REFACTORING

Refactoring: transformación de código que preserva el comportamiento, pero mejora el diseño. Con la refactorización se puede tomar un mal diseño, el caos incluso, y reelaborarlo en un código bien diseñado. Cada paso es simple, incluso simplista. Refactoring no agrega funcionalidad ⇒ puede complicar deadlines.

¿POR QUÉ REFACTORING ES IMPORTANTE?

- Ganar en la comprensión del código
- Reducir el costo de mantenimiento debido a los cambios inevitables que sufrirá el sistema (por ejemplo, código duplicado que haya que cambiar)
- Facilitar la detección de bugs
- Generar codigo legible
- La clave: poder agregar funcionalidad más rápido después de refactorizar

Además, permite recrear CLEAN Code:

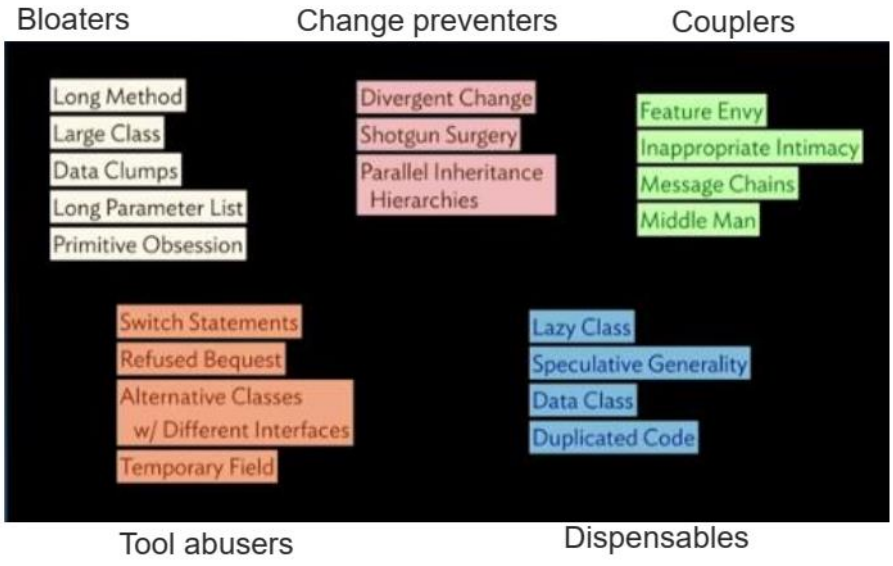
- Cohesive: Significa que un módulo, clase o función debe tener una única responsabilidad o propósito claro. Todo lo que hace debe estar relacionado con esa única tarea.
- Loosely coupled: Se refiere a que los componentes del sistema (clases, módulos, etc.) deben depender lo menos posible unos de otros. Si un componente cambia, no debería afectar drásticamente a los demás.
- Encapsulated: Implica ocultar los detalles internos de cómo funciona algo y solo exponer lo necesario a través de una interfaz clara.
- Assertive: El código debe ser claro y directo en lo que hace, evitando ambigüedades.
- Non-redundant: No debe haber duplicación en el código. Si algo se repite, debería abstraerse en una función, clase o módulo reutilizable.

Se toma un código que “huele mal”, producto de mal diseño y se lo trabaja para obtener un buen diseño.

CODE SMELLS

- Indicadores de posibles problemas en el diseño del código fuente que requieren la aplicación de refactorings.
- Destacan áreas en las que el código podría estar violando principios de diseño fundamentales.
- No son errores, pero en el futuro el desarrollo podría ser más difícil y demandar más horas.

CATEGORIZACIÓN DE BAD SMELLS



1. Bloaters (Infladores): Son problemas relacionados con el crecimiento excesivo o innecesario del código, lo que lo hace más grande, complicado o difícil de manejar.

- El código se vuelve difícil de leer y mantener porque está "inflado" con más de lo necesario.

2. Change Preventers (Preventores del Cambio): Son estructuras en el código que dificultan realizar modificaciones porque un cambio en un lugar requiere ajustes en muchos otros.

- Hacer cambios se convierte en una pesadilla porque el código está rígido o demasiado interconectado.

3. Couplers (Acopladores): Se refiere a un acoplamiento excesivo entre partes del código, es decir, cuando los módulos o clases dependen demasiado unos de otros.

- El alto acoplamiento hace que el sistema sea frágil; un cambio en un componente puede romper otros inesperadamente.

4. Tool Abusers (Abusadores de Herramientas): Ocurre cuando se usan mal las herramientas, patrones o características del lenguaje de programación, a menudo por seguir modas o no entender su propósito.

- El mal uso de herramientas lleva a código confuso o innecesariamente complejo.

5. Dispensables (Dispensables): Elementos en el código que no aportan valor y podrían eliminarse sin afectar la funcionalidad.

- Añaden ruido y confusión, aumentando el esfuerzo para mantener el código sin ningún beneficio.

TODOS LOS BAD SMELLS - EXPLICACIÓN

- Duplicated code: El mismo código, o código muy similar, aparece en muchos lugares.
    - Hace el código más largo de lo que necesita ser
    - Es difícil de cambiar, dificil de mantener
    - Un bug fix en un clone no es fácilmente propagado a los demás clones
- Long Method: Un método tiene muchas líneas de código.
    - Cuanto más largo es un método, más difícil es entenderlo, cambiarlo y reusarlo
    - Un bloque de código con un comentario que dice lo que está haciendo puede ser reemplazado por un método que esté basado en el comentario mismo.
- Large Class: Una clase intenta hacer demasiado trabajo. Tiene muchas variables de instancia. Tiene muchos métodos.
    - Indica un problema de diseño (baja cohesión)
    - Algunos métodos pueden pertenecer a otra clase
    - Generalmente tiene código duplicado
- Long Parameter List: Un método con una larga lista de parámetros es más dificil de entender.
    - También es difícil obtener todos los parámetros para pasarlos en la llamada entonces el método es más difícil de reusar
    - La excepción es cuando no quiero crear una dependencia entre el objeto llamador y el llamado
- Divergent Change: El cambio divergente ocurre cuando una clase se cambia comúnmente de diferentes maneras para diferentes razones.
    - Cualquier cambio para controlar una variación debe cambiar una sola clase, y todos los tipos en la nueva clase deben expresar la variación
- Shotgun Surgery: Es el opuesto de Divergent Change. Hueles esto cuando cada vez que se hace un tipo de cambio, se tienen que hacer un montón de pequeños cambios en muchas clases diferentes
    - Cuando los cambios están sobre todo el lugar, son difíciles de encontrar y es muy fácil perder cambios importantes.

- Feature Envy: Un método en una clase usa principalmente los datos y métodos de otra clase para realizar su trabajo (se muestra “envidiosa” de las capacidades de otra clase)
    - Indica un problema de diseño
    - Idealmente se prefiere que los datos y las acciones sobre los datos vivan en la misma clase
    - “Feature Envy” indica que el método fue ubicado en la clase incorrecta
- Data Clumps: Los conjuntos de datos que permanecen juntos deberían convertirse en un objeto propio.
    - Si desea asegurarse de que algunos datos sean un conjunto, simplemente elimine uno de los valores y compruebe si los demás siguen teniendo sentido. De no ser así, es una buena señal de que este grupo de variables debe combinarse en un objeto.
- Primitive Obsession: Los primitivos, que incluyen números enteros, cadenas, dobles, matrices y otros elementos del lenguaje de bajo nivel, son genéricos porque mucha gente los usa.
    - Fowler y Beck explican cómo la obsesión por lo primitivo se manifiesta cuando el código depende demasiado de primitivos.
    - Esto suele ocurrir cuando aún no se ha visto cómo una abstracción de alto nivel puede aclarar o simplificar el código.
- Switch Statements: Cuando sentencias condicionales contienen lógica para diferentes tipos de objetos. Cuando todos los objetos son instancias de la misma clase, eso indica que se necesitan crear subclases.
    - La misma estructura condicional aparece en muchos lugares (duplicación de código).
    - Si se añade una nueva cláusula al switch se deben encontrar todas las declaraciones del mismo y cambiarlas.
- Parallel Inheritance Hierarchies: Es un caso especial de Shotgun Surgery. En este caso, cada vez que se crea una subclase de una clase, también se debe crear una subclase de otra.
    - Se puede reconocer este olor porque los prefijos de los nombres de clase en una jerarquía son los mismos que en otra.
- Lazy Class (Freeloader): Cada clase que se crea cuesta dinero para mantenerla y comprenderla. Una clase que no se autofinancia lo suficiente debe eliminarse.
    - A menudo, puede tratarse de una clase que antes se autofinanciaba, pero que se ha reducido mediante refactorización. O puede ser una clase que se añadió debido a cambios planificados, pero que no se implementaron.
- Speculative Generality: Este olor se presenta cuando se tiene código genérico o abstracto que no se necesita actualmente. Dicho código suele existir para respaldar comportamientos futuros, que podrían o no ser necesarios en el futuro.
    - La generalidad especulativa se puede detectar cuando los únicos usuarios de un método o clase son casos de prueba.
- Temporary Field: A veces, los objetos contienen campos que no parecen ser necesarios constantemente. El resto del tiempo, el campo está vacío o contiene datos irrelevantes, lo cual es difícil de entender.
    - Un caso común de campo temporal ocurre cuando un algoritmo complejo necesita varias variables. Como el implementador no quería pasar una lista enorme de, los colocó en campos que son sólo válidos durante el algoritmo.
- Message Chains: Se produce cuando se observa una larga secuencia de llamadas a métodos o variables temporales para obtener datos.
    - Esta cadena hace que el código dependa de las relaciones entre muchos objetos potencialmente no relacionados.
    - Cualquier cambio en las relaciones intermedias obliga al cliente a cambiar.
- Middle Man: Un exceso de delegación puede dar lugar a objetos que no aportan ningún valor y que simplemente pasan mensajes a otro objeto.
    - La clase permanece como un cascarón vacío que no hace nada más que delegar.
- Inappropiate Intimacy: A veces, las clases se vuelven demasiado íntimas y dedican demasiado tiempo a utilizar los campos y métodos internos de otra.

    - La herencia a menudo puede generar una intimidad excesiva. Las subclases siempre sabrán más sobre sus padres de lo que estos desearían que supieran.
- Alternative Classes with Different Interfaces: Ocurre cuando las interfaces de dos clases son diferentes y, sin embargo, las clases son bastante similares.
    - Duplicación de código.
    - Menor reutilización.
    - Las interfaces diferentes pueden dificultar que los desarrolladores o usuarios del sistema comprendan cómo interactuar con las clases de manera uniforme.
- Incomplete Library Class: Se produce cuando surgen responsabilidades en nuestro código que claramente deberían trasladarse a una clase de biblioteca, pero no podemos o no queremos modificar la clase de biblioteca para aceptar estas nuevas responsabilidades.
- Data Class: Una clase que solo tiene variables y getters/setters para esas variables. Actúa únicamente como contenedor de datos
    - En general sucede que otras clases tienen métodos con “envidia de atributo”
    - Esto indica que esos métodos deberían estar en la “data class”
    - Suele indicar que el diseño es procedural
- Refused Bequest: Este olor resulta de heredar código que no se desea. En lugar de tolerar la herencia, se escribe código para rechazar el legado.
    - Causa confusión y problemas.
    - El olor a legado rechazado es mucho más fuerte si la subclase reutiliza el comportamiento, pero no quiere soportar la interfaz de la superclase. La subclase depende de la superclase para algunas cosas, pero rechaza otras, lo que sugiere que la relación de herencia no es adecuada.
- Comments: Muchas veces los comentarios aparecen porque el código es malo. Los métodos deben ser explicativos y entendibles, y un comentario no debería explicar que se trató de realizar en el mismo.

SMELLS - REFACTORINGS

Smell	Refactoring
Alternative Classes with Different Interfaces	Rename Method
	Move Method
	Extract Superclass
Comments	Rename Method
	Extract Method
	Introduce Assertion
Data Class	Move Method
	Encapsulate Field
	Encapsulate Collection
	Remove Setting Method
	Extract Method
	Hide Method
Data Clumps	Extract Class
	Preserve Whole Object
	Introduce Parameter Object
Divergent Change	Extract Class
Duplicated Code	Extract Method
	Pull Up Field
	Form Template Method
	Substitute Algorithm
	Extract Class
	Introduce Null Object
Feature Envy	Extract Method
	Move Method

	Move Field
Lazy Class (Freeloader)	Collapse Hierarchy
	Inline Class
Inappropriate Intimacy	Move Method
	Move Field
	Change Biridectional Association to Unidirectional Association
	Extract Class
	Hide Delegate
	Replace Inheritance with Delegation
Incomplete Library Class	Introduce Foreign Method
	Introduce Local Extension
Large Class	Extract Class
	Extract Subclass
	Extract Interface
	Replace Data Value with Object
Long Method	Extract Method
	Introduce Parameter Object
	Decompose Conditional
	Preserve Whole Object
	Replace Method with Method Object
	Replace Temp with Query
Long Parameter List	Replace Parameter with Method
	Introduce Parameter Object
	Preserve Whole Object
Message Chains	Hide Delegate
	Extract Method
	Move Method
Middle Man	Remove Middle Man
	Inline Method
	Replace Delegation with Inheritance
Parallel Inheritance Hierarchies	Move Method
	Move Field
Primitive Obsession	Replace Data Value with Object
	Introduce Parameter Object
	Extract Class
	Replace Type Code with Class
	Replace Code with State/Strategy
	Replace Type Code with Subclasses
	Replace Array with Object
Refused Bequest	Push Down Field
	Push Down Method
	Replace Inheritance with Delegation
Shotgun Surgery	Move Method
	Move Field
	Inline Class
Speculative Generality	Collapse Hierarchy
	Rename Method
	Remove Parameter
	Inline Class
Switch Statement	Extract Method
	Move Method
	Replace Type Code with Subclasses
	Replace Type Code with State/Strategy
	Replace Conditional with Polymorphism
	Replace Parameter with Explicit Methods

	Introduce Null Object
Temporary Field	Extract Class
	Introduce Null Object

REFACTORINGS

COMPOSICIÓN DE METODOS

Permiten distribuir el código adecuadamente. Los métodos largos son problemáticos ya que contienen mucha información y lógica compleja.

- Extract Method: Mueve fragmento de código a un nuevo método con nombre descriptivo.
- Inline Method: Sustituye una llamada a método por el contenido del mismo.
- Replace Temp with Query: Sustituye variables temporales por métodos.
- Split Temporary Variable: Divide variables temporales con múltiples asignaciones.
- Replace Method with Method Object: Convierte un método en una clase para manejar más complejidad.
- Substitute Algorithm: Reemplaza un algoritmo por uno más claro o eficiente.

MOVER ASPECTOS ENTRE OBJETOS

Ayudan a mejorar la asignación de responsabilidades.

- Move Method: Mueve un método a la clase donde más se utiliza.
- Move Field: Mueve un atributo a otra clase que lo necesita más.
- Extract Class: Crea una nueva clase cuando una clase tiene demasiadas responsabilidades.
- Inline Class: Elimina una clase poco útil moviendo sus miembros a otra clase.
- Remove Middle Man: Elimina métodos que solo delegan.
- Hide Delegate: Oculta el uso de un delegado exponiendo métodos propios.

ORGANIZACIÓN DE DATOS

Facilitan la organización de atributos.

- Self Encapsulate Field: Accede siempre a un campo a través de métodos.
- Encapsulate Field / Collection: Restringe el acceso directo a campos/colecciones.
- Replace Data Value with Object: Sustituye un valor simple por un objeto con comportamiento.
- Replace Array with Object: Sustituye un array por un objeto con campos nombrados.
- Replace Magic Number with Symbolic Constant: Usa constantes con nombre en lugar de números mágicos.

SIMPLIFICACIÓN DE EXPRESIONES CONDICIONALES

Ayudan a simplificar los condicionales.

- Decompose Conditional: Separa condiciones y acciones en métodos distintos.
- Consolidate Conditional Expression: Combina múltiples condiciones en una sola.
- Consolidate Duplicate Conditional Fragments: Mueve código duplicado fuera de condicionales.
- Replace Conditional with Polymorphism: Usa polimorfismo en lugar de condicionales para comportamiento variante.

SIMPLIFICACIÓN EN LA INVOCACIÓN DE MÉTODOS

Sirven para mejorar la interfaz de una clase.

- Rename Method: Cambia el nombre de un método para hacerlo más claro.
- Preserve Whole Object: En lugar de pasar datos individuales, pasa el objeto completo.
- Introduce Parameter Object: Agrupa varios parámetros relacionados en un solo objeto.
- Parameterize Method: Usa un parámetro en lugar de duplicar un método similar.

MANIPULACIÓN DE LA GENERALIZACIÓN

Ayudan a mejorar las jerarquías de clases.

- Push Up / Down Field: Mueve campo/método a la superclase.
- Pull Up / Down Method: Mueve campo/método a las subclases.
- Pull Up Constructor Body: Mueve lógica de construcción común a la superclase.
- Extract Subclass / Superclass: Crea subclase/superclase para separar responsabilidades.
- Collapse Hierarchy: Elimina clases innecesarias fusionándolas.
- Replace Inheritance with Delegation: Usa delegación en lugar de herencia.
- Replace Delegation with Inheritance: Usa herencia cuando la delegación agrega complejidad innecesaria.

BIG REFACTORINGS

Implican cambios estructurales importantes en el diseño del sistema. A menudo requieren una revisión profunda del código y afectan a múltiples clases o componentes.

- Tease Apart Inheritance: Separar una jerarquía de herencia mal estructurada en jerarquías más claras o en composición.
- Convert Procedural Design to Objects: Transformar código basado en procedimientos a un diseño orientado a objetos.
- Separate Domain from Presentation: Dividir la lógica de negocio (modelo de dominio) de la capa de presentación (UI), por ejemplo, aplicando MVC.
- Extract Hierarchy: Crear una jerarquía de clases desde una clase con muchas responsabilidades o condicionales complejos.
- Refactoring to Patterns: Transforman estructuras de código existentes en implementaciones de patrones de diseño, como Strategy, State, Decorator, Composite, etc.
- Introduce Framework: Integrar un framework, lo que implica adaptar la arquitectura para aprovecharlo.

EXTRACT METHOD

Motivación:

- Métodos largos
- Métodos muy comentados
- Incrementar reuso
- Incrementar legibilidad

Mecánica:

1. Crear un nuevo método cuyo nombre explique su propósito
2. Copiar el código a extraer al nuevo método
3. Revisar las variables locales del original
4. Si alguna se usa sólo en el código extraído, mover su declaración
5. Revisar si alguna variable local es modificada por el código extraído. Si es solo una, tratar como query y asignar. Si hay más de una no se puede extraer
6. Pasar como parámetro las variables que el método nuevo lee
7. Compilar después de que se hayan tratado todas las variables

8. Reemplazar código en método original por llamada
9. Compilar

REPLACE TEMP WITH QUERY

Motivación: usar este refactoring:

- Para 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

Mecánica:

1. Verifica que la variable esté completamente definida antes de su uso y que el código que la calcula no produzca un valor diferente cada vez que se ejecuta.
2. Si la variable no es de solo lectura y puede convertirse en de solo lectura, hazla de solo lectura.
3. Prueba el código para asegurarte de que todo funciona correctamente.
4. Extrae la asignación de la variable a una función nueva.
5. Si el nombre de la variable y el de la función no pueden ser iguales, usa un nombre temporal para la función.
6. Asegúrate de que la función extraída no tenga efectos secundarios. Si los tiene, aplica el patrón Separate Query from Modifier.
7. Prueba nuevamente el código.
8. Usa Inline Variable para eliminar la variable temporal.

MOVE METHOD

Motivación:

- Un método está usando o usará muchos servicios que están definidos en una clase diferente a la suya.
- Sus llamadores están en otro lugar o se necesita llamarla desde un nuevo contexto.
- Una función auxiliar dentro de otra puede tener valor propio, por lo que moverla a un lugar más accesible es beneficioso.
- Un método de una clase puede ser más fácil de usar si se traslada a otra clase.

Mecánica:

1. Revisar las v.i. usadas por el método a mover. ¿Tiene sentido moverlas también? Si se encuentra una función llamada que también debería moverse, moverla primero. Comenzar con la función que tenga la menor dependencia de otras en el grupo para facilitar el proceso.
2. Revisar super y subclases por otras declaraciones del método. Si hay otras tal vez no se pueda mover
3. Crear un nuevo método en la clase target cuyo nombre explique su propósito
4. Copiar el código a mover al nuevo método. Ajustar lo que haga falta
5. Compilar la clase target
6. Determinar cómo referenciar al target desde el source
7. Reemplazar el método original por llamada a método en target
8. Compilar y testear
9. Decidir si remover el método original o mantenerlo como delegación

PULL UP METHOD

Motivación:

- Eliminar código duplicado con Pull Up Method para prevenir bugs, ya que la duplicación aumenta el riesgo de modificar una copia y olvidar la otra, aunque detectar duplicados puede ser difícil.

- Caso simple: métodos con el mismo cuerpo (probable copia y pega); comparar diferencias ayuda a identificar comportamientos no probados.
- Alternativa: si los métodos comparten un flujo similar, pero difieren en detalles, considera Form Template Method.
- Análisis de dependencias: examina los elementos usados por la función en su contexto; mueve funciones dependientes comenzando por las de menor dependencia.
- Finalización: usa Inline Variable (123) para eliminar la variable temporal tras mover la función.

Mecánica:

1. Asegurarse que los métodos sean idénticos. Si no, parametrizar
2. Si el selector del método es diferente en cada subclase, renombrar
3. Si el método llama a otro que no está en la superclase, declararlo como abstracto en la superclase
4. Si el método llama a un atributo declarado en las subclases, usar “Pull Up Field” o “Self Encapsulate Field” y declarar los getters abstractos en la superclase
5. Crear un nuevo método en la superclase, copiar el cuerpo de uno de los métodos a él, ajustar, compilar
6. Borrar el método de una de las subclases
7. Compilar y testear
8. Repetir desde 6 hasta que no quede en ninguna subclase

REPLACE CONDITIONAL WITH POLYMORPHISM

Motivación:

- La lógica condicional compleja es difícil de razonar, por lo que se busca estructurarla.
- Separar casos en clases con polimorfismo hace la lógica más clara.
- Aplicable cuando tipos (ej. libros, música) varían en manejo o hay switches repetidos.
- Usar una superclase para el caso base y subclases para variantes, destacando diferencias.
- El polimorfismo es poderoso, pero no debe usarse para toda lógica condicional; if/else o switch suelen bastar.

Mecánica:

1. Crear la jerarquía de clases con una superclase y subclases para cada caso del condicional.
2. Mover la lógica condicional a un método en la superclase, extrayéndola con Extract Function si no está aislada.
3. Por cada variante, crear un método en cada subclase que sobrescriba el método de la superclase.
4. Copiar al método de cada subclase la parte correspondiente del condicional y ajustarla según sea necesario.
5. Compilar y testear tras configurar cada método de subclase para verificar el comportamiento.
6. Eliminar la rama copiada del condicional en el método de la superclase.
7. Repetir los pasos 3 a 6 para cada rama del condicional hasta completar todas las variantes.
8. Hacer el método de la superclase abstracto si corresponde, o dejar un caso por defecto, y compilar y testear para confirmar el comportamiento polimórfico.

REPLACE MAGIC NUMBER WITH SYMBOLIC CONSTANT

Motivación:

- Los números mágicos son valores literales sin significado claro, dificultando la comprensión del código.
- Reemplazarlos con constantes simbólicas mejora la legibilidad y el mantenimiento.
- Reduce errores al centralizar el valor en una sola definición.
- Facilita cambios futuros, ya que basta con modificar la constante.
- Aclara el propósito del valor con un nombre descriptivo.

Mecánica:

1. Identificar el número mágico en el código.
2. Crear una constante con un nombre descriptivo que refleje el propósito del número.
3. Si el número mágico está en una lógica compleja, usar Extract Method para encapsular la lógica en una función y luego reemplazar el número con la constante dentro de ella.
4. Reemplazar todas las instancias del número mágico con la constante (o la llamada al método extraído, si aplica).
5. Compilar y testear para asegurar que el comportamiento del código no ha cambiado.
6. Si el número mágico aparece en múltiples contextos con diferentes significados, crear constantes separadas para cada caso.
7. Verificar si la constante puede agruparse en una enumeración o clase si representa un conjunto de valores relacionados.
8. Compilar y testear nuevamente tras cada cambio para confirmar la correctitud.

DECOMPOSE CONDITIONAL

Motivación:

- La lógica condicional compleja es una fuente común de complejidad en un programa.
- Las funciones largas con múltiples condiciones dificultan la lectura y comprensión del código.
- El código condicional muestra qué sucede, pero a menudo oculta por qué sucede.
- Descomponer el código en funciones con nombres que reflejen la intención aclara el propósito de cada bloque.
- Aplicar Extract Function al condicional y sus ramas resalta la condición y la razón del branching, mejorando la legibilidad.

Mecánica:

1. Identificar la lógica condicional (por ejemplo, un bloque if-else o switch) en el código.
2. Aplicar Extract Function (106) al bloque de la condición para crear una función con un nombre que describa la intención de la verificación.
3. Aplicar Extract Function (106) a cada rama del condicional, creando funciones separadas con nombres que reflejen el propósito de cada caso.
4. Reemplazar el código original de la condición y sus ramas con llamadas a las nuevas funciones.
5. Compilar y testear para asegurar que el comportamiento del código no ha cambiado.
6. Si las funciones extraídas se usan en múltiples lugares, verificar si es necesario refactorizar para evitar duplicación.
7. Compilar y testear nuevamente para confirmar la correctitud tras todos los cambios.

RENAME METHOD

Motivación:

- El nombre de un método debe reflejar claramente su propósito y funcionalidad para facilitar la comprensión del código.
- Un nombre poco claro o confuso dificulta el mantenimiento y puede llevar a errores de interpretación.
- Renombrar un método mejora la legibilidad, comunica mejor la intención y alinea el código con los cambios en los requisitos o el diseño.
- Un buen nombre reduce la necesidad de comentarios y hace que el código sea más auto explicativo.

Mecánica:

1. Verificar si el método tiene un nombre único en la jerarquía de clases; si no, identificar todas las declaraciones afectadas.
2. Crear un nuevo método con el nombre deseado, copiando el cuerpo del método original.

3. Asegurarse de que el nuevo método tenga la misma firma (parámetros y tipo de retorno) que el original.
4. Reemplazar todas las llamadas al método original con el nuevo nombre.
5. Compilar y testear para confirmar que el comportamiento no ha cambiado.
6. Si el método original está en una interfaz o superclase, actualizar todas las implementaciones o subclases correspondientes.
7. Eliminar el método original una vez que todas las referencias usen el nuevo nombre.
8. Compilar y testear nuevamente para verificar la correctitud del cambio.

REFACTORING TOOLS

Debemos aplicar refactoring en el contexto TDD (Test Driven Development), cuando se descubre código con mal olor (al menos debemos mejorarlo un poco, dependiendo del tiempo que lleve y de lo que esté haciendo), cuando no puedo entender el código (aprovechar el momento en que lo logro entender) y cuando encuentro una mejor manera de codificar algo.

La idea es que hay dos sombreros, uno que usamos cuando queremos añadir funcionalidad, explorar ideas y corregir bugs, y otro el cual es utilizado a la hora de hacer refactoring. Puedo cambiar de sombrero frecuentemente pero solo puedo usar un sombrero por vez.

AUTOMATIZACIÓN DEL REFACTORING

Refactorizar a mano es demasiado costoso: lleva tiempo y puede introducir errores. Contamos con herramientas de refactoring que cuenta con estas características:

- Potentes para realizar refactorings útiles.
- Restrictivas para preservar comportamiento del programa (uso de precondiciones).
- 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.
- Pueden ser demasiado conservativas (no realizan un refactoring si no pueden asegurar preservación de comportamiento) o asumir buenas técnicas de programación.

Ejemplos: SmallTalk RB, IntelliJ IDEA y Sonar qube.

PATRONES

- Un patrón es un par problema-solución.
- Los patrones tratan con problemas recurrentes y buenas soluciones (probadas) a esos problemas.
- La solución es suficientemente genérica para poder aplicarse de diferentes maneras.

ORGANIZACIÓN

		Propósito		
		De Creación	**Estructurales**	**De comportamiento**
Ámbito	**Clase**		Adapter (de clases)	Template Method
	Objeto	Builder	Adapter (de objetos) Composite Decorator Proxy	State Strategy

El propósito refleja qué hace un patrón. Los patrones pueden tener un propósito de creación, estructural o comportamiento. Los patrones de creación tienen que ver con el proceso de creación de objetos. Los patrones estructurales tratan con la composición de clases u objetos. Los de comportamiento caracterizan el modo en que las clases y objetos interactúan y se reparten la responsabilidad.

El segundo criterio, denominado ámbito, especifica si el patrón se aplica principalmente a clases o a objetos. Los patrones de clases se ocupan de las relaciones entre las clases y sus subclases. Los patrones de objetos tratan con las relaciones entre objetos, que pueden cambiarse en tiempo de ejecución y son más dinámicas.

Los patrones de creación de clases delegan alguna parte del proceso de creación de objetos en las subclases, mientras que los patrones de creación de objetos lo hacen en otro objeto. Los patrones estructurales de clases usan la herencia para componer clases, mientras que los de objetos describen formas de ensamblar objetos. Los patrones de comportamiento de clases usan la herencia para describir algoritmos y flujos de control, mientras que los de objetos describen cómo cooperan un grupo de objetos para realizar una tarea que ningún objeto puede llevar a cabo por sí solo.

- Intención: Responde a las siguientes cuestiones: ¿Qué hace este patrón de diseño? ¿En qué se basa? ¿Cuál es el problema concreto de diseño que resuelve?
- Aplicabilidad: ¿En qué situaciones se puede aplicar el patrón de diseño? ¿Qué ejemplos hay de malos diseños que el patrón puede resolver? ¿Cómo se puede reconocer dichas situaciones?
- Estructura: Una representación gráfica de las clases del patrón.
- Participantes: Las clases y objetos participantes en el patrón de diseño, junto con sus responsabilidades.
- Colaboraciones: Cómo colaboran los participantes para llevar a cabo sus responsabilidades.
- Consecuencias: ¿Cómo consigue el patrón sus objetivos? ¿Cuáles son las ventajas e inconvenientes y los resultados de usar el patrón? ¿Qué aspectos de la estructura del sistema se pueden modificar de forma independiente?
- Implementación: ¿Cuáles son las dificultades, trucos o técnicas que deberíamos tener en cuenta a la hora de aplicar el patrón?
- Relación con otros patrones: ¿Qué patrones de diseño están estrechamente relacionados con éste? ¿Cuáles son las principales diferencias? ¿Con qué otros patrones deberían usarse?

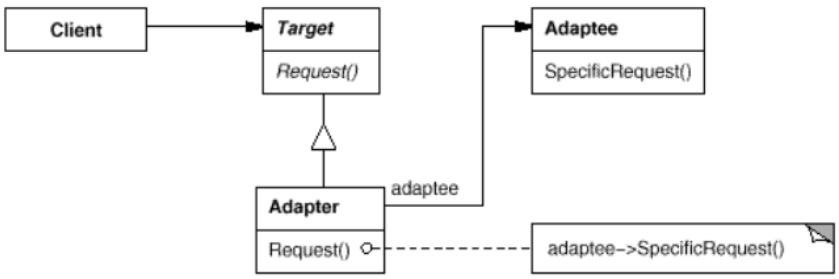
ADAPTER

Intención: 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 por tener interfaces incompatibles.

Aplicabilidad: Debería usarse el patrón Adapter cuando:

- Se quiere usar una clase existente y su interfaz no concuerda con la que necesita.
- Se quiere crear una clase reutilizable que coopere con clases no relacionadas o que no han sido previstas.
- Un adaptador de objetos puede adaptar la interfaz de su clase padre.

Estructura:



Participantes:

- Target: Define la interfaz específica que usa el cliente
- Client: Colabora con objetos que satisfacen la interfaz de Target
- Adaptee: Define una interfaz que precisa ser adaptada

- Adapter: Adapta la interfaz del Adaptee a la interfaz de Target

Colaboraciones:

- Los objetos Client llaman a las operaciones en la instancia del Adapter
- A su vez, el Adapter llama a las operaciones definidas en el Adaptee

Consecuencias:

- Una misma clase Adapter puede usarse para muchos Adaptees (el Adaptee y todas sus subclases)
- El Adapter puede agregar funcionalidad a los adaptados
- Se generan más objetos intermediarios

Implementación: pluggable adapters, parameterized adapters.

Relación con otros patrones: El patrón Decorator decora otro objeto sin cambiar su interfaz. Un decorador es por tanto más transparente a la aplicación que un adaptador. Como resultado, el patrón Decorator permite la composición recursiva, lo que no es posible con adaptadores puros.

El patrón Proxy define un representante o sustituto de otro objeto sin cambiar su interfaz.

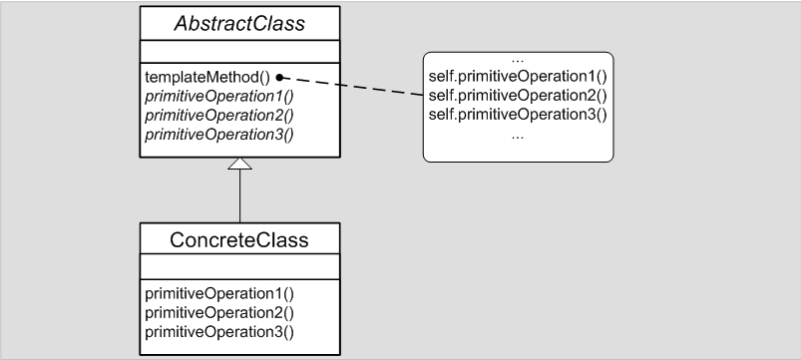
TEMPLATE METHOD

Intención: Definir el esqueleto de un algoritmo en un método, difiriendo algunos pasos a las subclases. Template Method permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.

Aplicabilidad: Usar Template Method:

- Para implementar las partes invariantes de un algoritmo una vez y dejar que las subclases implementen los aspectos que varían
- Para evitar duplicación de código entre subclases
- Para controlar las extensiones que pueden hacer las subclases

Estructura:



Participantes:

- AbstractClass
  - Implementa un método que contiene el esqueleto de un algoritmo (el template method). Ese método llama a operaciones primitivas, así como operaciones definidas en AbstractClass.
  - Declara operaciones primitivas abstractas que las subclases concretas deben definir para implementar los pasos de un algoritmo.
- ConcreteClass
  - Implementa operaciones primitivas que llevan a cabo los pasos específicos del algoritmo.

Colaboraciones:

- ConcreteClass confía en que la superclase implemente las partes invariantes del algoritmo

Consecuencias:

- Técnica fundamental de reuso de código
- Lleva a tener inversión de control (la superclase llama a las operaciones definidas en las subclases)
- El template method llama a dos tipos de operaciones:
    - operaciones primitivas (abstractas en AbstractClass y que las subclases tienen que definir)
    - operaciones concretas definidas en AbstractClass y que las subclases pueden redefinir si hace falta (hook methods)

Implementación: minimizar la cantidad de operaciones primitivas que las subclases deben redefinir. Muchas operaciones a sobrescribir = más trabajo tedioso para el cliente.

Relación con otros patrones: Los métodos de plantilla utilizan la herencia para modificar parte de un algoritmo. Las estrategias utilizan la delegación para modificar todo el algoritmo.

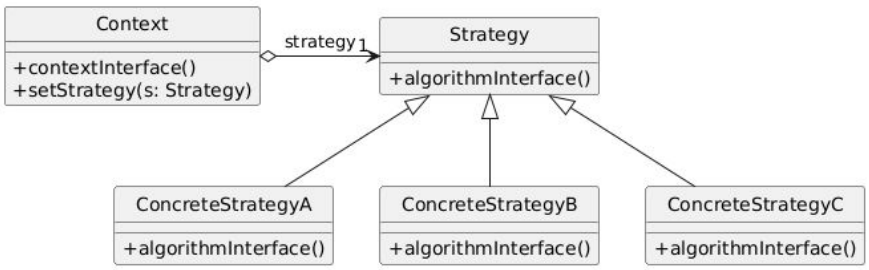
STRATEGY

Intención: 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. Permite cambiar (en forma dinámica), el algoritmo que se utiliza. Brindar flexibilidad para agregar nuevos algoritmos que llevan a cabo una función determinada.

Aplicabilidad: Uso el patrón Strategy cuando:

- Existen muchos algoritmos para llevar a cabo una tarea.
- No es deseable codificarlos todos en una clase y seleccionar cuál utilizar por medio de sentencias condicionales.
- Cada algoritmo utiliza información propia. Colocar esto en los clientes lleva a tener clases complejas y difíciles de mantener.
- Es necesario cambiar el algoritmo en forma dinámica, en tiempo de ejecución.

Estructura:



Participantes:

- Strategy: declara una interfaz común a todos los algoritmos permitidos. El Context usa esta interfaz para llamar al algoritmo definido por una ConcreteStrategy.
- ConcreteStrategy: implementa el algoritmo utilizando la interfaz de Strategy.
- Context
    - Se configura con un objeto ConcreteStrategy.
    - Mantiene una referencia a un objeto Strategy.
    - Puede definir una interfaz que permita a la Strategy acceder a sus datos.

Colaboraciones:

- Strategy y Context interactúan para implementar el algoritmo elegido. Un contexto puede pasar a la estrategia todos los datos requeridos por el algoritmo cada vez que se llama a éste. Otra alternativa es que el contexto se pase a sí mismo como argumento de las operaciones de Strategy. Eso permite a la estrategia hacer llamadas al contexto cuando sea necesario.
- Un contexto redirige peticiones de los clientes a su estrategia. Los clientes normalmente crean un objeto ConcreteStrategy, el cual pasan al contexto; por tanto, los clientes interactúan exclusivamente con el contexto. Suele haber una familia de clases ConcreteStrategy a elegir por el cliente.

Consecuencias:

- Puntos a favor:
    - Mejor solución que subclasificar el contexto, cuando se necesita cambiar dinámicamente.
    - Desacopla al contexto de los detalles de implementación de las estrategias.
    - ○ Se eliminan los condicionales.
- Puntos en contra:
    - La clase <<cliente>> debe conocer las diferentes estrategias para poder elegir.
    - Overhead en la comunicación entre contexto y estrategias.

Implementación:

- El contexto debe tener métodos en su protocolo que permitan cambiar la estrategia
- Parámetros entre el contexto y la estrategia: Hay que analizar qué datos se necesitan particularmente en cada caso

Relación con otros patrones: Se parece al Template Method porque ambos permiten variar partes de un algoritmo, pero mientras Template usa herencia, Strategy lo hace por composición. También se vincula con el patrón State, ya que ambos encapsulan comportamientos, aunque Strategy representa algoritmos intercambiables y State modela comportamientos según el estado del objeto.

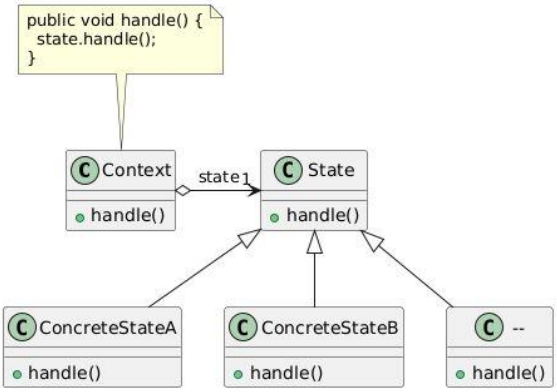
STATE

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

Aplicabilidad: Uso el patrón State cuando:

- El comportamiento de un objeto depende del estado en el que se encuentre.
- 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. El patrón State reemplaza el condicional por clases (es un uso inteligente del polimorfismo)
    - Desacoplar el estado interno del objeto es una jerarquía de clases.
    - Cada clase de la jerarquía representa un estado concreto en el que puede estar el objeto.
    - Todos los mensajes del objeto que dependan de su estado interno son delegados a las clases concretas de la jerarquía (polimorfismo).

Estructura:



Participantes:

- Context
  - Define la interfaz que conocen los clientes.
  - Mantiene una instancia de alguna clase de ConcreteState que define el estado corriente.
- State
  - Define la interfaz para encapsular el comportamiento de los estados de Context.
- ConcreteState
  - Cada subclase implementa el comportamiento respecto al estado específico.

Colaboraciones:

- Context delega las peticiones que dependen del estado en el objeto ConcreteState actual.
- Un contexto puede pasar a sí mismo como parámetro para que el objeto State maneje la petición. Esto permite al objeto State acceder al contexto si fuera necesario.
- Context es la interfaz principal para los clientes. Los clientes pueden configurar un contexto con objetos State. Una vez que está configurado el contexto, sus clientes ya no tienen que tratar con los objetos State directamente.
- Cualquiera de las subclases de Context o de ConcreteState pueden decidir qué estado sigue a otro y bajo qué circunstancias.

Consecuencias:

- Puntos a favor:
  - Localiza el comportamiento relacionado con cada estado.
  - Las transiciones entre estados son explícitas.
  - En el caso que los estados no tengan variables de instancia pueden ser compartidos.
- Puntos en contra:
  - En general hay bastante acoplamiento entre las subclases de State porque la transición de estados se hace entre ellas, por lo que deben conocerse entre sí.

Implementación:

- Se debe decidir quién maneja las transiciones entre estados: el Contexto o los Estados.
- Hay dos formas de gestionar los objetos Estado: Crear todos al inicio y reutilizarlos, o crear y destruir según se necesiten.
- Se pueden usar clases anidadas si los estados son específicos del contexto y no se reutilizan fuera de él.

Relación con otros patrones:

State	Strategy

El comportamiento de un objeto depende del estado en el que se encuentre	Necesito uno de diferentes algoritmos opcionales para realizar una misma tarea
El patrón State es útil para una clase que debe realizar transiciones entre estados fácilmente	El patrón Strategy es útil para permitir que una clase delegue la ejecución de un algoritmo a una instancia de una familia de estrategias
En State, los diferentes estados: son internos al contexto, no los eligen las clases clientes; la transición se realiza entre los estados mismos y cada State puede definir muchos mensajes	En Strategy, las diferentes estrategias: son conocidas desde afuera del contexto, por las clases clientes del contexto; el contexto del Strategy debe contener un mensaje público para cambiar el ConcreteStrategy
El estado es privado del objeto, ningún otro objeto sabe de él	EL Strategy suele setearse por el cliente, que debe conocer las posibles estrategias concretas
Los states concretos se conocen entre sí. Saben a cuál estado se debe pasar en respuesta a algún mensaje	Los strategies concretos no se conocen entre sí

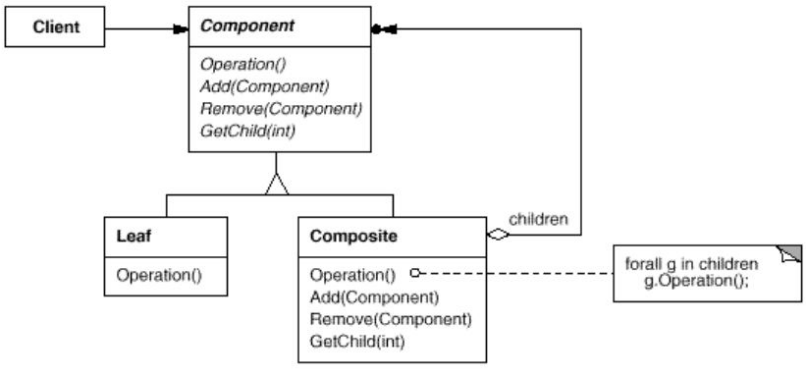
COMPOSITE

Intención: Combina 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.

Aplicabilidad: Use el patrón Composite cuando:

- Quiere representar jerarquías parte-todo de objetos.
- Quiere que los objetos "clientes" puedan ignorar las diferencias entre composiciones y objetos individuales. Los clientes tratarán a los objetos atómicos y compuestos uniformemente.

Estructura:



Participantes:

- Component:
  - Declara la interfaz para los objetos de la composición.
  - Implementa comportamientos default para la interfaz común a todas las clases.
  - Declara la interfaz para definir y acceder "partes de la composición".
- Leaf:
  - Las hojas no tienen sub-árboles.
  - Define el comportamiento de objetos primitivos en la composición.
- Composite:
  - Define el comportamiento para componentes complejos.
  - Almacena componentes hijos.
  - Implementa operaciones para manejar el sub-árboles.
- Client:
  - Manipula objetos en la composición a través de la interfaz Component.

Colaboraciones: Los Clients usan la interfaz de la clase Component para interactuar con los objetos de la estructura compuesta. Si el recipente es una Leaf, la petición se trata correctamente. Si es un Component, normalmente redirige las peticiones a sus componentes hijos, posiblemente realizando operaciones adicionales antes o después.

Consecuencias:

- (+) Define "jerarquías" de objetos primitivos y compuestos.
- (+) Los objetos primitivos pueden componerse en objetos complejos, los que a su vez pueden componerse recursivamente.
- (+) Simplifica los objetos cliente. Los clientes usualmente no saben (y no deberían preocuparse) acerca de si están manejando un compuesto o un simple.
- (+) Hace más fácil el agregado de nuevos tipos de componentes porque los clientes no tienen que cambiar cuando aparecen nuevas clases componentes.
- (-) Debe ser "customizado" con reglas de composición (si fuera necesario).

Implementación:

- Referencias explícitas a la raíz de una hoja
- Maximizar el protocolo de la clase/interfaz Component
- Orden de las hojas
- Borrado de componentes
- Búsqueda de componentes(fetch) por critérios
- Diferentes estructuras de datos para guardar componentes

Relación con otros patrones: El patrón Decorator suele juntarse junto con el Composite. Cuando se usan juntos decoradores y compuestos, normalmente ambos tendrán una clase padre común. Por tanto, los decoradores tendrán que admitir la interfaz Component con operaciones como Añadir, Eliminar y ObtenerHijo.

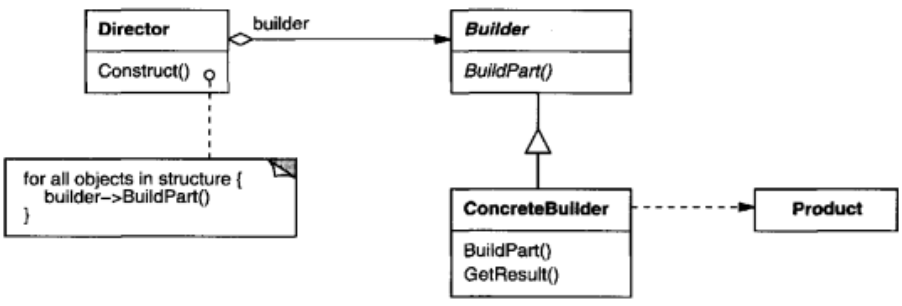
BUILDER

Intención: Separa la construcción de un objeto complejo de su representación (implementación), de forma que el mismo proceso de construcción pueda crear diferentes representaciones (implementaciones).

Aplicabilidad: Úsese el patrón Builder cuando:

- El algoritmo para crear un objeto complejo debiera 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.

Estructura:



Participantes:

- Builder: especifica una interface abstracta para crear partes de un Producto
- Concrete Builder: construye y ensambla partes del producto.
    - Guarda referencia al producto en construcción
- Director: conoce los pasos para construir el objeto
    - Utiliza el Builder para construir las partes que va ensamblando
    - En lugar de pasos fijos puede seguir una "especificación"
- Product: es el objeto complejo a ser construido

Colaboraciones:

- Client crea el objeto Director y lo configura con el objeto Builder deseado.
- Director notifica al constructor cada vez que hay que construir una parte de un producto.
- Builder maneja las peticiones del director y las añade al producto.
- El Client obtiene el producto del constructor.

Consecuencias:

- Permite variar la representación interna de un producto: El patrón Builder permite construir diferentes representaciones de un mismo producto sin modificar el proceso de construcción. Solo hace falta definir un nuevo tipo de constructor.

- Aísla el código de construcción y representación: Se mejora la modularidad ocultando cómo se construyen y representan los objetos. El cliente no necesita conocer los detalles internos del producto.
- Proporciona un control más fino sobre el proceso de construcción: A diferencia de otros patrones de creación, Builder permite construir el producto paso a paso, con mayor control sobre cada etapa del proceso, lo que también permite un mayor control sobre su estructura interna.

Implementación: Los constructores arman productos paso a paso, por lo que la interfaz debe ser lo suficientemente general para adaptarse a distintos tipos de productos y procesos de construcción. No se recomienda usar clases abstractas comunes para todos los productos, ya que sus estructuras internas pueden variar mucho.

Relación con otros patrones: Muchas veces lo que construye el constructor es un Composite.

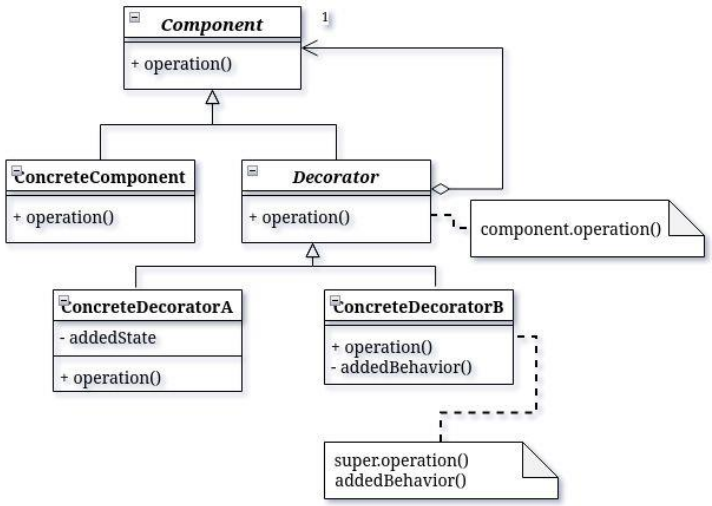
DECORATOR

Intención: Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

Aplicabilidad: Usar Decorator para:

- Agregar responsabilidades a objetos individualmente y en forma transparente (sin afectar a otros objetos)
- Quitar responsabilidades dinámicamente
- Cuando subclasificar es impráctico

Estructura:



Participantes:

- Component: define la interfaz para objetos a los que se puede añadir responsabilidades dinámicamente.
- ConcreteComponent: define un objeto al que se pueden añadir responsabilidades adicionales.
- Decorator: mantiene una referencia a un objeto Component y define una interfaz que se ajusta a la interfaz del Component.
- ConcreteDecorator: añade responsabilidades al componente.

Colaboraciones:

- El Decorator redirige peticiones a su objeto Component. Opcionalmente puede realizar operaciones adicionales antes y después de reenviar la petición.

Consecuencias:

- Puntos a favor:
  - Permite mayor flexibilidad que la herencia.
  - Permite agregar funcionalidad incrementalmente.
- Puntos en contra:
  - Mayor cantidad de objetos, complejo para depurar.

Implementación:

- Misma interface entre componente y decorador. Tanto la imagen base como los decoradores implementan la misma interfaz.
- No hay necesidad de la clase Decorator abstracta, si se tiene un solo decorador.
- Cambiar el “skin” vs cambiar sus “guts”
  - Decorator puede verse como una “piel” que modifica el comportamiento externo, no la estructura interna (los “guts”).
  - Es decir: no cambiamos su estructura interna

Relación con otros patrones:

Decorator	Adapter
Ambos patrones “decoran” el objeto para cambiarlo	
Decorator preserva la interface del objeto para el cliente, sólo cambia las responsabilidades de un objeto	Adapter convierte la interface del objeto para el cliente
Decorators pueden y suelen anidarse	Adapters no se anidan

Decorator	Composite
Mantiene una estructura con sólo un siguiente	Mantiene una estructura de tipo árbol, un composite usualmente se compone de varias partes
El propósito es agregar funcionalidad dinámicamente	El propósito es componer objetos y tratarlos de manera uniforme

Decorator	Strategy
Propósito: permitir que un objeto cambie su funcionalidad dinámicamente (agregando o cambiando el algoritmo que utiliza)	
El Decorator cambia el algoritmo por fuera del objeto	El Strategy cambia el algoritmo por dentro del objeto

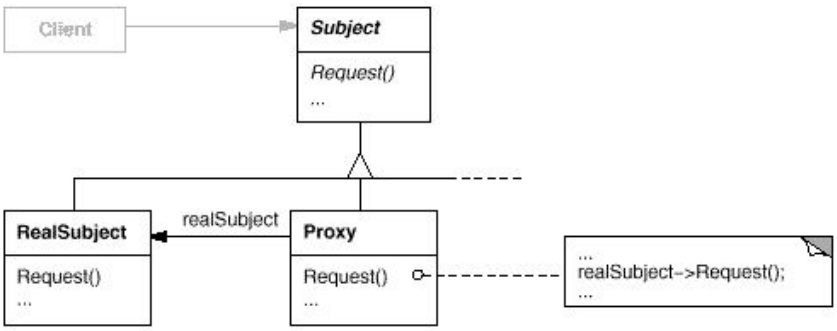
PROXY

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

Aplicabilidad: cuando se necesita una referencia más flexible hacia un objeto.

- Virtual proxy: demorar la construcción de un objeto hasta que sea realmente necesario, cuando sea poco eficiente acceder al objeto real.
- Protection proxy: Restringir el acceso a un objeto por seguridad.
- Remote proxy: Representar un objeto remoto en el espacio de memoria local. Es la forma de implementar objetos distribuidos. Estos proxies se ocupan de la comunicación con el objeto remoto, y de serializar/deserializar los mensajes y resultados.

Estructura:



Participantes:

- Proxy:
  - Mantiene una referencia que permite acceder al objeto real. El proxy puede referirse a un Subject en caso de que las interfaces de RealSubject y Subject sean la misma.
  - Proporciona una interfaz idéntica a la de Subject, de manera que un proxy pueda ser sustituido por el sujeto real.
  - Controla el acceso al sujeto real, y puede ser responsable de su creación y borrado.
    - Proxy remoto: Envía peticiones a un objeto real ubicado en otra máquina o dirección.
    - Proxy virtual: Retrasa la creación o el acceso al objeto real, guardando información previa.
    - Proxy de protección: Verifica si el usuario tiene permisos para acceder al objeto.
- Subject:

    - Define la interfaz común para el RealSubject y el Proxy, de modo que pueda usarse un Proxy en cualquier sitio en el que se espere un RealSubject.
- RealSubject:
    - Define el objeto real representado.

Colaboraciones:

- El Proxy redirige peticiones al RealSubject cuando sea necesario, dependiendo del tipo de proxy.

Consecuencias:

- Puntos a favor: Indirección en el acceso al objeto: El patrón Proxy introduce un nivel de indirección que permite controlar el acceso al objeto real, lo que ofrece flexibilidad en su uso.
- Puntos en contras: Complejidad adicional.

Implementación:

- Si se usa una interfaz o clase base, no hace falta una clase proxy por cada clase concreta.
- Esto mejora la reutilización, salvo que el proxy deba crear instancias del sujeto real (como el proxy virtual), donde sí necesita conocer la clase concreta.

Relación con otros patrones:

Proxy	Adapter	Decorator
Todos son patrones estructurales, todos con diagramas de objetos similares, y se los llama “wrappers”		
Proporciona la misma interfaz que su sujeto	Proporciona una interfaz diferente al objeto que adapta	Mantiene la misma interfaz que el objeto que decora
Controla el acceso a un objeto	Permite que clases con interfaces incompatibles colaboren	Agrega dinámicamente una o más responsabilidades a un objeto

Un proxy de protección podría implementarse exactamente como un decorador. Por otro lado, un proxy remoto no contendrá una referencia directa a su sujeto real sino sólo una referencia indirecta, como “un ID de máquina y la dirección local en dicha máquina”. Un proxy virtual empezará temiendo una referencia indirecta como un nombre de fichero, pero podrá al final obtener y utilizar una referencia directa.

REFACTORING TO PATTERNS

El refactoring nos permite introducir patrones recién cuando el software que construimos evoluciona al punto que son necesarios. No necesitamos adivinar o prever de antemano.

FORM TEMPLATE METHOD

Propósito: Definir el esqueleto de un algoritmo en una operación, y diferir algunos pasos a las subclases. Template Method permite que las subclases redefinan algunos pasos de un algoritmo sin cambiar la estructura del algoritmo. Template Method reduce o elimina el código repetido en métodos similares de las subclases en una jerarquía.

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.

Mecánica:

1. Encontrar el método que es similar en todas las subclases y extraer sus partes en: métodos idénticos (misma signatura y cuerpo en las subclases) o métodos únicos (distinta signatura 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.

Pros:

- Elimina código duplicado en las subclases moviendo el comportamiento invariante a la superclase.
- Simplifica y comunica efectivamente los pasos de un algoritmo genérico.
- Permite que las subclases adapten fácilmente un algoritmo.

Contras:

- Complica el diseño cuando las subclases deben implementar muchos métodos para sustanciar el algoritmo

EXTRACT ADAPTER

Propósito: Introducir una clase Adapter para conectar una clase existente con una interfaz esperada por el cliente, sin modificar la clase original.

Motivación: Cuando una clase no cumple con la interfaz que otro componente necesita, o cuando se quiere desacoplar el sistema de detalles concretos de una librería o API externa, puede resultar útil introducir un Adapter que actúe como intermediario. Esto permite que el sistema dependa de una interfaz propia, no de implementaciones concretas.

Solución: Crear una clase Adapter que implementa la interfaz esperada y delega las llamadas al objeto adaptado, traduciendo si es necesario entre los formatos o nombres de métodos.

Mecánica:

1. Identificar la interfaz que el cliente necesita (si no existe, definirla).
2. Crear una nueva clase Adapter que implemente esa interfaz.
3. Agregar una instancia del objeto adaptado como atributo del Adapter.
4. Implementar los métodos de la interfaz esperada delegando en el objeto adaptado.
5. Reemplazar las referencias directas al objeto adaptado por el Adapter.
6. Aplicar “Introduce Interface” o “Move Method” si es necesario para facilitar el desacople.

Pros:

- Aísla dependencias con librerías o clases externas.
- Permite reutilizar clases que no encajan directamente con lo requerido.

Contras:

- Añade una clase más al sistema.
- Puede ocultar la complejidad real del objeto adaptado.

REPLACE IMPLICIT TREE WITH COMPOSITE

Propósito: Transformar una estructura de árbol implícita en una estructura explícita basada en el patrón Composite para unificar el tratamiento de nodos simples y compuestos.

Motivación: A veces las estructuras de árbol están representadas de forma implícita. Esto dificulta la extensión, reutilización y navegación del árbol. Al reemplazar la estructura implícita con el patrón Composite, se obtiene una jerarquía clara y polimórfica, donde nodos y ramas pueden ser tratados uniformemente.

Solución: Aplicar el patrón Composite: definir una clase componente común (interfaz o abstracta) y crear implementaciones concretas para nodos hoja y compuestos. Las operaciones se definen en la clase base y se implementan en cada tipo de componente según su naturaleza.

Mecánica:

1. Identificar cómo está representado actualmente el árbol.
2. Definir una clase o interfaz común Component con las operaciones que deben ser soportadas por todos los nodos.
3. Crear subclases concretas: una para las hojas (Leaf) y otra para los compuestos (Composite).
4. Mover la lógica desde la estructura implícita a estas nuevas clases.
5. Reemplazar el uso de estructuras ad hoc por instancias de estas clases.
6. Refactorizar métodos que operan sobre el árbol para que trabajen con polimorfismo (sin condicionales de tipo).

Pros:

- Permite tratar uniformemente nodos simples y compuestos.
- Facilita agregar nuevos tipos de nodos o nuevas operaciones.
- Mejora la legibilidad y extensibilidad del árbol.

Contras:

- Aumenta el número de clases involucradas.
- Puede ser excesivo para árboles simples o con lógica poco cambiante.
- Introduce un diseño más rígido si la estructura cambia frecuentemente.

REPLACE CONDITIONAL LOGIC WITH STRATEGY

Propósito: Permitir definir una familia de algoritmos, encapsular cada uno y hacerlos intercambiables. Strategy permite que el algoritmo varíe independientemente del cliente que lo utiliza.

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.

Mecánica:

1. Crear una clase Strategy.
2. Aplicar “Move Method” para mover el cálculo con los condicionales del contexto al strategy.
    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. - 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.

Pros:

- Elimina código condicional complejo y repetido.
- Aumenta la flexibilidad: se pueden intercambiar algoritmos en tiempo de ejecución.
- Hace explícitas las variantes de comportamiento.

Contras:

- Aumenta la cantidad de clases (una por estrategia).
- Puede ser excesivo si sólo hay una o dos variantes y no se prevé que aumenten.
- La creación y gestión de estrategias puede ser más compleja si hay dependencias o se requiere compartir estado con el contexto.

REPLACE STATE-ALTERING CONDITIONALS WITH STATE

Propósito: Reemplazar condicionales complejos que alteran el estado interno de un objeto por un diseño basado en objetos (patrón State), permitiendo que cada estado se represente como una clase y maneje su propio comportamiento.

Motivación: Las expresiones condicionales que controlan las transiciones de estado de un objeto son complejas. Obtener una mejor visualización con una mirada global, de las transiciones entre estados.

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

Mecánica:

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.

Pros:

- Mejora la claridad del flujo de estados del objeto.
- Facilita la extensión de nuevos estados y transiciones.
- Elimina condicionales repetitivos y difíciles de mantener.
- Permite validar transiciones de manera más precisa y estructurada.

Contras:

- Aumenta el número de clases (una por estado).
- Puede resultar excesivo si hay pocos estados o transiciones simples.
- Si las transiciones son muy dinámicas o dependen de datos externos, el diseño puede volverse rígido.
- Requiere mayor esfuerzo inicial de diseño y testeo.

MOVE EMBELISHMENT TO DECORATOR

Propósito: Separar responsabilidades de embellecimiento (adición de comportamiento) de un objeto sin alterar su estructura base, mediante el patrón Decorator.

Motivación: A veces una clase tiene responsabilidades adicionales que decoran su comportamiento principal (como logging, formateo, trazas, etc.). Estas responsabilidades pueden hacer que la clase principal crezca innecesariamente y se vuelva difícil de mantener. Al mover ese comportamiento a decoradores, se obtiene una estructura más flexible, reutilizable y con menor acoplamiento.

Solución: Aplicar el patrón Decorator, extrayendo las funcionalidades adicionales (embellecedoras) en clases separadas que implementan la misma interfaz o heredan de la misma superclase que el objeto decorado. El decorador mantiene una referencia al objeto decorado y delega en él, agregando o modificando el comportamiento según sea necesario.

Mecánica:

1. Identificar la superclase (o interfaz) del objeto a decorar (la "Component" del patrón Decorator). Si no existe, crearla para unificar la abstracción.
2. Aplicar “Replace Conditional Logic with Polymorphism”: extraer el comportamiento de embellecimiento a una subclase (el Decorator), eliminando condiciones si las hubiera.
3. Aplicar “Replace Inheritance with Delegation”: en lugar de heredar directamente, hacer que el decorador contenga una instancia del componente decorado y le delegue.
4. Aplicar “Extract Parameter” en el decorador: para que pueda recibir el componente decorado como parámetro y mantener la composición.
5. Repetir si hay múltiples decoradores posibles (anidables o combinables).

Pros:

- Separa responsabilidades y mantiene la clase base más simple.
- Permite composición de comportamientos de forma flexible (incluso en tiempo de ejecución).
- Reduce la duplicación cuando varios objetos requieren decoraciones similares.

Contras:

- Puede introducir muchas clases pequeñas, lo que aumenta la complejidad estructural del sistema.
- Es más difícil de rastrear el comportamiento total de un objeto decorado cuando hay muchos decoradores anidados.
- El orden de aplicación de decoradores puede ser difícil de manejar y crítico para el resultado final.

INTRODUCE NULL OBJECT

Propósito: Proporcionar un sustituto para otro objeto que comparte la misma interfaz, pero no hace nada. El objeto nulo encapsula las decisiones de implementación sobre cómo "no hacer nada" y oculta esos detalles a sus colaboradores.

Fuerzas del problema:

- Hay lógica condicional que testea por null en muchos métodos de mi clase
- La lógica condicional testea la existencia o no de un colaborador para delegarle una tarea
- Quisiéramos poder tratar al null como un objeto más, que simplemente no hace nada al recibir el
- mensaje
- De esta forma también evitamos el condicional y los errores o excepciones que se producen al olvidarnos de chequear por null

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

Aplicabilidad:

- Un objeto tiene un colaborador, que algunas veces no hace nada
- 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

Mecánica:

1. Crear el *null object* 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 *null check* 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 *null object* para que implemente el comportamiento alternativo. Compilar
3. Repetir el paso 2 para todos los *null checks* 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 *null object* lo antes posible. Compilar
5. Para cada lugar elegido en el paso 4, eliminar los *null checks* asociados

Consecuencias:

- Define jerarquías de clase que consisten de objetos reales y null objects. En cualquier lugar que el cliente espera un objeto real también puede tomar un null object
- Hace el código del cliente más simple. Los clientes pueden tratar a sus colaboradores de manera uniforme
- Encapsula el comportamiento de hacer nada en un objeto, que puede ser reusado por múltiples clientes
- Requiere crear una clase NullObject cada vez que en una jerarquía necesitemos el comportamiento nulo
- Podría ser difícil de implementar si la usan varios clientes y no hay un acuerdo de cual debería ser el comportamiento nulo

DEUDA TÉCNICA

- Concepto que introdujo Ward Cunningham para explicar a los stakeholders la necesidad de refactoring
- Está bien tomar prestado o endeudarse cuando *estratégicamente* conviene entregar código rápido para ganar feedback y aprender, y el código refleja nuestro entendimiento actual del problema
- El peligro es cuando esa deuda no se paga
- Cada minuto dedicado a código que acarrea deuda cuenta como interés. Y hasta que no se aplique refactoring para pagar esa deuda, seguiremos acumulando interés
- El código que escribimos debería ser lo suficientemente limpio para refactorizarlo fácilmente a medida que lo entendemos mejor
- Capital de la deuda: costo de remediar los problemas de diseño (costo del refactoring)
- Interés de la deuda: costo adicional o esfuerzo extra por acarrear un mal diseño
- Varias IDEs tienen la capacidad de cuantificar y visualizar el capital de la TD

TEST DOUBLES

TEST DE UNIDAD

- Testeo de la mínima unidad de ejecución/funcionalidad.
- En OOP, la mínima unidad es un método.
- Objetivo: aislar cada parte de un programa y mostrar que funciona correctamente.
- Cada test confirma que un método produce el output esperado ante un input conocido.
- Es como un contrato escrito de lo que esa unidad tiene que satisfacer.

TEST DOUBLE

Problema general

- El SUT (System Under Test) depende de un módulo/objeto externo.
- El módulo/objeto real no se puede usar en el entorno de pruebas.
- Las pruebas pueden necesitar:
    - Configuraciones válidas del sistema.
    - Salidas indirectas del sistema.
    - Lógica del sistema.
    - Protocolos de comunicación.

Qué es un Test Double

- Es una maqueta (objeto simulado) del objeto/módulo requerido.
- Debe ser polimórfico respecto al objeto real.
- Se utiliza según sea necesario en la prueba.
- Test Double se considera un lenguaje de patrones utilizado en pruebas.

Rangos de implementación:

- Desde un cascarrón vacío hasta una simulación completa.
- Se aplican distintos patrones según el tipo de prueba a realizar.

Escenarios en donde es necesario usar TestDoubles:

- Cuando un objeto del cual se depende no está desarrollado aún.
- Es necesario probar la funcionalidad de un objeto independientemente del comportamiento de otro objeto, cuyos mensajes son invocados por la funcionalidad siendo testeada.

Patrones (Cascarón vacío → Simulación)

- Test Stub: cascarón vacío. Sirve para que el SUT envíe los mensajes esperados
- Test Spy: Test Stub + registro de mensajes recibidos
- Mock Object: Test Stub + comprueba la validez de los mensajes recibidos
- Fake Object: imitación. Simula el comportamiento del módulo real (protocolos, tiempos de respuesta, etc)

FRAMEWORKS

Un framework es una aplicación “semi-completa”, “reusable”, que puede ser especializada para producir aplicaciones a medida… un conjunto de clases concretas y abstractas, relacionadas (por herencia, conocimiento, envío de mensajes) para proveer una arquitectura reusable que implementa una familia de aplicaciones (relacionadas)...

- Framework controla la ejecución (execution thread)
    - Inversión de control
- Cookbook: reglas de uso
    - Instanciación: implementar aplicación
        - White-box: thread incompleto
        - Black-box: thread configurable
    - Extensión: agregar opciones

SERVIDORES TCP

Es un programa que implementa una de las partes de la arquitectura Cliente/Servidor.

- 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:

- EchoServer: Recibe un mensaje de texto y Responde el mensaje que el cliente envió
- SMTPServer: emails
- DayTimeServer: fecha actual
- HTTPServer: envío y recepción de documentos web

FLUJO DE CONTROL

1. Crear un socket (IP_address + port)
2. Escucha en el socket
3. Aceptar al cliente ⇒ Sesión
4. Recibir un mensaje
    a. Condición ⇒ *Responder*
    b. !Condición ⇒ Cerrar sesión e ir a paso 2

La idea es reusar el "flujo de control". Este está incompleto (generalmente no compila), y debemos hacer que "responda", es decir "enganchar" (usando *hooks obligatorios*) la funcionalidad.

INVERSIÓN DE CONTROL

Es un principio de diseño donde el control del flujo de un programa se delega a un componente externo, como un framework, en lugar de que el código de la aplicación lo gestione directamente. Esto significa que el framework asume la responsabilidad de crear, configurar y gestionar los objetos (o componentes) y sus interacciones, en lugar de que el desarrollador lo haga manualmente.

HOTSPOTS VS FROZENSPOT

FrozenSpot: aspecto del framework que afecta a todas las instanciaciones y que no se puede modificar (marca indeleble)

- Comportamiento invariable en un framework orientado a objetos

HotSpot: estructura en el código que permite modificar el comportamiento del framework, para instanciar y para extender. Hooks Methods

- Punto de variabilidad de un framework orientado a objetos

FRAMEWORK DE CAJA BLANCA (WHITE BOX FRAMEWORK)

- La instanciación hereda y completa el loop de control agregando código
    - Ejercitando un hotspot con herencia
    - Modificando código fuente del framework
    - Es posible que requiera agregar métodos a clases del framework
- Demanda conocimiento del código del framework ⇒ es una Caja Blanca
- Loop de control suele ser Template Method

FRAMEWORK DE CAJA NEGRA (BLACK BOX FRAMEWORK)

- La instancia se obtiene mediante composición y completa el loop de control agregando código
    - Usando interfaces o clases proporcionadas por el framework

    - Configurando componentes sin modificar el código fuente
    - No requiere herencia directa ni cambios en el framework
- Demanda conocimiento limitado del código del framework ⇒ es una Caja Negra
- Loop de control suele ser manejado por un contenedor o configurador (no Template Method)

PLANTILLAS Y GANCHOS

- Las plantillas y los ganchos se pueden utilizar en un framework para separar lo que se mantiene constante (frozenspot) de lo que varía (hotspots).
- 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 (como el clásico patrón Método Plantilla). Si uso composición, un objeto implementa la plantilla y delega la implementación de los ganchos en sus partes.

PLANTILLAS Y HERENCIA

El patrón de diseño "método plantilla" documentado en el libro "Patrones de diseño" (de Gamma y otros) propone abstraer los pasos canónicos de la operación en un método implementado en la clase abstracta y redefinir los pasos variables en sus subclases. De esta forma, la herencia es el mecanismo principal para separar lo que es constante, de lo que varía. El método plantilla es parte del frozenspot mientras que las operaciones abstractas (o los ganchos) dan lugar a la implementación de los hotspots.

- Es más simple para casos con pocas alternativas y/o pocas combinaciones
- Al implementar los métodos gancho puedo utilizar las variables de instancia y todo el comportamiento heredado de la clase abstracta
- Si hay muchas variantes o combinaciones, empiezo a tener muchas clases y duplicación de código

PLANTILLAS Y COMPOSICIÓN

La otra estrategia es separar lo constante de lo que cambia componiendo objetos. La separación de lo que es constante y lo que varía no solo se separa en métodos diferentes, sino que también se encapsula en objetos diferentes.

El método plantilla ya no envía mensajes al objeto mismo (como lo hacía en el caso de herencia) sino que delega las operaciones en sus partes. Entonces, la plantilla está en el robot, y los ganchos en las partes.

- Evita la duplicación de código y el creciente número de clases cuando existen muchas alternativas y combinaciones posibles
- Al implementar los métodos gancho debo pasar como parámetro todo lo que necesiten - no tienen acceso a las variables de instancia del objeto
- Puedo cambiar el comportamiento en tiempo de ejecución sin mayor dificultad

SIMPLETHREADTCPSERVER (WHITEBOX)

Extensión

- No existen puntos de extensión

Ejemplos

- VoidServer
- EchoServer

Instanciación (Cookbook)

1. Subclasificar SimpleThreadTCPServer
    a. Debe implementar Main(String[])
        i. crear una instancia
        ii. enviar método startLoop(String[])
    b. Debe implementar handleMessage(String)
    c. Métodos que podría implementar
        i. acceptAndDisplaySocket(ServerSocket).
        ii. checkArguments(String[])
        iii. displayAndExit(int)
        iv. displaySocketData(Socket)
        v. displayUsage()
        vi. displaySocketInformation()
        vii. displayWarning()

```
12      public final void startLoop(String[] args) {
13          checkArguments(args);
14
15          int portNumber = Integer.parseInt(args[0]);
16
17
18          try (ServerSocket serverSocket = new ServerSocket(portNumber)) {
19              displaySocketInformation(portNumber);
20              while (true) {
21                  Socket clientSocket = acceptAndDisplaySocket(serverSocket);
22                  handleClient(clientSocket);
23              }
24          } catch (IOException e) {
25              displayAndExit(portNumber);
26          }
27      }
```

Todos los servidores pueden redefinir: checkArguments(), displaySocketInformation() y displayAndExit()

SIMPLETHREADTCPSERVER.HANDLECLIENT (SOCKET)

```
62      private final void handleClient(Socket clientSocket) {
63
64          try (
65                  PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
66                  BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));) {
67              String inputLine;
68              while ((inputLine = in.readLine()) != null) {
69                  System.out.println("Received message: " + inputLine + " from "
70                          + clientSocket.getInetAddress().getHostAddress() + ":" + clientSocket.getPort());
71                  handleMessage(inputLine, out);
72                  if (inputLine.equalsIgnoreCase("")) {
73                      break; // Client requested to close the connection
74                  }
75              }
76              System.out.println("Connection closed with " + clientSocket.getInetAddress().getHostAddress() + ":"
77                      + clientSocket.getPort());
78          } catch (IOException e) {
79              System.err.println("Problem with communication with client: " + e.getMessage());
80          } finally {
81              try {
82                  clientSocket.close();
83              } catch (IOException e) {
84                  System.err.println("Error closing socket: " + e.getMessage());
85              }
86          }
87      }
88  }
```

Todos los servidores deberán definir: handleMessage(String, PrintWriter)

ECHOSERVER.JAVA

SingleThreadTCPServer (hotspot herencia)

```java
import java.io.PrintWriter;

public class EchoServer extends SingleThreadTCPServer {

    public void handleMessage(String message, PrintWriter out) {
        out.println(message);
    }

    Run | Debug
    public static void main(String[] args) {

        new EchoServer().startLoop(args);

    }
}
```

1. startLoop(args) arranca el loop de control del framework
    a. El Framework toma el control (inversión de control)
    b. Ahora está completo porque se implementa handleMessage(...)
2. handleMessage(...) es invocado desde alguna parte del framework
    a. Completa el loop de control
    b. "Hollywood Principle": no nos llames, nosotros te contactaremos
    c. EchoServer hereda el loop de control no existe encapsulamiento ⇒ es una Caja Blanca
    d. La ejecución del server depende de la correcta implementación de handleMessage() porque es parte del loop (incompleto) de control

TCP.SERVER.REPLY (BLACKBOX)

Cookbook

1. En un objeto "contexto"
    a. instanciar un MessageHandler (puede ser Strategy o Command)
        i. Echo,
        ii. Void
    b. Instanciar ConnectionHandler con el MessageHandler
        i. SimpleConnectionHandler
        ii. MultiConnectionHandler
    c. Instanciar TCPControlLoop con ConnectionHandler
    d. Enviar método startLoop() al TCPControlLoop

```java
import tcp.server.reply.*;

public class EchoApp  {

    Run | Debug
    public static void main(String[] args) {

        new TCPControlLoop(new SingleConnectionHandler(new EchoHandler())).startLoop(args);

    }
}
```

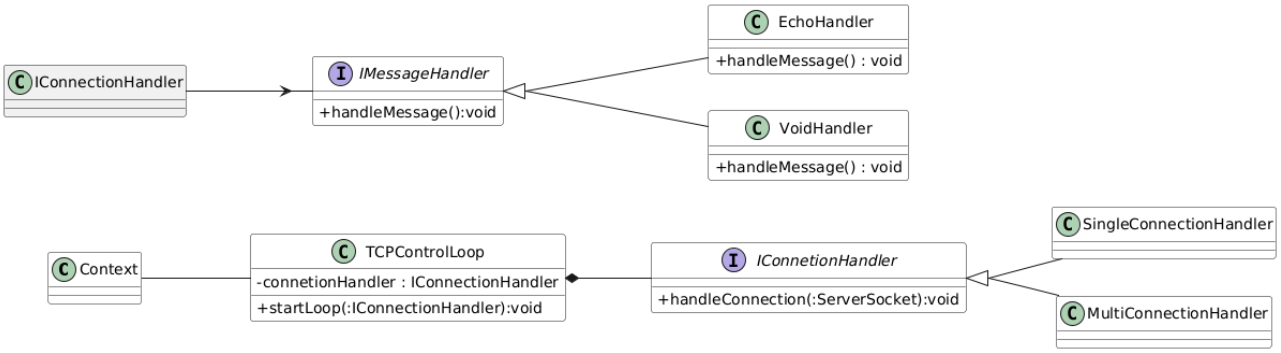
tcp.server.reply
(hotspot composición)

```java
import tcp.server.reply.*;

public class MultiEchoApp  {

    Run | Debug
    public static void main(String[] args) {

        new TCPControlLoop(new MultiConnectionHandler(new EchoHandler())).startLoop(args);
    }
}
```

tcp.server.reply
multisession, echo handler

```java
import tcp.server.reply.*;

public class TestApp  {

    Run | Debug
    public static void main(String[] args) {

        new TCPControlLoop().startLoop(args);

    }
}
```

tcp.server.reply
singlesession, void handler



Context no es parte del framework

FrozenSpot:

- Condición de corte de la conección
- Un tipo de conexión (runtime)
- Un tipo de MessageHandler (runtime)

HotSpot:

- Nuevos MessageHandlers s/funcionalidad: Hash, timestamp, etc
- Nuevos ConnectionHandlers: Timeout, recording, etc

wrap up

- TCPControlLoop se configura con TCPConnection y MessageHandler
- El contexto puede ser:
    - servidor
    - parte de una aplicación
- Posibles mejoras:
    1. Crear una Superclase de SingleConnectionHandler y MultiConnectionHandler
    2. Crear la jerarquía de EndSessionPolicy
    3. Modelar el concepto de Session
        - a. En SingleConnectionHandler la sesión es el loop que procesa mensajes
        - b. En MultiConnectionHandler la sesión es el TCPWorker (subclase de Thread)

JAVA.UTIL.LOGGING

- Los logs son útiles para desarrolladores, administradores y usuarios
- Define jerarquía de “labels”
- Activar/Desactivar logs (sin tocar código)
- Generar reportes en varios formatos (txt, json,xml) y destinos (file, screen, socket)
- La aplicación
    - Configura al framework
    - Manda mensajes a objetos Logger

- El Framework se encarga de:
  - Cómo se crean, organizan y recuperan esos objetos
  - Cómo se configuran
  - Cómo se activan y desactivan
  - A qué prestan atención y a qué no
  - Cómo se formatean los logs
  - A dónde se envían los logs

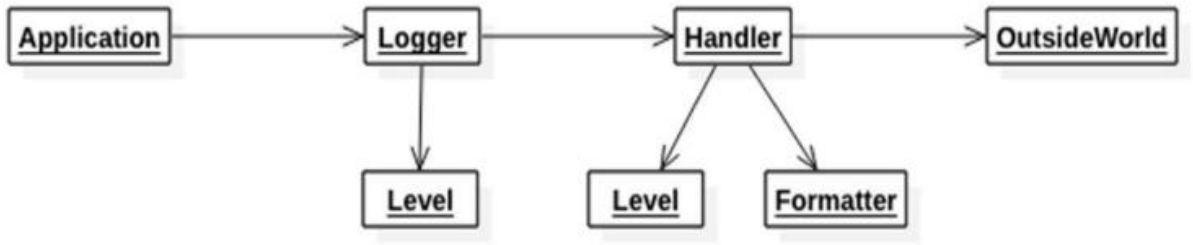
```java
import java.util.logging.Logger;

public class SimpleLoggingExample {

    private static final Logger logger = Logger.getLogger(SimpleLoggingExample.class.getName()

    public static void main(String[] args) {
        logger.info("Application started");

        try {
            int result = 10 / 0; // Simulate an error
        } catch (ArithmeticException e) {
            logger.severe("An error occurred: " + e.getMessage());
        }

        logger.info("Application finished");
    }
}
```

Variante

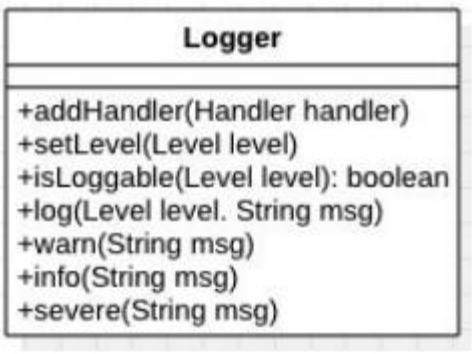
```java
public class Sandbox {
    public static void main(String[] args) throws IOException {
        Logger.getLogger("app.main").addHandler(new FileHandler("log.txt"));
        Logger.getLogger("app.main").log(Level.INFO, "App iniciada");
        try {
            // Acá que hace algo que "podría" 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");
    }
}
```

Logger
- mantiene un "registry" de sus instancias.
- getLogger() es un lazy-initializer

Arquitectura visible

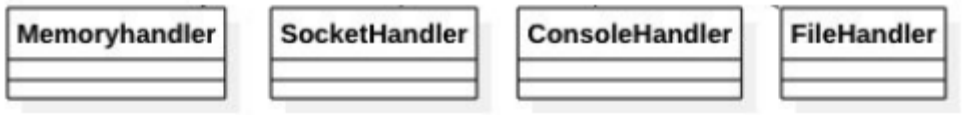


Logger: objeto al que le pedímos que emita un mensaje de log



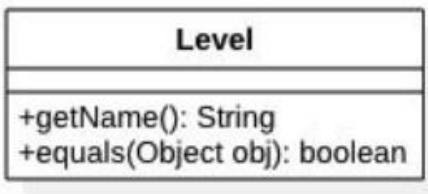
- Podemos definir tantos como necesitemos
  - Instancias de la clase Logger
  - Las obtengo con Logger.getLogger(String nombre)
- Cada uno con su filtro y handler/s
- Se organizan en un árbol (en base a sus nombres)
  - Heredan configuración de su padre (handlers y filters)
- log(Level, String) agrega un mensaje al log
  - Alternativamente uso warn(), info(), severe() …

Handler: recibe los mensajes del Logger y determina como “exportarlos”



- Puede filtrar por nivel
- Tiene un Formatter

Level: indica la importancia de un mensaje



- Cada vez que pido que se loggee algo, debo indicar un nivel
- Los Loggers y Handler comparan el nivel de un cada mensaje con el suyo para decidir si les interesa o no
- Si te interesa un nivel, también te interesan los que son más importantes que ese
- Hay niveles predefinidos, en variables estáticas de la clase Level (SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINIEST, OFF)

Formatter: determina cómo se ”presentará” el mensaje

- El Formatter recibe un mesaje de log (un objeto) y lo transforma a texto
- Son instancias de: SimpleFormatter o XMLFormatter

- Cada handler tiene su formatter
    - Los FileHandler tienen un XMLFormatter por defecto
    - Los ConsoleHandler tienen un SimpleFormatter por defecto

EXTENDIENDO EL FRAMEWORK

Y, mirando adentro, puedo agregar nuevas clases de Formater, Handler y Filter

- Nuevo Formatter: Subclasifico la clase abstracta Formatter o alguna de sus subclases
- Nuevo Handler: Subclasifico la clase abstracta Handler o alguna de sus subclases
- Nuevo Filter: Implemento la interfaz Filter

TEST DRIVEN DEVELOPMENT (TDD)

PROBLEMAS DEL DESARROLLO TRADICIONAL

- Testing se realiza al final del proceso.
- Costo alto de programación y de cambios.
- Especialización de roles genera mala comunicación:
    - Analista ≠ Desarrollador.
- El código sigue una arquitectura fija.
- Se acumulan errores que se detectan tarde.
- Cambios en requerimientos son comunes.

EXTREME PROGRAMMING (XP)

- Cambios de bajo costo → más fácil de evolucionar el código.
- Desarrollo incremental, automático, que conserva comportamiento.
- Arquitectura emerge desde el código.
- Uso de backlog de historias cortas.
- Tareas pequeñas: 1 o 2 días.
- Prácticas:
    - Pair Programming.
    - Tests como contratos.

TDD

Ciclo:

1. Red: Escribir un test que falla.
2. Green: Escribir el código justo para que pase los tests.
3. Refactor: Mejorar el diseño del código manteniendo los tests verdes.
4. Repetir el ciclo.

Combina:

- Test First Development: escribir el test antes del código que haga pasar el test
- Refactoring

Objetivo:

- Pensar en el diseño y qué se espera de cada requerimiento antes de escribir código
- Escribir código limpio que funcione (como técnica de programación)
- Validar requisitos de forma automática

Granularidad de los Test

- Test de unidad
  - Aislar cada unidad de un programa y mostrar que funciona correctamente.
  - Escritos desde la perspectiva del programador
- Test de aceptación
  - Por cada funcionalidad esperada.
  - Escritos desde la perspectiva del cliente

FILOSOFÍA DE TDD

- Escribir los tests primero antes que el código.
- Se escriben tests funcionales para capturar use cases que se validan automáticamente
- Se escriben tests de unidad para enfocarse en pequeñas partes a la vez y aislar los 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 tests pasan, se refactoriza para asegurar que se mantenga una buena calidad en el código.

PROCESO DE TDD

1. Captura de requerimientos:
   a. Historias cortas.
   b. Comunicación directa Cliente ↔ Programador.
2. Uso de frameworks de testing (xUnit). Pruebas:
   a. Unidad.
   b. Regresión.
   c. Integración.
3. Integración Continua (CI).

BENEFICIOS

- Historias de Usuario y Tareas más Pequeñas
- Criterios de Aceptación Claros y Realizables
- Priorización Basada en el Valor de Negocio y el Riesgo
- Diseño Emergente
- Refinamiento Continuo
- Colaboración y Entendimiento Compartido

REQUERIMIENTOS

- Comunicación y Colaboración Efectivas (que incluye al “cliente”)
- Backlog Refinado con Criterios de Aceptación Testeables
- Equipo con Cultura orientada al Testing
- Frameworks de Pruebas Unitarias Adecuados
- Herramientas de Cobertura de Código (Opcional Inicialmente)
- Integración Continua (CI)