Ingeniería del Software II

Tema 2: Mantenimiento del Software

A. Goñi, J. Iturrioz

Indice

- Introducción
- Ciclo de vida del mantenimiento
- Tipos de mantenimiento
- Refactorización

Mantenimiento del software.

Definición

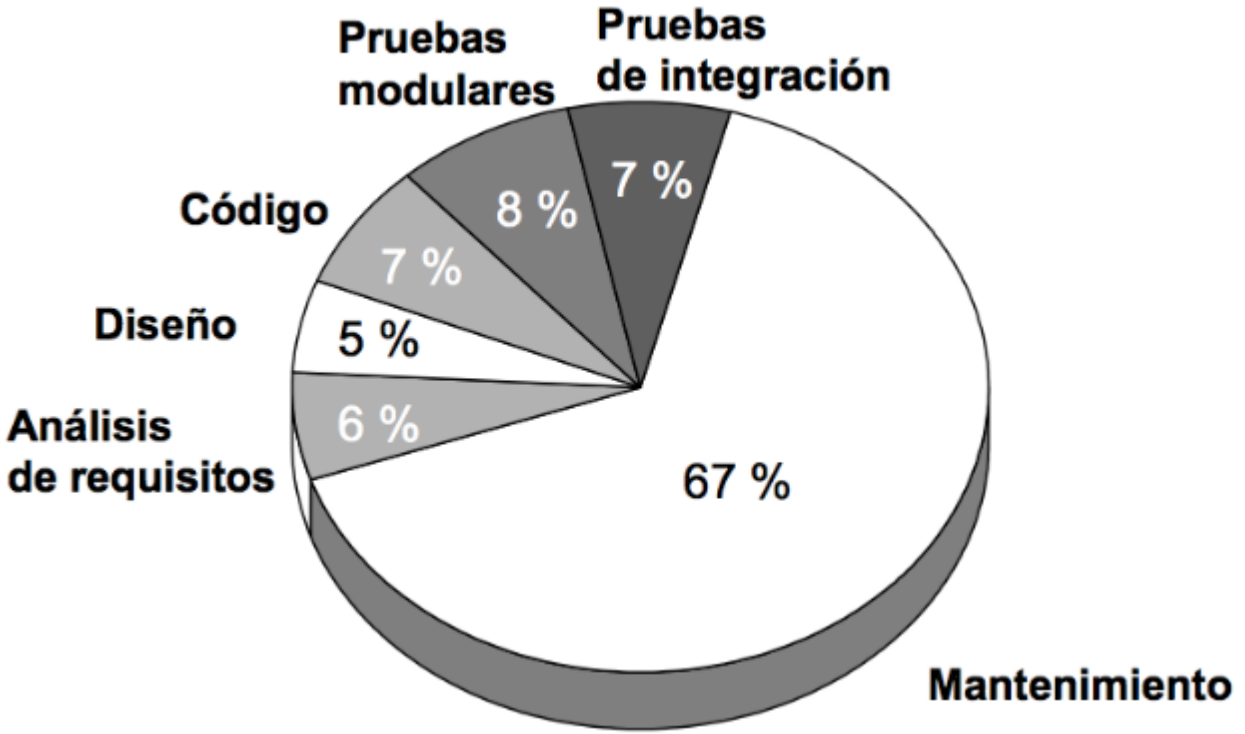
Modificación de un producto software después de su entrega al cliente o usuario para **corregir defectos**, para **mejorar el rendimiento** u otras propiedades deseables, o para **adaptarlo** a un cambio de entorno. [1]

[1] IEEE. Standard for Software Maintenance. 1998. https://ieeexplore.ieee.org/document/720567/

Mantenimiento del software.

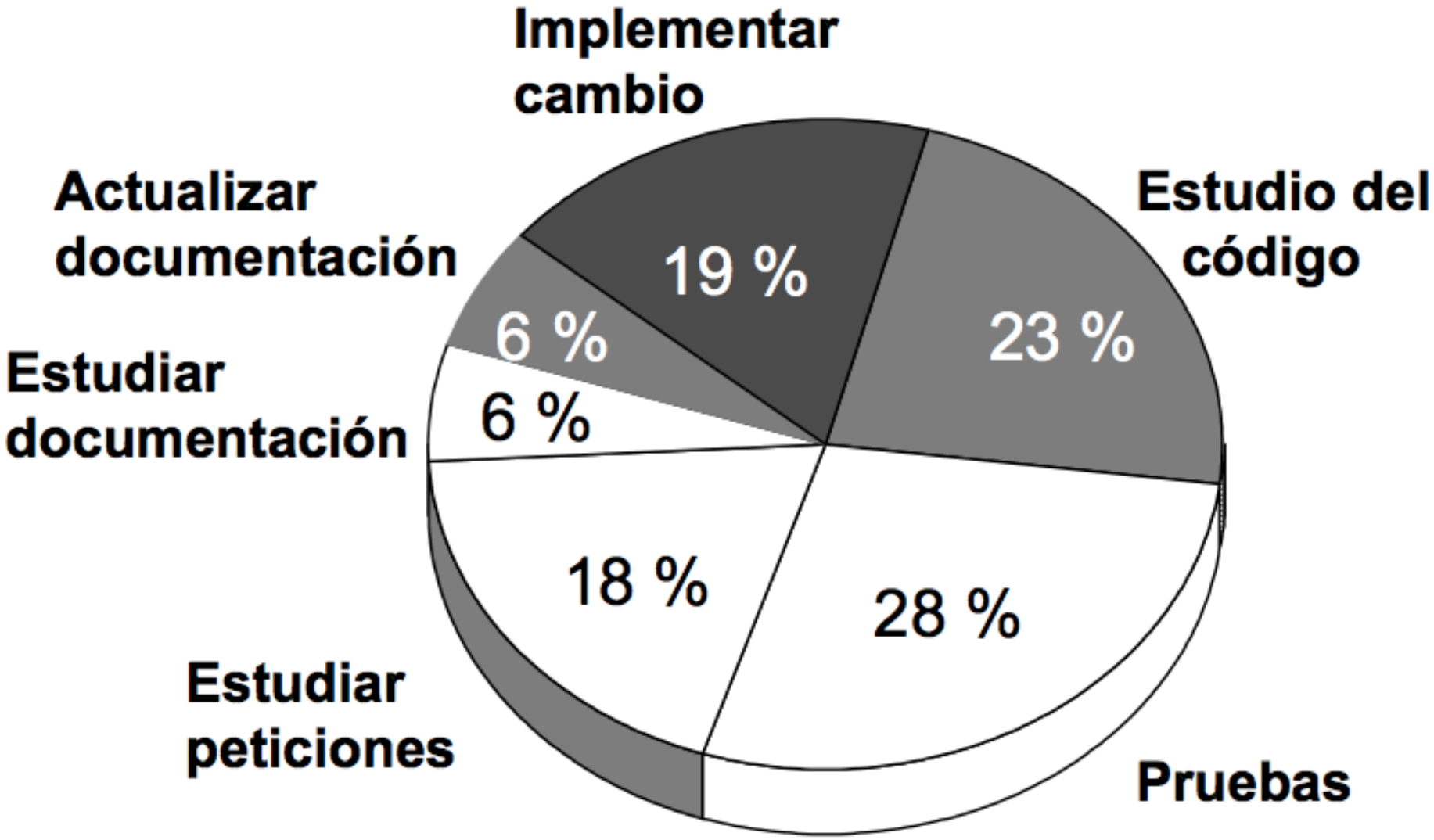
El mantenimiento en el ciclo de vida software

- El mantenimiento de software, es una de las fases en el Ciclo de Vida de Desarrollo de Software.
- Es la fase que consume más tiempo dentro del ciclo de vida.



Mantenimiento del software.

Tiempo empleado en cada tarea de mantenimiento

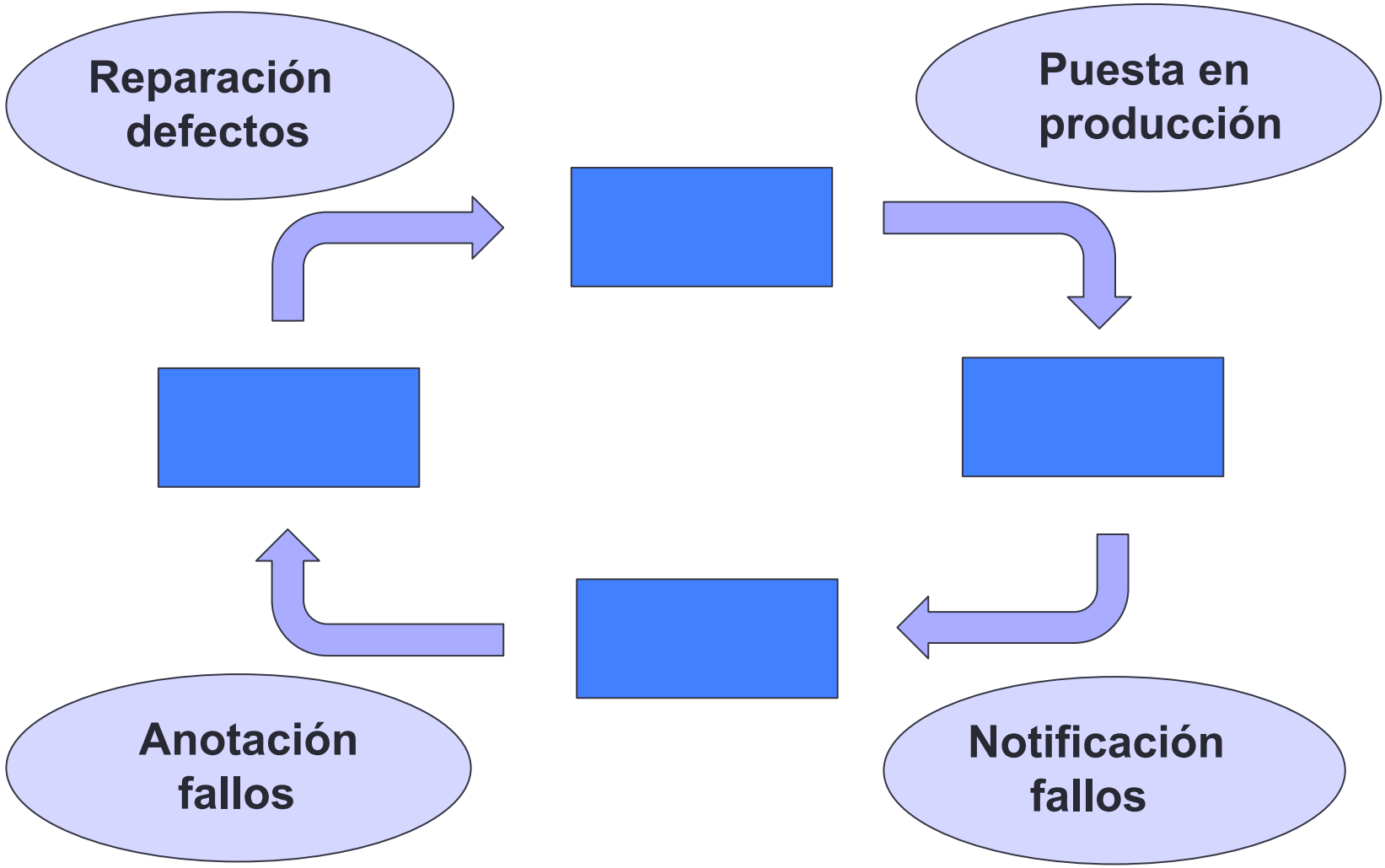


Mantenimiento del software.

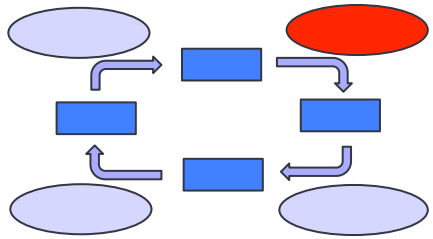
Problemática asociada al mantenimiento

- Inexistencia de métodos, técnicas y herramientas que puedan proporcionar una solución global al mantenimiento.
- La complejidad de los sistemas se incrementa paulatinamente por la realización de continuas modificaciones.
- La documentación del sistema es defectuosa o inexistente.
- Se considera el mantenimiento como una actividad poco creativa, a diferencia del desarrollo.
- Las actividades del mantenimiento se suelen realizar bajo presión de tiempo.
- Escasa formación de los profesionales.

Ciclo de vida en el mantenimiento

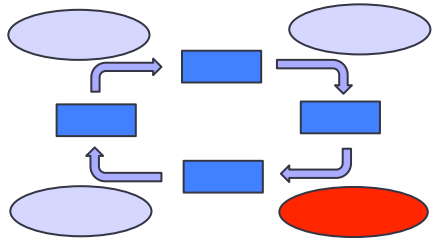


Puesta en producción



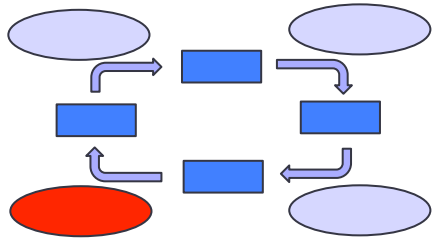
- El Software (como muchos otros productos) se lanza al mercado con un conjunto conocido de defectos y deficiencias.
- El software se lanza con esos defectos conocidos porque la organización decide que la utilidad y el valor del software en un determinado nivel de calidad compensa el impacto de los defectos conocidos.
- Las defectos conocidos se documentan en una carta de consideraciones operacionales o **notas de lanzamiento** (*release notes*) de forma que los usuarios del software puedan trabajar evitando estos defectos y sabiendo cuándo el uso del software puede ser inadecuado para algunas tareas específicas (es decir, fallos).

Notificación de fallos



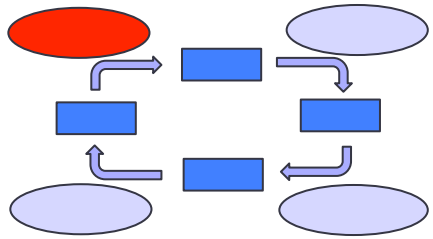
Con el lanzamiento del software (*software release*), los usuarios del software descubrirán nuevos fallos no documentados, y los notificarán al desarrollador del software.

Anotación de fallos



Cuando el desarrollador del software reciba la notificación de un fallo, lo guardará en el **sistema de rastreo.**

Reparación de defectos



El desarrollador del software, clasifica los fallos en base a una prioridad, establece un plan para la reparación de algunos de los defectos que han generado esos fallos y preparará un nuevo lanzamiento del software. Este plan consiste en:

- **Analizar** el fallo y localizar el defecto que lo ocasiona.
- **Diseñar** la solución.
- **Modificar** el software.
- **Probar** el software reparado.

Tipos de mantenimiento: Perfectivo

Mejora del software (**rendimiento**, **flexibilidad**, **reusabilidad**..) o implementación de nuevos requisitos. También se conoce como mantenimiento **evolutivo**.

Firma digital en banca online.

Tipos de mantenimiento: Adaptativo

Adaptación del software a cambios en su entorno tecnológico (nuevo hardware, otro sistema de gestión de **bases de datos**, otro **sistema operativo**...)

Pantallas táctiles, entorno web.

Tipos de mantenimiento: Correctivo

Corrección de fallos detectados en el software durante la fase de explotación. Estos pueden estar relacionados, con la funcionalidad, pero también pueden estar relacionados con el rendimiento o escalabilidad del sistema.

Agujeros de seguridad.

Tipos de mantenimiento: Preventivo

Realización de modificaciones en el software con el objetivo de facilitar el mantenimiento futuro del sistema (mejorar legibilidad...).

Añadir comentarios, reestructurar métodos.

Refactorización

Definición

- Uso común
  - Reorganizar un programa (o algo).
- Como nombre
  - Cambio realizado en la estructura interna de algún componente software para hacerlo más fácil de entender y más barato para modificar, sin modificar el comportamiento observable del software.
- Como verbo
  - Actividad de reestructuración del software aplicando una serie de refactorizaciones sin modificar el comportamiento observable del software.

Qué es refactorizar

- **Pequeños** cambios en el software que cambian su **estructura interna** sin modificar su **comportamiento externo –** Martin Fowler
- Después de refactorizar, hay que verificar que no cambia el comportamiento externo:
  - Testing.
  - Utilizando herramientas de análisis de código.
  - Siendo muy, muy cuidadoso.

Cuando el software deja de ser «soft»

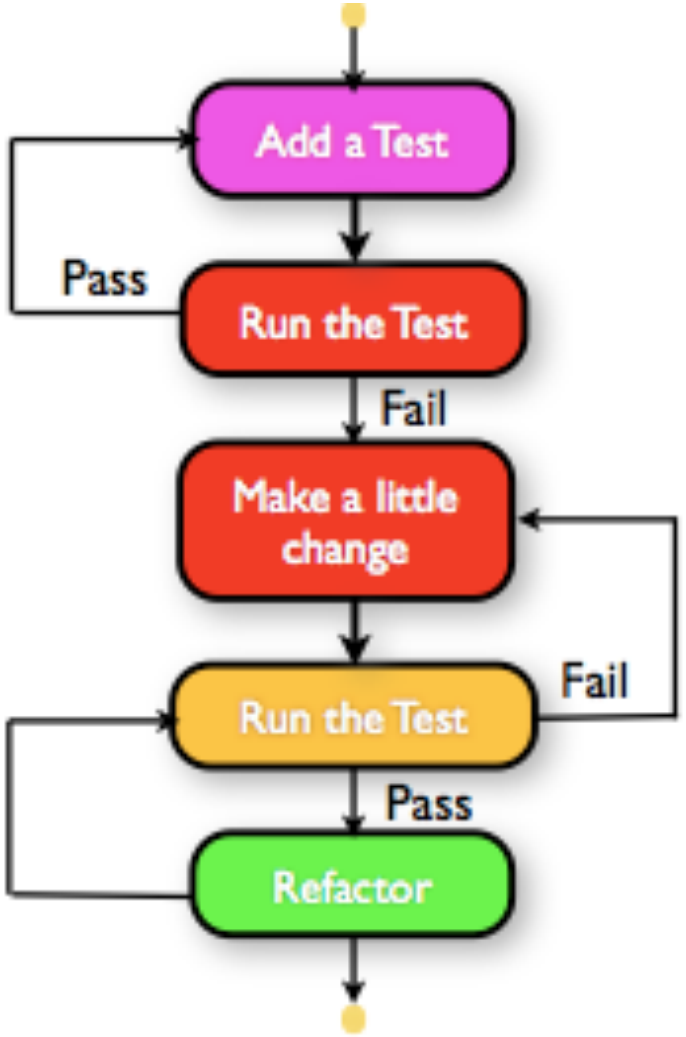
- ¿Te ha tocado trabajar con software difícil de cambiar?
  - No sabes dónde hay que modificar el código.
  - No sabes si es el único lugar donde hay que modificar el código.
  - Al modificar el código, no tienes certeza si el programa va a fallar en alguna otra parte.
  - Hay demasiadas dependencias internas.
  - No hay un diseño claro.
  - Es difícil probar un módulo de forma aislada.
  - Es difícil intercambiar una pieza con otra.
- ***El software se convierte en una gran bola de lodo.***

¿Por qué refactorizar?

- Con tiempo el software adquiere **Deuda técnica.**
  - Crecimiento no regulado.
- Debido a:
  - Desarrollo Quick and dirty. Resultados inmediatos a corto plazo.
- Esto nos lleva a:
  - Información duplicada y globalizada.
  - Estructura erosionada.
- Para mitigarlo, necesitamos:
  - Escribir software más fácil de entender.
  - Escribir código para las personas, no para el compilador.
- Si funciona, no lo toques
  - ¿Sabiduría de ingeniería, o es la señal de que ya no tenemos control?

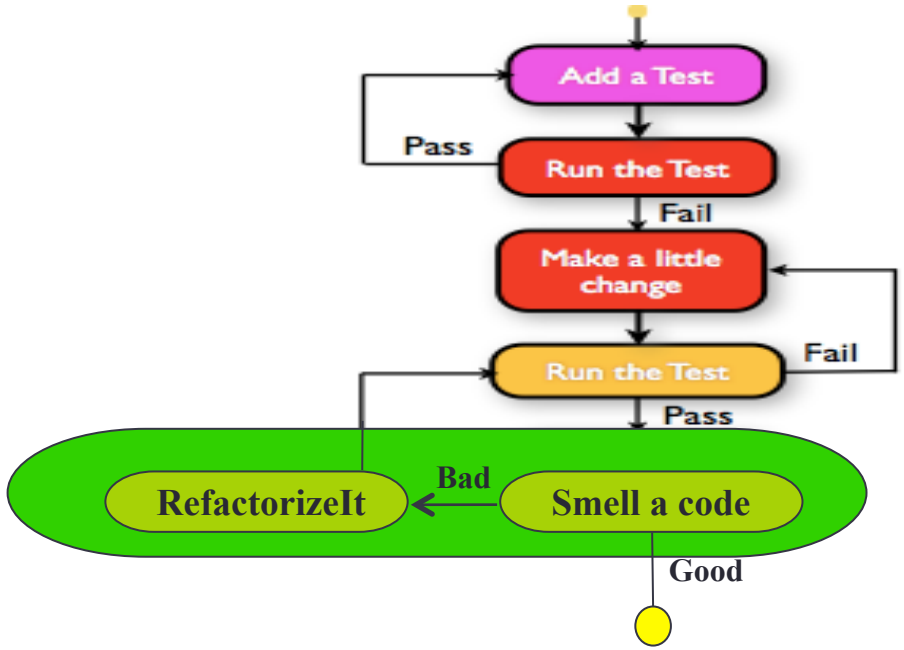
Refactoring y Test Driven Development (TDD)

- Una vez finalizado el proceso de test, pasamos a la etapa de refactorización, donde procedemos a realizar pequeños cambios con el objetivo de mejorar el código existente.



Refactoring y Test Driven Development (TDD)

- Descubre cómo «huele» (***code smell***) el código.
- Refactoriza (eliminando el mal olor).
- Ejecuta las pruebas.



Malos olores del código fuente

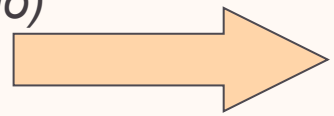
- Metáfora para problema potencial en código.
- Es un síntoma.
- Es difícil imponer métricas o reglas exactas
  - ¿Cuál es el numero máximo de líneas que un método debe tener?
  - ¿Y número mínimo?
  - Herramientas de análisis estática de código fuente ayudan… pero hasta cierto punto.

Los "*Bad Smells*" propuestos por Fowler [2]

- Descripción de 22 defectos de diseño o ***Bad-Smells***
  - Bajo nivel. A nivel de clase y método
  - Ventajas: propuesta de refactorización.
  - Modificar el diseño en base al "Bad Smell"

Duplicate code (código duplicado)

Long method(métodos largos)



```
Refactorización: Extract
Method
Extrae un fragmento en un
método, cuyo nombre sea su
propósito
```

```
void printTrabajador() {
  printPhoto();

  System.out.println ("name: " + _name);
  System.out.println ("amount " + getSueldo());
 }
```

```
void printTrabajador() {
  printPhoto();
  printDetalles(getSueldo());
}

void printDetalles(float sueldo);
  System.out.println ("name: " + _name);
  System.out.println ("sueldo " + sueldo);
 }
```

[2] Libro Refactoring: Improving the Design of Existing Code. M. Fowler, J. Brant, W. Opdyke, D. Roberts. Addison Wesley, 1999

Clasificación de "*Bad Smells*"

The bloaters (los infladores), agrupa "aromas" que indican el crecimiento excesivo de algún aspecto que hacen incontrolable el código.

The Object-Orientation Abusers (los maltratadores de la orientación a objetos), que aglutina *code smells* que indican que no se está aprovechando la potencia de este paradigma.

The change preventers (los previsores de cambios). Cambios en los requisitos afectan a muchos artefactos del diseño.

The Dispensables (los prescindibles), pistas aportadas por porciones de código innecesarias que podrían y deberían ser eliminadas.

The couplers (Los emparejadores), son *smells* que alertan sobre problemas de acoplamiento entre componentes, a veces excesivo y otras demasiado escaso.

Categorización de Bad smells

BLOATERS

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- DataClumps

OO-ABUSERS

- Switch Statements
- Temporary Field
- Refused Bequest
- Alternative Classes with Different Interfaces

CHANGE PREVENTERS

- Divergent Change
- Shotgun Surgery
- Parallel Inheritance Hierarchies

DISPENSABLES

- Lazy class
- Data class
- Duplicate Code
- Dead Code
- Speculative Generality

COUPLERS

- Feature Envy
- Inappropriate Intimacy
- Message Chains
- Middle Man

Ejemplo Bad smell: Long Method

Definición

- Olores que desprende:
  - Difícil de modificar, mantener, depurar
  - Difícil de entender
  - Difícil de reutilizar
- Cómo detectarlo: ¿Cuál es el numero máximo de las líneas de código fuente para un método bien hecho?
- Ejemplo:
- Fuentes:
  - http://sourcemaking.com/refactoring/extract-method
  - http://lostechies.com/seanchambers/2009/08/10/refactoring-day-10-extract-method/
- Refactorizaciones: **Extraer método (opcional desarrollarlo)**

Ejemplo Bad smell: Long Method

Código inicial

```
public class Receipt {
  private Vector discounts;
  private Vector itemTotals;

  public float calculateGrandTotal() {
    float subTotal = 0;
    Enumeration items=itemTotals.elements();
    while (items.hasMoreElements())
     subTotal += (Float)items.nextElement();

    if (!discounts.isEmpty()) {
     Enumeration discs=discounts.elements();
     while (discs.hasMoreElements())
     subTotal -= (Float)discs.nextElement();
    }
    double tax = subTotal*0.04;
    subTotal += tax;
    return subTotal;
  }
}
```

Ejemplo Bad smell: Long Method

Código refactorizado aplicando Extract Method

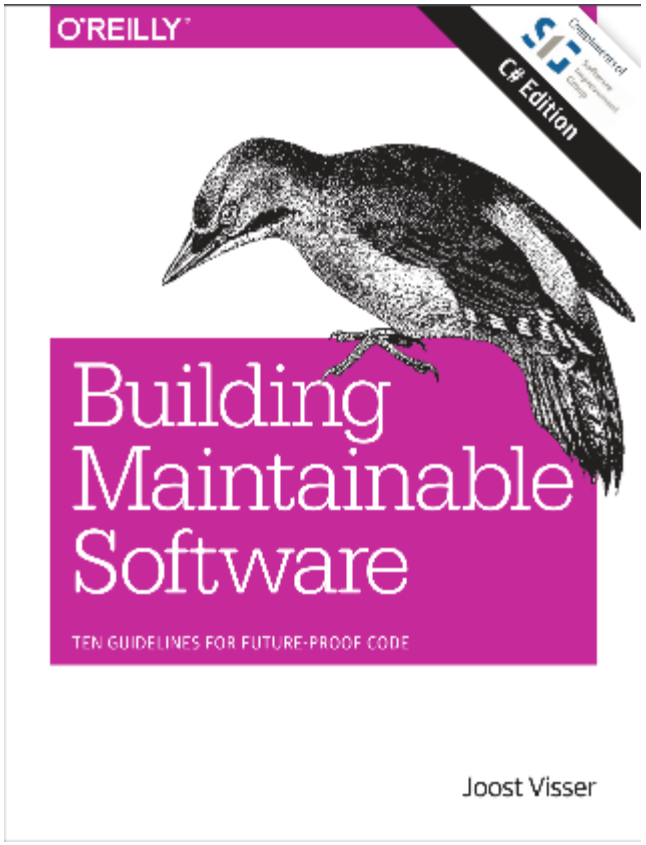
```
public class Receipt {
    private Vector discounts;
    private Vector itemTotals;

public float calculateGrandTotal() {
   float subTotal = calculateSubTotal();
   subTotal = calculateDiscount(subTotal);
   subTotal = calculateTax(subTotal);
   return subTotal;
 }
public float calculateSubTotal() {
 float subTotal = 0;
 Enumeration items=itemTotals.elements();
 while (items.hasMoreElements())
  subTotal += (Float)items.nextElement();
  return subTotal;
 }
public float calculateDiscounts(float subTotal){
 if (!discounts.isEmpty()) {
  Enumeration discs=discounts.elements();
  while (discs.hasMoreElements())
    subTotal -= (Float)discs.nextElement();
 }
 return subTotal;
}
public float calculateTax(float subTotal) {
 double tax = subTotal*0.04;
 subTotal += tax;
 return subTotal;
}
}
```

¿Son los comentarios un Bad Smell?

- Los comentarios mienten, el código no miente.
  - El código se ejecuta, los comentarios no.
- Llegan a ser obsoletos con facilidad.
- Son buena señal de que el código está poco claro, demasiado complejo.
- Muchas veces son indicador de que un bloque de código merece ser un método aparte.
- Las pruebas unitarias son la mejor alternativa para documentar el código.
- Posibles Refactorizaciones: Extraer método, Renombrar.
- Siempre disponer de las pruebas unitarias necesarias para verificar la refactorización realizada.

Otros “Bad smells” [3]



- ¿Cuántas líneas de código un método?
- ¿Cuántas bifurcaciones un método?
- ¿Cuántos parámetros un método?
- ¿Qué pasa con el código replicado?

[3] Libro Building Maintainable Software.
https://www.sig.eu/wp-content/uploads/2017/02/Building_Maintainable_Software_C_Sharp_SIG.pdf