UNIVERSIDAD DE LIMA

60 años

REFACTORIZACIÓN

UNIDAD 3: PRÁCTICAS DE DISEÑO ÁGIL

Temario

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

UNIVERSIDAD DE LIMA | 60 años

Conceptos y principios de Refactorización

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

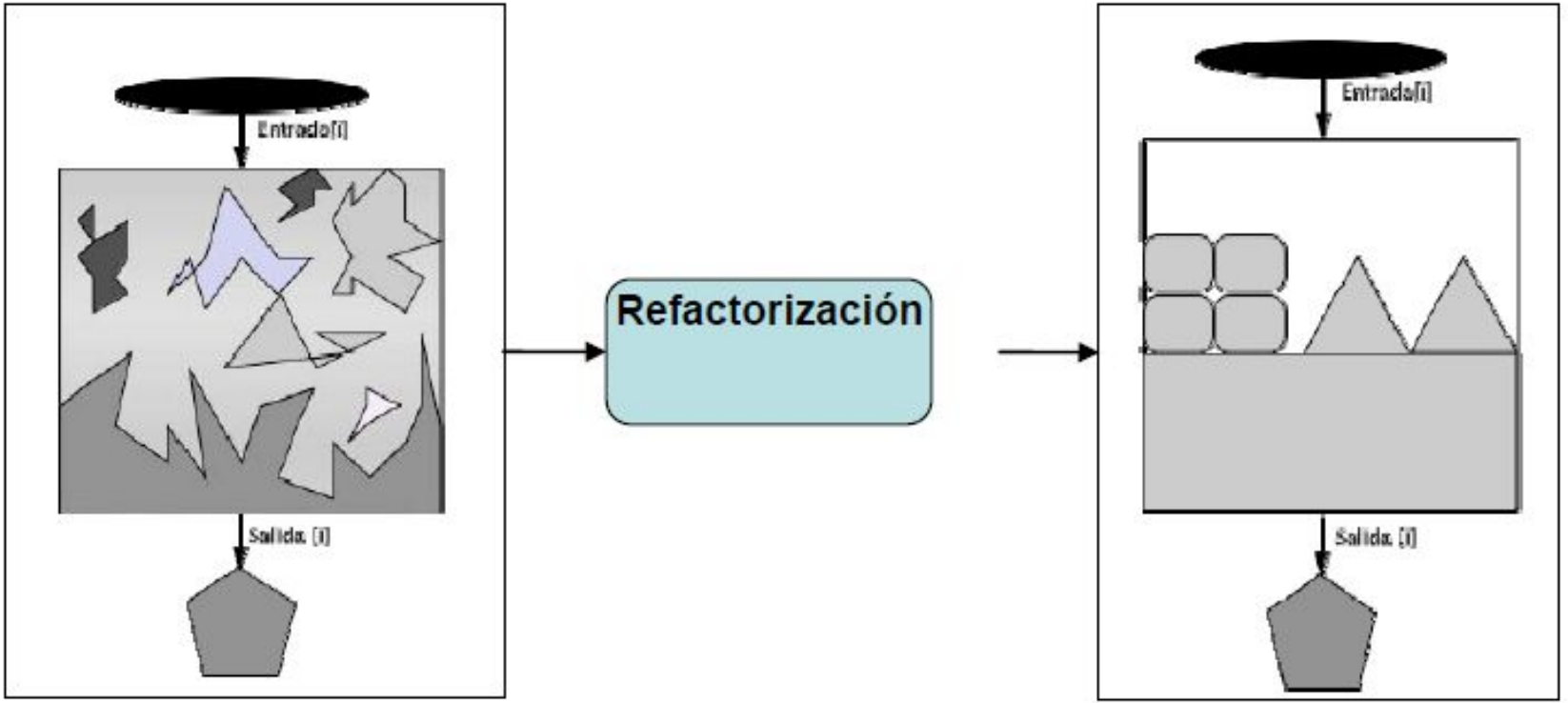
Introducción

- **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

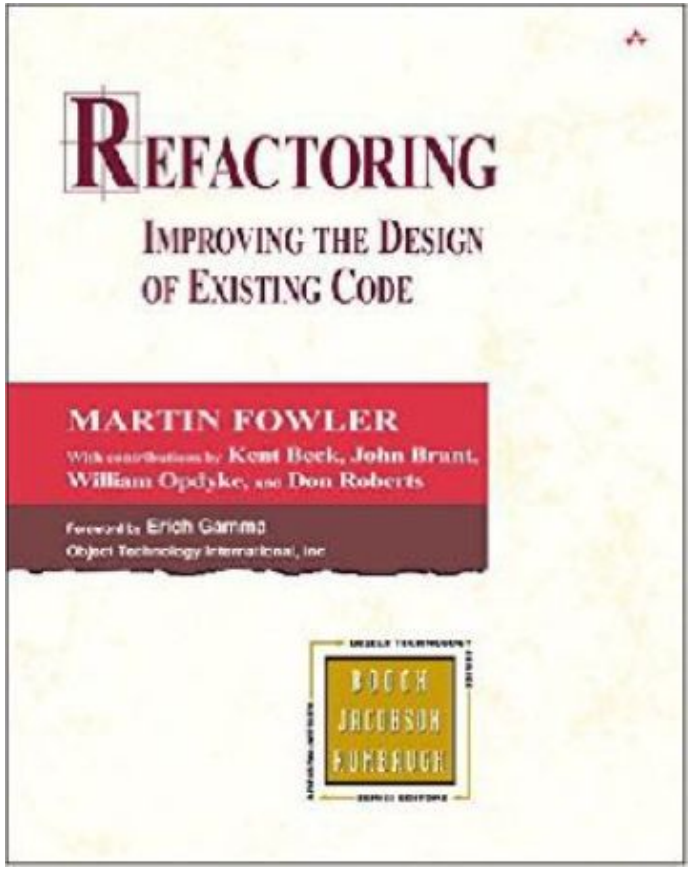


Refactorización: antes y después

RSIDAD
A

Refactoring

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



UNIVERSIDAD DE LIMA

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

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

UNIVERSIDAD DE LIMA

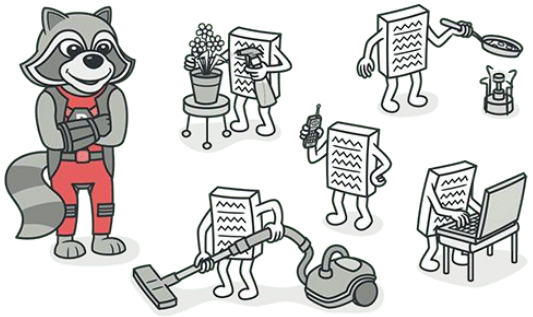
Momentos para refactorizar

- 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**

UNIVERSIDAD DE LIMA

Momentos para no refactorizar

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



UNIVERSIDAD DE LIMA

UNIVERSIDAD DE LIMA | 60 años

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

Revisar:

https://github.com/HugoMatilla/Refactoring-Summary

UNIVERSIDAD DE LIMA | 60 años

Antipatrones

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

Antipatrones (Anti-patterns)

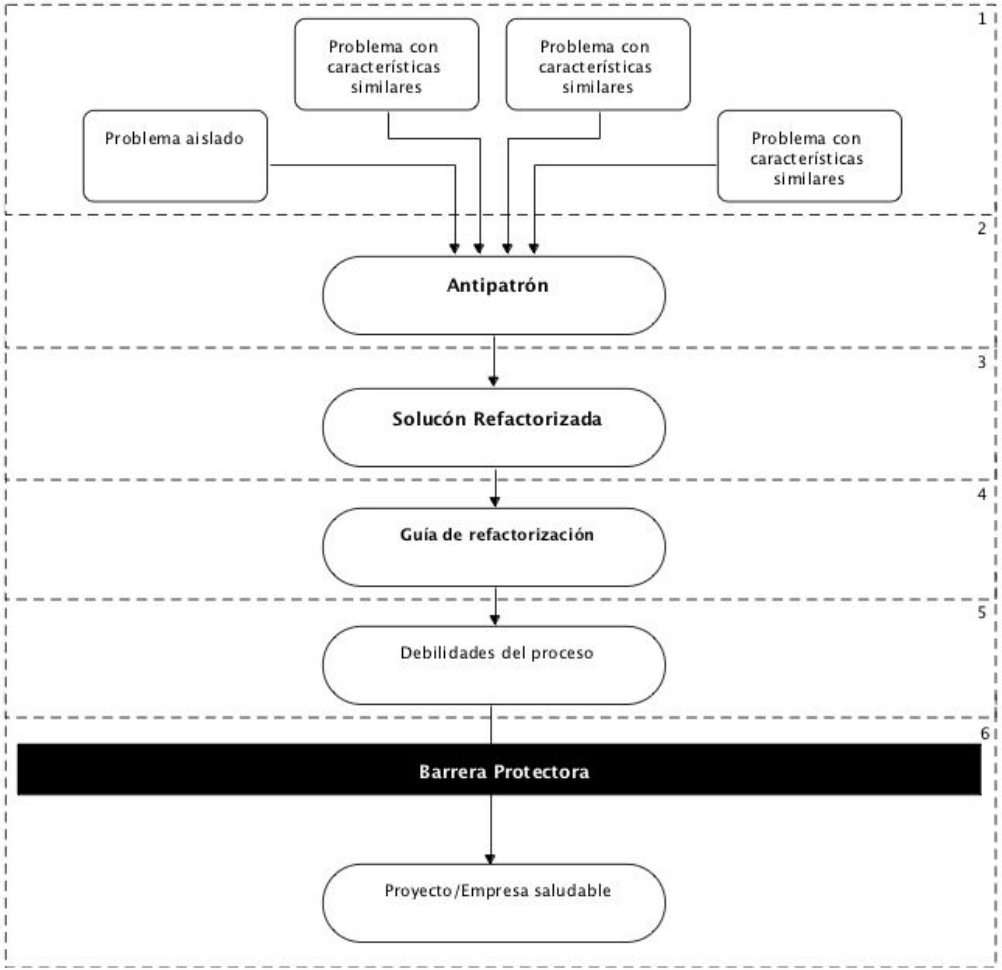
- Decisión equivocada cuando se resuelve un determinado problema
- Aplicación correcta de un **patrón de diseño** en el contexto equivocado.
  - soluciones con efectos negativos, contrario a los **patrones de diseño**
- Usar **Antipatrones** permite evitar cometer errores recurrentes
- **Refactorización**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

Relación de antipatrones con patrones de diseño y refactorizaciones

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

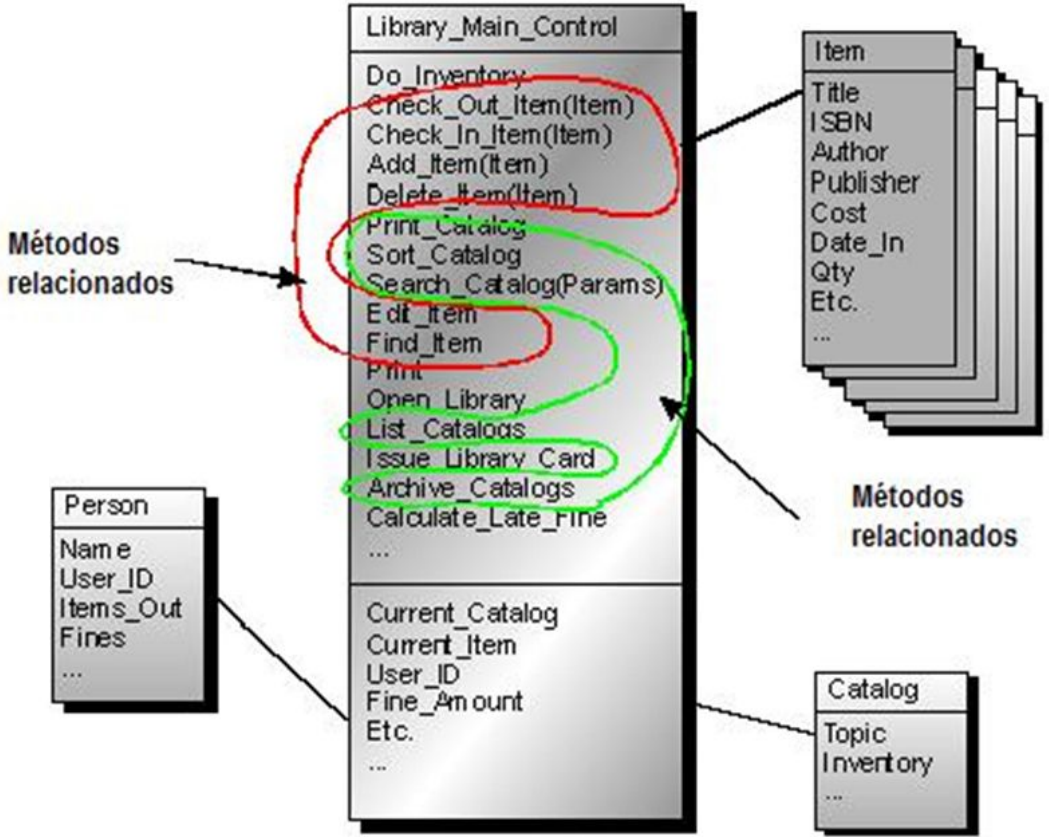
Proceso para el uso de antipatrones

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



Catálogo de antipatrones de desarrollo

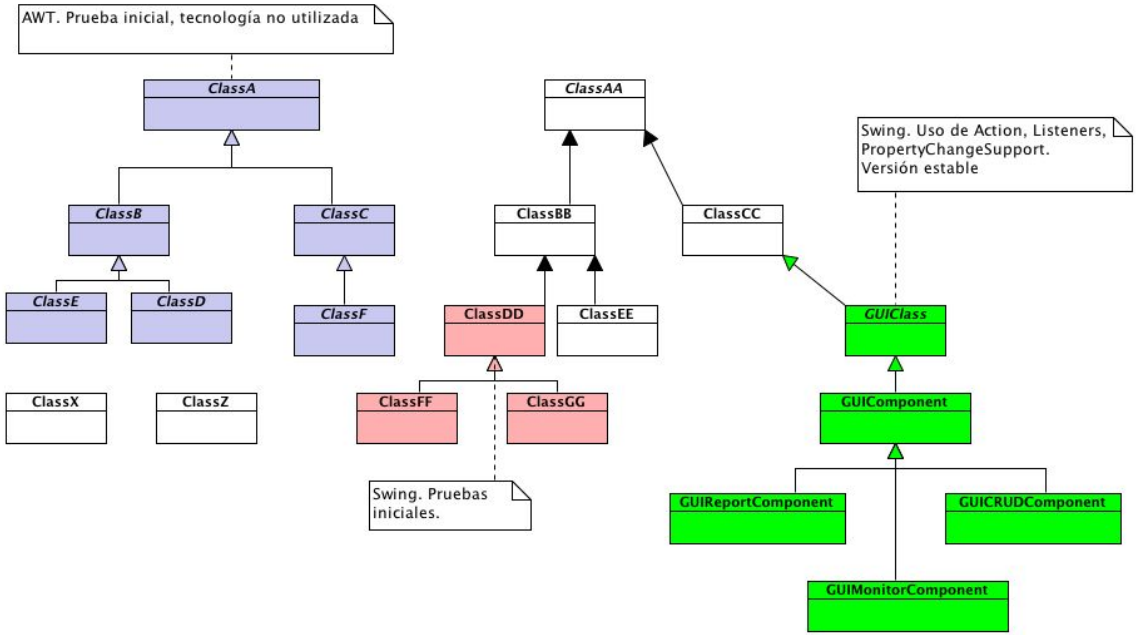
1. **The Blob**: God Class o Winnebago es una **clase**, o componente, que **conoce o hace demasiado**



Catálogo de antipatrones de desarrollo

2. Lava Flow: Dead Code

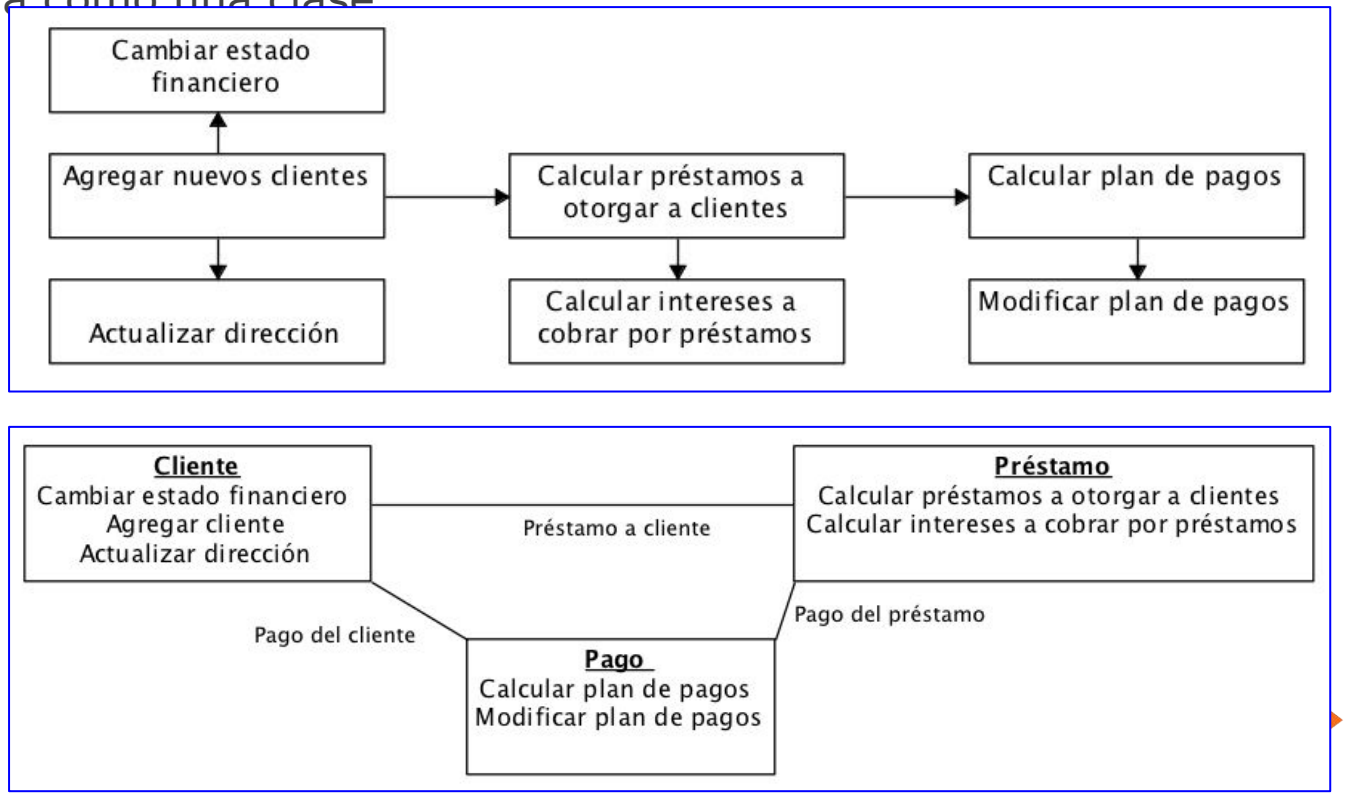
Aparece principalmente en aquellos sistemas que comenzaron como investigación o pruebas de concepto y luego llegaron a producción.



UNIVERSIDAD DE LIMA

Catálogo de antipatrones de desarrollo

3. Functional Decomposition: No object oriented Traducen cada subrutina como una clase



Catálogo de antipatrones de desarrollo

4. Poltergeists: Otros nombres:

- Gipsy
- Proliferation of Classes
- Big DoIt Controller Class.

Las clases fantasmas tienen pocas responsabilidades y un ciclo de vida breve. "**Aparecen**" solamente para iniciar algún método.

Son de relativa facilidad de encuentro ya que sus nombres suelen llevar el sufijo "**controller**" o "**manager**".

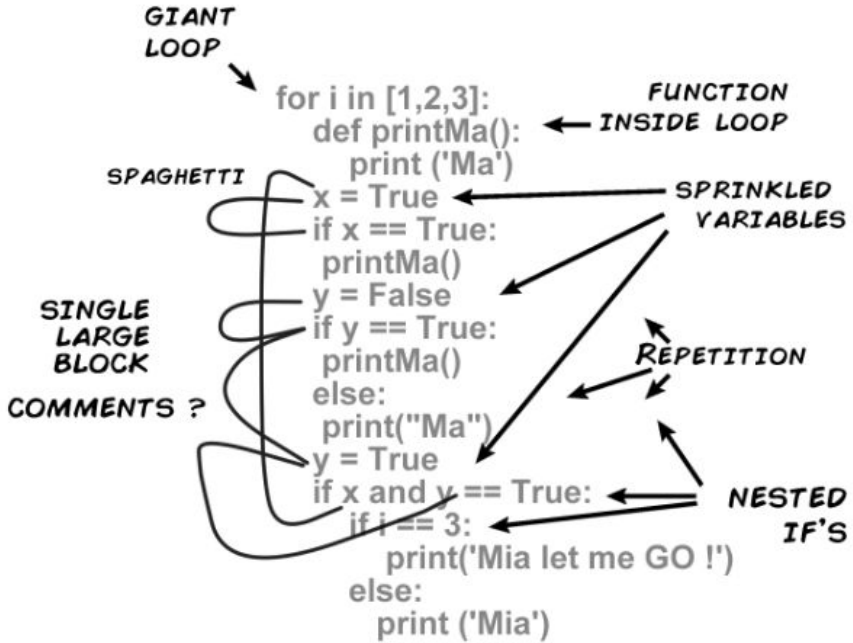
Catálogo de antipatrones de desarrollo

5. 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

6. 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.png

UNIVERSIDAD DE LIMA

Catálogo de antipatrones de desarrollo

7. 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. */

    // ...
}
```

DAD