Resumen Orientación a Objetos 2

Clase 1

Contexto: métodos y prácticas de desarrollo de hoy

- Métodos ágiles
  - Se desarrolla en iteraciones
  - cada iteración extiende código existente con nueva funcionalidad
  - poca documentación
  - testing automático e integración continua
- Arquitecturas complejas (arquitecturas modulares, capas con distintos concerns: interfaz de usuario, persistencia, seguridad, ...), arquitecturas orientadas a servicios (conexión con código ajeno, desarrollo de APIs, ...)
- El cambio es continuo, el crecimiento es continuo, la complejidad es cada vez mayor

Contexto: herramientas para el desarrollo

- Version Control Systems: Branching & merging, DevOps
  - qué implica?
    - entender código existente (50% del tiempo de mantenimiento)
    - extender código existente
    - code review antes de cada merge
    - entrega continua de alta calidad
- Herramientas de IA: Copilot, ChatGPT, Gemini, etc.
  - Qué implica?
    - Una gran ayuda para tareas rudimentarias
    - Necesitamos desarrollar un pensamiento crítico

Introducción a Refactoring

Refactoring

Transformación de código que **PRESERVA** el comportamiento pero mejora el diseño. Su objetivo es hacer que el código sea más fácil de comprender y mantener.
Mecánicas: Conjunto atómico de pasos para realizar la transoformación

Usos:

- Eliminar duplicaciones
- Simplificar lógicas complejas
- Clarificar códigos

Escenarios:

- Una vez que tengo código que funciona y pasa los tests
- A medida que voy desarrollando:
  - cuando encuentro código difícil de entender (ugly code)
  - cuando tengo que hacer un cambio y necesito reorganizar primero

¿Por qué refactorizar?

- Mejora la legibilidad del código.
- Facilita la detección y corrección de errores.
- Reduce la complejidad del sistema.
- Permite realizar cambios futuros con menor riesgo de introducir errores.
- La clave: poder agregar funcionalidad más rápido después de refactorizar

¿Cuándo refactorizar?

- Antes de añadir una nueva funcionalidad, para facilitar su implementación.
- Al corregir errores, para evitar problemas estructurales que dificulten la depuración.
- Durante revisiones de código, cuando se detectan oportunidades de mejora.

Desafíos y problemas de la refactorización

- Puede introducir errores si no se cuenta con un buen conjunto de pruebas automatizadas.
- Puede afectar el rendimiento si se realizan cambios sin considerar optimizaciones.
- No siempre es bien vista en entornos con alta presión por entregar nuevas características.

Refactoring modifica los programas en pequeños pasos. Si ocurre un problema, es fácil encontrar el bug.
Aclaración: Refactoring no agrega funcionalidad ⇒ puede complicar deadlines

Pasos de un refactoring

- Verificar Precondiciones
- Transformaciónes
- Compilar&Testear

La lección más importante es el ritmo del refactoring: realizar pequeños cambios de forma continua, siempre probando después de cada ajuste. Este enfoque permite que el refactoring se haga de manera rápida y segura. Con esta comprensión, ya podemos avanzar hacia los principios y la teoría detrás de este proceso.

Entonces -> Refactoring

- Se toma un código que “huele mal” producto de mal diseño
  - Codigo duplicado, ilegible, complicado
- y se lo trabaja para obtener un buen diseño
- Cómo?
  - Moviendo atributos / métodos de una clase a otra
  - Extrayendo código de un método en otro método

    - Moviendo código en la jerarquía
    - Etc etc etc …

Code Smell: Código duplicado

- Indicadores de posibles problemas en el diseño del código fuente
- Destacan áreas en las que el código podría estar violando principios de diseño fundamentales
- No son errores, pero para futuros desarrollo pueden:
    - Ser más difíciles
    - Demandar más horas hombre ⇒ costo (1)
- Problemas:
    - 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

Clase grande

- Una clase intenta hacer demasiado trabajo
- Tiene muchas variables de instancia
- Tiene muchos métodos
- Problema:
    - Indica un problema de diseño (baja cohesión).
    - Algunos métodos puede pertener a otra clase
    - Generalmente tiene código duplicado

Método largo

- Un método tiene muchas líneas de código
- Cúanto es muchas LOCs?
    - Más de 20? 30?
    - También depende del lenguaje
- Problemas:
    - Cuanto más largo es un método, más difícil es entenderlo, cambiarlo y reusarlo

Envidia de atributo

- 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)
- Problema
    - 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

Clase de datos

- Una clase que solo tiene variables y getters/setters para esas variables
- Actúa únicamente como contenedor de datos
- Problemas:
  - 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

Condicionales

- 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.
- Problema: la misma estructura condicional aparece en muchos lugares

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 los objetos llamador y el llamado

Velocity

Cantidad de tareas(funcionalidad) / periodo de tiempo

Deuda técnica

costo de rehacer una tarea que tiempo atrás fue resuelta con una solución rápida y desprolija (quick&dirty)

Clase 2

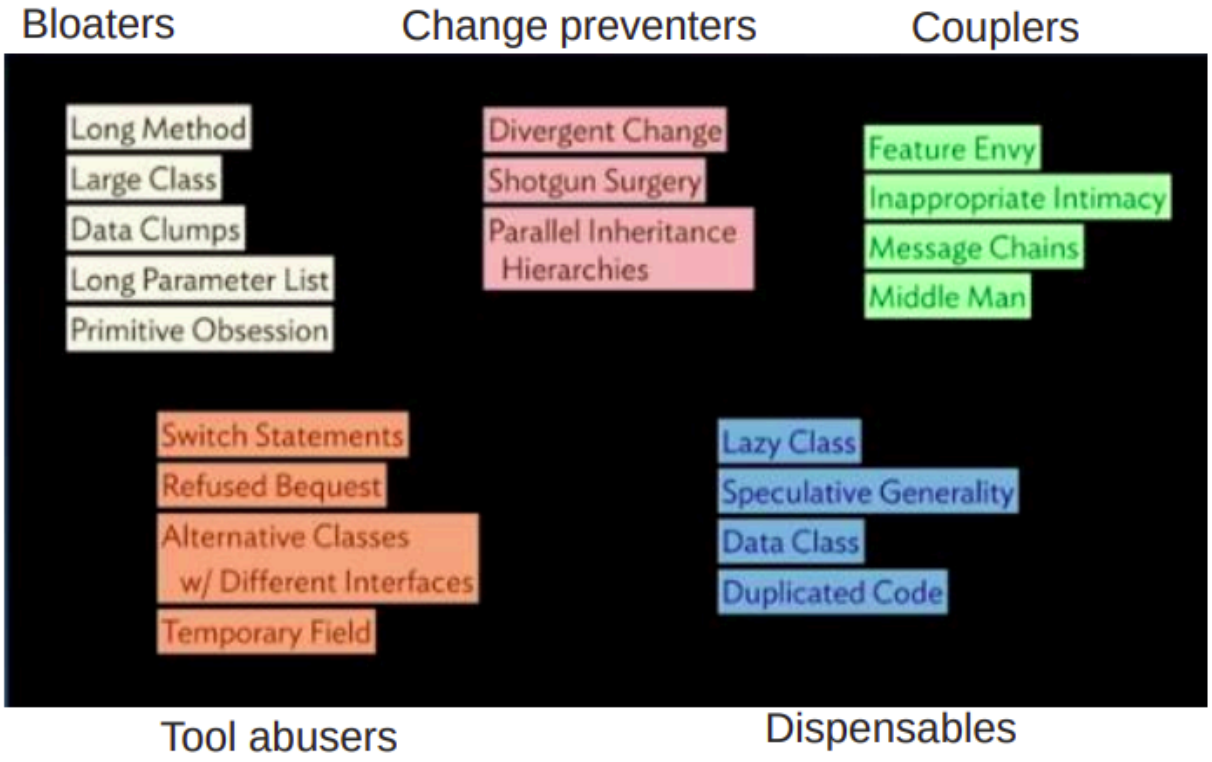
Bad Smells

Indicios de problemas que requieren la aplicación de refactorings.
Algunos Bad Smells:

- Duplicate Code
  - Extract method
  - Pull up method
  - Form template method
- Large Class
  - Extract Class
  - Extract Subclass
- Long Method
  - Extract method

    - Descompose Conditional
    - Replace Temp with query
- Data Class
    - Move Method
- Feature Envy
    - Move Method
- Long Parameter List
    - Replace Parameter with Method
    - Preserve Whole Object
    - Introduce Parameter Object
- Switch Statements
    - Replace Conditional with Polymorphism
- Cambios divergentes (Divergent Change)
    - Extract Class
- “Shotgun surgery”
    - Move Method/Field
- Generalidad especulativa
    - Collapse Hierarchy
    - Inline Class
    - Remove Parameter
- Cadena de mensajes
    - Hide Delegate
    - Extract method & Move method
- Middle man
    - Remove middle man
- Inappropiate intimacy
    - Move Method/Field
- Legado rechazado (Refused bequest)
    - Push Down Method/Field
- Comentarios
    - Extract Method
    - Rename Method

Categorización



Catálogo de refactorings

- Refactoring manual
- Formato:
  - Nombre
  - Motivación
  - Mecánica
  - Ejemplo

Organización catálogo Fowler

- Composición de métodos
- Mover aspectos entre objetos
- Organización de datos
- Simplificación de expresiones condicionales
- Simplificación en la invocación de métodos
- Manipulación de la generalización
- Big refactorings

Composición de métodos

- Permiten "distribuir" el código adecuadamente
- Métodos largos son problemáticos
- Contienen:
  - mucha información
  - lógica compleja
- Extract Method

- Inline Method
- Replace Temp with Query
- Split Temporary Variable
- Replace Method with Method Object
- Substitute Algorithm

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 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.
  5. Pasar como parámetro las variables que el método nuevo lee
  6. Compilar
  7. Reemplazar código en método original por llamada
  8. Compilar y testear

Inline Method

Es utilizada cuando una función no aporta suficiente claridad y su cuerpo es tan comprensible como su nombre. En estos casos, eliminar la función y reemplazar su llamada por su contenido puede hacer que el código sea más directo y fácil de leer, evitando una indirection innecesaria.

También se usa cuando un grupo de funciones está mal estructurado: primero se combinan en una función más grande (*inlining*), y luego se reorganizan extrayendo solo aquellas partes que realmente aportan claridad.

Otro caso común es cuando hay demasiada delegación entre funciones, lo que hace difícil seguir el flujo del código. *Inline Function* ayuda a identificar qué funciones realmente valen la pena mantener y cuáles pueden eliminarse, reduciendo la complejidad innecesaria.

- Mecánica:
  - Verificar que este no esté en un método polimorfico
    - Si este es un método en una clase, y tiene subclases que lo anulan, entonces no puedo insertarla directamente
  - Buscar/encontrar todas las llamadas/ o los que llaman a la función a la función
  - Reemplazar cada llamada con cuerpo de la función
  - Testear desp de cada sustitución
  - Eliminar la definición de la función

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 Exctract method, para evitar parámetros innecesarios
- Solución:
  - Extraer la expresión de un método
  - Reemplazar TODAS las referencias a la var. temporal por la expresión
  - El nuevo método luego puede ser usado en otros métodos
- Mecánica:
  1. Encontrar las vars. temporales con una sola asignación (si no, Split Temporary Variable)
  2. Extraer el lado derecho de la asignación (tener cuidado con los efectos colaterales; si no, Separate Query From Modifier)
  3. Reemplazar todas las referencias de las var. temporal por el nuevo método
  4. Eliminar la declaración de la var. temporal y las asignaciones
  5. Compilar y testear

Replace Method with Method Object

Aunque las funciones son esenciales en la programación, a veces es útil encapsular una función en un objeto llamado *command object* o *command*. Esto ofrece más flexibilidad que una función simple, permitiendo operaciones como *undo*, la construcción progresiva de parámetros y personalizaciones mediante herencia o *hooks*. Sin embargo, esta flexibilidad aumenta la complejidad, por lo que se prefiere usar funciones en la mayoría de los casos, solo recurriendo a un *command* cuando es estrictamente necesario.

- Mecánica
  - Crear una clase vacía para la función. Nombrarla en base a la función
  - Usar Move functión para mover la función a la clase vacía
  - Considere la posibilidad de hacer un campo para cada argumento, y mover los argumentos al constructor.

Substitute Algorithm

Cuando encuentro una forma más clara de resolver un problema, sustituyo la solución complicada por la más simple. La refactorización puede simplificar un algoritmo complejo, pero a veces es más efectivo reemplazarlo por uno más sencillo, especialmente cuando aprendo más sobre el problema o uso una biblioteca que reemplaza mi código. Este enfoque también facilita hacer cambios cuando se requiere una adaptación del algoritmo. Al realizar este reemplazo, aseguro haber descompuesto el método al máximo para que la sustitución sea manejable.

- Mecánicas
  - Organizar el código a reemplazar para que rellene una función completa.
  - Preparar test usando solamente esta función, para capturar su comportamiento
  - Preparar un algoritmo alternativo

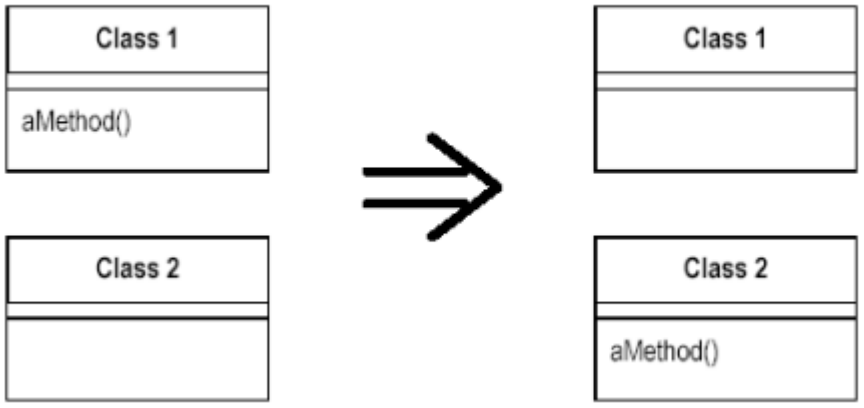
- Ejecutar chequeos estáticos
- Ejecutar los test para comparar la salida del algoritmo viejo con la del nuevo. Si son iguales, ya está. De lo contrario, utilice el algoritmo antiguo para la comparación en las pruebas y depuración.

Mover aspectos entre objetos

- Ayudan a mejorar la asignación de responsabilidades
- Move method
- Move Field
- Extract Class
- Inline Class
- Remove Middle Man
- Hide Delegate

Move Method

- Motivación:
  - Un método esta usando o usará muchos servicios que están definidos en una clase diferente a la suya (Feature envy)
- Solución:
  - Mover el método a la clase donde están los servicios que usa.
  - Convertir el método original en un simple delegación o eliminarlo



- Mecánica
  1. Revisar otros atributos y métodos en la clase original (puede que también haya que moverlos)
  2. Chequear subclases y superclases de la clase original por si hay otras declaraciones del método (puede que no se pueda mover)
  3. Declarar el método en la clase destino
  4. Copiar y ajustar el código (ajustando las referencias desde el objeto origen al destino); chequear manejo de excepciones
  5. Convertir el método original en una delegación
  6. Compilar y testear
  7. Decidir si eliminar el método original → eliminar las referencias

8. Compilar y testear

Move Field

Cuando identifico que una estructura de datos no es adecuada, es esencial cambiarla para evitar que errores futuros compliquen el código. A menudo, muevo datos cuando siempre necesito pasar un campo de un registro junto con otro. Si varios datos deben pasarse juntos con frecuencia, lo mejor es agruparlos en un solo registro. También, si un cambio en un registro afecta a otro, es señal de que un campo está en el lugar equivocado. Si un campo debe actualizarse en múltiples estructuras, lo ideal es moverlo a un lugar donde solo se actualice una vez.

- Mecánicas:
  - Asegurarse que el campo fuente está encapsulado.
  - Testear
  - Crear un campo (y accesorios) en el destino
  - Ejecutar chequeos estáticos
  - Asegurarse que existe una referencia desde el objeto fuente hacia el objeto destino
  - Ajustar accesorios para utilizar el campo de destino
  - Testear
  - Eliminar el campo de origen/ campo fuente
  - Testear

Extract Class

Las clases deben ser abstracciones claras con responsabilidades definidas, pero con el tiempo pueden volverse demasiado complejas. Cuando una clase crece en datos y métodos, es importante identificar partes que puedan separarse. Un buen indicador es cuando ciertos datos y métodos están estrechamente relacionados o cambian juntos. También es útil analizar qué sucede al eliminar un método o dato. Otra señal de sobrecarga aparece cuando el subtipado afecta solo a algunas partes de la clase o de formas inconsistentes.

Mecánicas:

- Decidir cómo dividir las responsabilidades de la clase.
- Crear una nueva clase secundaria para expresar las responsabilidades divididas.
- Si las responsabilidades de la clase padre original ya no coinciden con su nombre, cambie el nombre el padre.
- Crear una instancia de la clase hija al construir el padre y añadir un enlace de padre a hijo.
- Use Move Field (207) en cada campo que desee mover. Pruebe después de cada movimiento.
- Use la Move function (198) para mover los métodos al nuevo hijo. Comience con el nivel inferior métodos (los que se llaman en lugar de llamar). Prueba después de cada movimiento.
- Revisar las interfaces de ambas clases, eliminar métodos innecesarios, cambiar los nombres para adaptarse mejor a las nuevas circunstancias.
- Decidir si exponer al nuevo hijo. En caso afirmativo, considere aplicar Change Reference to Value (252) a la clase hijo.

Inline Class

Es el inverso a Extract Class, se usa cuando una clase ha perdido su propósito y debe integrarse en otra. Esto suele ocurrir tras refactorizaciones que trasladan responsabilidades fuera de la clase, dejándola casi vacía. También es útil al reorganizar dos clases en una nueva distribución de características: primero se combinan con **Inline Class** y luego se separan con **Extract Class**. Esta estrategia facilita la reestructuración eficiente del código.

Mecánicas:

- En la clase de destino, cree funciones para todas las funciones públicas de la clase source. Estas funciones deberían delegarse a la clase source.
- Cambiar todas las referencias a los métodos de clase fuente para que utilicen la clase de destino delegadores en su lugar. Prueba después de cada cambio.
- Mover todas las funciones y datos de la clase fuente al destino, probando después cada movimiento, hasta que la clase source esté vacía.
- Eliminar la clase de origen

Remove middle man

Se aplica cuando una clase actúa solo como intermediaria para un objeto delegado, volviéndose innecesaria. Esto ocurre al añadir demasiados métodos delegados, lo que puede volverse tedioso. En estos casos, es mejor permitir que el cliente acceda directamente al delegado. La cantidad adecuada de encapsulación varía con el tiempo, y herramientas como **Hide Delegate** y **Remove Middle Man** permiten ajustar el código según evoluciona el sistema.

Mecánica:

- Crear un getter para el delegado
- Por cada uso de cliente de un metodo delegado, reemplazar la llamada al método encadenando a través del accesor. Testear despues de cada reemplazo
- Si todas las llamadas a el método son reemplazadas, puedes eliminar el método
- Con refactorings automatizados, podemos usar Encapsulate variable en la delegación al campo y luego Inline function para todos los métodos que lo utilicen.

Hide delegate

La **encapsulación** reduce la dependencia entre módulos, facilitando los cambios sin afectar todo el sistema. Inicialmente, se asocia con ocultar campos, pero también se aplica a otros aspectos, como la delegación de métodos.

Cuando un cliente llama directamente a un método de un objeto delegado dentro de un servidor, depende de su estructura interna. Si el delegado cambia su interfaz, todos los clientes que lo usan deben actualizarse.

Para evitar esto, se aplica **Hide Delegate**, agregando un método en el servidor que encapsula la llamada al delegado. Así, los clientes solo interactúan con el servidor y cualquier cambio en el delegado afecta únicamente al servidor, no a los clientes.

Mecánica:

- Para cada método en la delegación, cree un método de delegación simple en el servidor.
- Ajusta el cliente para llamar al servidor. Prueba después de cada cambio.
- Si ningún cliente necesita acceder al delegado, retire el accesor del servidor para el delegado.
- Testear

Manipulación de la generalización

- Ayudan a mejorar las jerarquías de clases
- Push Up / Down Field
- Push Up / Down Method
- Pull Up Constructor Body
- Extract Subclass / Superclass
- Collapse Hierarchy
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance

Pull Up Method

Veo dos métodos en diferentes clases que pueden parametrizarse de tal manera que terminen siendo esencialmente el mismo método

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, declarado 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

Organización de datos

- Facilitan la organización de atributos
- Self Encapsulate Field
- Encapsulate Field / Collection
- Replace Data Value with Object
- Replace Array with Object
- Replace Magic Number with Symbolic Constant

Simplificación de expresiones condicionales

- Ayudan a simplificar los condicionales
- Descompose Conditional

- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Replace Conditional with Polymorphism

Replace Conditional with Polymorphism

1. Crear la jerarquía.
2. Por cada variante, crear un método en cada subclase que redefina el de la superclase.
3. Copiar al método de cada subclase la parte del condicional correspondiente.
4. Compilar y testear.
5. Borrar de la superclase la sección (branch) del condicional que se copió.
6. Compilar y testear.
7. Repetir para todos los branchs del condicional.
8. Hacer que el método de la superclase sea abstracto.

Simpificación de invocación de métodos

- Sirven para mejorar la interfaz de una clase
- Rename Method
- Preserve Whole Object
- Introduce Parameter Object
- Parameterize Method

Refactoring tools

Volviendo a la pregunta que se planteo anteriormente. ¿Cuando aplicar Refactoring?

- En el contexto de TDD
- Cuando se descubre código con mal olor, aprovechando la oportunidad
  - dejarlo al menos un poco mejor, 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
- Cuando encuentro una mejor manera de codificar algo

Automatización del Refactoring

- Refactorizar a mano es demasiado costoso: lleva tiempo y puede introducir errores
- Herramientas de refactoring
- Características de las herramientas:
  - 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

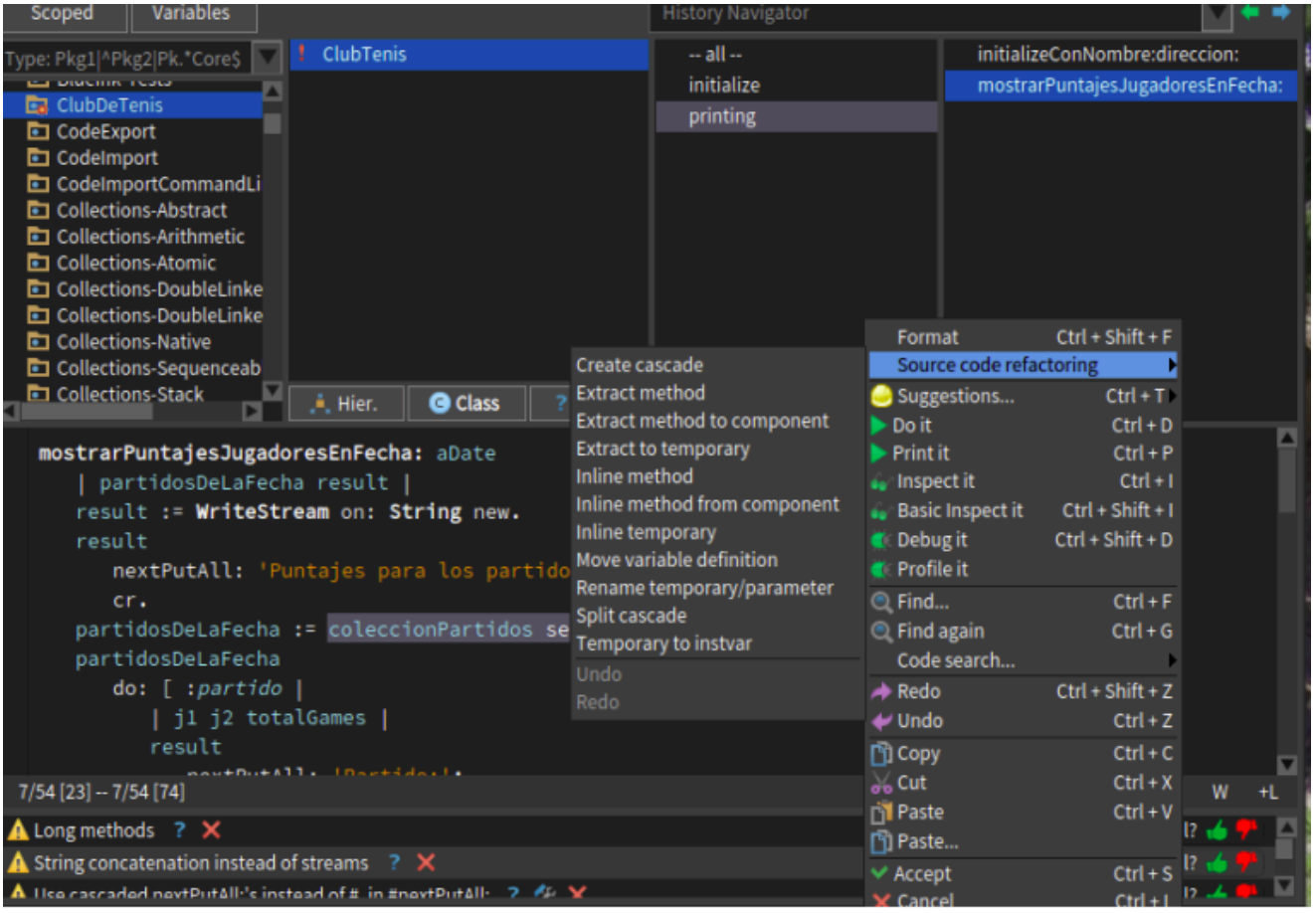
Nacimiento con Smalltalk

- Primera herramienta de refactoring: : Refactoring Browser (RB) (en UIUC by John Brant & Don Roberts del grupo de Ralph Johnson)
- Practicamente todos los lenguajes tienen una herramienta de refactoring hoy en día, y copian la misma arquitectura / técnica del RB
- Más adelante: herramienta Code Critic que detecta code smells
- Smalltalk 1ro en:
    - XUnit
    - Refactoring

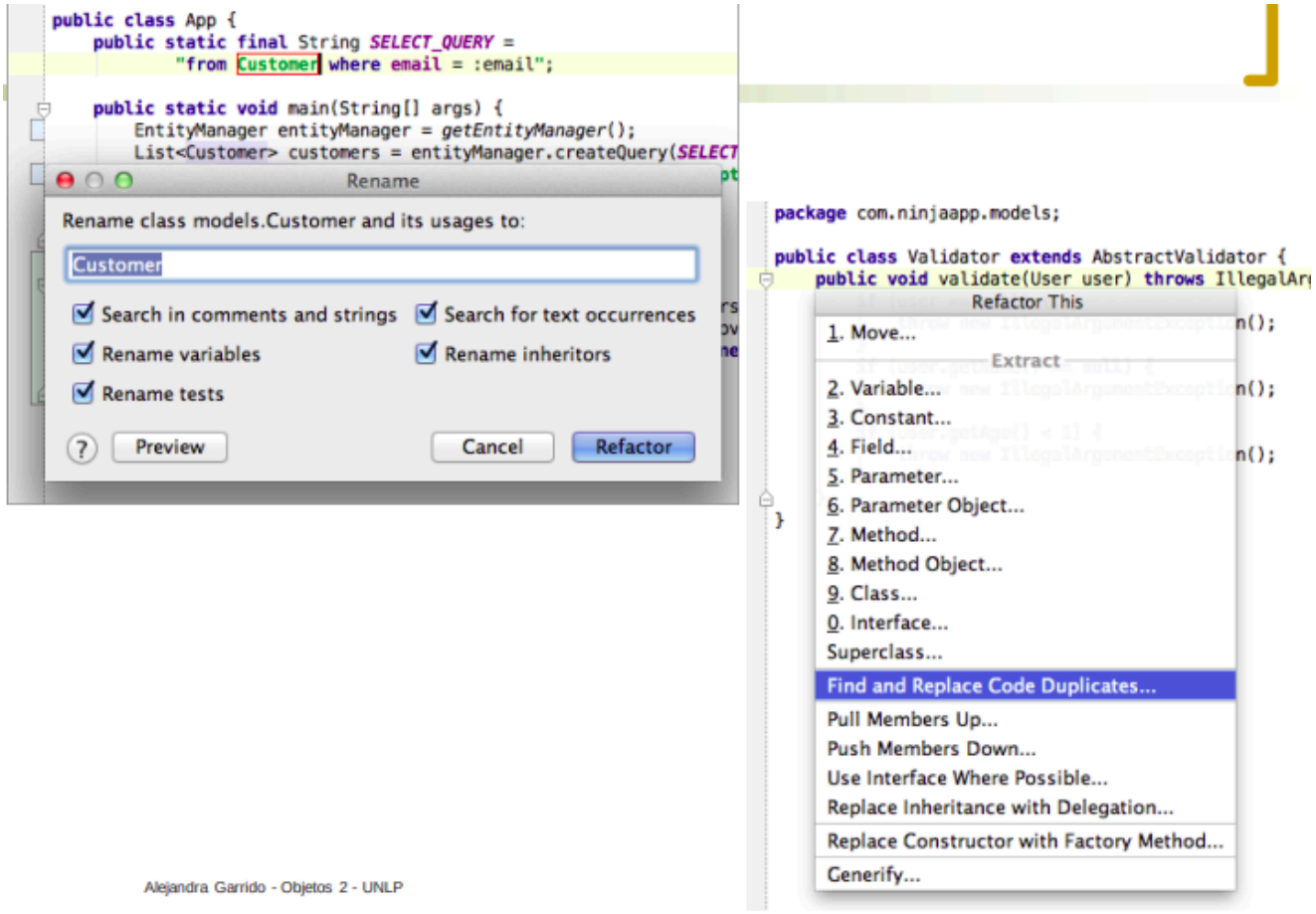
Las herramientas

- 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

Smalltalk RB



IntellijJ IDEA



Clase 3

Los expertos saben que no conviene resolver cada problema desde cero. En lugar de eso, reutilizan soluciones que ya les han resultado útiles, y al hacerlo, reconocen patrones recurrentes en clases y comunicaciones entre objetos. Esa experiencia es parte de lo que los convierte en expertos. Por eso, los patrones de diseño permiten lograr un buen diseño de forma más rápida y efectiva. En definitiva, los patrones de diseño ayudan a un diseñador a lograr un buen diseño más rápidamente.

De la arquitectura al software...

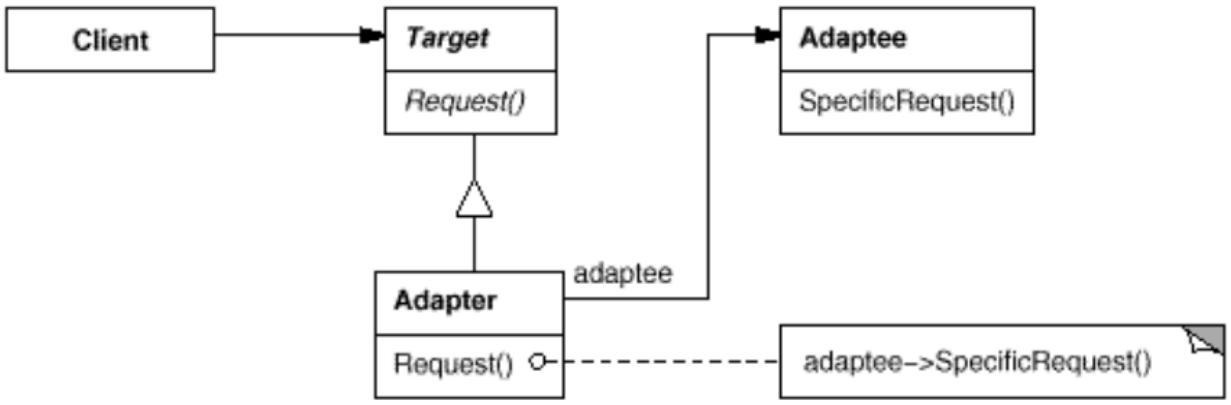
- ¿Que es importante?
  - Un patrón es un par problema-solución
  - Los patrones tratan con problemas recurrentes y buenas soluciones (probadas) a esos problemas
  - La solución es suficientemente genérica para poder aplicarse de diferentes maneras

Patrón Adapter

- **Intención:**
  Convertir la interfaz de una clase en otra que el cliente espera.
  Adapter permite que ciertas clases con interfaces incompatibles puedan trabajar en conjunto
- **Aplicabilidad:**
  Use Adapter cuando usted quiere usar una clase existente y su interfaz no es compatible con lo que precisa

Se quiere crear una clase reutilizable que coopere con clases no relacionadas o que no han sido previstas, es decir, clases que no tienen por qué tener interfaces compatibles.
(solamente en el caso de un adaptador de objetos) Es necesario usar
varias subclases existentes, pero no resulta práctico adaptar su interfaz
heredando de cada una de ellas. Un adaptador de objetos puede adaptar la interfaz de su clase padre.

- **Estructura**



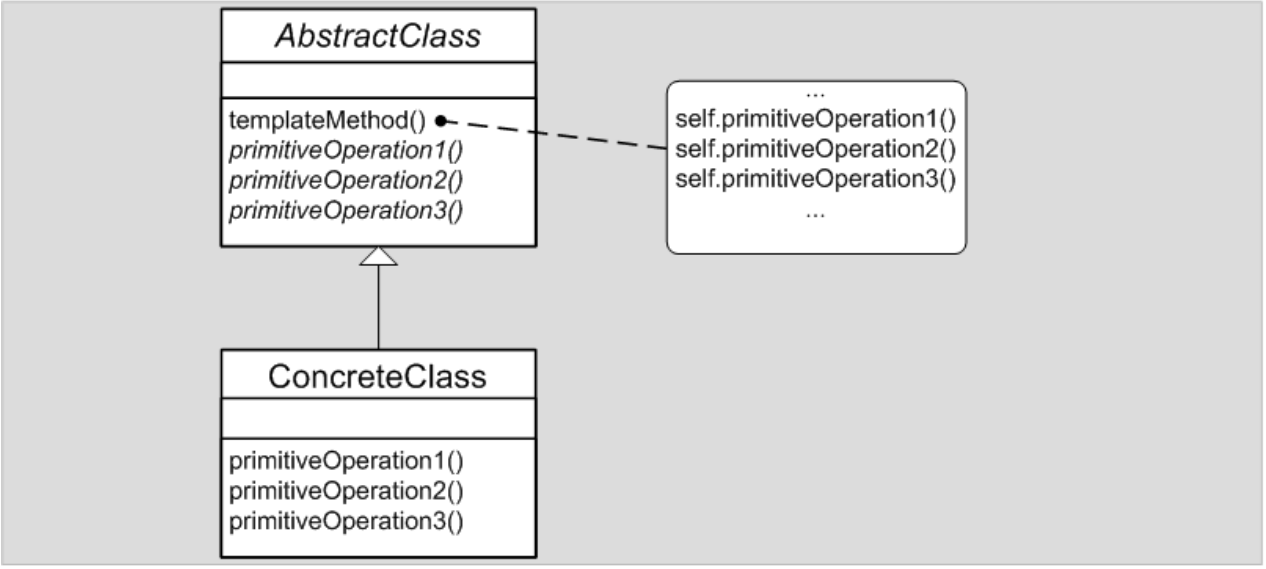
- **Participantes**
  Target
  Define la interfaz específica que usa el cliente
  Client
  Colabora con objetos que satisfacen la interfaz de Target
  Adaptee
  Define una interfaz que precisa ser adaptada
  Adapter
  Adapta la interfaz del Adaptee a la interfaz de Target
- **Elementos**
  Colaboraciones
  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
- Consecuencias
  Los adaptadores de clases y de objetos tienen diferentes ventajas e inconvenientes. Un adaptador de clases
  - Adapta una clase Adaptable a Objetivo, pero se refiere únicamente a una clase Adaptable concreta. Por tanto, un adaptador de clases no nos servirá cuando lo que queremos es adaptar una clase y todas sus subclases.

    - Permite que Adaptador redefina parte del comportamiento de Adaptable, por ser Adaptador una subclase de Adaptable
    - Introduce un solo objeto, y no se necesita ningún puntero de indirección adicional para obtener el objeto adaptado.
      Por su parte, un adaptador de objetos
    - Permite que un mismo Adaptador funcione con muchos Adaptables —es decir, con el Adaptable en sí y todas sus subclases, en caso de que las tenga—. El Adaptador también puede añadir funcionalidad a todos los Adaptables a la vez.
    - Hace que sea más difícil redefinir el comportamiento de Adaptable. Se necesitará crear una subclase de Adaptable y hacer que el Adaptador se refiera a la subclase en vez de a la clase Adaptable en sí.
- Cosas a tener en cuenta
    - **Cantidad de adaptación necesaria:**
      El trabajo que realiza el adaptador varía según cuán diferente sea la interfaz de *Adaptable* respecto a la de *Objetivo*. Puede ir desde simples cambios de nombres hasta la incorporación de operaciones completamente nuevas.
    - **Adaptadores conectables:**
      Cuanto menos supongan las clases sobre otras, más reutilizables serán. Adaptar interfaces dentro de una clase permite integrarla en sistemas existentes con distintas expectativas de interfaz. Así, por ejemplo, un componente como un *VisualizadorDeArboles* se vuelve reutilizable si permite adaptarse a diferentes estructuras jerárquicas sin exigir una interfaz común.
    - **Adaptadores bidireccionales:**
      A veces, un objeto adaptado ya no cumple con la interfaz de *Adaptable*, lo que impide su uso en ciertos contextos. En estos casos, los adaptadores bidireccionales permiten que el mismo objeto sea compatible con distintas interfaces.

Patrón 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 varian;
    - 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
- Elementos
  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

Strategy

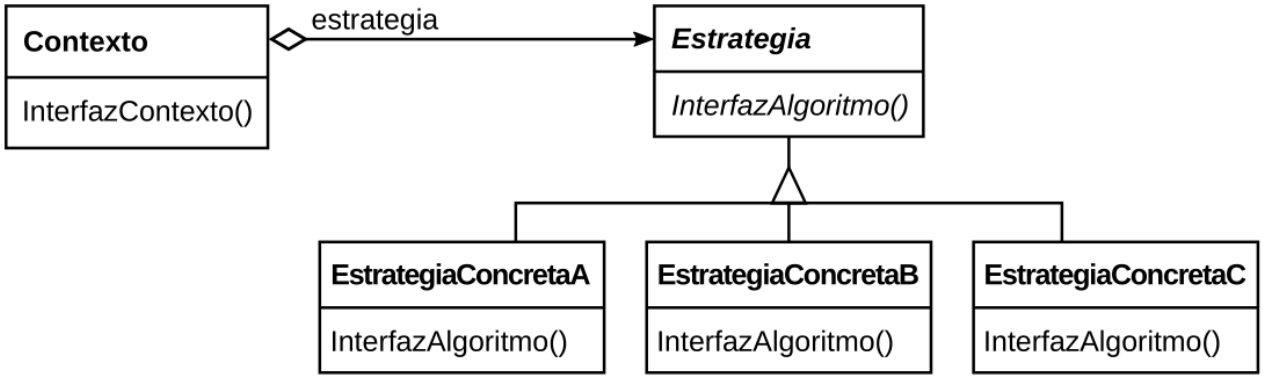
Propósito

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

Aplicabilidad

- Muchas clases relacionadas difieren sólo en su comportamiento. Las estrategias permiten configurar una clase con un determinado comportamiento de entre muchos posibles.
- Se necesitan distintas variantes de un algoritmo, for ejemplo, podríamos definir algoritmos que reflejasen distintas soluciones de compromiso entre tiempo y espacio. Pueden usarse estrategias cuando estas variantes se implementan como una jerarquía de clases de algoritmos.
- Un algoritmo usa datos que los clientes no deberían conocer. Úsese el patrón Strategy para evitar exponer estructuras de dalos complejas y dependientes del algoritmo.
- Una clase define muchos comportamientos, y éstos se representan como múltiples sentencias condicionales en sus operaciones. En vez de tener muchos condicionales, podemos mover las ramas de éstos a su propia clase Estrategia.

Estructura



Participantes

Estrategia (Componedor)
Declara una interfaz común a todos los algoritmos permitidos. El Contexto usa esta interfaz para llamar al algoritmo definido por una EstrategiaConcreta.
EstrategiaConcreta (ComponedorSimple, ComponedorTeX, ComponedorMatriz)
Implementa el algoritmo usando la interfaz Estrategia.
Contexto (Composición)
se configura con un objeto EstrategiaConcreta.
Mantiene una referencia a un objeto Estrategia.
puede definir una interfaz que permita a la Estrategia acceder a sus datos.

Colaboraciones

- Estrategia y Contexto 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 Estrategia. 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 EstrategiaConcreta, el cual pasan al contexto; por tanto, los clientes interactúan exclusivamente con el contexto. Suele haber una familia de clases EstrategiaConcreta a elegir por el cliente.

Consecuencias

1. Familias de algoritmos relacionados. Las jerarquías de clases Estrategia definen una familia de algoritmos o comportamientos para ser reutilizados por los contextos. La herencia puede ayudar a sacar factor común: de la funcionalidad de estos algoritmos.
2. Una alternativa a la herencia. La herencia ofrece otra forma de permitir una variedad de algoritmos o comportamientos. Se puede heredar directamente de una clase Contexto para proporcionar diferentes comportamientos. Pero esto liga el comportamiento al Contexto, mezclando la implementación del algoritmo con la del Contexto, lo que hace que éste sea más difícil de comprender, mantener y extender. Y no se puede modificar el algoritmo dinámicamente.
3. Las estrategias eliminan las sentencias condicionales. El patrón Strategy ofrece una alternativa a las sentencias condicionales para seleccionar el comportamiento deseado. Cuando se juntan muchos comportamientos en una clase es difícil no usar sentencias condicionales para seleccionar el comportamiento correcto. Encapsular el comportamiento en clases Estrategia separadas elimina estas sentencias condicionales.
4. Una elección de implementaciones. Las estrategias pueden proporcionar distintas implementaciones del mismo comportamiento. El cliente puede elegir entre estrategias con diferentes soluciones de compromiso entre tiempo y espacio.
5. Los clientes deben conocer las diferentes Estrategias, El patrón tiene el inconveniente potencial de que un cliente debe comprender cómo difieren las Estrategias antes de seleccionar la adecuada. Los clientes pueden estar expuestos a cuestiones de implementación.
6. Costes de comunicación entre Estrategia y Contesto. La interfaz de Estrategia es compartida por todas las clases EstrategiaConcreta, ya sea el algoritmo que implementa trivial o complejo. Por tanto, es probable que algunos objetos EstrategiaConcreta no usen toda la información que reciben a través de dicha interfaz
7. Mayor número de objetos. Las estrategias aumentan el número de objetos de una aplicación. A veces se puede reducir este costo implementando las estrategias como objetos sin estado que puedan ser compartidos por el contexto.

Implementación

1. Definir las interfaces Estrategia y Contesto. Las interfaces Estrategia y Contexto deben permitir a una EstrategiaConcreta acceder de manera enciente a cualquier dato que ésta necesite del contexto, y viceversa.
   Un enfoque es hacer que Contexto pase los datos como parámetros a las operaciones de Estrategias.
   Otra técnica consiste en que un contexto se pase a sí mismo como argumento, y que la estrategia pida los datos explícitamente al contexto. Como alternativa, la estrategia puede guardar una referencia a su contexto, eliminando así la necesidad de pasar nada.
2. Estrategias como parámetros de plantillas.
3. Hacer opcionales los objetos Estrategia. La clase Contexto puede simplificarse en caso de que tenga sentido no tener un objeto Estrategia. Contexto comprueba si tiene un objeto Estrategia antes de acceder a él. En caso de que exista, lo usa normalmente. Si no hay una estrategia, Contexto realiza el comportamiento predeterminado. La ventaja de este enfoque es que los

clientes no tienen que tratar con los objetos Estrategia a menos que no les sirva el comportamiento predeterminado.

Cosas importantes recordar

- Propósito
- Estructura:
  - clases que componen el patrón (roles)
  - cómo se relacionan (jerarquías, clases abstractas/interfaces, métodos abstractos – protocolo de interfaces, conocimiento/composición)
- Variantes de implementación •Consecuencias positivas y negativas
- Relación con otros patrones

Composite

Propósito

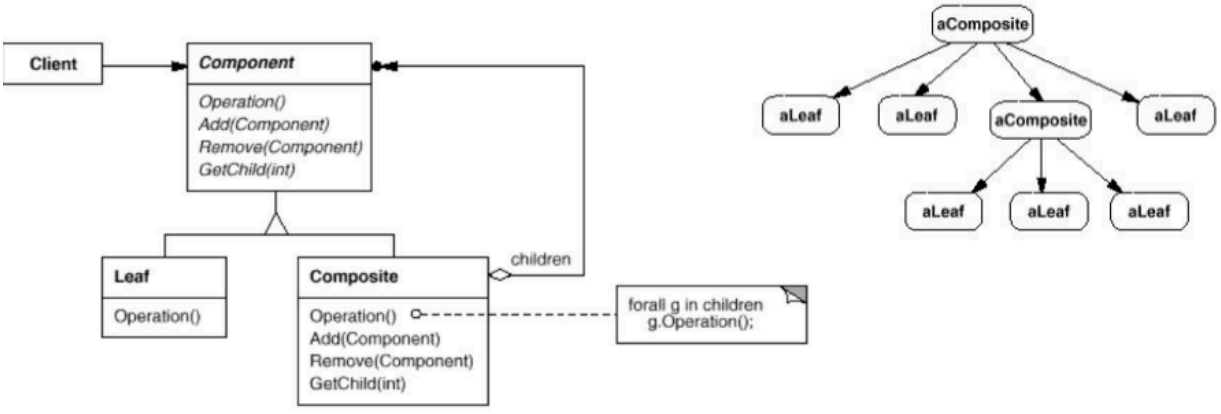
Componer objetos en estructuras de árbol para representar jerarquías parte-todo. El composite permite que los clientes traten a los objetos atómicos y a sus componentes uniformemente.

Aplicabilidad

Utilícelo cuando:

- quiere representar jerarquías parte-todo de objetos
- quiere que los objetos "clientes" puedan ignorar las diferencias entre composiciones y objetos individuales. Los clientes tratan 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.
- Implementa operaciones para manejar el sub-árboles
  Client
- manipula objetos en la composición a través de la interfaz Componente.

Colaboraciones

Los Clientes usan la interfaz de la clase Componente para interactuar con los objetos de la estructura compuesta. Si el recipiente es una Hoja, la petición se trata correctamente. Si es un Compuesto, normalmente redirige las peticiones a sus componentes hijos, posiblemente realizando operaciones adicionales antes o después.

Consecuencias

+bueno
-malo

- Define jerarquías de clases formadas por 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)

Cuestiones de implementación

- Referencias explícitas a la raíz de una hoja. Mantener referencias de los componentes hijos a sus padres puede simplificar el recorrido y la gestión de una estructura compuesta. La referencia al padre facilita ascender por la estructura y borrar un componente.
- Compartir componentes. Muchas veces es útil compartir componentes, por ejemplo para reducir los requisitos de almacenamiento. Pero cuando un componente no puede tener más de un padre, compartir componentes se hace más difícil.
- Maximizar el protocolo de la clase/interfaz Component Uno de los objetivos del patrón Composite es hacer que los clientes se despreocupen de las clases Hoja o Compuesto que están usando. Para conseguirlo, la clase Componente debería definir tantas operaciones comunes a las clases Compuesto y Hoja como sea posible. La clase Componente normalmente proporciona implementaciones predeterminadas para estas operaciones, que serán redefinidas por las subclases Hoja y Compuesto.

- Orden de las hojas, muchos diseños especifican una ordenación de los hijos de Compuesto. Cuando la ordenación de los hijos es una cuestión a tener en cuenta, debemos diseñar las intertaces de acceso y gestión de hijos cuidadosamente para controlar la secuencia de hijos.
- Borrado de componentes en lenguajes sin recolección de basura, normalmente es mejor hacer que un Compuesto sea el responsable de borrar sus hijos cuando es destruido. Una excepción a esta regla es cuando los objetos Hoja son inmutables y pueden por tanto ser compartidos.
- **Búsqueda de componentes (fetch) por criterios.** Si necesitamos recorrer composiciones o realizar búsquedas frecuentes dentro de ellas, la clase `Compuesto` puede almacenar información sobre sus elementos hijos que facilite estas operaciones. Por ejemplo, puede guardar resultados parciales o datos auxiliares que le permitan reducir el tiempo de recorrido o búsqueda.
- Diferentes estructuras de datos para guardar componentes La elección de la estructura de datos depende (como siempre) de la eficiencia. De hecho, ni siquiera es necesario usar una estructura de datos de propósito general. A veces los compuestos tienen una variable para cada hijo, aunque esto requiere que cada subclase de Compuesto implemente su propia interfaz de gestión.

Usos a tener en cuenta

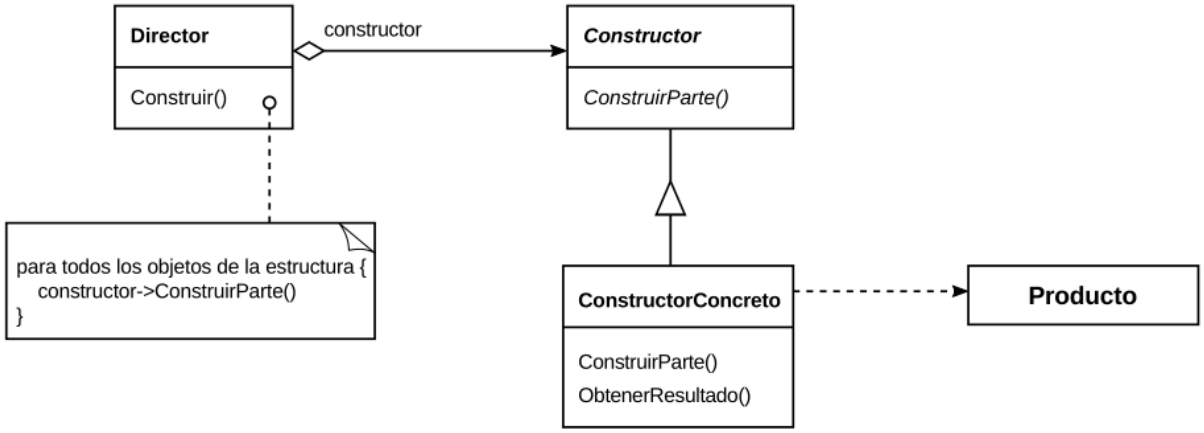
- Ejemplos comunes:
  - group/ungroup de elementos gráficos
  - Carpetas, archivos y link simbolicos (sistemas de archivos)
- Garantías en operaciones
  - Alquileres de inmuebles (garantía: Sueldo, Inmueble)
  - Prod. Financieros (collaterals: acciones, bonos, plazos fijos)
  - Prestamos prendarios (lo visto en el ejemplo)
- Agrupamiento
- Combos (cajita feliz, talcos, útiles escolares)
  - Productos financieros (prestamos hipotecarios)
  - Pixels/Celdas en sensado remoto (imagenes satelitales)
  - Group/Ungroup de elementos gráficos ○ Paquetes de Alojamiento…
- Es fundamental entender
  - el comportamiento de las hojas y los sub-árboles
  - Cuáles son las reglas de composición (configuración)
    - Solapamiento, continuidad, circularidad, mutua exclusión, etc
  - ¿Qué objetos/métodos se encargan de crear las configuraciones?

Builder

Intención

separa la construcción de un objeto complejo de su representacion (implementación) de tal manera que el mismo proceso puede construir diferente representaciones (implementaciones)

Estructura



Participantes

- Constructor (ConvertidorDeTexto) especifica una interfaz abstracta para crear las partes de un objeto Producto.
- Constructor Concreto (ConvertidorASCII, ConvertidorTeX, ConvertidorUtilDeTexto)
  - implementa la interfaz Constructor para construir y ensamblar las partes del producto.
  - define la representación a crear.
  - proporciona una interfaz para devolver el producto (p. ej., ObtenerTextoASCII, ObtenerUtilDeTexto).
- Director (LectorRTF) construye un objeto usando la interfaz Constructor.
- Producto (TextoASCII, TextoTeX, UtilDeTexto)
  - representa el objeto complejo en construcción. El ConstructorConcreto construye la representación interna del producto y define el proceso de ensamblaje.
  - incluye las clases que definen sus partes constituyentes, incluyendo interfaces para ensamblar las partes en el resultado final.

Colaboraciones

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

Vale la pena Builder?

Si!

- Abstrae la construcción compleja de un objeto complejo
- Permite variar lo que se construye Director <-> Builder
- Da control sobre los pasos de contrucción
  No!
- Requiere diseñar y implementar varios roles
- Cada tipo de producto requiere un ConcreteBuilder
- Builder suelen cambiar o son parsers de specs (> complejidad)

Consecuencias

1. Permite variar la representación interna de un producto. El objeto Constructor proporciona al director una interfaz abstracta para construir el producto. La interfaz permite que el constructor oculte la representación y la estructura interna del producto. También oculta el modo en que éste es ensamblado. Dado que el producto se construye a través de una interfaz abstracta, todo lo que hay que hacer para cambiar la representación interna del producto es definir un nuevo tipo de constructor
2. Aísla el código de construcción y representación. El patrón Builder aumenta la modularidad al encapsular cómo se construyen y se representan los objetos complejos. Los clientes no necesitan saber nada de las clases que definen la estructura interna del producto
3. Proporciona un control más fino sobre el proceso de construcción. A diferencia de los patrones de creación que construyen los productos de una vez, el patrón Builder construye el producto paso a paso, bajo el control del director. El director sólo obtiene el producto del constructor una vez que éste está terminado.

Consideraciones

1. El Director solo sabe hacer una cosa
2. Los Builders pueden saber hacer cosas que no requiera un Director pero si otro a. Ej: componentePsicologico()
3. Los Builders funcionan (generalmente) “injectando dependencias” para armar configuraciones en run-time.
4. Otros Directors pueden usar los mismos Builders.
5. Nuevas definiciones de Viajes de Egresado ⇒nuevos directores
6. Nuevos servicios ⇒ nuevos Builders

Errores comunes

Errores comunes

Composite	La jerarquía no es polimorfico Composite no tiene relación con otras clases de Jerarquía Composite no implementa add/remove Composite no delega en la coleccion
Builder	Director construye partes Director necesita diferentes builders al mismo tiempo Builder manda mensajes a Director Solo hay un Builder

Clase 4

Wrappers

Decorator

Objetivo

Agregar comportamiento a un objeto dinámicamente y en forma transparente

Problema

Cuando queremos agregar comportamiento adicional a ciertos objetos de una clase, una opción es usar herencia.
Sin embargo, presenta limitaciones cuando necesitamos que ese comportamiento se pueda agregar o quitar dinámicamente en tiempo de ejecución.
En esos casos, la herencia no es adecuada, ya que implica una decisión estática. Esta rigidez hace que la herencia no sea flexible para escenarios donde los objetos necesitan "mutar de clase" o modificar su comportamiento de forma dinámica.

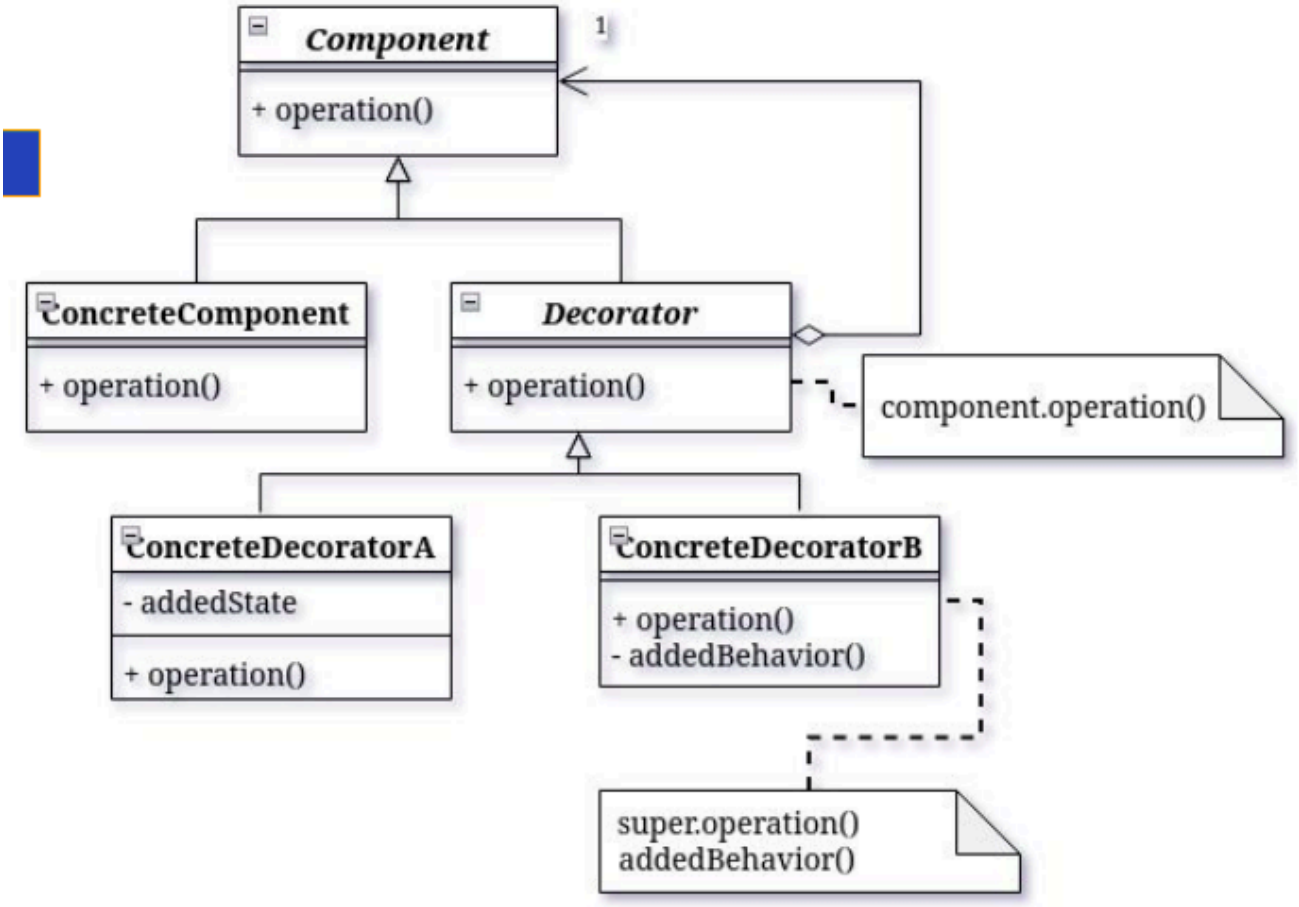
Usar decorator para

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

Solución

Definir un decorador que agregue comportamiento cuando sea necesario

Estructura



Consecuencias

- Permite mayor flexibilidad que la herencia
- Permite agregar funcionalidad incrementalmente
- Mayor cantidad de objetos, complejo para depurar

Implementación

- 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

Relacion con otros patrones

Decorator	Adapter
Decora el objeto para cambiarlo	Decora el objeto para cambiarlo
Decorator preserva la interfaz del objeto para el cliente	Convierte la interfaz del objeto para el cliente

Decorator	Adapter
Pueden y suelen anidarse	No se anidan

Decorator	Composite
Mantiene una estructura con sólo un siguiente	Mantiene una estructura 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
Su propósito es permitir que un objeto cambie su funcionalidad dinámicamente (agregando o cambiando el algoritmo que utiliza)	Su propósito es permitir que un objeto cambie su funcionalidad dinámicamente (agregando o cambiando el algoritmo que utiliza)
Cambia el algoritmo por fuera del objeto	Cambia el algoritmo por dentro del objeto

Proxy

Propósito

Proporcionar un intermediario de un objeto para controlar su acceso.

- Una de las razones para controlar el acceso a un objeto es posponer el costo de su creación e inicialización hasta que realmente necesitemos usarlo.

Aplicabilidad

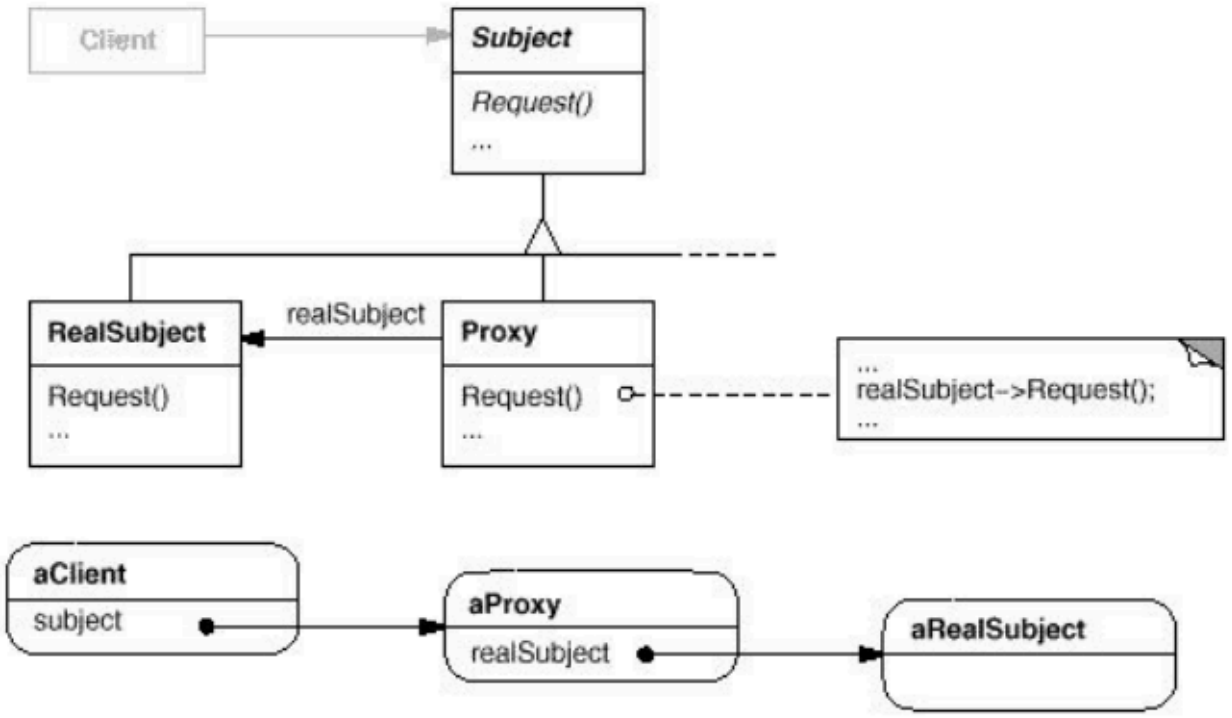
Cuando se necesito una referencia más flexible hacia un objeto

Aplicaciones del proxy

- 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

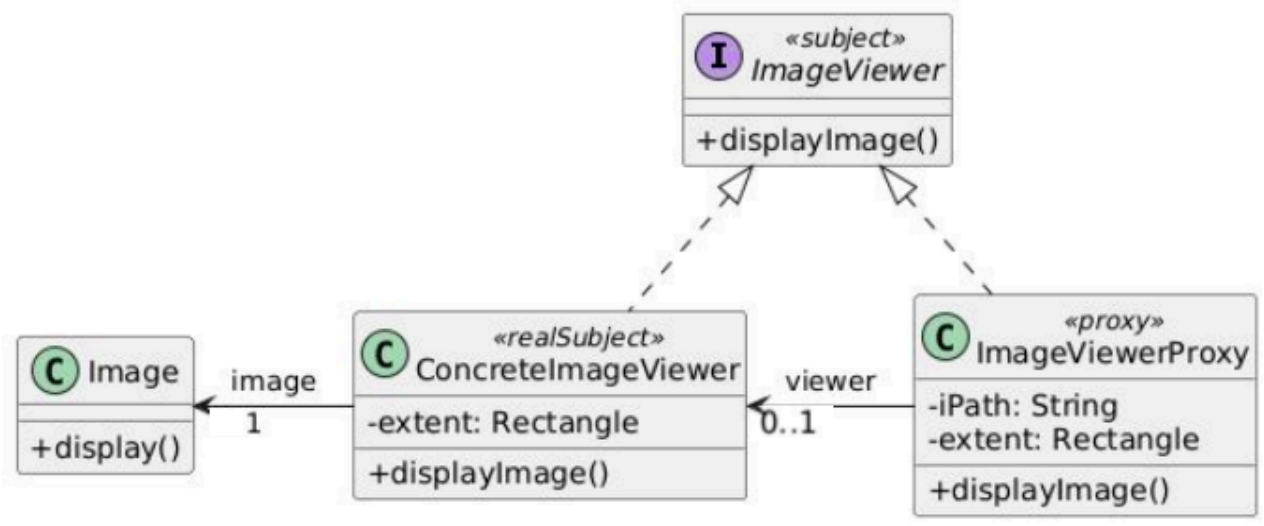
Estructura

- Colocar un objeto intermedio que respete el protocolo del objeto que está reemplazando.
- Algunos mensajes se delegarán en el objeto original. En otros casos puede que el proxy colabore con el objeto original o que reemplace su comportamiento.



Ejemplos de cada proxy

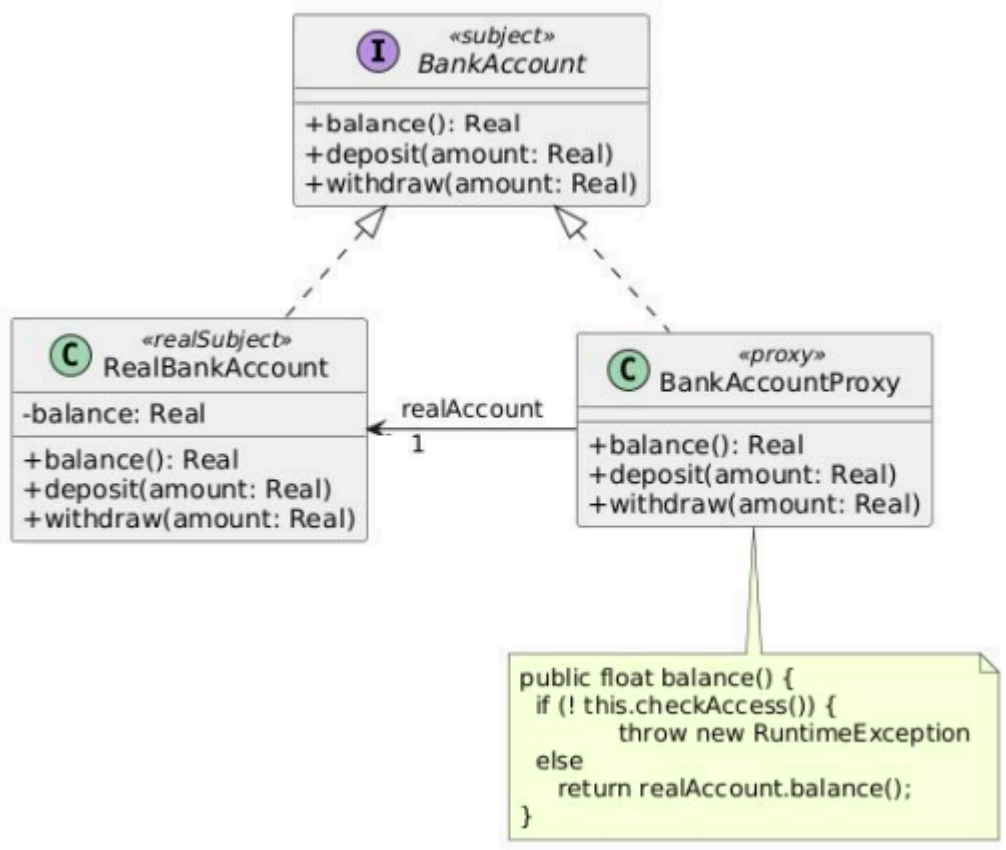
Virtual




```
public class ImageViewerProxy implements mageViewer {

private String iPath;
private Rectangle extent;
private ConcreteImageViewer viewer;

public ImageViewerProxy(String path, Rectangle ec) {
iPath = path;
extent = rec;
}
```

```
public void displayImage() {

    if (viewer == null) {
        viewer = new ConcreteImageViewer(iPath, extent);
    }
    viewer.displayImage();
}
}

public class ConcreteImageViewer implements ImageViewer {

private Image image;
private Rectangle extent;

public ConcreteImageViewer (String path, Rectangle rec) {
    // Costly operation
    image = Image.load(path);
    extent = rec;
    }

public void displayImage() {
    // Costly operation
    image.display();
    }

}
```

Protection



```
public class BankAccountProxy implements BankAccount {

private BankAccount realAccount;

public BankAccountProxy(BankAccount anAccount) {
    realAccount = anAccount;
}

public float balance() {
    if (! this.checkAccess()) {
        throw new RuntimeException("accesso denegado");
    }
    return realAccount.balance();
}

public void deposit(float amount) {
    if (this.checkAccess)
        realAccount.deposit(amount);
}
public void withdraw(float amount) {
    if (this.checkAccess)
        realAccount.withdraw(amount);
}
private boolean checkAccess() ...
}

public class RealBankAccount implements BankAccount {

private float balance;

public float balance() {
    return balance;
}

public void deposit(float amount) {
    balance += amount;
}
public void withdraw(float amount) {
    balance -= amount;
}

}
```

Acceso remoto

No se provee ejemplo por ahora pero si este texto:
Para acceder a objetos que se encuentran en otro espacio de memoria, en una arquitectura distribuida.
El proxy empaqueta el request, lo envía a través de la red al objeto real, espera la respuesta, desempaqueta la respuesta y retorna el resultado.

En este contexto el proxy suele utilizarse con otro objeto que se encarga de encontrar la ubicación del objeto real. Este objeto se denomina Broker, del patrón de su mismo nombre.

Consecuencias

A favor:

- Indirección en el acceso al objeto: El patrón introduce un nivel de indirección que permite controlar el acceso al objeto real, lo que ofrece flexibilidad en su uso
  En contra:
- Complejidad adicional

Relacion con otros patrones

Proxy	Adapter
Proporciona la misma interfaz que su sujeto	Proporciona una interfaz diferente al objeto que adapta

Proxy	Decorator
Controla el acceso a un objeto	Agrega una o más responsabilidades a un objeto

```
Ambos tienen propósitos diferentes
```

Clase 5

Refactoring to Patterns

La panacea de los patrones

- Los patrones son tentadores para no quedarnos envueltos y arrastrar un mal diseño.
- También nos pueden llevar al otro extremo. Por esto es muy importante conocer las consecuencias tanto positivas como negativas de un patrón.

Cómo ayuda el refactoring?

- Una vez que tengo código que funciona y pasa los tests
- Provee mecanismos que solucionan problemas de diseño
- A través de cambios pequeños
  - Hacer muchos cambios pequeños es más fácil y más seguro que un gran cambio
  - Me permite testear después de cada cambio
  - Cada pequeño cambio pone en evidencia otros cambios necesarios
- Una secuencia de refactorings me puede llevar a aplicar un gran cambio

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

Ejemplos:

- 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

Form Template method

- Dos o más métodos en subclases realizan pasos similares en el mismo orden, pero los pasos son distintos
    - Generalizar los métodos excluyendo sus pasos en métodos de la misma signatura y luego subir a la superclase común en el 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 y contras

Pros	Contras
- 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 facilmente un algoritmo	- Complica el diseño cuando las subclases deben implementar muchos métodos para sustanciar el algoritmo

Replace conditional Logic with Strategy

- Existe lógica condicional en un método que controla qué variante ejecutar entre distintas posibles
    - 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.

Como setear la estrategia en el contexto?
Si no hay muchas combinaciones de Strategies y contextos, es una buena práctica aislar el código del cliente de preocuparse de cómo instanciar las subclases de Strategy.
Encapsulate Classes with Factory [Kerievsky]: definir un método en el contexto que retorne una instancia del mismo con el strategy correspondiente, por cada subclase de Strategy.

Replace State-Altering Conditionals with State

- Las expresiones condicionales que controlan las transiciones de estado de un objeto son complejas.
    - Reemplazar los condicionales con States que manejen estados específicos y transiciones entre ellos.

Motivación

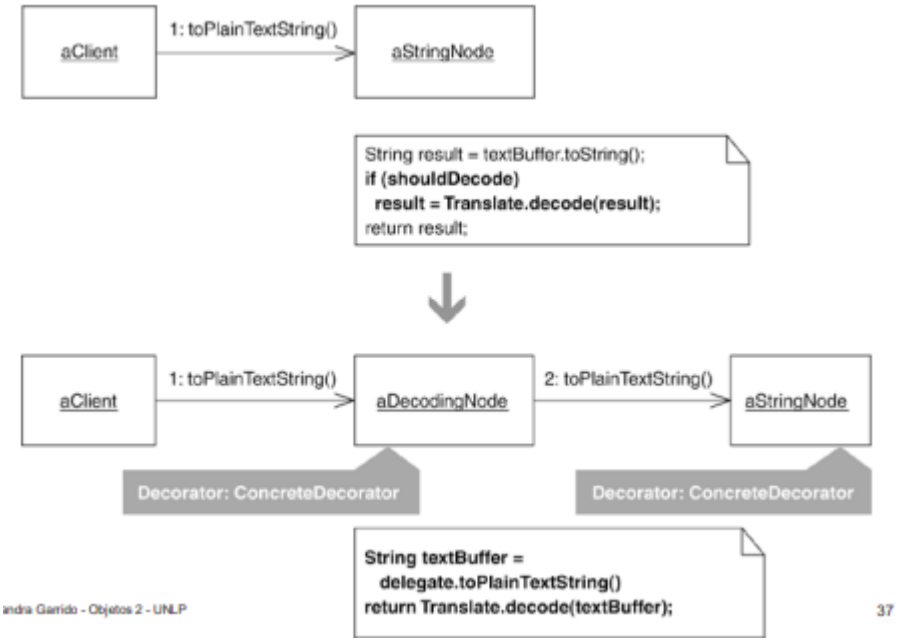
- Obtener una mejor visualización con una mirada global, de las transiciones entre estados.
- Cuando la lógica condicional entre estados dejó de ser fácil de seguir o extender.
- Cuando aplicar refactorings más simples, como “Extract Method” o “Consolidate Conditional Expressions” no alcanzan

Mecánica

1. Aplicar "Replace Type-code with class" para crear una clase que será la superclase del State a partir de una v.i. que mantiene el estado

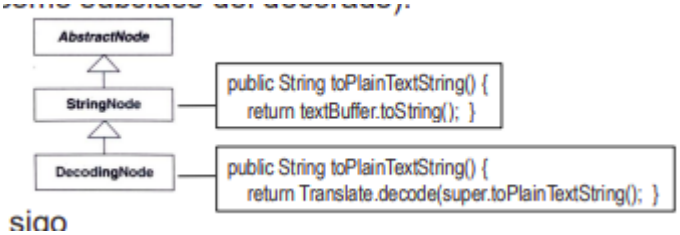
2. Aplicar “Extract Subclass” [F] para crear una subclase del State por cada uno de los estados de la clase contexto.
3. Por cada método de la clase contexto con condicionales que cambiar el valor del estado, aplicar “Move Method” hacia la superclase de State.
4. Por cada estado concreto, aplicar “Push 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 Embellishment to Decorator



Mecánica

1. Identificar la superclase (or interface) del objeto a decorar (clase Component del patrón). Si no existe, crearla
2. Aplicar Replace Conditional Logic with Polymorphism (crea decorator como subclase del decorado)



sigo

3. Aplicar Replace Inheritance with delegation (decorator delega en decorado como clase hermana)
4. Aplicar Extract parameter en decorator para asignar decorado

Patrón 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 hace nada" y oculta esos detalles a sus colaboradores

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

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

Refactoring: Introduce null object

La lógica para manejarse con un valor nulo en una variable está duplicado por todo el código

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

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

Conceptos básicos

- 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

Clase 6

Reuso, Liberías y Frameworks

Reuso

Ventajas	Desventajas

- Menor Esfuerzo
- Menor tiempo de desarrollo
- Menor incertidumbre
- Mayor consistencia
- Mayor confiabilidad
  - Funciono para n casos anteriores
- Promueve trabajo en equipo
- Adquisición tecnológica
  - Llave en mano
  - Know-how
	- Requiere mantener configuraciones de software (versiones de dependencias)
	- Riesgo de dependencias no deseadas
	- Curva de aprendizaje
	- Código externo
	  - Dependencia tecnológica
	  - Resistencia a adopción
	  - Riesgo de Seguridad
	- Desarrollo de Código reusable
	  - Requiere tiempo
	  - Requiere conocimiento detallado
	  - Basado en casos
	  - Experto en el tema

Librería OO

- Se reusa código
- (OO) ⇒ Conjunto de clases
  - Funcionalidad
    - Instancias
    - Clase (Estática)
  - El sistema/módulo
    - Crea instancias
    - Invoca la funcionalidad

Framework

- Se reusa
  - Una manera de ejecutar
  - Una manera de instanciar
  - Una manera de extender
- Es incompleto. No hace nada concreto
- (OO) ⇒ Conjunto de clases
  - 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

Frameworks

Es una aplicación “semi-completa”, “reusable”, que puede ser especializada para producir aplicaciones a medida…
…un conjunto de clases concretas y abstractas, relacionadas para proveer una arquitectura reusable que implementa una familia de aplicaciones (relacionadas)…

Framework de caja blanca

- 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
  ⇒ es una Caja Blanca

HotSpots vs FrozenSpot

FrozenSpot: aspecto del framework que afecta a todas las instanciaciones y que no se puede modificar (marca indeleble)
HotSpot: estructura en el código que permite modificar el comportamiento del framework, para instanciar y para extender.

Resumen de Frameworks

- 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 instanciación
- Tipos de frameworks
  - Aplicación: Desktop, webapps, tcpservers
  - Manejo Datos: ORDB, pipelines, NRDB
  - Sistemas Distribuidos: mensajes, eventos, rpc
  - Testing: unit, web pages

- oo2.next()
  - Blackbox frameworks: instanciación por composición
  - Frameworks & Design Patterns