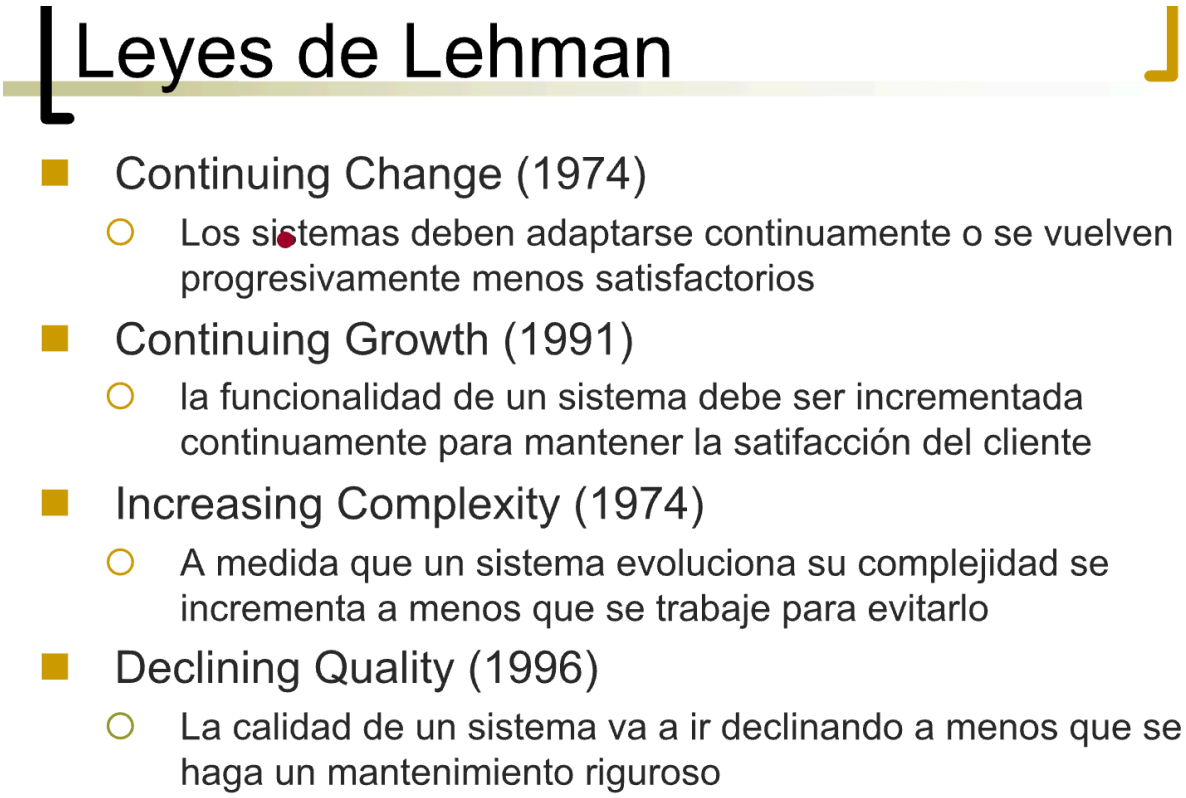
Refactoring y Refactoring to Patterns



Refactoring: Transformación que preserva el comportamiento, pero mejora el diseño.

- mejora la **organización**, **legibilidad**, **adaptabilidad** y **mantenibilidad** del código luego de que haya sido escrito.
- NO altera el comportamiento externo del sistema.

El refactoring es una técnica escencial del desarrollo ágil.

- Eliminar código duplicado
- Simplificar lógicas complejas
- Clarificar códigos

Cuándo

- 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

Testear después de cada cambio. No significa ningún cambio para el cliente, ven el sistema exactamente igual.

Hacerlo manualmente es muy costoso. Se usan herramientas de refactoring. En Eclipse seleccionando el texto y con click derecho se pude hacer.

No puedo agregar funcionalidad y hacer refactoring a la vez. Hay que cambiar el "modo" de trabajo, dependiendo de lo que se está haciendo.

No tengo que preocuparme en este apartado por la performance. Eso es algo a futuro, luego de tener el código refactorizado.

¿Por qué refactoring es importante?

- Ganar en la compresión del código.
- Reducir el costo de mantenimiento debido a los cambios inevitables que sufrirá el sistema.
- Facilitar la detección de bugs.
- La clave principal es poder agregar funcionalidad más rápido después de refactorizar.

Extract Method

Código que realiza exactamente lo mismo y aparece dos o más veces. Se lo hace un método y se lo llama desde todos los lugares en los que se lo necesita.

Motivación

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

▼ **Mecánica**

- **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 extraido. 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
  8. Reemplazar código en método original por llamada
  9. Compilar

Move Method

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

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

▼ **Mecánica**

- Mecánica:
  1. Revisar las v.i. usadas por el método a mover. Tiene sentido moverlas también?
  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 como 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

Replace Conditional With Polymorpshism

Reemplazar condicionales con polimorfismo. Controlar si el condicional está en la clase adecuada, ya que puede pasar que antes se tenga que realizar un Move Method. Y si el condicional es parte de un método largo: Extract Method.

▼ **Mecánica**

- Mecánica:
  1. Crear la jerarquía de clases necesaria
  2. Si el condicional es parte de un método largo: Extract Method
  3. Por cada subclase:
     1. Crear un método que sobreescribe al método que contiene el condicional
     2. Copiar el código de la condición correspondiente en el método de la subclase y ajustar
     3. Compilar y testear
     4. Borrar la condición y código del branch del método en la superclase
     5. Compilar y testear
  4. Hacer que el método en la superclase sea abstracto

Replace Temp With Query

Reemplazar temporal con una llamada. **Motivación**:

- 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.

Solución:

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

Pull Up Method

Varias subclases con el mismo método. Ese método es llevado a la superclase.

▼ **Mecánica**

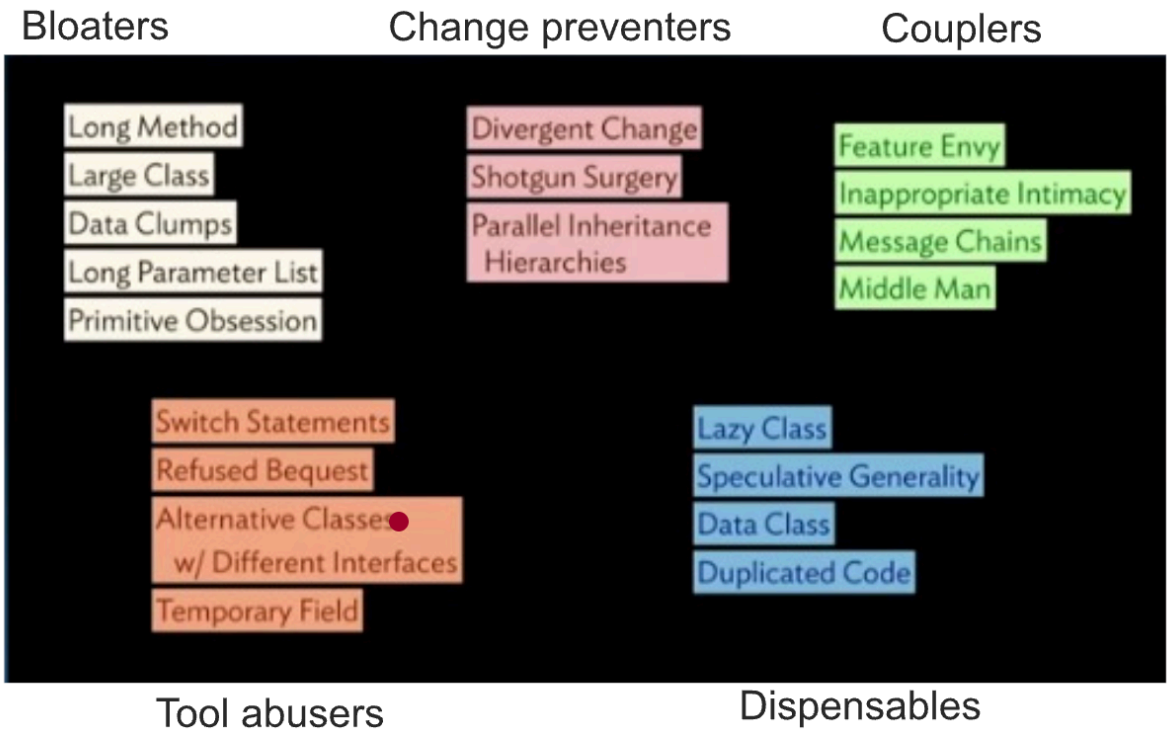
1. Asegurarse que los métodos sean idénticos. Si no, parametrizar
2. Si el selector del método es diferente en cada subclase, renombrar
3. Si el método llama a otro que no está en la superclase, declararlo como abstracto en la superclase
4. Si el método llama a un atributo declarado en las subclases, 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

- **Preserve Whole Object**: En vez de mandar una lista de parámetros larga de un objeto, se envía el objeto entero.
- EJ5 de REFACTORING, Cuando se crean subclases que extienden y reemplazan una variable string tipo:
  - El más adecuado es **Replace Conditional With Polymorphism**.
  - **Replace Type Code with Subclasses**: es un refactoring más específico para reemplazar un `String tipo` por subclases, y muchas veces se combina con el anterior.
  - **Replace Type Code with State/Strategy**: si el comportamiento varía mucho y necesitás inyectar la lógica, pero no necesariamente usar herencia.

Bad Smells

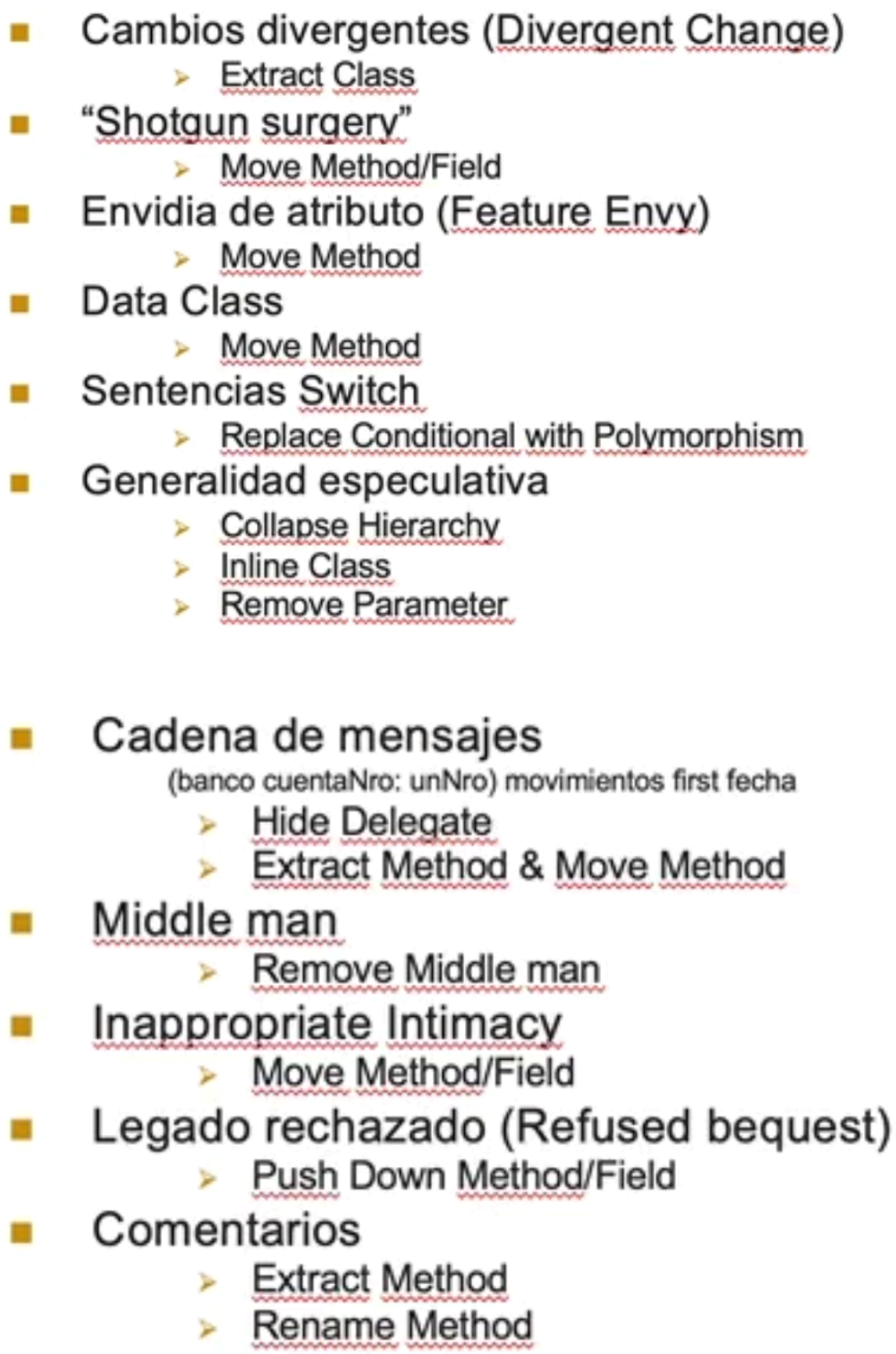
Indicios de problemas que requieren la aplicación de refactorings. Posteriormente llamados **code smells**.

- Duplicated Code
- Large Class
- Long Method
- Data Class: Clase que sólo tiene datos, getters y setters
- Feature Envy: Envidia
- Long Parameter List: Difícil de reusar
- Switch Statements



- **Bloaters**: Hace las cosas más grande de lo que en realidad deberían ser.
- **Tool Abusers**: Abusar de herramientas.
- **Change Preventers**: Previene que yo me anime a ser cambios. Código muy difícil de cambiar. Para hacer un cambio tengo que ir a arreglar un montón de agujeros.
- **Dispensables**: Cosas que realmente no necesito.
- **Couplers**: Producen acoplamiento.
    - **Inappropiate Intimacy**: Clases que se conocen demasiado entre sí.

- Código duplicado
  - Extract Method
  - Pull Up Method
  - Form Template Method
- Métodos largos
  - Extract Method
  - Decompose Conditional
  - Replace Temp with Query
- Clases grandes
  - Extract Class
  - Extract Subclass
- Muchos parámetros
  - Replace Parameter with Method
  - Preserve Whole Object
  - Introduce Parameter Object



Shotgun Surgery: Código que al hacer un cambio mínimo requiere muchos cambios en el resto del código.

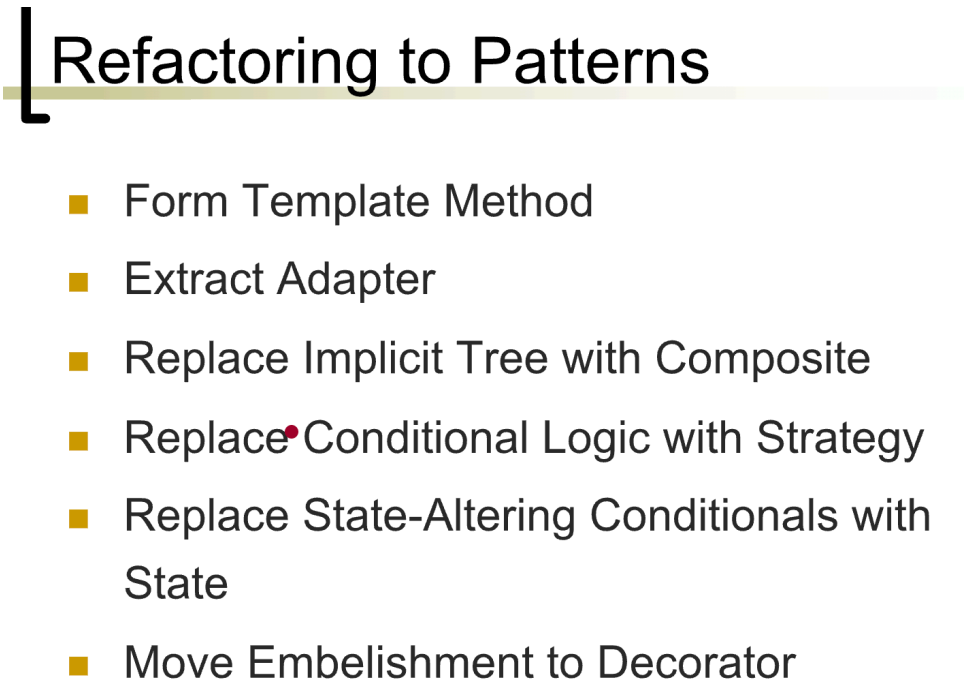
Legado Rechazado: Una subclase que redefine mucho lo de su superclase o no lo termina de utilizar.

Refactoring to Patterns

El refactoring también se realiza para simplificarle la tarea al siguiente programador que tenga que leer mi código. Estos refactoring de patrones son realizados para llegar a introducir los patrones en el código.

Aplicar patrones de manera excesiva termina siendo contraproducente.

Algunos de los que aparecen en el libro:



Ninguna herramienta logra realizar este tipo de refactoring, ya que es muy complejo.

Form Template Method

Refactoring “Form 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

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

▼ Pros y Contras del Patrón Form **Template Method**

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

Replace Conditional Logic With Strategy

(Replace Conditional With Strategy)

Replace Conditional Logic with 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.

- **Extract Parameter**: consiste en **extraer un valor que se calcula o accede dentro de un método y convertirlo en un parámetro del mismo**.
- Pasos:
  - Se crea la clase abstracta de Strategy.
  - Se hace Move Method del método con los condicionales a la clase de Strategy.
  - Se aplica el Refactoring **Replace Conditional With Polymorphism**.

Replace State-Alterning Conditionals with State

(Replace Conditional With State)

Replace State-Altering Conds. with State. Mecánica

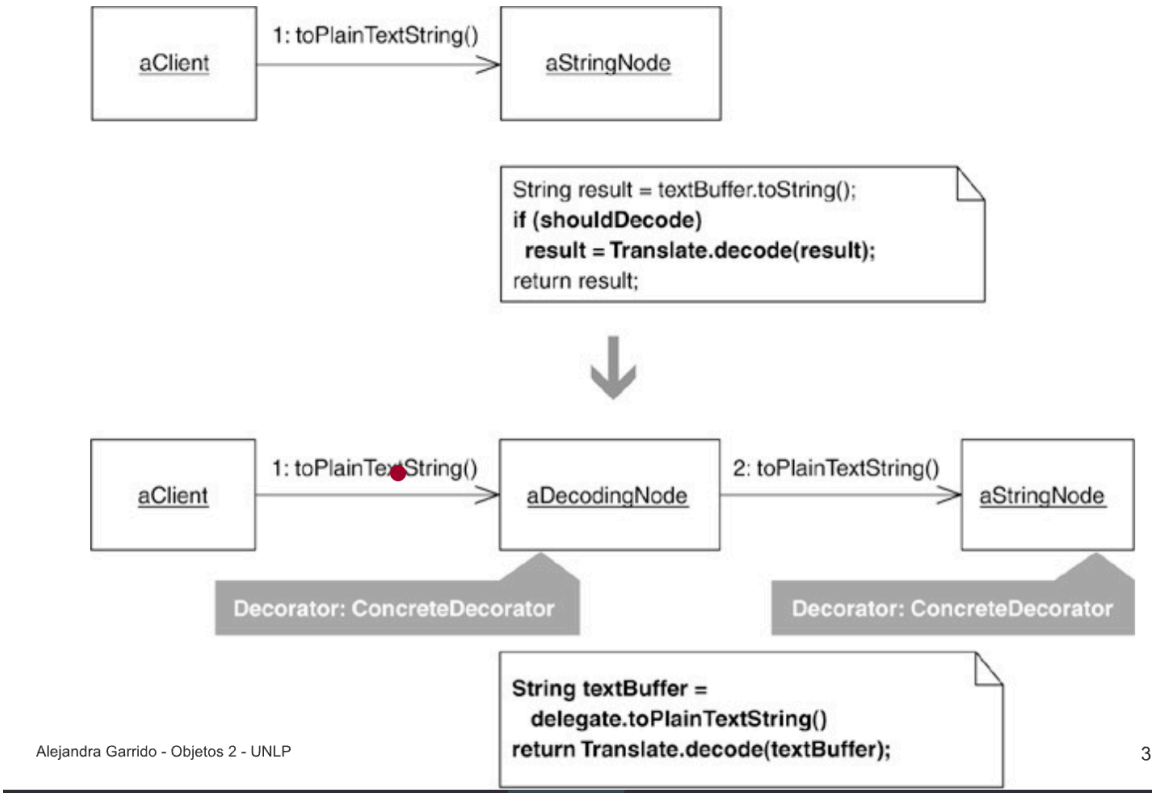
1. Aplicar *"Replace Type-Code with Class"* para crear una clase que será la superclase del State a partir de la v.i. que mantiene el estado
2. Aplicar "*Extract Subclass"* [F] para crear una subclase del State por cada uno de los estados de la clase contexto.
3. Por cada método de la clase contexto 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.

- 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.

Move Embellishment to Decorator (Decorador)

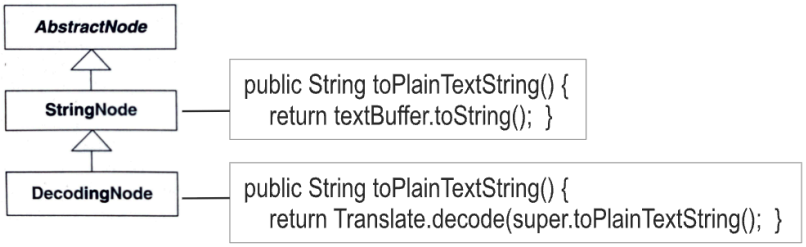
▼ Imagen de ejemplo



- Se elimina el if. Si hay que aplicar el decode se hace con el Decorador, sino no.
- Pero a su vez aplicar un Decorator por un if, es en parte un sobrediseño. Incorpora toda la complejidad del Decorator por un if.

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).



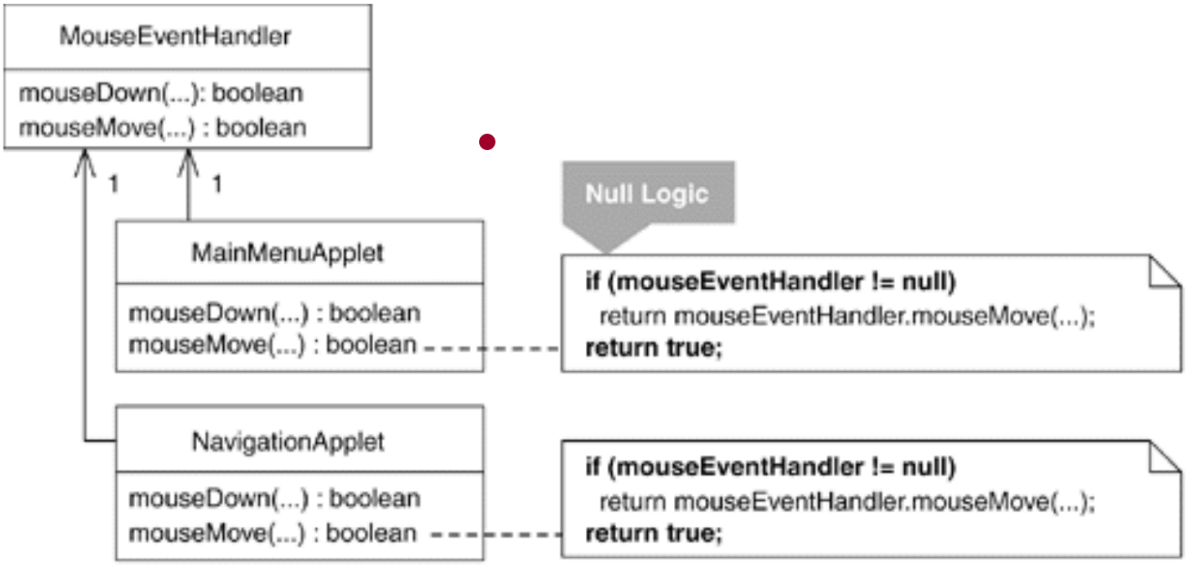
Alcanza? Si no sigo

3. Aplicar *Replace Inheritance with Delegation* (decorator delega en decorado como clase “hermana”)
4. Aplicar *Extract Parameter* en decorator para asignar decorado

Introduce Null Object

Introduce Null Object

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



- Asegurar que nadie se olvide de chequar por null. Se utiliza un objeto que representa el Null Object. Se elimina la necesidad de controlar por null.
- Este refactoring me permite introducir este objeto Null, eliminando los condicionales.

Herramientras de Refactoring

- Refactorizar a mano es demasiado costoso, lleva tiempo y puede introducir errores.
- Características de las herramientras:
  - potentes para realizar refactorings útiles
  - restricivas para preservar comportamiento del programa
  - interactivas, de manera que el chequeo de precondiciones no debe ser extenso

Las herramientas

- Sólo chequean lo que sea posible desde el árbol de sintaxis y la tabla de símbolos.
- Puedes ser demasiado conservativas (no realizan un refactoring si no pueden asegurar preservación del comportamiento) o asumir buenas técnicas de programación.
- Cada vez más potentes en la detección de code smells.

Deuda Técnica

- Permite visualizar las consecuencias de un diseño quick & dirty.
- **Capital de la deuda**: costo de remediar los problemas de diseño (costo del refactoring).
- **Interés de la deuda**: Costo adicional en el futuro por mantener software con deuda técnica acumulada.

La deuda técnica permite demostrar la importancia del refactoring y la necesida de hacerlo. Cómo también la importancia de no postergarlo. Es como una bola de nieve que cada vez se va haciendo más grande, ya que cada vez es más costoso solucionar los bad smells.

Cuánto más flexible es mi código (menos bad smells) más fácil es implementar nueva funcionalidad.

También se van solucionando los bad smells que más riesgo me suponen, o del área en la cual quiero extender funcionalidad.

Importancia del Refactoring

- Nuestra única defensa contra el deterioro del software.
- Facilita la incorporación de código.
- Permite preocuparse por la generalidad mañana.
- Permite ser ágil en el desarrollo.