Universidad Nacional Autónoma de México

RED UNIVERSITARIA DE COLABORACIÓN
EN INGENIERÍA DE SOFTWARE Y BASES DE DATOS

Análisis estático de código fuente

Presenta: Daniel Barajas González

Septiembre de 2018

ldanielbg@comunidad.unam.mx

Contenido

1. Problemática
2. Análisis estático de código fuente
3. Mejorar la calidad de nuestras bases de código
4. Conclusiones

Bienvenido, eres nuestro nuevo programador

Tu asignación:

Enfrentar una base de código ajena, para:

- Modificar alguna funcionalidad
- Agregar nueva funcionalidad
- Corregir bugs
- Modernizar la aplicación

… Y no, no hay tiempo de volver a escribir la aplicación.

Problemática

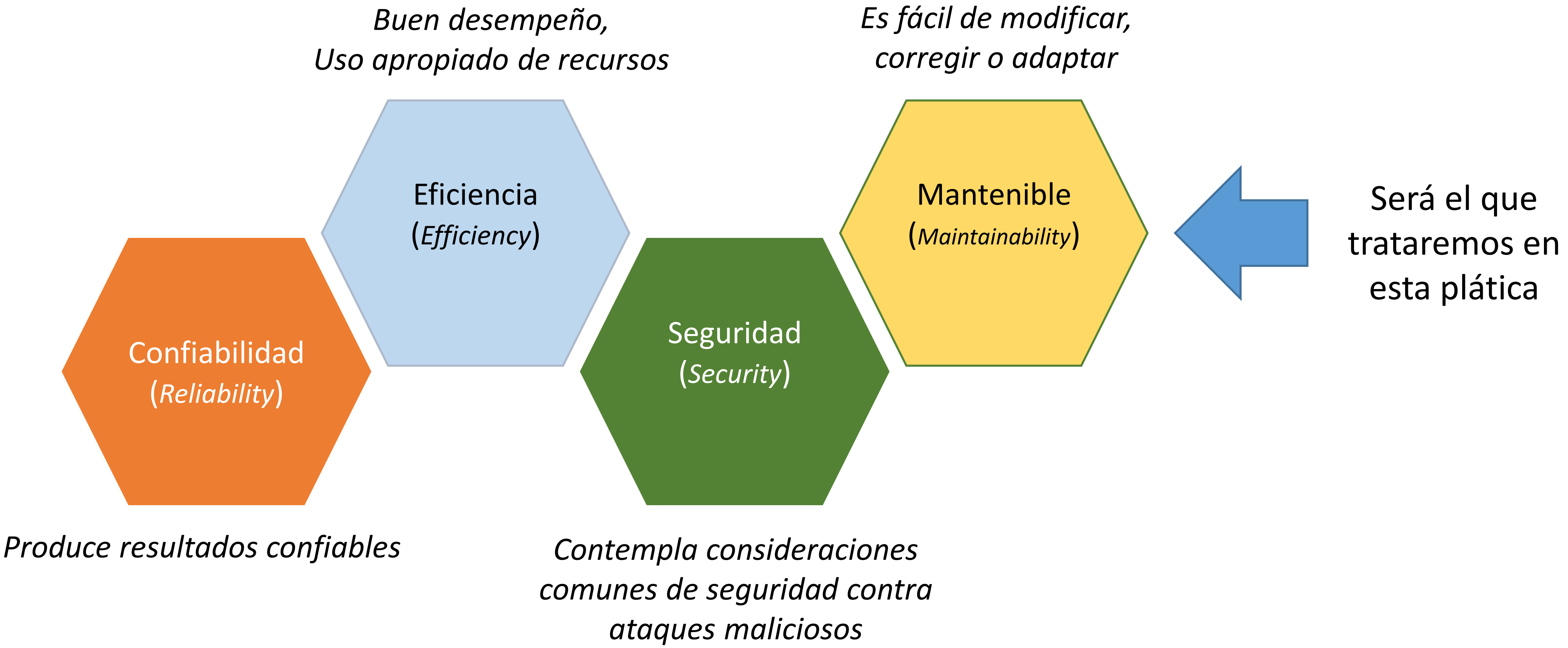
La problemática con las bases de código aparece cuando es difícil de mantenerlas

- Complejidad excesiva
- Carece de pruebas unitarias
- Clases, métodos o funciones muy grandes
- Numerosas sentencias IF-ELSE
- Sin comentarios en el código
- Documentación nula o irrelevante
- Código muerto
- Nombres de variables poco significativos

*** inserte otra característica aquí ***

Cualidades de una buena base de código

Pueden variar entre programadores y niveles de experiencia



Cómo detectar problemas en el código

Code smells o la Hediondez del código

- Los **code smells** son una metáfora para referirse a problemas en el código que pueden identificarse a simple vista

- Clases o métodos demasiado largos
- Sentencias switch
- Listas largas de parámetros
- Código duplicado
- Nombres demasiado cortos
- Nombres demasiado largos
- Líneas de código demasiado largas
- Demasiadas sentencias “return”
- Excesiva Complejidad *ciclomática*

Ver: https://martinfowler.com/bliki/CodeSmell.html | https://en.wikipedia.org/wiki/Code_smell | https://sourcemaking.com/refactoring/smells

Cómo detectar problemas en el código

Anti-Patrones: soluciones (inapropiadas) para problemas comunes

- Los anti-patrones son soluciones a las que recurrimos y que conducen a un software difícil de comprender y modificar
- También existen anti-patrones en el desarrollo de software, en la arquitectura de software y en la administración de proyectos
  - **Código espagueti** – Ocasiona estructuras difíciles de “seguir”
  - **Hard code** – Ocurre al adoptar supuestos sobre el entorno del sistema. Por ejemplo, rutas de ejecución, credenciales de conexión.
  - **Complejidad accidental –** Soluciones demasiado complejas a problemas simples
  - **Confianza ciega** – Descuidar la verificación de resultados. Por ejemplo, valores que devuelve una función
  - **Ocultación de errores** – Capturar un error y mostrar un mensaje irrelevante al usuario

Ver: https://sourcemaking.com/antipatterns

Consecuencias

Las consecuencias de una mala calidad en las bases de código

Una base de código con mala calidad, puede elevar el costo de mantenimiento de una aplicación hasta llevarla al abandono

- Dado el **alto costo** (tiempo, dinero, personas) que implica el desarrollo de software, es **conveniente** que permanezca **fácil de entender y modificar**

- Los **requerimientos** del software **cambian** constantemente para adaptarse al negocio. El software debe adaptarse

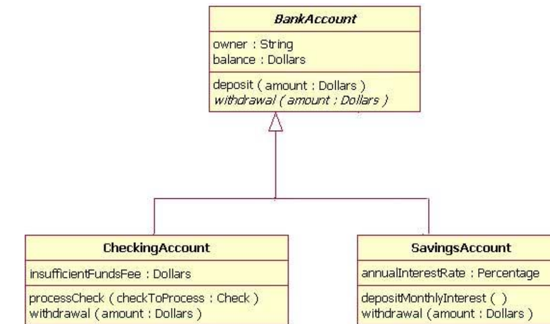
- Una base de código de mala calidad puede **costar** hasta 4 veces más **mantenerla** de lo que costó construirla

Reduce la vida útil del software

Análisis estático de código

Qué es el análisis estático de código

- Se enfoca en examinar los **elementos estructurales** que conforman el software (funciones, clases, variables)



- Se realiza sobre el **código fuente sin ejecutar** el software

- Puede apoyarse en **herramientas automáticas**

- También puede ser realizado por un ser humano (**revisiones de código**)

Herramientas para Análisis estático

- Permiten **detectar problemas** en una base de código
- Trabajan con **conjuntos de reglas** predefinidos
- Las hay diversos tipos de licencia, **comerciales** y también **open source**
- La mayoría de **entornos integrados de desarrollo** (IDEs) brindan soporte a estas herramientas
- Algunas soportan **múltiples lenguajes de programación** y otras son especializadas en uno solo

Herramienta	URL
SonarQube	https://www.sonarqube.org/
Yasca	http://scovetta.github.io/yasca/
Sonargraph	http://www.hello2morrow.com/products/sonargraph
Infer	http://fbinfer.com/

Ver: https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

Herramientas para Java

Algunas de las herramientas de análisis para Java que son gratuitas

Herramienta	URL
Checkstyle	http://checkstyle.sourceforge.net/
Error Prone	http://errorprone.info/
FindBugs	http://findbugs.sourceforge.net/
PMD	https://pmd.github.io/
Squale	http://www.squale.org/squale-user-site/downloads.html
Soot	https://sable.github.io/soot/

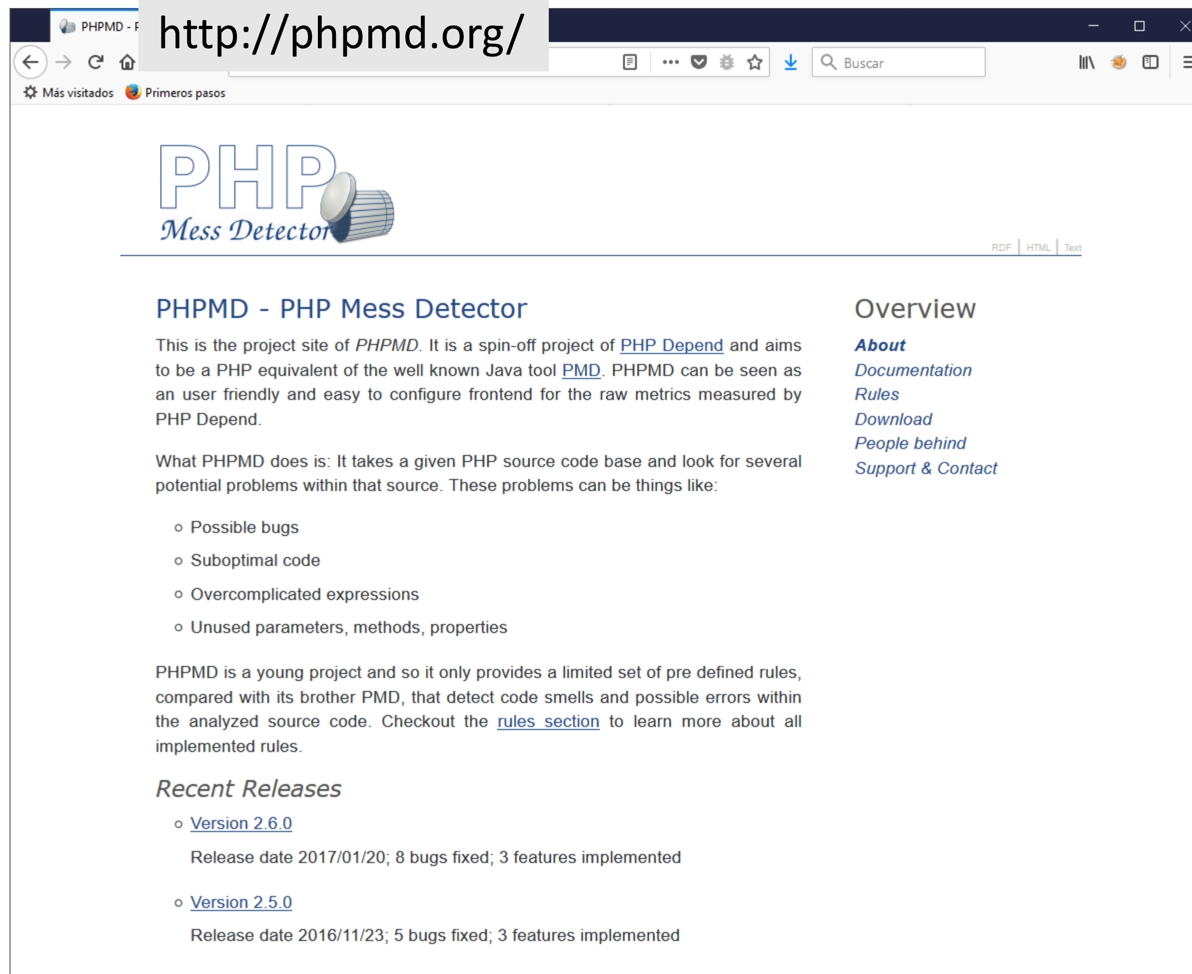
Herramientas para PHP

Algunas de las herramientas de análisis para PHP

Herramienta	URL
PHPMD	https://phpmd.org/
PDepend	https://pdepend.org/
PHPMetrics	https://www.phpmetrics.org/
phpcs	https://github.com/squizlabs/PHP_CodeSniffer
phploc	https://github.com/sebastianbergmann/phploc

PHP Mess Detector

Es el equivalente a la herramienta PMD para Java



- Es una herramienta para detectar problemas en una base de código escrita en **PHP**
- Utiliza un conjunto predefinido de **reglas**
- Posee una **buena documentación** incluso de sus reglas
- Puede generar diferentes **formatos de salida**: texto plano, xml y html
- Licencia **GPL**

Ejemplo con PHP Mess Detector

$ phpmd <*directorio*> <*formato-de-salida*> <*reglas-a-revisar*>

Text

XML

HTML

```
/application/models/checklist_model.php:201    The method obtener_listado() has 140 lines of code. Current threshold is set to 100. Avoid really long methods.
/application/models/contacto_model.php:161     The method regresa_listado_contacto() has a Cyclomatic Complexity of 10. The configured cyclomatic complexity threshold is 10.
/application/models/contacto_model.php:161     The method regresa_listado_contacto() has an NPath complexity of 384. The configured NPath complexity threshold is 200.
```

Archivos

Problemas encontrados

Reglas predefinidas de PHP Mess Detector

Conjunto de Reglas	Qué señalan
Clean Code rules	• Uso de banderas lógicas • Uso de sentencias “else” • Acceso estático a métodos y atributos
Code Size rules	• Complejidad ciclomática • Complejidad Npath • Longitud excesiva en clases, métodos • Listas de parámetros demasiado largas • Demasiados métodos públicos • Demasiados atributos en una clase • Demasiados métodos en una clase
Naming rules	• Nombres demasiado cortos • Nombres demasiado largos • Omitir convenciones de nombrado para constantes, constructores y *getters* lógicos

Reglas predefinidas de PHP Mess Detector

Conjunto de Reglas	Qué señalan
Unused code rules	• Campos privados declarados y no utilizados • Variables locales declarados y no utilizados • Métodos privados declarados y no utilizados • Parámetros declarados y no utilizados
Controversial rules	• Omitir el uso de la notación camel case al nombrar clases, métodos y variables
Design rules	• Uso de instrucciones “exit”, “goto”, “eval”. • Relaciones de herencia muy profundas. • Alto acoplamiento entre clases

Complejidad ciclomática

1 de 2

- Introducida por Thomas McCabe en ‘76
- Es una **métrica cuantitativa** de la complejidad lógica de un programa
- Es **independiente** del lenguaje
- Cuantifica los **puntos de decisión** dentro de un fragmento de código

Umbrales	Descripción
1 a 10	Programa Simple, sin mucho riesgo
11 a 20	Más complejo, riesgo moderado
21 a 50	Complejo, Programa de alto riesgo
> 50	Programa no testeable, Muy alto riesgo

Ver: https://code2read.com/2015/03/25/code-metrics-complejidad-ciclomatica/

```php
<?php
class Foo
{
1   public function example() {
2       if ($a == $b) {
3           if ($a1 == $b1) {
                fiddle();
4           } else if ($a2 == $b2) {
                fiddle();
            } else {
                fiddle();
            }
5       } else if ($c == $d) {
6           while ($c == $d) {
                fiddle();
            }
7       } else if ($e == $f) {
8           for ($n = 0; $n > $h; $n++) {
                fiddle();
            }
        } else {
            switch ($z) {
9               case 1:
                    fiddle();
                    break;
10              case 2:
                    fiddle();
                    break;
11              case 3:
                    fiddle();
                    break;
12              default:
                    fiddle();
                    break;
            }
        }
    }
}
```

Ejemplo 1

1. Empieza agregando 1 punto por la declaración de la función
2. Agrega 1 punto por cada IF, WHILE, CASE, FOR

La complejidad ciclomática de este fragmento de código es 12, se considera una complejidad alta.

Ver: https://modess.io/npath-complexity-cyclomatic-complexity-explained/

Complejidad ciclomática y pruebas

¿Cuántas pruebas debo realizar para lograr cobertura del 100% en mi set de pruebas?

```
void ejemplo2(int indice){
    if (indice == 0){
        //Algo pasa
    } else if (indice == 1) {
        //Haz esto
    } else {
        //Haz aquello
    }
}
```

Complejidad ciclomática (CC) = 3

Flujo normal	1 punto
If (índice == 0)	1 punto
If (índice == 1)	1 punto

Ver: https://code2read.com/2015/03/25/code-metrics-complejidad-ciclomatica/

Complejidad NPath

- Es una **métrica cuantitativa** de la complejidad lógica de un programa
- Es **independiente** del lenguaje
- Cuantifica los **caminos distintos** que puede tomar el flujo dentro de un programa
- Puede calcularse a partir de la complejidad ciclomática
- La complejidad **Npath** se comporta de forma exponencial

Umbrales	Complejidad
1 a 16	Baja
17 a 128	Moderada
129 a 1,024	Alta
>= 1,025	Muy alta

Si estas leyendo esto una de dos, estas muy interesado o estas muy aburrido. Espero que sea la primera

Ejemplo de complejidad NPath

```php
<?php
function foo($a, $b)
{
    if ($a > 10) {
        echo 1;
    } else {
        echo 2;
    }
    if ($a > $b) {
        echo 3;
    } else {
        echo 4;
    }
}

foo(1, 2); // Outputs 24
foo(11, 1); // Outputs 13
foo(11, 20); // Outputs 14
foo(5, 1); // Outputs 23
```

Los posibles caminos son:

1) Se cumple el primer IF, se cumple el segundo IF
2) Se cumple el primer IF, no se cumple el segundo IF
3) No se cumple el primer IF, se cumple el segundo IF
4) No se cumple el primer IF, no se cumple el segundo IF

La complejidad NPath es de 4

2 sentencias IF * 2 posibles resultados = 4 caminos posibles

Agregamos otra sentencia con 2 posibles resultados:

2 * 2 * 2 = 8

Si estas leyendo esto una de dos, estas muy interesado o estas muy aburrido. Espero que sea la primera

Cálculo de Npath usando CC

CC = Complejidad ciclomática

- En algunos casos, no devuelve el valor real sino al valor máximo del Npath
- La fórmula es la siguiente

$$Npath = 2 ^ (CC - 1)$$

Donde CC es la complejidad ciclomática

```
void ejemplo2(int indice){
    if (indice == 0){
        //Algo pasa
    } else if (indice == 1) {
        //Haz esto
    } else {
        //Haz aquello
    }
}
```

CC = 3
Npath = 2 ^ (3 – 1)

```
function foo($a, $b)
{
    if ($a > 10) {
        echo 1;
    } else {
        echo 2;
    }
    if ($a > $b) {
        echo 3;
    } else {
        echo 4;
    }
}
```

CC = 3
Npath = 2 ^ (3 – 1)

Ver: *http://www.michael-pratt.com/blog/17/Complejidad-N-Path/*

Casos reales

```
The method guardar() has a Cyclomatic Complexity of 28. The configured cyclomatic complexity threshold is 10.
The method guardar() has an NPath complexity of 25200. The configured NPath complexity threshold is 200.
The method guardar() has an NPath complexity of 818251200. The configured NPath complexity threshold is 200.
```

```
The method guardar() has an NPath complexity of 197837640165.
```

Modelos y métricas

- Existen muchos modelos de medición y métricas. Algunas son métricas de complejidad y otras de orientación a objetos.
  - Robert C. Martin metrics
  - McCabe metrics
  - Halstead complexity measures
  - Moose, MOOD, MOOD2 object-oriented metrics
  - Card & Agresti complexity metrics
- Tomar aquellas que te sean significativas y útiles
- Las métricas pueden variar entre herramientas así como los umbrales

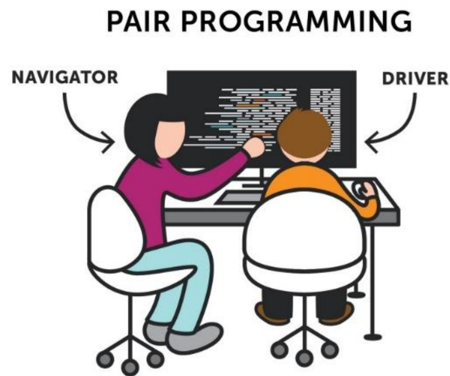
#	Métrica	Descripción	Modelo
1	ca	Afferent coupling : Number of classes affected by this class	Martin Metrics - Ca
2	bugs	Number of estimated bugs by file (Halstead metric)	Halstead metrics
3	commw	Comment weight measure the ratio between logical code and comments	
4	cc	Cyclomatic complexity number measures the number of linearly independent paths through a program's source code	McCabe
5	dc	Data Complexity (Card and Agresti metric)	Card and Agresti metrics
6	diff	Difficulty of the code (Halstead metric)	Halstead
7	ce	Efferent coupling : Number of classes that the class depend	Martin Metrics - Ce
8	effort	Effort to understand the code (Halstead metric)	
9	instability	Ratio between ce and ca (I = ce / (ce + ca). Indicates the class's resilience to change	Martin Metrics - I
10	IC	Intelligent content	?
11	lcom	Lack of cohesion of methods measures the cohesiveness of a class	MOOSE - LCOM
12	length	Length (Halstead metric)	Halstead
13	loc	Number of lines of code	
14	lloc	Number of logical lines of code	
15	MI	Maintenability index is based on Halstead's metrics, LOC and Cyclomatic complexity number	?
16	MIwC	Maintenability Index without comments	?
17	distance	Myer's distance is derivated from Cyclomatic complexity	
18	interval	Myer's interval indicates the distance between cyclomatic complexity number and the number of operators	
19	noc	Number of classes	
20	noca	Number of abstract classes	
21	nocc	Number of concrete classes	
22	operators	Number of operators	
23	rdc	Relative data complexity (Card and Agresti metric)	Card and Agresti metrics
24	rsc	Relative structural complexity (Card and Agresti metric)	Card and Agresti metrics
25	rsysc	Relative System complexity (Card and Agresti metric)	Card and Agresti metrics
26	sc	System complexity (Card and Agresti metric)	Card and Agresti metrics
27	sysc	Total System complexity (Card and Agresti metric)	Card and Agresti metrics
28	time	Time to read and understand the code (Halstead metric)	Halstead metrics
29	vocabulary	Vocabulary used in code (Halstead metric)	Halstead metrics
30	volume	Volume (Halstead metric)	Halstead metrics

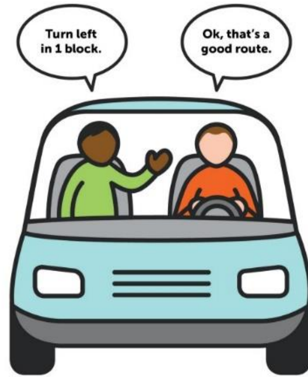
Avisos y recomendaciones

1. Cumplir con las métricas no debería volverse ser una meta por sí misma, úsalas como una **guía para mejorar tus prácticas** de construcción y diseño de software

2. En ocasiones las herramientas pueden arrojar **falsos positivos** (Ej. Models en CakePHP v.2) ¡Cuidado!

Avisos y recomendaciones





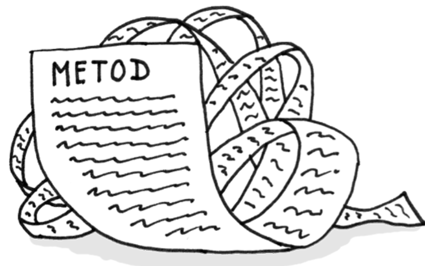
- Los resultados del análisis estático del código deben tomarse como indicadores de problemas más profundos. No son “números que hay que bajar”, el problema detrás a menudo es **comunicación** y/o **niveles de experiencia**

- La meta real debe ser **lograr una base de código fácil de mantener** para que la vida útil del software sea larga

Resolver problemas del código

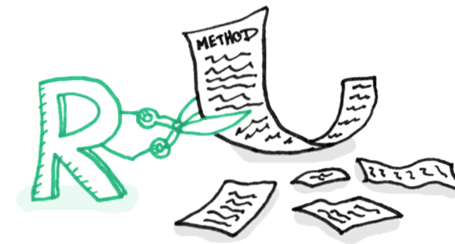
*Por lo regular para cada **code smell**, **anti-patrón** y problema existirá un **tratamiento**, un **refactoring** o algún **patrón de diseño** efectivo para resolver el problema.*

Ver: https://sourcemaking.com/refactoring



Problema

- Método demasiado largo



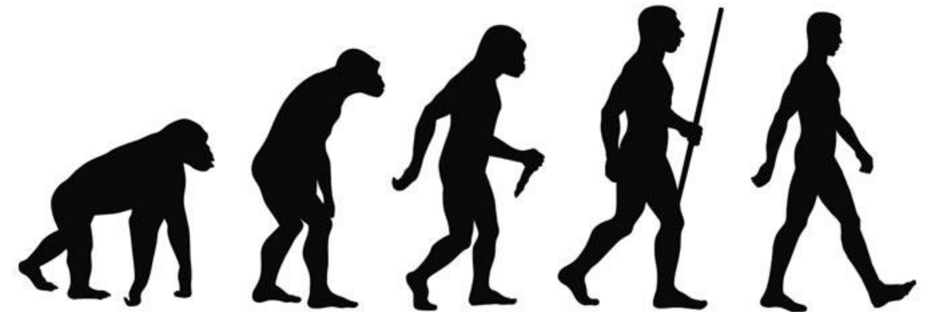
Técnicas de refactoring recomendadas:

- Extract method
- Replace method with method object
- Decompose conditional

¿Refactoring?

Refactorizar código

- Una **técnica** de la ingeniería de software para **reestructurar código fuente**
- Consiste en **modificar la estructura interna** del código **sin alterar el comportamiento externo**
- Busca **mejorar el diseño** después de que el código ha sido escrito



Catálogos de técnicas de Refactoring

Catálogo de Martin Fowler https://refactoring.com/catalog/

Guía de SourceMaking https://sourcemaking.com/refactoring

Acciones preventivas

1. **Define estándares de programación y apégate a ellos**
   - Usa listas de verificación para asegurar consistencia entre bases de código

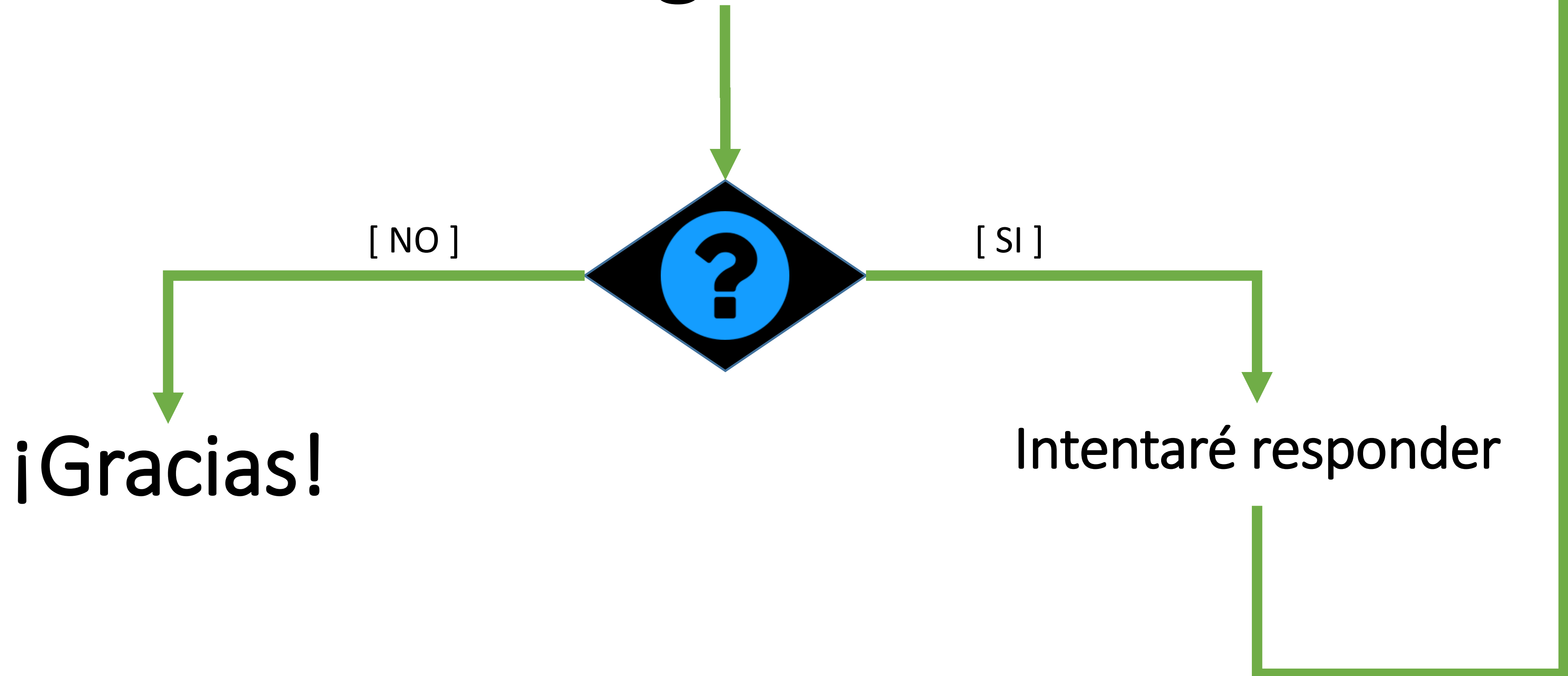
2. **Realiza revisiones de tu código**
   - Revisión de autor@
   - Revisión de pares (cruzadas)
   - Revisión automatizadas
   - Para descubrir problemas potenciales
   - Para mejorar su legibilidad

3. **Utiliza una herramienta de control de versiones**
   - Versiona tu código, tu documentación y hasta tus scripts SQL

Acciones preventivas

4. **Crea pruebas unitarias**
   - Para identificar el impacto de los cambios en tu base de código
5. **Refactoriza tu código**
   - Para que sea más legible
   - Para que sea fácil de mantener y modificar
6. **Documenta tu software**
   - Como si mañana fueras a perder la memoria
   - Comentarios pertinentes en tu código fuente
   - Las razones que sustentan tus decisiones de diseño y desarrollo
   - Las consideraciones especiales para que funcione el software
   - Bugs conocidos
   - Lo que necesites para entender cómo funciona tu software

https://www.snapdeal.com/product/futaba-japanese-sakura-bonsai-seeds/673767715909


¿Preguntas?
[NO]
[SI]
¡Gracias!
Intentaré responder

Análisis estático de `código fuente`

Presenta: Daniel Barajas González

Septiembre de 2018

ldanielbg@comunidad.unam.mx