

REFACTORIZACIÓN

UNIDAD 3: PRÁCTICAS DE DISEÑO ÁGIL



Temario

- Conceptos y principios de Refactorización.
- Bad Smells in Code.
- Anti-patrones.

Conceptos y principios de Refactorización

El proceso de refactoring es un proceso sistemático, en el cual vamos a conseguir mejorar nuestro código sin crear nuevas funcionalidades.

Introducción

- **Stovepipe systems**, sistemas que **no pueden adaptarse** al cambio - Flexibilidad
- Herencia, encapsulamiento y polimorfismo por sí solos no son **suficientes**
- Recolectar **experiencia**:
 1. Patrones de Diseño - buenas experiencias
 2. Antipatrones - las malas experiencias

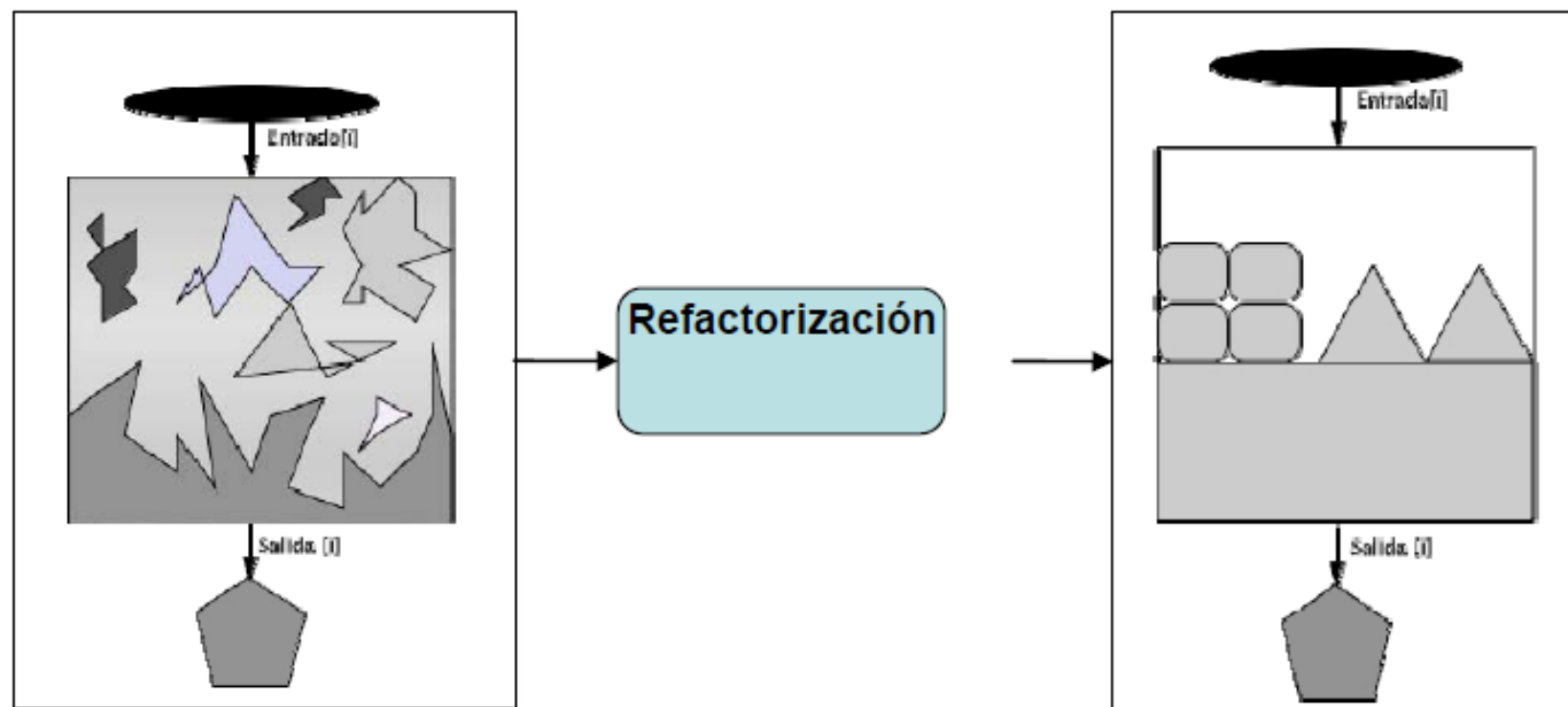


Refactoring

- **Definición**

- **Transformación controlada del código** fuente de un sistema que no altera su **comportamiento observable**
 - Hacer más comprensible y de más fácil mantenimiento el código.
 - Forma disciplinada de limpiar el código minimizando las probabilidades de introducir defectos.
- Proceso que toma **diseños defectuosos**, con código mal escrito (duplicidad, complejidad innecesaria, por ejemplo) y adaptarlo a uno **bueno, más organizado**.
 - El diseño no se da solo al inicio, sino también a lo largo del ciclo de desarrollo, durante la codificación, de manera tal que el diseño original no decaiga.

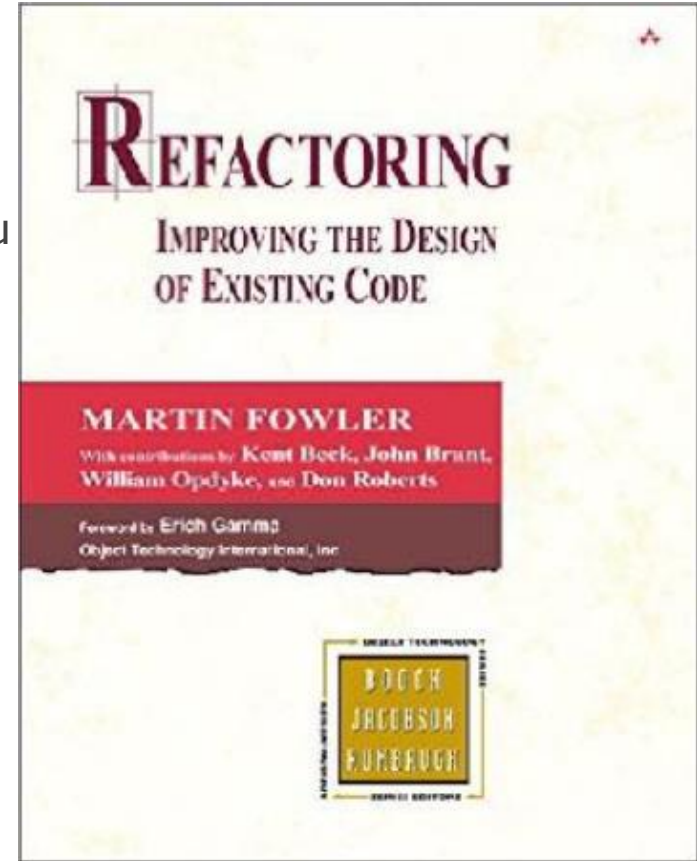
Refactoring



Refactorización: antes y después

Refactoring

- Pequeños cambios en el software que cambian su estructura interna sin modificar su comportamiento externo – **Martin Fowler**



Proceso de refactorizar

- Requisitos:
 1. Buen lote de **casos de prueba** que sean:
 - Automáticos - Ejecutarse todos a la vez
 - Auto Verificables - reporte de resultados
 - Independientes
 2. Los casos de prueba útil para verificar el **comportamiento observable** sin cambios.
- Pasos:
 1. Ejecutar las pruebas antes de cualquier cambio
 2. Analizar los cambios a realizar
 3. Aplicación del cambio
 4. Volver a ejecutar las pruebas
- **¿Refactorizar es lo mismo que optimizar?**

Beneficios de refactoring

- Continua mejora del diseño de nuestro software.
 - Evitar que el diseño original se vaya desvaneciendo
- Incremento de facilidad de lectura y comprensión del código fuente - **auto-documentable**
 - A más código, más complicado modificarlo correctamente
 - Contratar nuevos programadores
- Detección temprana de fallos
 - Mejora la **robustez del código** escrito.
- Aumenta la velocidad de programación (**Productividad**).
 - Disponer de **buenos diseños** de base
 - Lecto-comprensión

Desventajas de refactoring

- Cambio en base de datos
 - Acoplamiento a esquemas de base de datos
 - Migración de datos costoso
- Cambio en Interfaces
 - Programación Orientada a Objetos
 - Interface publicada (published interface)
 - No se dispone del código fuente modificable

```

<refactoring>(<código>, <diseño>): {
  do {
    <cuero>
  } while (<condición>);
}

```

```

<refactoring>(<código>, <diseño>): {
  do {
    <microPasos>
  } while (!<código>.<calidad>);
}

```

```

<refactoring>(<código>, <diseño>): {
  assert <red de seguridad de pruebas>
  do {
    <micro-pasos>
    assert <pasar pruebas>;
  } while (!<código>.<calidad>);
  assert <código>.<calidad>
}

```

```

<refactoring>(<código>, <diseño>): {
  assert <pre-condición>;
  do {
    <microPasos>
    assert <invariante>;
  } while (!<código>.<calidad>);
  assert <post-condición>;
}

```

```

<refactoring>(<código>, <diseño>): {
  assert <red de seguridad de pruebas>
  do {
    switch (<código>.<smellCode> | <inadecuación>) {
      case METODO_LARGO:
        <aplicar>(<microPasos>, <extraerMétodo> | <extraerClase> | ...)
        break;
      case OTRO_SMELL_CODE:
        <aplicar>(<microPasos>, <otraRefactorización> | ...)
        break;
    }
    assert <pasar pruebas>;
  } while (!<código>.<calidad>);
  assert <código>.<calidad>
}

```



```

<TDD|RUP>. <inspeccionar>(<código>) {
  do {
    try {
      <leer>(<código>, <diseño>);
    } catch (Exception <diseño>. <smellCode>|<incompatible>) {
      <desarrollo>(<diseño>, <pruebas>, <código>);
      <refactoring>(<código>, <diseño>);
    }
  } while (!<inspeccionado>(<código>, <diseño>, <condición>));
}

```

```

<TDD>. <desarrollo>(<requisito>, <historiaUsuario>) {
  do {
    <inspeccionar>(<código>, <diseño>);
    <desarrollo>(<requisito>, <pruebas>, <código>. <interfaz>);
    <desarrollo>(<pruebas>, <código>. <implementación>);
    <inspeccionar>(<código>, <diseño>);
  } while (!<funcionando>(<requisito>));
}

```

```

<RUP>. <desarrollo>(<requisito>, <casoUso>)
  do {
    <analizar|diseñar>(<requisito>, <diseño>, <nuevo>);
    <inspeccionar>(<código>, <diseño>);
    <desarrollo>(<diseño>, <pruebas>, <código>. <interfaz>);
    <desarrollo>(<pruebas>, <código>. <implementación>);
    <inspeccionar>(<código>, <diseño>);
  } while (!<funcionando>(<requisito>));
}

```

Análisis de la necesidad de refactorizar

- Bad smells - Malos olores
- Composing methods
- Moving features between elements
- Organizing data
- Simplifying conditional expressions
- Making method calls simpler
- Dealing with generalization
- Big refactorings



```
void printOwing() {  
    printBanner();  
  
    // Print details.  
    System.out.println("name: " + name);  
    System.out.println("amount: " + getOutstanding());  
}
```

```
void printOwing() {  
    printBanner();  
    printDetails(getOutstanding());  
}  
  
void printDetails(double outstanding) {  
    System.out.println("name: " + name);  
    System.out.println("amount: " + outstanding);  
}
```



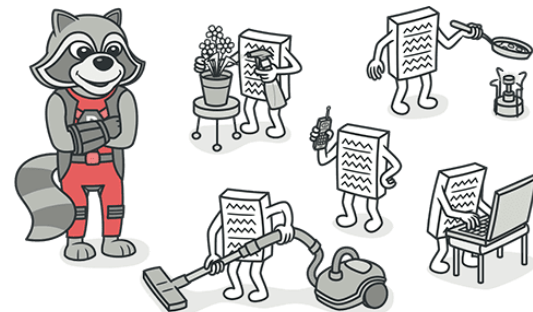
Momentos para refactorizar

- Regla de los tres strikes
 - A la tercera vez que se realice el mismo trabajo se debe refactorizar
- Al momento de agregar funcionalidad
- Al momento de resolver una falla
- Al momento de realizar una revisión de código
 - **Pair programming de eXtreme Programming**



Momentos para no refactorizar

- Código simplemente no funciona
- Esfuerzo necesario demasiado grande
- **Próximo a una entrega ¿?**



Bad smells in code

Código que huele o apesta



BAD SMELLS IN CODE

Code smells: es cualquier síntoma en el código fuente de un programa que posiblemente indica un problema más profundo.

- | | | |
|------------------------|--------------------------|---------------------------|
| 1. Duplicated code | 8. Data Clumps | 15. Message Chain |
| 2. Long Method | 9. Primitive Obsession | 16. Middle Man |
| 3. Large Classes | 10. Switch Statements | 17. Inappropriate |
| 4. Long Parameter List | 11. Parallel Inheritance | Intimacy |
| 5. Divergent Change | Hierarchies | 18. Alternative Classes |
| 6. Shotgun Surgery | 12. Lazy Class | with Different Interfaces |
| 7. Feature Envy | 13. Speculative | 19. Incomplete Library |
| | Generality | Class |
| | 14. Temporary Field | 20. Data Class |
| | | 21. Refused Bequest |
| | | 22. Comments |

Revisar: <https://github.com/HugoMatilla/Refactoring-Summary>



1. Código Duplicado

Según Fowler, “*number one in the stink parade*”.

Ejemplos:

- La misma expresión en dos métodos de la misma clase.

Solución: Extract Method

- La misma expresión en dos clases hermanas.

Solución: Extract Method, Pull Up Method, o Substitute Algorithm

- Código duplicado en clases diferentes

Solución: Extract Class

Bad smells



2. Método largo/extenso

Lo ideal es producir métodos cortos y específicos. Las razones son obvias: *fácil de mantener/comprender/reemplazar/reusar*

La solución es *descomponer* el método en submétodos.

Algunas refactorizaciones útiles:

Extract Method, Introduce Parameter Object, Replace Method with Method Object, Decompose Conditional

3. Clase larga/extensa

También identificado a veces como el antipatrón *Blob*

La solución es disminuir la complejidad y las responsabilidades

Algunas refactorizaciones útiles:

Extract Class, Extract Subclass



4. Lista de parámetros larga/extensa

En los inicios de la programación, todo lo necesario para una rutina era pasado como parámetro (cada uno!)

En orientación a objetos esto no es completamente cierto, ya que la rutina puede comunicarse con **un objeto que le provea lo necesario**.

Las listas de parámetros extensas pueden simplificarse con algunas refactorizaciones clásicas:

*Replace Parameter with Method,
Preserve Whole Object,
Introduce Parameter Object*



5. Cambio divergente

El cambio divergente ocurre cuando una clase es cambiada de forma diferente por diferentes razones. *“Debo cambiar estos tres métodos cada vez que usamos una nueva base de datos; debo cambiar estos cuatro métodos cada vez que tenemos una nueva herramienta financiera”*

Solución posible: *Extract Class.*

6. Shotgun Surgery

Similar al anterior, pero ocurre cuando al hacer un cambio, se requieren muchos pequeños cambios en clases diferentes.

Solución posible: *Move Method, Move Field.*

Bad smells



7. Data clumps

Son agrupaciones de datos que son usadas en conjunto en muchos lugares.

Deberían probablemente ser parte de un objeto.

Soluciones:

Extract Class, Introduce Parameter Object, Preserve Whole Object

7. Grupos de datos

8. Primitive Obsession

Muchos datos contruidos en base a tipos primitivos, en lugar de encapsularlos en pequeños objetos. Típicamente: arreglos, strings, enteros.

Soluciones:

Replace Data Value With Object, Replace Type Code With Class, Replace Type Code with State/Strategy

Los mismos de Data Clumps.

Bad smells



9. Lazy Class

Una clase que no se justifica, redundante, o de poca utilidad.

Entorpece la comprensión del sistema.

A veces es producto de refactorizaciones anteriores

Soluciones: *Collapse Hierarchy*

10. Inappropriate Intimacy

Clases demasiado íntimas, que constantemente indagan en aspectos privados de ambas.

Soluciones :

Move Method, Move Field, Change Bidirectional Association to Unidirectional, Hide Delegate



Refactorizaciones Comunes



Composing methods - Extract Method

Es uno de las principales refactorizaciones que apuntan a empaquetar código apropiadamente. Básicamente, es un **fragmento de código** que puede ser agrupado y apartado.

```
void printOwing() {  
    printBanner();  
  
    //print details  
    System.out.println ("name:  " + _name);  
    System.out.println ("amount  " + getOutstanding());  
}
```



```
void printOwing() {  
    printBanner();  
    printDetails(getOutstanding());  
}  
  
void printDetails (double outstanding) {  
    System.out.println ("name:  " + _name);  
    System.out.println ("amount  " + outstanding);  
}
```

Pasos:

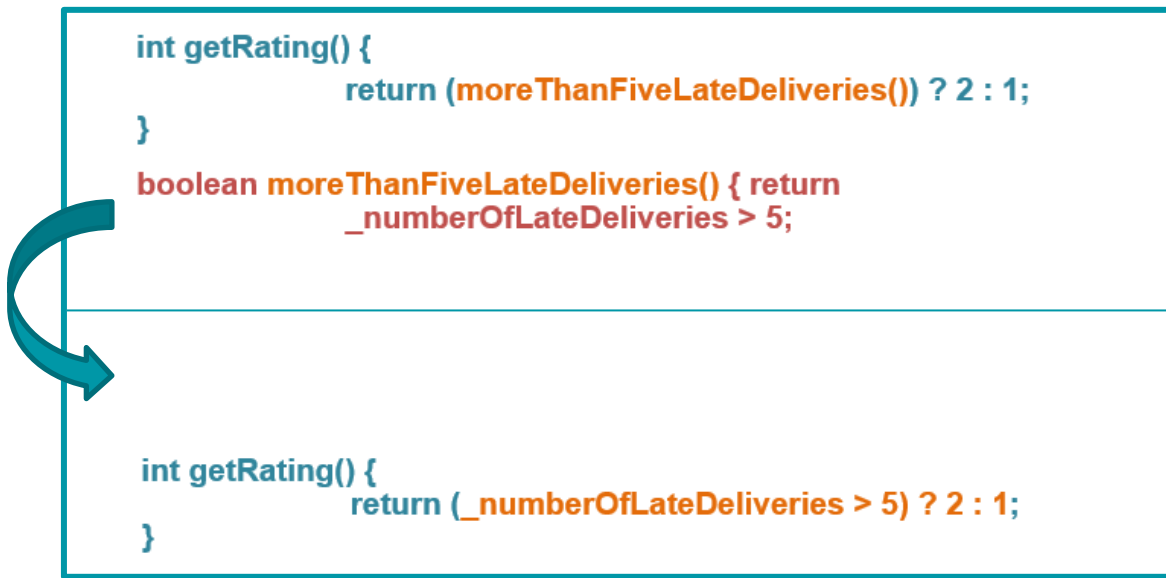
1. Crear un **nuevo método**, nombrarlo apropiadamente según lo que hace.
2. **Copiar** el código extraído del origen a este nuevo método
3. Escanear el código extraído por referencias a variables locales al origen.
4. Escanear el código extraído por variables temporales.
5. Evaluar el uso de las variables (son modificadas? son solo de lectura?)
6. Pasar al nuevo método las variables necesarias como parámetros.
7. Compilar este nuevo método.
8. Reemplazar el código extraído por una llamada al nuevo método.



Composing methods - Inline Method

Ocurre cuando el cuerpo de un método es tan claro como su nombre.

La solución es eliminar el método trasladando el cuerpo a sus invocadores






Composing methods - Replace Temp with Query

Se usa una variable temporal para guardar el resultado de una expresión.

El problema es que son temporales y locales y (a veces) tienden a generar métodos extensos.

La solución es convertir la expresión en un método y reemplazar su uso por la invocación correspondiente

```
double basePrice = _quantity * _itemPrice;  
if (basePrice > 1000)  
    return basePrice * 0.95;  
else  
    return basePrice * 0.98;
```



```
if (basePrice() > 1000)  
    return basePrice() * 0.95;  
else  
    return basePrice() * 0.98;
```

```
...  
double basePrice() {  
    return _quantity * _itemPrice;  
}
```

Composing methods - Introduce explaining variable



Existe una expresión muy complicada.

La solución es usar variables temporales con nombres significativos para simplificar la expresión



```
if ( (platform.toUpperCase().indexOf("MAC") > -1) &&  
      (browser.toUpperCase().indexOf("IE") > -1) &&  
      wasInitialized() && resize > 0 )  
{  
    ...  
}
```

```
final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;  
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;  
final boolean wasResized = resize > 0;  
  
if (isMacOs && isIEBrowser && wasInitialized() && wasResized)  
{  
    ...  
}
```

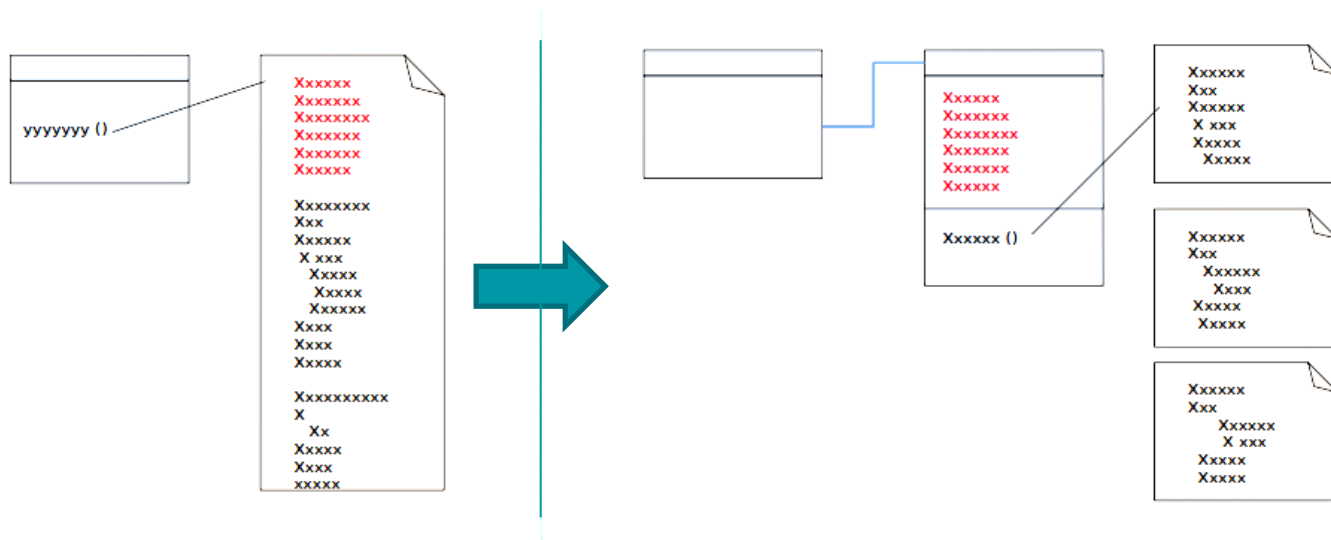
Composing methods - Replace Method with Method Object



Existe un método extenso que usa variables locales de forma tal que no se puede aplicar Extract Method.

La solución es convertir el método en un objeto de manera tal que las variables locales son ahora campos del objeto.

Luego se puede descomponer el método en otros métodos dentro de la misma clase



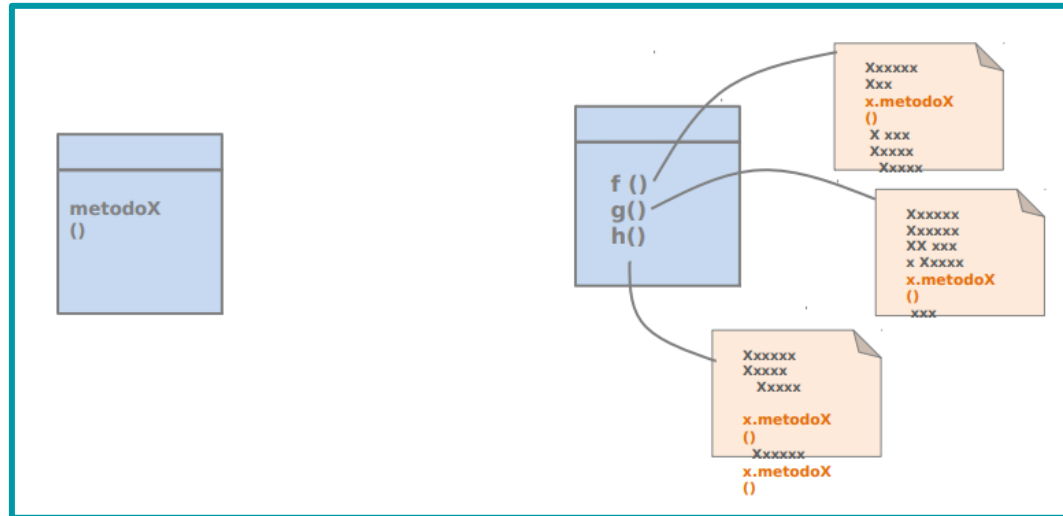


Moving Features – Moving Method

Existe un método que es o será usado por más aspectos de otra clase que de la clase misma en la que es definido.

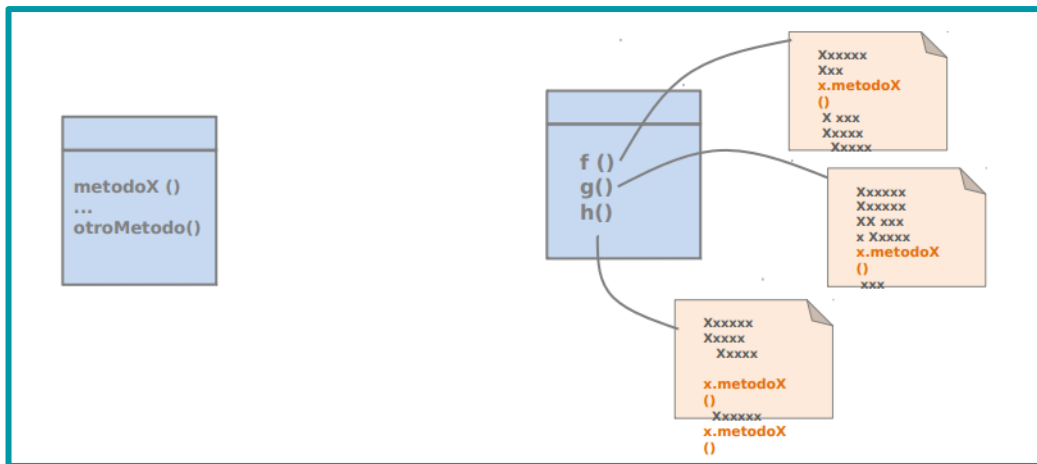
Puede ocurrir cuando una clase tiene muchas responsabilidades, o cuando las clases colaboran demasiado y existe mucho acoplamiento.

La solución es crear un nuevo método con un cuerpo similar en la clase que mas lo use.





Moving Features – Moving Method



Mecánica del proceso

1. Examinar **todo lo que debe moverse** a otra clase.
2. Controlar las **dependencias** con las subclases y las superclases
3. Declarar el nuevo método en la clase destino.
4. Copiar el código del origen al destino y hacer los ajustes necesarios.
5. Compilar la clase destino.
6. Determinar el vínculo de la clase origen a la clase destino.
7. Convertir el código origen en un delegando
8. Compilar y testear

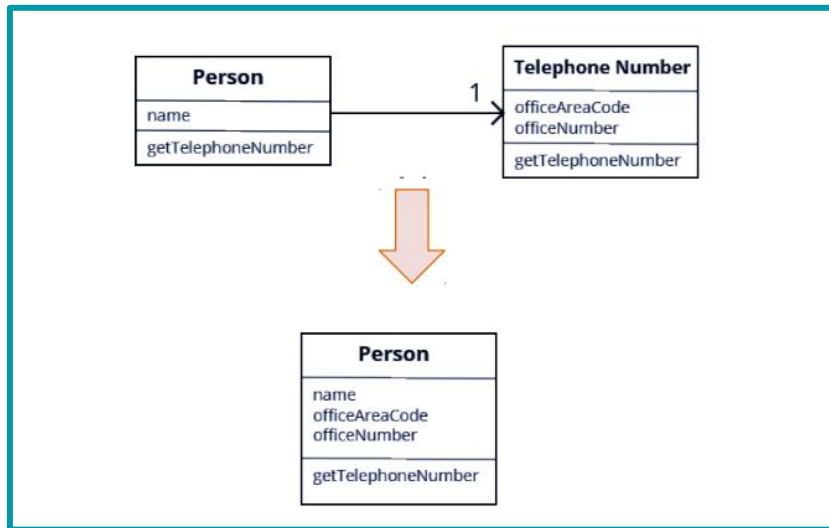


Moving Features – Extract Class

Hay una clase que está haciendo una tarea que deberían hacerla dos.

Demasiados métodos, muchos datos, exceso de responsabilidades.

La solución es crear una nueva clase y trasladar los campos y métodos relevantes de la clase vieja a la clase nueva



En esta refactorización **una clase absorbe** a otra pues esta última no tiene muchas responsabilidades asociadas.

Mecánica del proceso

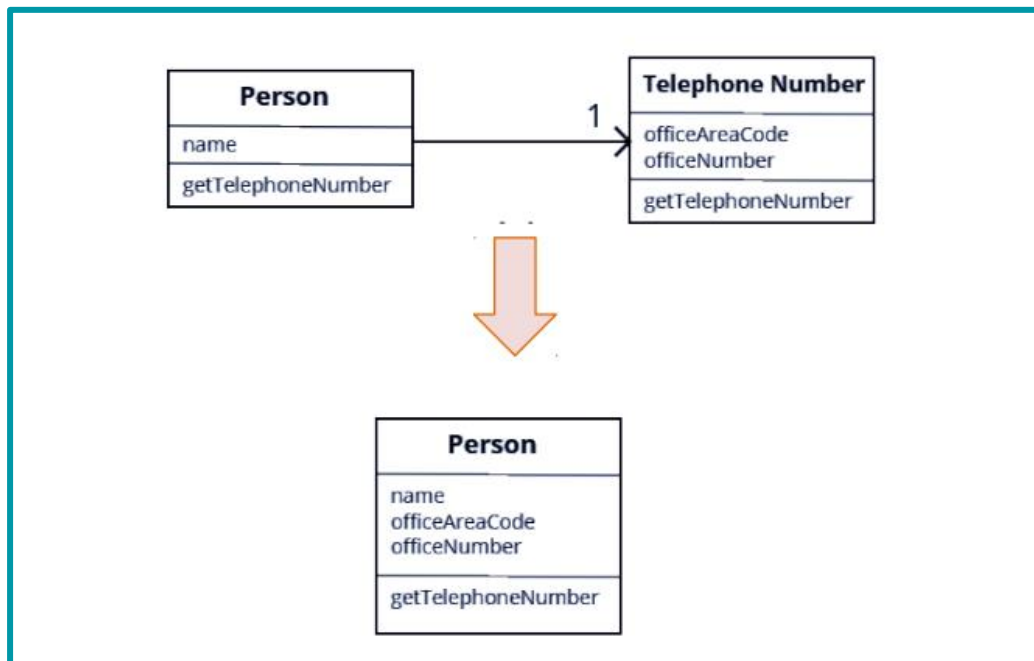
1. Decidir cómo dividir las responsabilidades de la clase
2. Crear una nueva clase que exprese esa división (tal vez deban renombrarse ambas)
3. Hacer un link desde la clase vieja a la nueva (tal vez una asociación bidireccional)
4. Usar **Move Field** y **Move Method**, y compilar luego de aplicar esas refactorizaciones.
5. Revisar las interfaces y reducirlas en lo necesario.
6. Examinar la visibilidad de la nueva clase en la clase vieja

* La inversa de Extract Class es **Inline Class**.



Moving Features – Inline Class.

- La inversa de **Extract Class** es **Inline Class**.
- En esta refactorización **una clase absorbe a otra** pues esta última no tiene muchas responsabilidades asociadas.



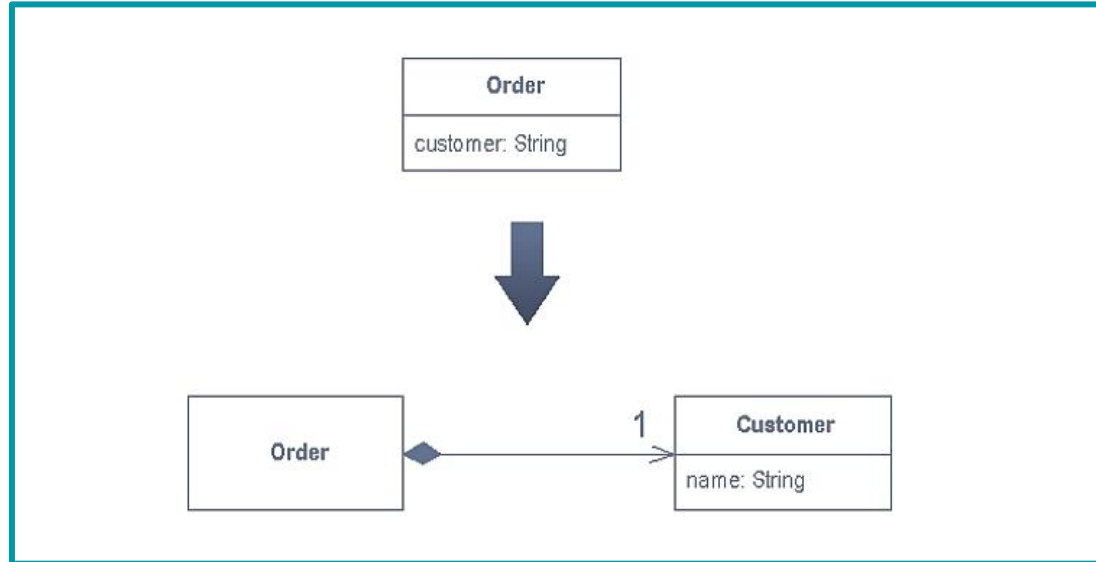
Organizing data – Replace data value with object



Existe un item de datos que necesita datos o comportamiento adicional

Es básicamente un conjunto de datos que en su momento no fue considerado como objeto del sistema.

La solución es convertir ese item de datos en un objeto



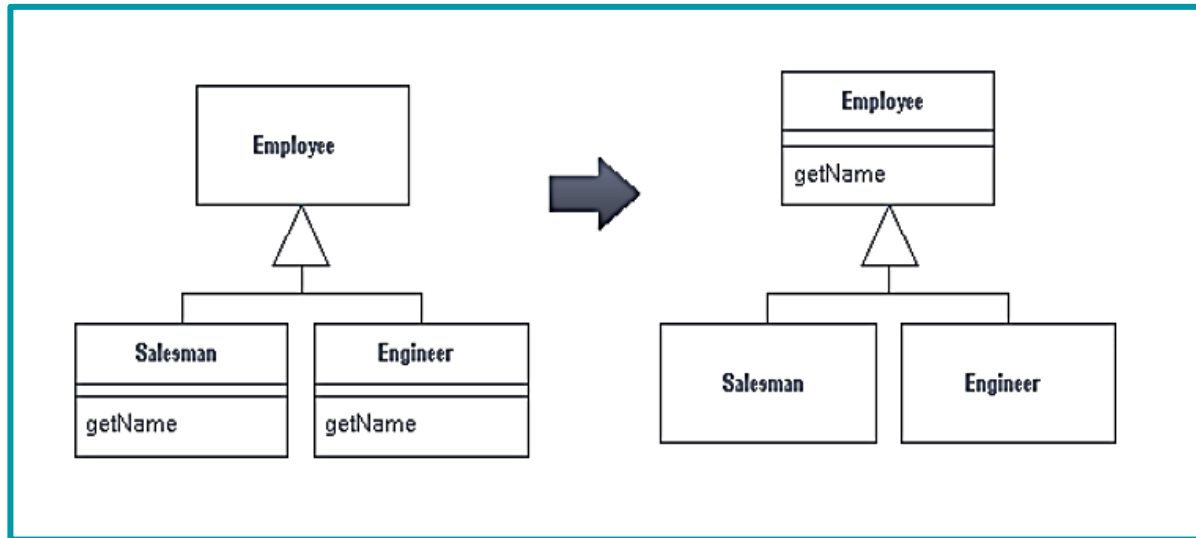


Generalización – Pull Up Method

Existen métodos con idénticos resultados en varias subclases.

Usualmente son consecuencia de copy+paste. También surge al refinar abstracciones incompletas.

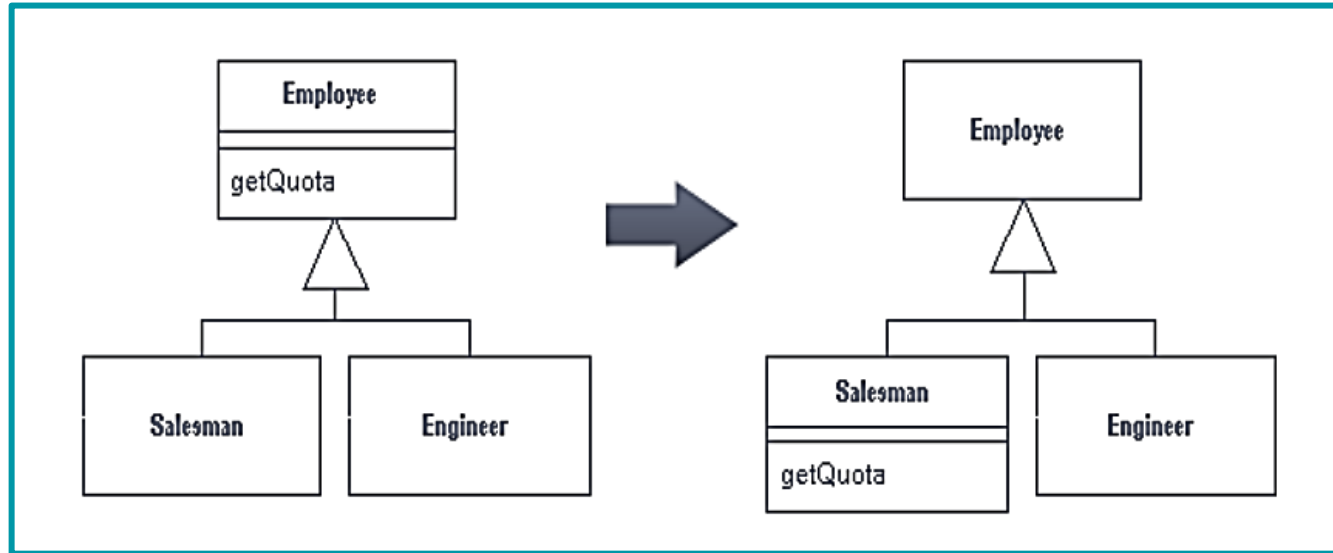
La solución es trasladar los métodos a las superclases.





Generalización – Pull down Method

Parte del comportamiento de una clase es relevante sólo a algunas subclases.
La solución es trasladar los métodos a las subclases.



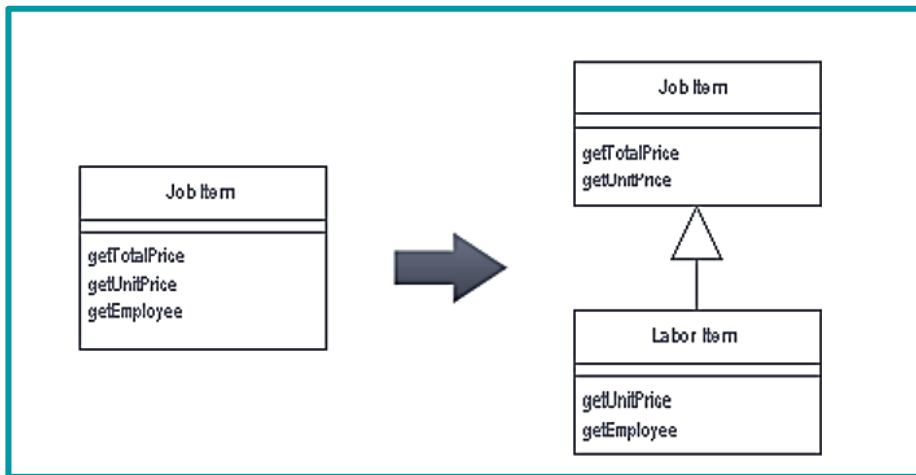


Generalización – Extract Subclass

Una clase tiene aspectos que son usados sólo en algunas instancias.

Puede ocurrir como una sobrecarga de responsabilidades, o una dependencia sobre los atributos. Algunas instancias los usan, otras no.

La solución es crear subclases para los subconjuntos de aspectos relevantes a ciertas instancias.



Mecánica del proceso

1. Definir una nueva subclase de la clase origen.
2. Proveer constructores para la nueva subclase. **Utilizar super con los argumentos adecuados**
3. Buscar todas las invocaciones a constructores de la superclase y reemplazarlo por el constructor de la subclase si es necesario.
4. **Si la superclase ya no puede ser instanciada, declararla abstracta.**
4. Aplicar **Push Down Method y Push Down Field** hasta que ya no sea necesario.
5. Eliminar campos que distinguían entre las instancias (usualmente booleans)
6. Compilar y testear luego de cada push down.

Antipatrones

Según (Brown et al, 1998) “Un antipatrón es una forma literaria que describe una solución recurrente que genera consecuencias negativas”

Antipatrones (Anti-patterns)

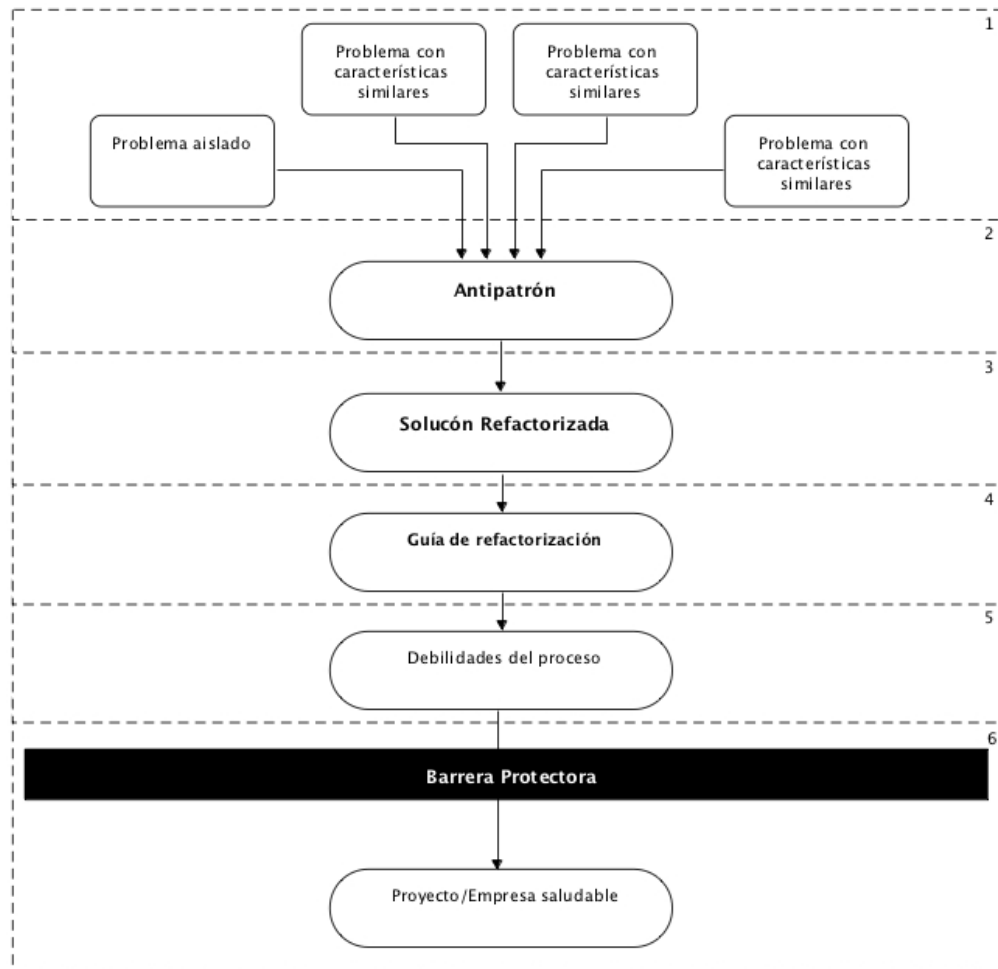
- Decisión equivocada cuando se resuelve un determinado problema
- Aplicación correcta de un **patrón de diseño** en el contexto equivocado.
 - soluciones con efectos negativos, contrario a los **patrones de diseño**
- Usar **Antipatrones** permite evitar cometer errores recurrentes
- **Refactorización** para asegurar reorganizaciones ordenadas de código fuente para mejorar la mantenibilidad, o salir de algún Antipatrón

Relación de antipatronos con patrones de diseño y refactorizaciones

- Ambos proveen un vocabulario común
- Ambos documentan conocimiento
- Los patrones de diseño documentan **soluciones exitosas**, los antipatronos documentan **soluciones problemáticas**
- Los antipatronos son el lado oscuro de los patrones de diseño
 - No se evalúa qué tan aplicables

Proceso para el uso de antipatrones

1. Encontrar el problema
2. Establecer un patrón de fallas
3. Refactorizar el código
4. Publicar la solución
5. Identificar debilidades, o posibles problemas del proceso.
6. Corregir el proceso



Antipatrones - descripción

NO existe un estándar para la descripción de antipatrones, pero los siguientes son algunos items recurrentes:

- **AntiPattern Name** – nombre descriptivo del antipatrón.
- **Also Known As** – otros nombres por los cuales es conocido.
- **Most Frequent Scale** – área en la cual el antipatrón es aplicable. (micro-architecture, framework, application, system, enterprise, global/industry)
- **Refactored Solution Name** – nombre significativo describiendo la solución al antipatrón.
- **Refactored Solution Type** – la categoría en la cual la solución refactorizada cae (software, technology, process, or role).
- **Root Causes** – raíz (causa) del problema.
- **Unbalanced Forces** – fuerzas (aspectos) desbalanceados en la causa del antipatrón. (functionality, management of performance, resources, complexity, change, IT resources, or technology transfer).
- **Anecdotal Evidence** – evidencia anecdótica de casos y consecuencias del antipatrón en acción
- **Background** – ejemplos del antipatrón.
- **General Form** – diagrama o detalles adicionales acerca del antipatrón.
- **Symptoms and Consequences** – síntomas y consecuencias generales.
- **Typical Causes** – causas específicas que conjuntamente con la raíz del problema llevan a la creación del antipatrón.
- **Known Exceptions** – instancias en las cuales el antipatrón es aceptable.
- **Refactored Solutions** – explica cómo una solución refactorizada resuelve el desbalance de las fuerzas antes mencionadas.
- **Variations** – variaciones comunes del antipatrón
- **Example** – ejemplo descriptivo de cómo la solución refactorizada puede aplicarse.
- **Related Solutions** – referencias cruzadas a otros antipatrones
- **Applicability to Other Viewpoints and Scales** – explica cómo el antipatron corresponde con otros puntos de vista y/o escalas.



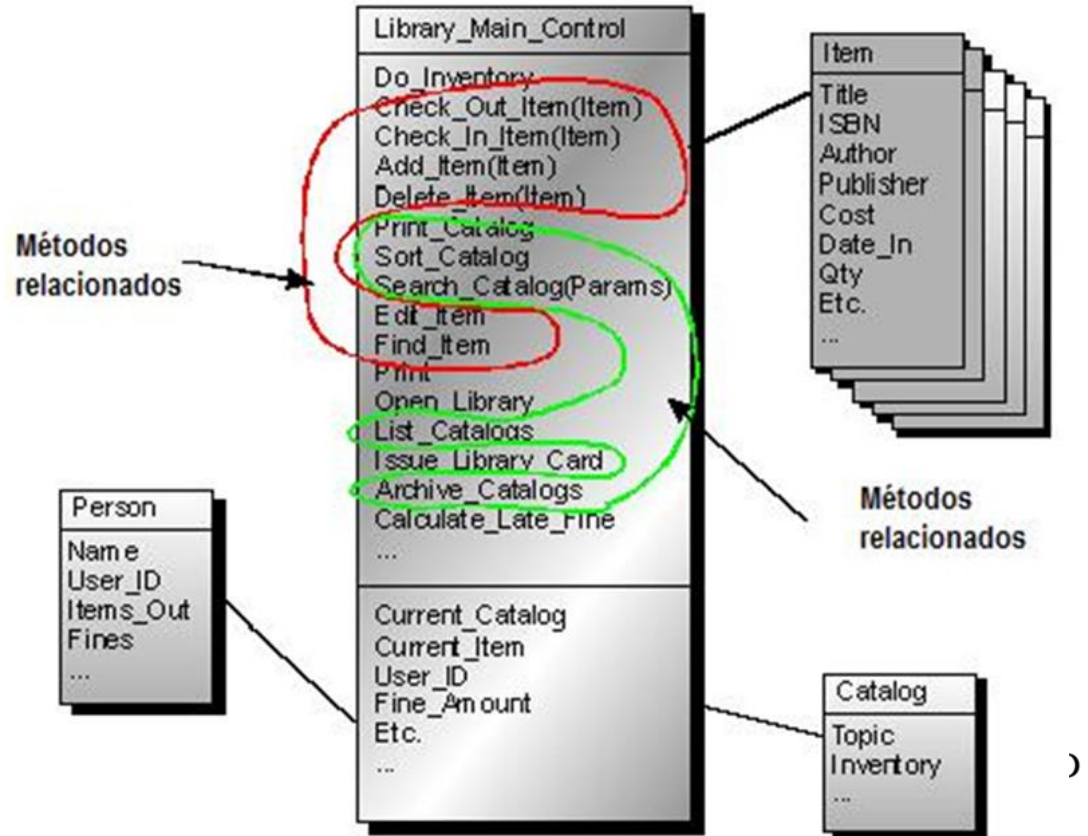
Antipatrones - Tipos

Los antipatrones pueden agruparse en

- **Antipatrones del desarrollo de software** :Describen situaciones usualmente candidatas a la refactorización.
- **Antipatrones de la arquitectura del software**: Se enfocan en la estructura general del sistema y sus componentes.
- **Antipatrones de Administración del Proyecto**: Se enfocan en aspectos relacionados con la administración del proyecto, de las personas involucradas y de la comunicación humana.

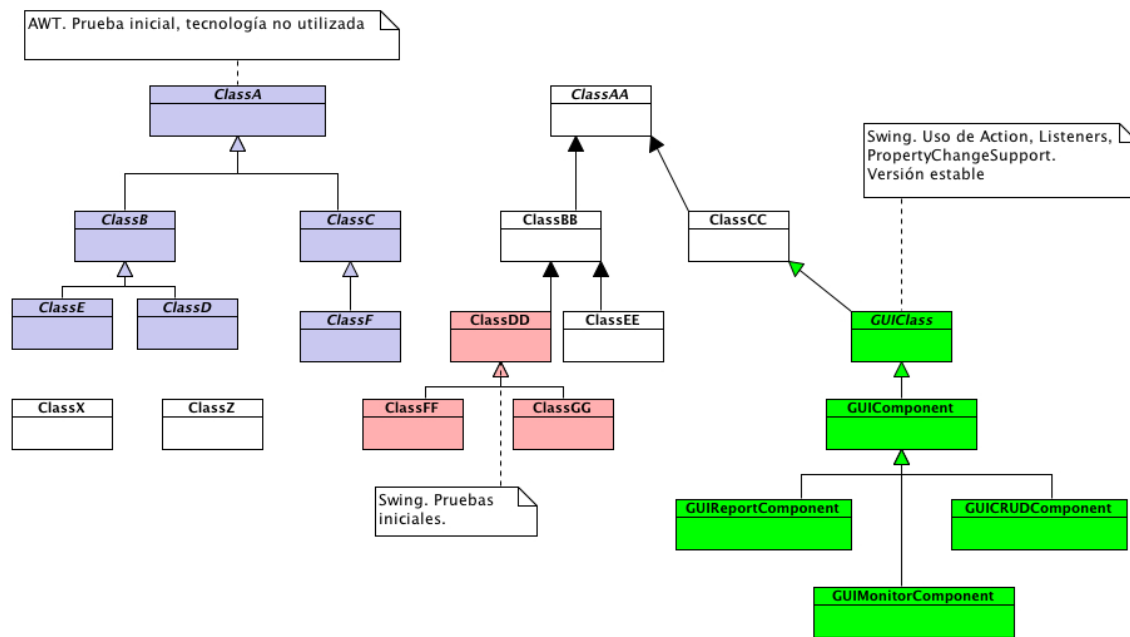
Catálogo de antipatrones de desarrollo

- 1. The Blob:** God Class o Winnebago es una **clase**, o componente, que **conoce o hace demasiado**



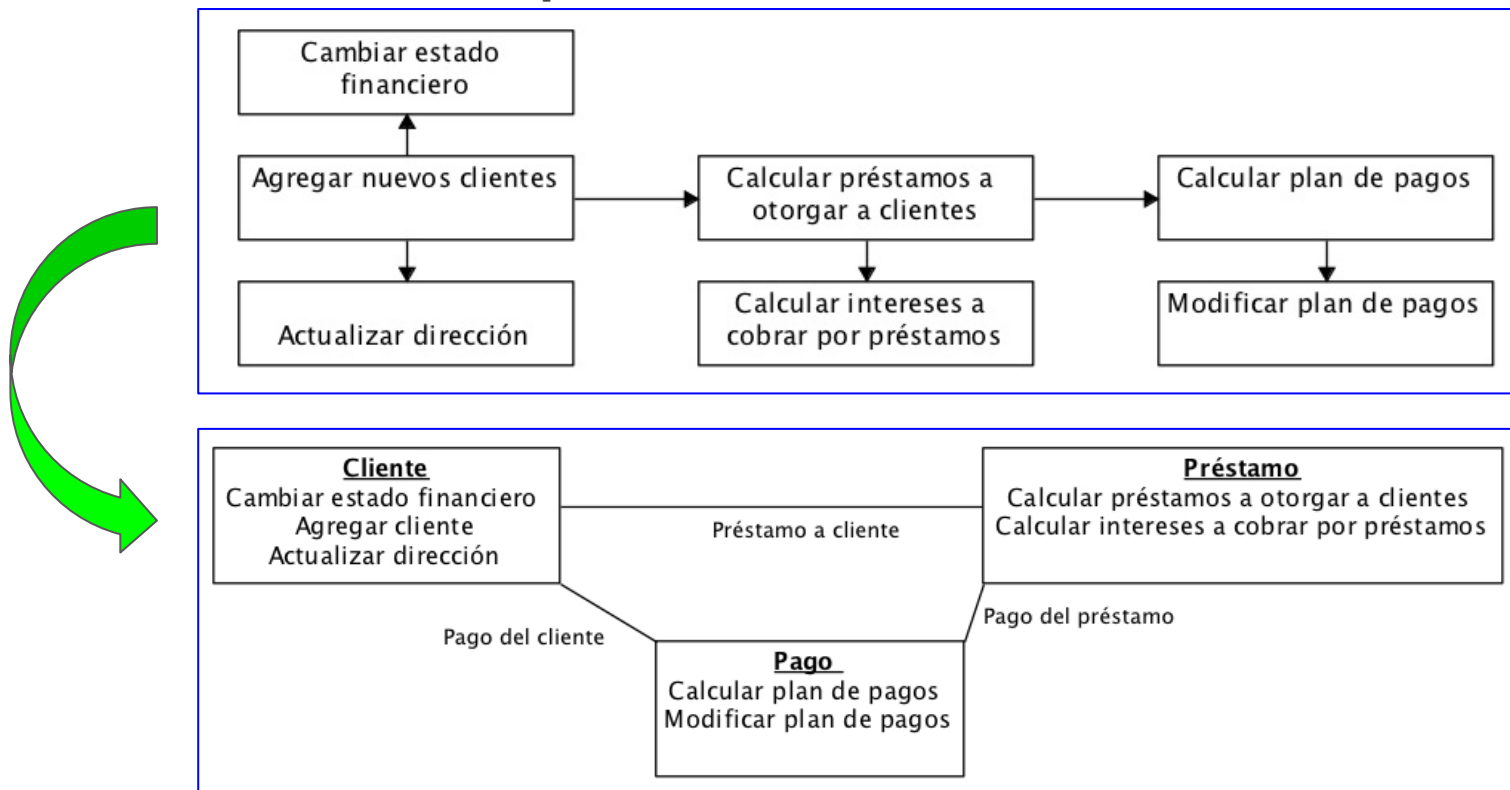
Catálogo de antipatrones de desarrollo

- 1. Lava Flow:** Dead Code aparece principalmente en aquellos sistemas que comenzaron como investigación o pruebas de concepto y luego llegaron a producción.



Catálogo de antipatronos de desarrollo

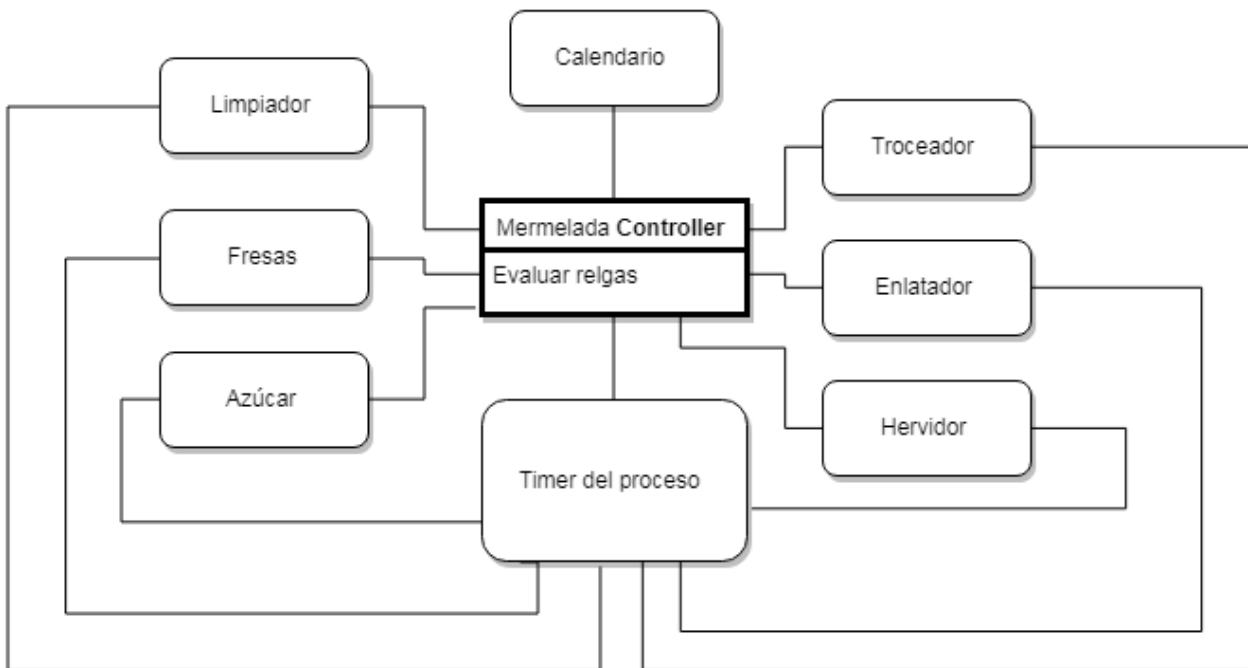
1. Functional Decomposition: Traducen cada subrutina como una clase.



Catálogo de antipatrones de desarrollo

1. **Poltergeists**: Gipsy, o Proliferation of Classes, o Big DoIt Controller Class. Las clases fantasmas tienen pocas responsabilidades y un ciclo de vida breve. “**Aparecen**” solamente para iniciar algún método.
2. Son de relativa facilidad de encuentro ya que sus nombres suelen llevar el sufijo “**controller**” o “**manager**”.

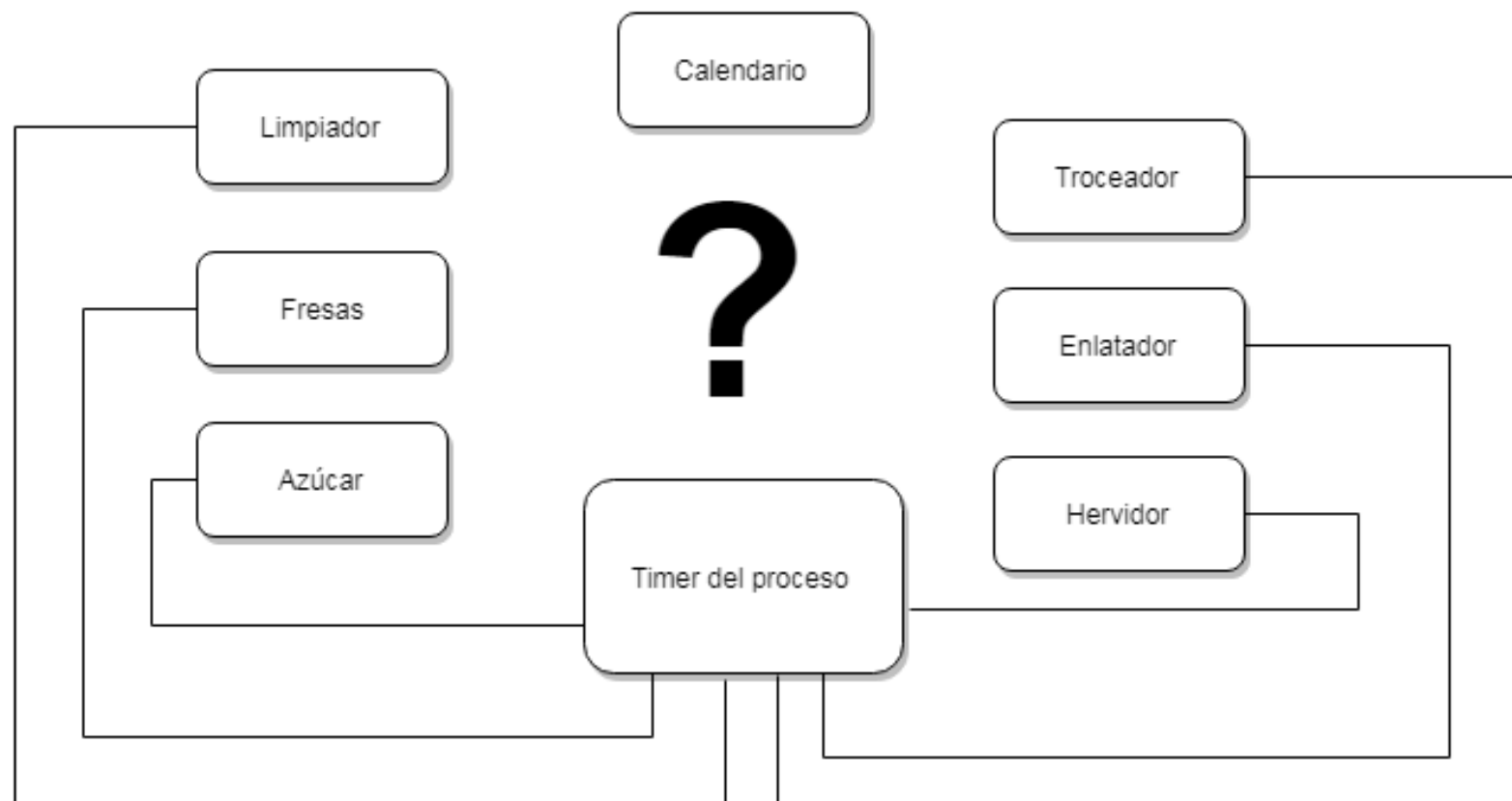
Catálogo de antipatronos de desarrollo



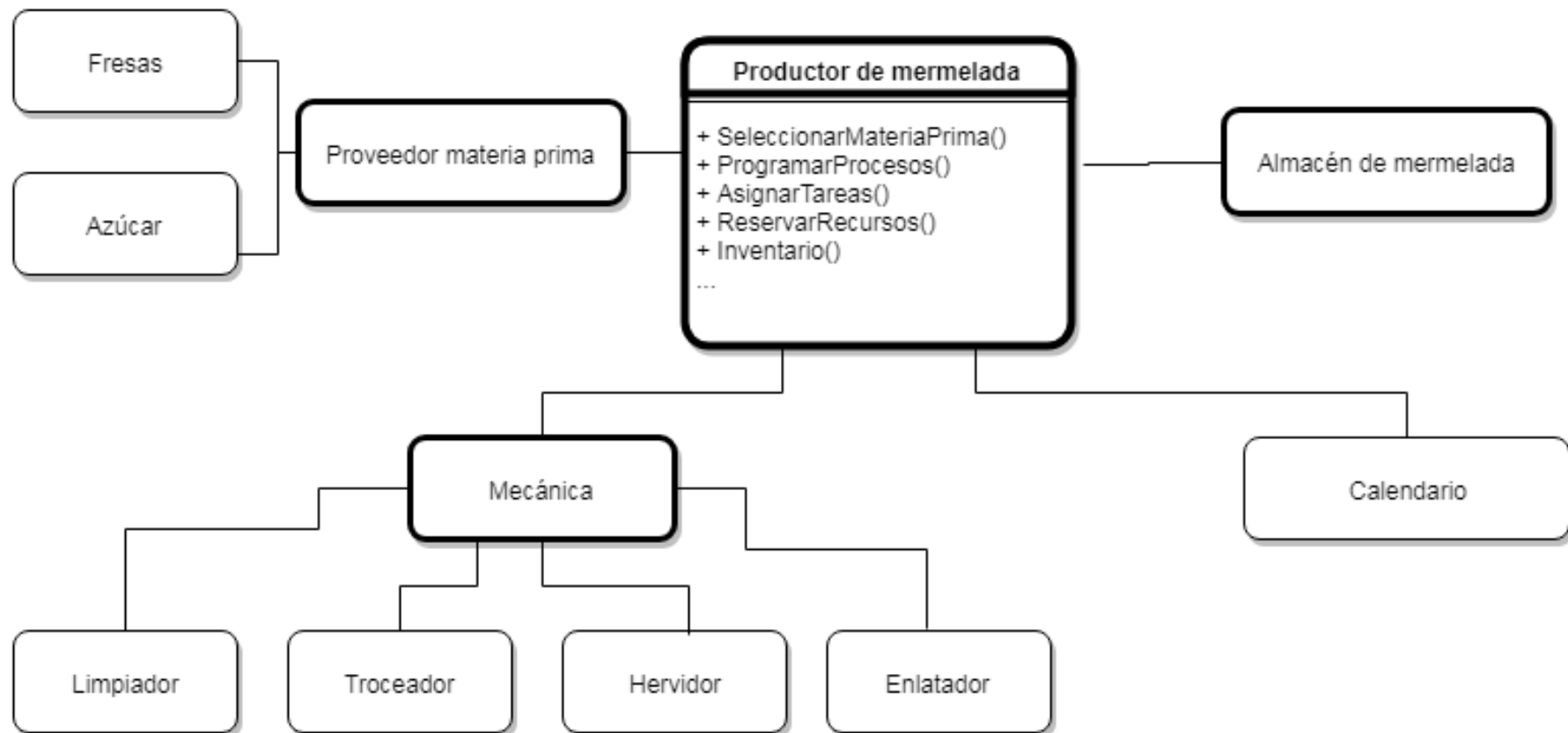
MermeladaController

- No tiene estado
- Sus instancias son temporales y aparecerán solamente para invocar a otras clases
- Todas sus asociaciones son transitorias
- Añade rutas de acceso a otras clases completamente

Catálogo de antipatrones de desarrollo



Catálogo de antipatrones de desarrollo

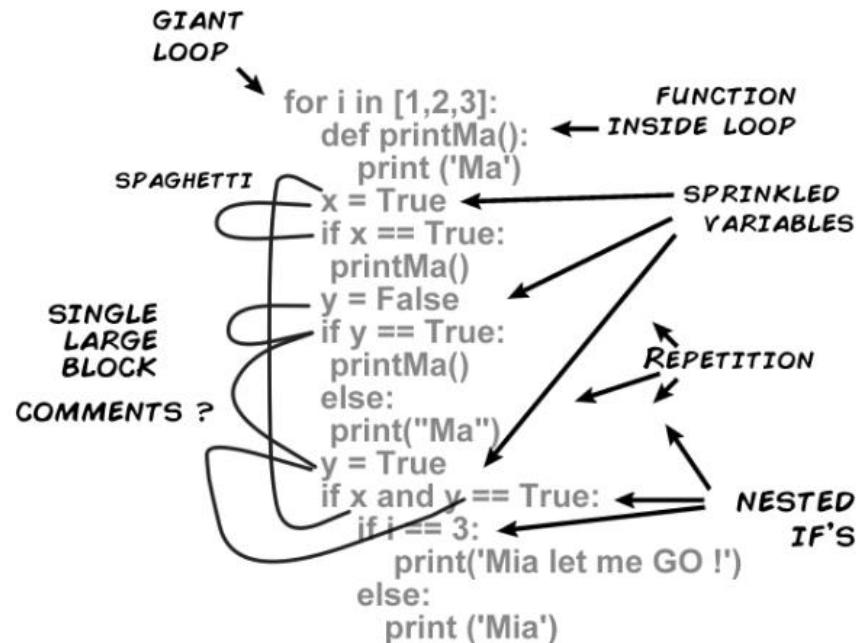


Catálogo de antipatrones de desarrollo

- 1. Golden Hammer:** Old Yeller, o Head in the Sand. Un **martillo de oro** es cualquier herramienta, tecnología o paradigma que, según sus partidarios, es capaz de **resolver** diversos tipos de problemas, incluso aquellos para los cuales **no fue concebido**
 - El lenguaje XML

Catálogo de antipatrones de desarrollo

- 1. Spaghetti Code:** Sistema con poca estructura donde los cambios y **futuras extensiones** se tornan **difíciles** por haber perdido **claridad en el código**, incluso para el autor del mismo.



https://miro.medium.com/max/1224/1*7Dt8oqdanszwwG_QNTN4Yg.pn

Catálogo de antipatrones de desarrollo

1. Copy-And-Paste Programming: más fácil modificar código preexistente que programar desde el comienzo.

```
abstract class Game {  
    /* Hook methods. Concrete implementation may differ in each subclass*/  
    protected int playersCount;  
    abstract void initializeGame();  
    abstract void makePlay(int player);  
    abstract boolean endOfGame();  
    abstract void printWinner();  
  
    /* A template method : */  
    public final void playOneGame(int playersCount) {  
        this.playersCount = playersCount;  
        initializeGame();  
        int j = 0;  
        while (!endOfGame()) {  
            makePlay(j);  
            j = (j + 1) % playersCount;  
        }  
        printWinner();  
    }  
}
```

```
/*Now we can extend this class in order  
//to implement actual games:  
  
class Monopoly extends Game {  
  
    /* Implementation of necessary concrete methods */  
    void initializeGame() {  
        // Initialize players  
        // Initialize money  
    }  
    void makePlay(int player) {  
        // Process one turn of player  
    }  
    boolean endOfGame() {  
        // Return true if game is over  
        // according to Monopoly rules  
    }  
    void printWinner() {  
        // Display who won  
    }  
    /* Specific declarations for the Monopoly game. */  
  
    // ...  
}
```

Referencias

1. Fowler, M. (2018). Refactoring: improving the design of existing code. Addison-Wesley Professional.
2. Campo, G. D. (2009). Patrones de Diseño, Refactorización y Antipatrones. Ventajas y Desventajas de su Utilización en el Software Orientado a Objetos. Cuadernos de Ingeniería, (4), 101-136.
3. Summary of "Refactoring: Improving the Design of Existing Code" by Martin Fowler: <https://github.com/HugoMatilla/Refactoring-Summary>