Cambios

Leyes de Lehman

- Continuing Change (1974): Los sistemas deben adaptarse continuamente o se vuelven progresivamente menos satisfactorios
- Continuing Growth (1991): La funcionalidad de un sistema debe ser incrementada continuamente para mantener la satifacción del cliente
- Increasing Complexity (1974): A medida que un sistema evoluciona su complejidad se incrementa a menos que se trabaje para evitarlo
- Declining Quality (1996): La calidad de un sistema va a ir declinando a menos que se haga un mantenimiento riguroso

Hay que prepararse para el cambio, reaccionar rápido. El mantenimiento puede ser: correctivo, evolutivo, adaptativo, perfectivo, preventivo. Entender código existente: 50% del tiempo de mantenimiento

La iteración es fundamental ya que los cambios de aplican de una iteración a otra. Los cambios de una iteración a la siguiente pueden involucrar únicamente cambios estructurales entre componentes existentes que no cambian la funcionalidad

Refactoring

Refactoring es una transformación que preserva el comportamiento, pero mejora el diseño. Es el proceso a través del cual se cambia un sistema de software

- para mejorar la organización, legibilidad, adaptabilidad y mantenibilidad del código luego que ha sido escrito
- que NO altera el comportamiento externo del sistema

- ✓ Refactoring (sustantivo): cada uno de los cambios catalogados
- ✓ Refactor (verbo): el proceso de aplicar refactorings

El refactoring:

Implica:

- Eliminar duplicaciones
- 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

¿Cómo ayuda el refactoring?

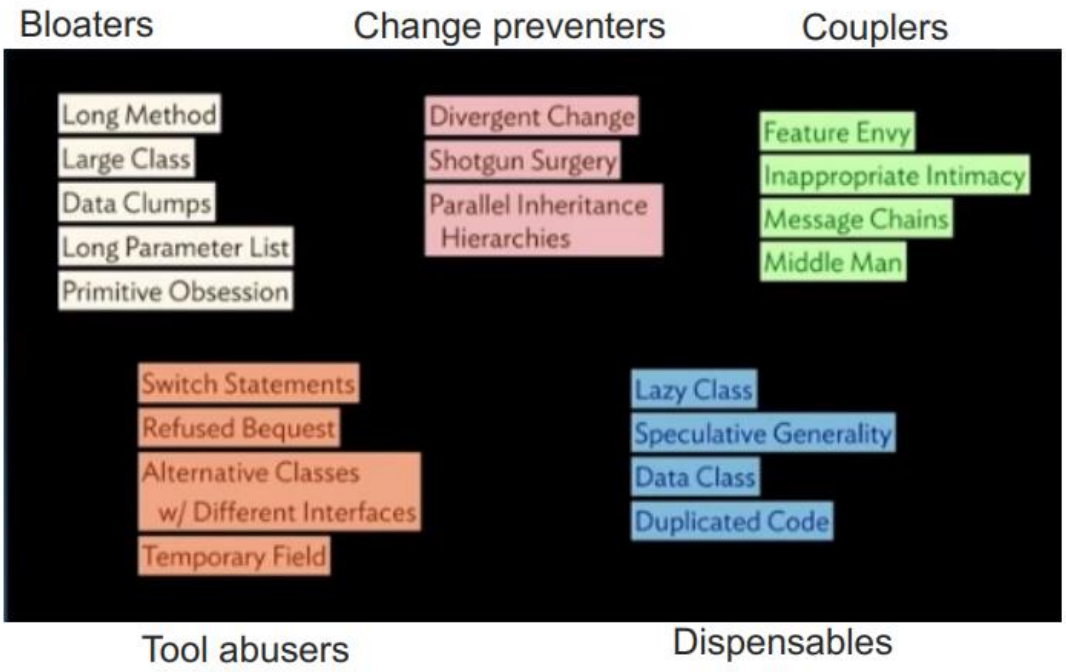
- Introduce mecanismos que solucionan problemas de diseño

- A través de cambios **pequeños**
    - Hacer muchos cambios pequeños es más fácil y seguro que un gran cambio
    - Cada pequeño cambio pone en evidencia otros cambios necesarios

Catálogo de BAD SMELLS

Indicios de problemas que requieren la aplicación de refactorings. Posteriormente llamados CODE SMELLS

Categorización:



Duplicated Code

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

Large Class

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

Large Method

- Un método tiene muchas líneas de código
- Problema:

- Cuanto más largo es un método, más difícil es entenderlo, cambiarlo y reusarlo

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)
- Problemas:
  - 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 Class

- 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 el objetos llamador y el llamado.

Catálogo de refactorings

Organización catálogo Fowler:

Composición de métodos

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

Extract Method:

- Problema: Tiene un fragmento de código que se puede agrupar.
- Solución: Mueva este código a un nuevo método (o función) separado y reemplace el código anterior con una llamada al método.

Cómo refactorizar

1. Cree un nuevo método y asígnele un nombre que haga evidente su propósito.
2. Copie el fragmento de código relevante en su nuevo método. Elimine el fragmento de su ubicación anterior y haga una llamada para el nuevo método allí.
    a. Encuentre todas las variables utilizadas en este fragmento de código. Si se declaran dentro del fragmento y no se usan fuera de él, simplemente déjelos sin cambios: se convertirán en variables locales para el nuevo método.

3. Si las variables se declaran antes del código que está extrayendo, deberá pasar estas variables a los parámetros de su nuevo método para poder usar los valores contenidos previamente en ellos. A veces es más fácil deshacerse de estas variables recurriendo a Replace Temp with Query .
4. Si ve que una variable local cambia en su código extraído de alguna manera, esto puede significar que este valor modificado será necesario más adelante en su método principal. ¡Doble verificación! Y si este es el caso, devuelva el valor de esta variable al método principal para que todo siga funcionando.

Inline Method

Replace Temp with Query:

- Problema: Usted coloca el resultado de una expresión en una variable local para su uso posterior en su código.
- Solución: Mueva la expresión completa a un método separado y devuelva el resultado. Consulta el método en lugar de usar una variable. Incorpore el nuevo método en otros métodos, si es necesario.

Cómo refactorizar

1. Asegúrese de que se asigna un valor a la variable una vez y solo una vez dentro del método.
2. Utilice Extract Method para colocar la expresión de interés en un nuevo método. Asegúrese de que este método solo devuelva un valor y no cambie el estado del objeto.
3. Reemplace la variable con una consulta a su nuevo método.

Split Temporary Variable

Replace Method with Method Object

Substitute Algorithm

Mover aspectos entre objetos

- Ayudan a mejorar la asignación de responsabilidades

Move Method:

- Problema: Un método se usa más en otra clase que en su propia clase.
- Solución: Cree un nuevo método en la clase que más utilice el método y, a continuación, mueva el código del método anterior allí. Convierta el código del método original en una referencia al nuevo método en la otra clase o elimínelo por completo.

Cómo refactorizar

1. Verifique todas las funciones utilizadas por el método anterior en su clase. Puede ser una buena idea moverlos también. Como regla general, si una característica es utilizada solo por el método en consideración, ciertamente debe mover la característica a él. Si la característica también es utilizada por otros métodos, también debe mover estos métodos. A veces es mucho más fácil mover una gran cantidad de métodos que establecer relaciones entre ellos en diferentes clases.
   Asegúrese de que el método no esté declarado en superclases y subclases. Si este es el caso, deberá abstenerse de moverse o implementar una especie de polimorfismo en la clase receptora para garantizar la funcionalidad variable de un método dividido entre las clases donantes.
2. Declare el nuevo método en la clase del destinatario. Es posible que desee dar un nuevo nombre al método que sea más apropiado para él en la nueva clase.
3. Declare el nuevo método en la clase del destinatario. Es posible que desee dar un nuevo nombre al método que sea más apropiado para él en la nueva clase.
   Ahora tiene una forma de referirse al objeto destinatario y un nuevo método en su clase. Con todo esto en su haber, puede convertir el antiguo método en una referencia al nuevo método.

Move Field

Extract class

Inline Class

Remove Middle Man

Hide Delegate

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

Decompose Conditional

Consolidate Conditional Expression

Consolidate Duplicate

Conditional Fragments

Replace Conditional with Polimorfism:

- Problema: Tiene un condicional que realiza varias acciones según el tipo de objeto o las propiedades.
- Solución: Cree subclases que coincidan con las ramas del condicional. En ellos, cree un método compartido y mueva el código de la rama correspondiente del condicional a él. Luego reemplace el condicional con la llamada al método relevante. El resultado es que la implementación adecuada se logrará a través del polimorfismo según la clase de objeto.

Como refactorizar

1) Si el condicional está en un método que también realiza otras acciones, ejecute el Extract Method.
2) Para cada subclase de jerarquía, redefina el método que contiene el condicional y copie el código de la rama condicional correspondiente a esa ubicación.
3) Elimina esta rama del condicional.
4) Repita el reemplazo hasta que el condicional esté vacío. Luego elimine el condicional y declare el método abstracto.

Simplificación en la invocación de métodos

- Sirven para mejorar la interfaz de una clase

Rename Method – Rename Field:

- Problema: El nombre de un método no explica lo que hace el método.
- Solución: Cambie el nombre del método.

Como refactorizar

1) Vea si el método está definido en una superclase o subclase. Si es así, también debe repetir todos los pasos en estas clases.
2) El siguiente método es importante para mantener la funcionalidad del programa durante el proceso de refactorización. Cree un nuevo método con un nuevo nombre. Copie el código del método anterior. Elimine todo el código del método anterior y, en su lugar, inserte una llamada para el nuevo método.
3) Encuentre todas las referencias al método anterior y reemplácelas con referencias al nuevo.
4) Eliminar el método antiguo. Si el método anterior es parte de una interfaz pública, no realice este paso. En su lugar, marque el método antiguo como obsoleto.

Preserve Whole Object

Introduce Parameter Object

Parameterize Method

Manipulación de la generalización

Ayudan a mejorar las jerarquías de clases

Pull Up

Push Down Field

Pull Up Method

- Problema: Sus subclases tienen métodos que realizan un trabajo similar.
- Solución: Haga que los métodos sean idénticos y luego muévalos a la superclase relevante.

Como refactorizar

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

Push Down Method

Pull Up Constructor Body

Extract Subclass / Superclass

Collapse Hierarchy

Replace Inheritance with

Delegation

Replace Delegation with

Inheritance

Code smells –Catálogo Fowler (1)

- 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

Code smells –Catálogo Fowler (2)

- Cambios divergentes (Divergent Change)
  - Extract Class
- "Shotgun surgery"
  - Move Method/Field
- Envidia de atributo (Feature Envy)
  - Move Method
- Data Class
  - Move Method
- Sentencias Switch
  - Replace Conditional with Polymorphism
- Generalidad especulativa
  - Collapse Hierarchy
  - Inline Class
  - Remove Parameter

Code smells –Catálogo Fowler (3)

- Cadena de mensajes
  (banco cuentaNro: unNro) movimientos first fecha
  - Hide Delegate
  - Extract Method & Move Method
- Middle man
  - Remove Middle man
- Inappropriate Intimacy
  - Move Method/Field
- Legado rechazado (Refused bequest)
  - Push Down Method/Field
- Comentarios
  - Extract Method
  - Rename Method

¿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
- Se usan herramientas de refactoring

¿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
- La clave: poder agregar funcionalidad más rápido después de refactorizar

Refactoring To Patterns

La sobre-ingeniería es tan peligrosa como la poca ingeniería.

- Sobre-ingeniería (over-engineering) significa construir software más sofisticado de lo que realmente necesita ser.
    - Por qué se hace?
        - Para acomodar futuros cambios (pero no se puede predecir el futuro)
        - Para no quedar inmerso y acarrear un mal diseño (pero a la larga, el encanto de los patrones puede hacer que perdamos de vista formas más simples de escribir código)
    - Consecuencias:
        - El código sofisticado, complejo, se queda y complica el mantenimiento.
        - Nadie lo entiende. Nadie lo quiere tocar.
        - Como otros no lo entienden generan copias, código duplicado.
- Poca ingeniería (under-engineering) significa producir software con un diseño pobre.

Introducimos patrones sólo cuando se necesitan

<u>Form Template Method:</u>

- Problema: 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

Como refactorizar

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

Extract Adapter

Replace Implicit Tree with Composite

<u>Replace Conditional Logic with Strategy:</u>

- Problema: 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

Como refactorizar

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

   a. Definir una v.i. en el contexto para referenciar al strategy y un setter (generalmente el constructor del contexto)
   b. Dejar un método en el contexto que delegue
   c. Elegir los parámetros necesarios para pasar al strategy (el contexto entero? Sólo algunas variables? Y en qué momento?)
   d. 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.
   a. 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 y contras

- ☑ Clarifica los algoritmos al reducir o remover la lógica condicional.
- ☑ Simplifica una clase moviendo variaciones de un algoritmo a una jerarquía separada
- ☑ Permite reemplazar un algoritmo por otro en runtime
- ☒ Complica el diseño cuando se podría solucionar con subclases o simplificando los condicionales.

¿State o Strategy?

- 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
- El estado es privado del objeto, ningún otro objeto sabe de él. vs. El Strategy suele setearse por el cliente, que debe conocer las posibles estrategias concretas.
- Cada State puede definir muchos mensajes. vs. Un Strategy suele tener un único mensaje público.
- Los states concretos se conocen entre si. vs Los strategies concretos no

<u>Replace State-Altering Conditionals with State</u>

- Problema: Las expresiones condicionales que controlan las transiciones de estado de un objeto son complejas
- Solución: 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

Como refactorizar

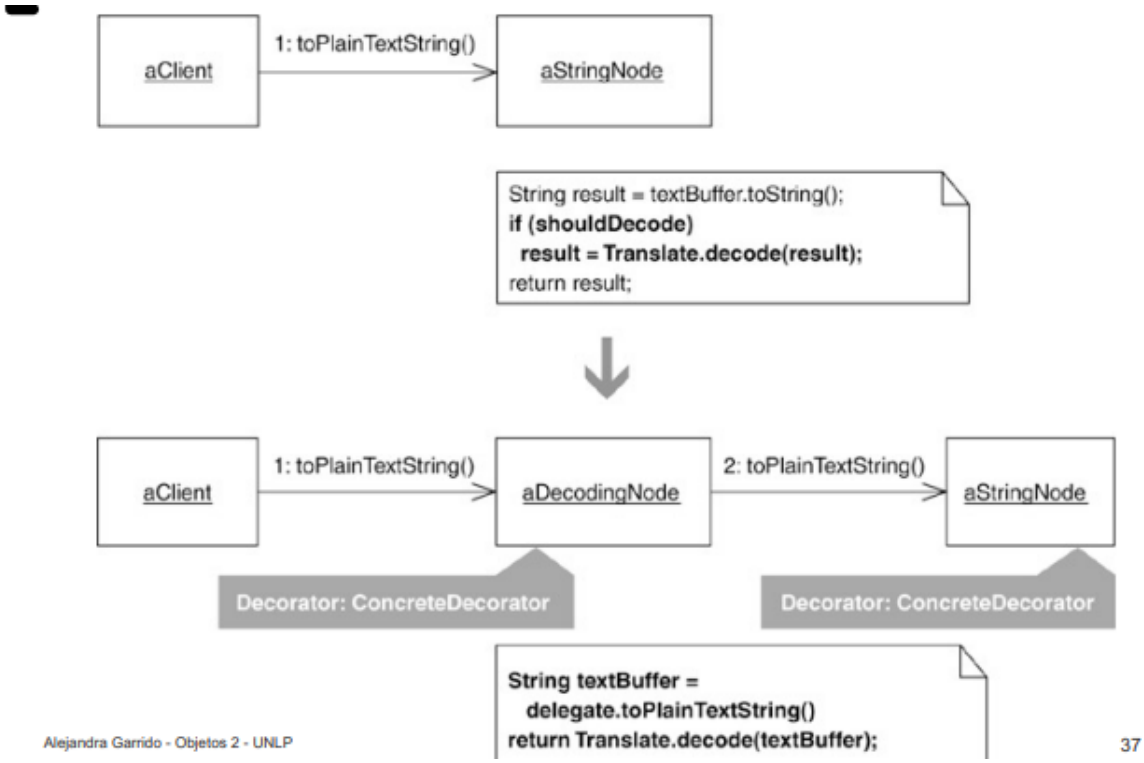
1) Aplicar “Replace Type-Code with Class” para crear una clase que será la superclase del State a partir de la v.i. que mantiene el estado

2) Aplicar “Extract Subclass” [F] para crear una subclase del State por cada uno de los estados de la clase context
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 y contras

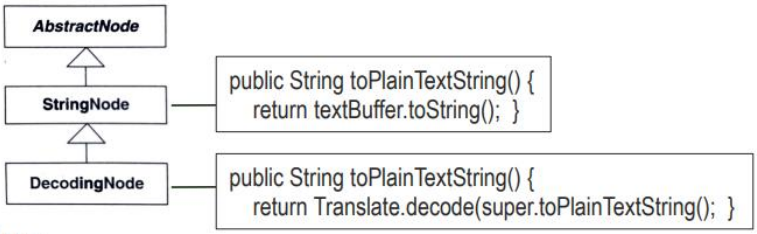
- ☑ Reduce o remueve la lógica condicional de cambio de estado.
- ☑ Simplifica la lógica compleja de transiciones.
- ☑ Provee una mejor visualización de alto nivel de los posibles estados y transiciones
- ☒ Complica el diseño cuando la lógica de transición de estados ya es fácil de seguir.

Move Embelishment to Decorator



Como refactorizar

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).



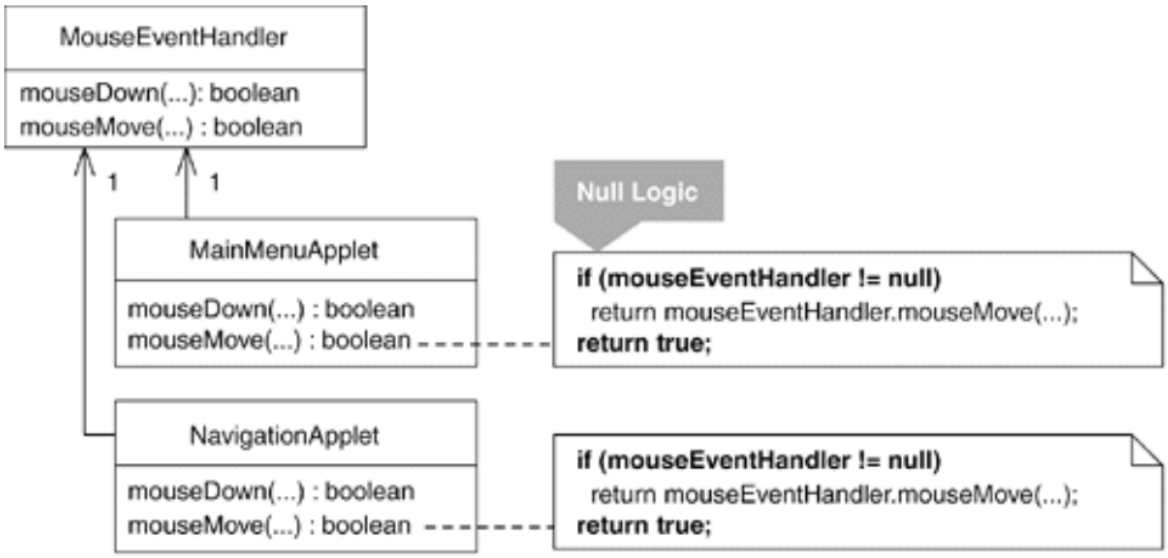
Alcanza? Si no sigo

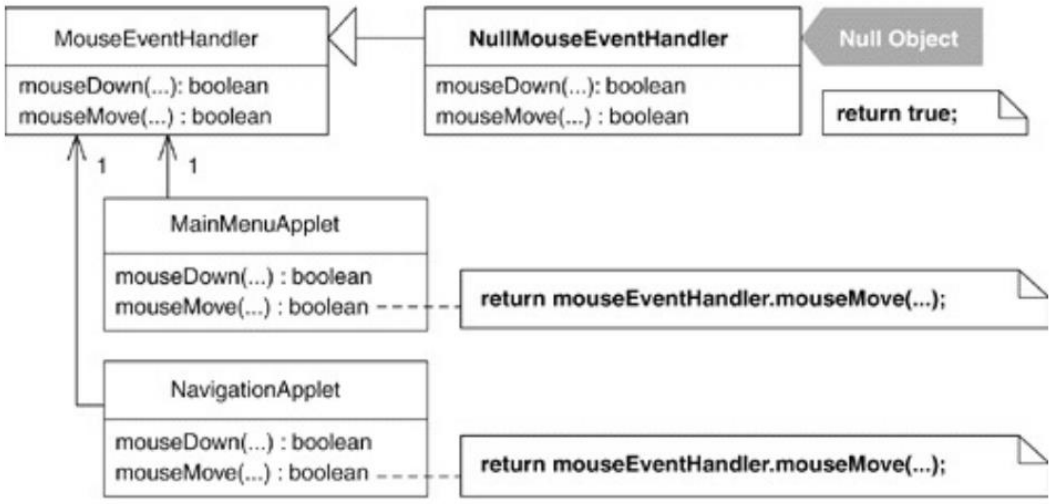
3) Aplicar Replace Inheritance with Delegation (decorator delega en decorado como clase "hermana")
4) Aplicar Extract Parameter en decorator para asignar decorado

Introduce Null Object

- Problema: Dado que algunos métodos regresan nullen lugar de objetos reales, tiene muchas comprobaciones nullen su código.
- Solución: En lugar de null, devuelva un objeto nulo que muestre el comportamiento predeterminado.

Reemplazar la lógica de testeo por null con un Null Object. La lógica para manejarse con un valor nulo en una variable está duplicado por todo el código





Concepto asociado al refactoring: Deuda Técnica

- Concepto que introdujo Ward Cunningham para explicar a los managers la necesidad de refactoring
- 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

Refactoring y testing

El refactoring puede romper un test aunque sea correcto, por eso tal vez hayan que cambiar los test, mas que nada el BeforeEach