina todos los condicionales que verifican si la referencia a un objeto es NULL
- Hace explícito elementos del dominio que hacen "nada"

Refactoring to patterns

REFACTORING TO PATTERNS	
Tipos	Form Template Method
<p>Refactoring to Patterns</p> <ul style="list-style-type: none"> ■ Form Template Method ■ Extract Adapter ■ Replace Implicit Tree with Composite ■ Replace Conditional Logic with Strategy ■ Replace State-Altering Conditionals with State ■ Move Embelishment to Decorator 	<ol style="list-style-type: none"> 1) Encontrar el método que es similar en todas las subclases y extraer sus partes en: métodos idénticos (misma firma y cuerpo en las subclases) o métodos únicos (distinta firma y cuerpo) 2) Aplicar "Pull Up Method" para los métodos idénticos. 3) Aplicar "Rename Method" sobre los métodos únicos hasta que el método similar quede con cuerpo idéntico en las subclases. 4) Compilar y testear después de cada "rename". 5) Aplicar "Rename Method" sobre los métodos similares de las subclases (esqueleto). 6) Aplicar "Pull Up Method" sobre los métodos similares. 7) Definir métodos abstractos en la superclase por cada método único de las subclases. 8) Compilar y testear
Replace Conditional Logic with Strategy	Replace State-Altering Conditionals with State
<ol style="list-style-type: none"> 1) Crear una clase Strategy. 2) Aplicar "Move Method" para mover el cálculo con los condicionales del contexto al strategy. <ol style="list-style-type: none"> 1) Definir una v.i. en el contexto para referenciar al strategy y un setter (generalmente el constructor del contexto) 2) Dejar un método en el contexto que delegue 3) Elegir los parámetros necesarios para pasar al strategy (el contexto entero? Sólo algunas variables? Y en qué momento?) 4) Compilar y testear. 3) Aplicar "Extract Parameter" en el código del contexto que inicializa un strategy concreto, para permitir a los clientes setear el strategy. <ul style="list-style-type: none"> - Compilar y testear. 4) Aplicar "Replace Conditional with Polymorphism" en el método del Strategy. 5) Compilar y testear con distintas combinaciones de estrategias y contextos. 	<ol style="list-style-type: none"> 1. Aplicar "Replace Type-Code with Class" para crear una clase que será la superclase del State a partir de la v.i. que mantiene el estado 2. Aplicar "Extract Subclass" [F] para crear una subclase del State por cada uno de los estados de la clase contexto. 3. Por cada método de la clase contexto con condicionales que cambiar el valor del estado, aplicar "Move Method" hacia la superclase de State. 4. Por cada estado concreto, aplicar "Push down method" para mover de la superclase a esa subclase los métodos que producen una transición desde ese estado. Sacar la lógica de comprobación que ya no hace falta. 5. Dejarlos estos métodos como abstractos en la superclase o como métodos por defecto.
Move Embelishment to Decorator	Introduce Null Object
<ol style="list-style-type: none"> 1. Identificar la superclase (or interface) del objeto a decorar (clase Component del patrón). Si no existe, crearla. 2. Aplicar <i>Replace Conditional Logic with Polymorphism</i> (crea decorador como subclase del decorado). <pre> classDiagram class AbstractNode { <<AbstractNode>> } class StringNode { <<StringNode>> } class DecodingNode { <<DecodingNode>> } AbstractNode < -- StringNode AbstractNode < -- DecodingNode StringNode <--> toPlainTextString : public String DecodingNode <--> toPlainTextString : public String code toPlainTextString() { return textBuffer.toString(); } code toPlainTextString() { return Translate.decode(super.toPlainTextString()); } </pre> <p>Alcanza? Si no sigo</p> <ol style="list-style-type: none"> 3. Aplicar <i>Replace Inheritance with Delegation</i> (decorador delega en decorado como clase "hermana") 4. Aplicar <i>Extract Parameter</i> en decorador para asignar decorado 	<ol style="list-style-type: none"> 1) Crear el <i>null object</i> aplicando "Extract Subclass" sobre la clase que se quiere proteger del chequeo por null (clase origen). Alternativamente hacer que la nueva clase implemente la misma interface que la clase origen. Compilar. 2) Buscar un <i>null check</i> en el código cliente, es decir, código que invoque un método sobre una instancia de la clase origen si la misma no es null. Redefinir el método en la clase del <i>null object</i> para que implemente el comportamiento alternativo. Compilar 3) Repetir el paso 2 para todos los <i>null checks</i> asociados a la clase origen. 4) Encontrar todos los lugares que pueden retornar null cuando se le pide una instancia de la clase origen. Inicializar con una instancia del <i>null object</i> lo antes posible. Compilar 5) Para cada lugar elegido en el paso 4, eliminar los <i>null checks</i> asociados

Test Double

Definición	Test Double es un lenguaje de patrones
Utilización	Cuando es necesario realizar pruebas de un SUT que depende de un módulo u objeto que no se puede utilizar en el ambiente de la pruebas
Tipos de pruebas	<ul style="list-style-type: none"> -Configuraciones válidas del sistema -“Salidas indirectas” del sistema -Lógica del sistema -Protocolos
Aplicabilidad	<ul style="list-style-type: none"> - Crear un objeto que es una maqueta (polimórfica) del objeto o módulo requerido - Utilizar la maqueta según se necesite
Rangos de implementación	<p style="text-align: center;">- Cascarón vacío → Simulación</p> <ul style="list-style-type: none"> - Se generan diferentes patrones que se aplican a cada caso
Implementación	<p>Van de más simple y barato a más caro y complejo (caro en tiempo de análisis y mantenimiento)</p> <p style="text-align: center;">Implementar según sea necesario</p> <ul style="list-style-type: none"> - Es necesario verificar funcionalidad de dependencia - Objetos que no están disponibles para probar

TIPOS DE TEST DOUBLE		
Van de más simple y barato a más caro y complejo	Test Stub	Cascarón vacío. Sirve para que el SUT envíe los mensajes esperados Recibe los mensajes, no hace nada.
	Test Spy	Test Stub + registro de mensajes recibidos Guarda registro de mensajes recibidos
	Mock Object	Test Spy + verificación de mensajes recibidos Comprueba la validez de los mensajes recibidos
	Fake Object	Imitación. Se comporta como el módulo real (protocolos, tiempos de respuesta, etc) Simula el comportamiento en tiempo y forma

Resumen Objetos 2

Frameworks

DEFINICIONES
Un framework es una aplicación “semi-completa”, “reusable”, que puede ser especializada para producir aplicaciones a medida
Es un conjunto de clases concretas y abstractas, relacionadas para proveer una arquitectura reusable que implementa una familia de aplicaciones (relacionadas)
Proveen una solución reusable para una familia de aplicaciones
Las clases en el framework se relacionan (herencia, conocimiento, envío de mensajes) de manera que resuelven la mayor parte del problema en cuestión
El código del framework controla/usa al código de la instanciaación

CONCEPTOS

Framework de Caja Blanca	<ul style="list-style-type: none"> - La instanciación hereda y completa el loop de control - Es posible que requiera agregar métodos a clases del framework - Demanda conocimiento del código del framework - Las instanciações “completan” el loop de control agregando código: <ul style="list-style-type: none"> ○ Ejercitando un hotspot con herencia ○ Modificando código fuente del framework
Framework de Caja Negra	<p>Las instanciações se basan en configuraciones</p> <p>A los que frameworks que utilizan composición les llamamos de Caja Negra (blackbox)</p> <p>Hay frameworks que se pueden usar, instanciando objetos y componiéndolos (sin entender como esas cosas funcionan o quien le manda mensajes a quien). Esto puede llevarse al extremo en donde uno ni siquiera escribe código sino que uno setea cosas simplemente (frameworks muy maduros)</p>
Hotspot	<p style="text-align: center;">Elementos del diseño pensadas para adaptarse</p> <p>Estructura en el código que permite modificar el comportamiento del framework, para instanciar y para extender</p>
Frozenspot	<p style="text-align: center;">Partes del diseño que no cambian</p> <p>Aspecto del framework que afecta a todas las instanciações y que no se puede modificar (marca indeleble)</p>
Inversión de control	<p style="text-align: center;">Este principio de diseño permite que el flujo de control del programa sea invertido</p> <p>En lugar de que el código de la aplicación controle el flujo de la ejecución, un framework o contenedor externo toma el control y llama a los componentes según sea necesario</p> <p style="text-align: center;">La inversión de control permite al framework determinar qué set de métodos específicos de la aplicación invocar en respuesta a eventos externos</p> <p>El código que escriben los programadores de un framework le envía mensajes a las clases escritas por mí</p> <p>Lo normal sería que las clases escritas por mí nunca sean instanciadas, así como los métodos implementados en dichas clases, nunca los ejecutaré desde mi código. A esas clases y esos mensajes los instancia y los envía el propio framework</p> <p>Por eso se dice, que el código del framework invoca a mi código. A esto se le llama Inversión de control. El código del framework controla el nuestro.</p>

Resumen Objetos 2

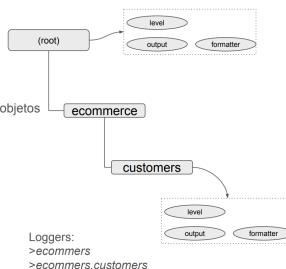
Por Composición	<p>Serían los frameworks de Caja Negra -> no conozco el código</p> <p>Es una relación en la que un objeto depende de otro. No puede funcionar sin dicho objeto</p> <p>La composición puede cambiar en tiempo de ejecución</p> <p>Cuando uso composición como usuario de un framework simplemente instancio y configuro, conecto a mi código con callbacks (paso funciones como parámetros), y no puedo cambiar o extender el framework</p>
Por Herencia	<p>Serían los frameworks de Caja Blanca -> heredo comportamiento</p> <p>La herencia es cuando un objeto hereda estado y comportamiento de otro. En el caso de las interfaces, no hay herencia, sino que las subclases de una interfaz implementan la interfaz, pero no heredan comportamiento alguno.</p> <p>La herencia es más estática y no puede ser fácilmente cambiada en tiempo de ejecución</p> <p>Cuando uso herencia como usuario de un framework podría cambiar cosas que el desarrollador no tuvo en cuenta, puedo extender el framework, uso variables y métodos heredados, Implemento, extiendo, y redefino métodos</p>
Hooks	<p>Métodos que implementan las subclases</p> <p>Opcionales: <u>protected</u> methods.</p> <p>Obligatorios: <u>abstract</u> methods</p>

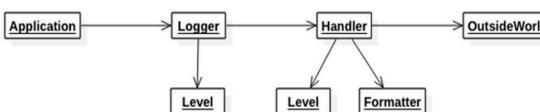
TIPOS DE FRAMEWORKS	
Aplicación: desktop, webapps, tcpservers	
Manejo Datos: ORDB, pipelines, NRDB	
Sistemas Distribuidos: mensajes, eventos, rpc	
Testing: unit, web pages	
SingleThreadTCPServer (whitebox)	Un servidor como extensión de una clase loop + sesión (singular)
Tcp.server.reply (blackbox)	Loop(sesión(msgHandler)). Sesión singular o multiple. Diferentes MsgHandler
Java.util.logging	Jerarquía de Labels + composición de filtros, formatos y salidas

Servidores TCP	
<p>Servidores TCP</p> <ul style="list-style-type: none"> Es un programa que implementa una de las partes de la arquitectura Cliente/Servidor <ul style="list-style-type: none"> Un Servidor escucha en un socket TCP (IP_address+port) Un Cliente establece una conexión con el servidor para enviar mensajes (plain text) El Servidor responde Ejemplos: <ul style="list-style-type: none"> EchoServer: <ul style="list-style-type: none"> Recibe un mensaje de texto Responde el mensaje que el cliente envió SMTPServer: emails DayTimeServer: fecha actual HTTPServer: envío y recepción de documentos web 	<p>Flujo de Control</p> <ol style="list-style-type: none"> Crear un socket (IP_address +port) Escucha en el socket Aceptar al cliente => Sesión Recibir un mensaje <ol style="list-style-type: none"> Condición ≠ Responder !Condición => <ol style="list-style-type: none"> Cerrar sesión Goto 2 <ul style="list-style-type: none"> Reusar el “flujo de control” <ul style="list-style-type: none"> El flujo de control es incompleto (generalmente no compila) “Responder” es donde se puede “enganchar” la funcionalidad “Responder” es un <u>hook (obligatorio)</u> El flujo de control pertenece a un framework

Resumen Objetos 2

Dominio: TCP Servers			
SingleThreadTCPServer (whitebox)	tcp.server.reply (blackbox)		
Cookbook:	Cookbook		
1. Subclasicar SimpleThreadTCPServer a. Debe implementar Main(String[]) <ul style="list-style-type: none"> i. crear una instancia ii. enviar método startLoop(String[]) Debe implementar handleMessage(String) hook	1. En un objeto "contexto" a. instanciar un MessageHandler <ul style="list-style-type: none"> i. Echo, ii. Void b. Instanciar ConnectionHandler con el MessageHandler <ul style="list-style-type: none"> i. SimpleConnectionHandler ii. MultiConnectionHandler c. Instanciar TCPControlLoop con ConnectionHandler Enviar método startLoop() al TCPControlLoop	<pre> 1 import java.io.PrintWriter; 2 3 public class EchoServer extends SingleThreadTCPServer { 4 5 public void handleMessage(String message, PrintWriter out) { 6 out.println(message); 7 } 8 9 Run Debug 10 public static void main(String[] args) { 11 new EchoServer().startLoop(args); 12 } 13 14 }</pre>	SingleThreadTCPServer (hotspot herencia)

Java.util.logging	
java.util.logging <ul style="list-style-type: none"> Agregamos código de loggin a nuestra aplicación para entender lo que pasa con ella, por ejemplo: <ul style="list-style-type: none"> Reportes de eventos importantes, errores y excepciones Pasos críticos en la ejecución Inicio y fin de operaciones complejas o largas Los logs son útiles para desarrolladores, administradores y usuarios Comentario al margen: ¡Los logs no reemplazan al testing! Mucho mejor que System.out.println, que es "rapido&sucio" <ul style="list-style-type: none"> Define jerarquía de "labels" Activar/Desactivar logs (sin tocar código) Generar reportes en varios formatos (txt, json,xml) y destinos (file, screen, socket) 	java.util.logging <ul style="list-style-type: none"> La aplicación <ul style="list-style-type: none"> Configura al framework Manda mensajes a objetos Logger El Framework se encarga de: <ul style="list-style-type: none"> Como se crean, organizan y recuperan esos objetos Como se configuran Como se activan y desactivan A qué prestan atención y a qué no Cómo se formatean los logs A dónde se envían los logs  <pre> graph TD root --> level1 root --> output1 root --> formatter1 ecommerce --> ecommerceLevel ecommerce --> ecommerceOutput ecommerce --> ecommerceFormatter ecommerceLevel --> customers ecommerceOutput --> customers ecommerceFormatter --> customers customers --> customersLevel customers --> customersOutput customers --> customersFormatter </pre> <p>Loggers: >ecommerce >ecommerce.customers</p>

Arquitectura visible	Logger
<ul style="list-style-type: none"> Logger: objeto al que le pedimos que emita un mensaje de log Handler: encargado de enviar el mensaje a donde corresponda Level: indica la importancia de un mensaje y es lo que mira un Logger Logger y un Handler para ver si le interesa Formatter: determina cómo se "presentará" el mensaje 	Logger <ul style="list-style-type: none"> Podemos definir tantos como necesitemos <ul style="list-style-type: none"> Instancias de la clase Logger Las obtengo con Logger.getLogger(String nombre) Cada uno con su filtro y handler/s Se organizan en un árbol (en base a sus nombres) <ul style="list-style-type: none"> Heridan configuración de su padre (handlers y filters) log(Level, String) agrega un mensaje al log <ul style="list-style-type: none"> Alternativamente uso warn(), info(), severe() ... <pre> Logger +addHandler(Handler handler) +setLevel(Level level) +isLoggable(Level level): boolean +log(Level level, String msg) +warn(String msg) +info(String msg) +severe(String msg) </pre>
<pre> 1 import java.util.logging.Logger; 2 3 public class SimpleLoggingExample { 4 5 private static final Logger logger = Logger.getLogger(SimpleLoggingExample.class.getName()); 6 7 public static void main(String[] args) { 8 logger.info("Application started"); 9 10 try { 11 int result = 10 / 0; // Simulate an error 12 } catch (ArithmaticException e) { 13 logger.severe("An error occurred: " + e.getMessage()); 14 } 15 16 logger.info("Application finished"); 17 } 18 } 19 </pre>	Variante de uso de Loggers <pre> public class Sandbox { public static void main(String[] args) throws IOException { Logger.getLogger("app.main").addHandler(new FileHandler("log.txt")); Logger.getLogger("app.main").log(Level.INFO, "App iniciada"); try { // Acá que hace algo que "podría" resultar en una excepción int explodesForSure = 1 / 0; } catch (Exception ex) { Logger.getLogger("app.main").log(Level.SEVERE, "Explotó!", ex); } Logger.getLogger("app.main").log(Level.INFO, "App terminada"); } } </pre> <p>Logger <ul style="list-style-type: none"> mantiene un "registry" de sus instancias. getLogger() es un lazy-initializer </p>

Resumen Objetos 2

Ejemplo avanzado

```
//Loggers apagados por defecto
Logger.getLogger("").setLevel(Level.OFF);

//Loggers encendidos en nivel SEVERE para ecommerce
//Utilizará un ConsoleHandler y un SimpleFormatter
Logger ecommerce = Logger.getLogger("ecommerce");
ecommerce.setLevel(Level.SEVERE);
```

Ejemplo avanzado

```
//Loggers apagados por defecto
OFF: Level :SimpleFormatter
SEVERE: Level :SimpleFormatter
WARNING: Level :SimpleFormatter

:Filter :ConsoleHandler :Filter :FileHandler
root: Logger ecommerce: Logger customersLogger: Logger
```

Ejemplo avanzado

```
// Este logger hereda de la raíz y no define nada propio, por lo tanto ignora el warning
Logger.getLogger("delivery").log(Level.WARNING, "Error in delivery");

// Este logger hereda de ecommerce por lo tanto ignora el warning
Logger.getLogger("ecommerce.products").log(Level.WARNING, "Stock inconsistency detected");

// A este logger le interesa el warning, que termina en un archivo con formato simple
Logger.getLogger("ecommerce.customers").log(Level.WARNING, "Stock inconsistency detected");
```

Extendiendo el framework

- Y, mirando adentro, puedo agregar nuevas clases de Formatter, Handler y Filter
 - Nuevo Formatter: Subclaseo la clase abstracta Formatter o alguna de sus subclases
 - Nuevo Handler: Subclaseo la clase abstracta Handler o alguna de sus subclases
 - Nuevo Filter: Implemento la interfaz Filter
- Esto no es hacking, sino algo previsto por los diseñadores

Ejemplo usando Logger, FileHandler y SimpleFormatter

```
private static final Logger uiLogger = Logger.getLogger("WallPostUI");

static {
    try {
        FileHandler fileHandler = new FileHandler("ui.log", true);
        fileHandler.setFormatter(new SimpleFormatter());
        uiLogger.addHandler(fileHandler);
        uiLogger.setUseParentHandlers(false);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```