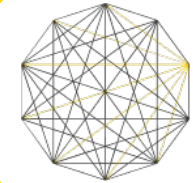


V&V

# Mejora III

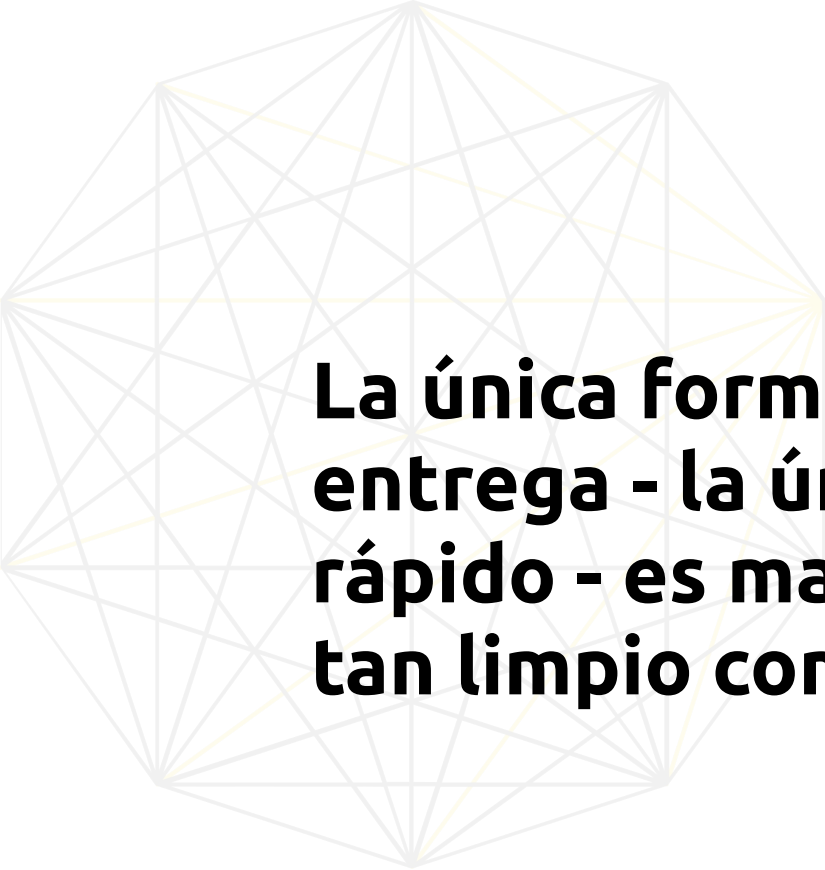
## Clean code



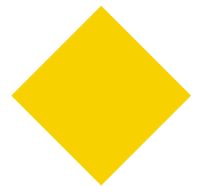
540  
quinientoscuarenta

¿Qué es "clean code"?

# Clean Code



**La única forma de cumplir la entrega - la única forma de ir rápido - es mantener el código tan limpio como sea posible**



# Clean code

- Código que se lee como si fuera una historia (como prosa).
- Código que se nota que ha sido escrito con cariño.
- Código simple:
  - Pasa los tests
  - No tiene duplicidad
  - Expresa las ideas
  - Minimiza el código

# Clean code

- El ratio de lectura respecto al de escritura es 10:1
- Constantemente leemos código como parte del esfuerzo de escribir nuevo código



**Para ir rápido y para que el  
código sea rápido de escribir,  
tiene que ser rápido de leer.**





**Regla del boy scout:**  
***Leave the campground cleaner than you found it***





**Bad smell**





# Bad smell

- Es algo en el código que no huele bien
- No tienen por qué ser un problema per se, pero son indicadores de que algo no va bien.
- Encontrarlos es algo a trabajar, code-sense.

# Nombres con sentido

Un nombre tiene que tener sentido en sí mismo. Debe ser directo y representar bien lo que significa, aunque sea un nombre largo.

Si necesita un comentario no es lo suficientemente bueno.

MAL: `int s; //size of the file`

BIEN: `int sizeofTheFile; int fileSizeInBytes`

# No desinformar

Escribir conceptos similares de forma similar es información. Utilizar nombres incoherentes es desinformación.

MAL: `hp; // hypotenuse.`

BIEN: `hypotenuse;`

MAL: `userList;`

BIEN: `users or userGroup;`

# Realizar distinciones con sentido

Distinguir nombres de tal manera que el lector sepa qué los diferencia. Evitar palabras que añaden ruido o que son redundantes (variable, table, string, boolean).

Product, ProductInfo, ProductData

MAL: customerObject, nameString

BIEN: customer, name

## Usar nombres que se puedan pronunciar, recordar, discutir

MAL: genymdhms, modymdhms

BIEN: generationTimestamp,  
modificationTimestamp

# Usar nombres que se puedan buscar

MAL:

```
for (int j=0; j<34; j++) {  
    s += (t[j]*4)/5;  
}
```

BIEN:

```
int realDaysPerIdealDay = 4;  
const int WORK_DAYS_PER_WEEK = 5;  
int sum = 0;  
for (int j=0; j < NUMBER_OF_TASKS; j++) {  
    int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;  
    int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);  
    sum += realTaskWeeks;  
}
```

# Evitar codificaciones

- No usar notación húngara: hacer que el nombre de una variable comience por una o más letras minúsculas que indiquen el tipo de dato de la variable.
- No utilizar prefijos
- No añadir "I" en las interfaces

# Nombres de clases

Utilizar nombres, no verbos

MAL: Manager, Processor, Data, Info



# Nombres de métodos

Utilizar verbos, no nombres

postPayment, deletePayment, save

get, set, is

# Una palabra por concepto

fetch, get, find, retrieve -> ¡Elige una!

# No usar la misma palabra para dos propósitos diferentes

Consistencia

add

insert

# El lector no tiene que traducir el nombre al concepto que conoce

Lenguaje ubicuo

- Nombres en el dominio del problema.
- Nombres en el dominio de la solución.



# Niveles de abstracción





**“Leer código de arriba a abajo:  
La regla descendente”.**





# Funciones



# Funciones

- Funciones cuanto más pequeñas mejor
- Hacer una sola cosa, hacerla bien, hacer solo eso.
- Extraer funciones hasta que el nombre de la función sea la descripción de la implementación.
- Código que se lee como una narrativa arriba-abajo: cada vez que bajamos leyendo las funciones, deberíamos descender un nivel de abstracción.



# Funciones

## Nombres descriptivos

- Cambiarlos tantas veces como haga falta.
- No importa que sean largos.
- Que exprese lo que hace.
- Son verbos

# Funciones

## Argumentos

- Cuantos menos, mejor.
- Dificultan el entendimiento.
- Evitar argumentos de salida.
- Evitar argumentos flag: la función entonces hace dos cosas.
- Los argumentos son nombres. Tratar que la función explique los argumentos.

# Funciones

Evitar efectos colaterales

- Cuidado con nombres que no transmiten el efecto.

# Funciones

Varios returns por función

# Switchs

- Evitarlos
- Tienen más de una responsabilidad, más de un motivo de cambio.
- Están cerrados a la extensión y abiertos a la modificación.



**Comentarios**  
**“Don’t comment bad code - Rewrite it”**



# Comentarios

- Compensan el fracaso al expresarnos mediante código
- Antes de escribir un comentario ver si es posible expresarlo de cualquier otra manera mediante código.
- Engañan. No siempre, no voluntariamente, pero muchas veces es así.
- Nos engañamos pensando que son mantenibles.

# Buenos comentarios

- Legales
- Informativos
- Expresar intención
- Clarificación
- Avisar de las consecuencias
- TODOs
- Amplificación
- Javadocs (API pública)



# Malos comentarios

- Balbucear
- Redundantes
- Confuso/engañoso
- Obligatorio
- Diario
- Ruido
- Marcadores
- Llaves de cierre
- Atribuciones
- Código comentado
- HTML
- Información no local
- Demasiada información
- Conexión no obvia
- Cabeceras de función
- Javadoc (API privada)

# Comentarios

¡EVITARLOS!

Solo si no hay otra forma o para explicar decisiones importantes.



**Formato**



## INTERNACIONAL

TITULAR

**Asesinada en Pakistán una líder del partido político de Imran Khan**

subtítulo

**Zahra Shahid Hussain ha muerto este sábado tras ser emboscada por dos hombres**

Entradilla

Una fundadora del partido Tehreek I Insaf de Imran Khan, la tercera fuerza más votada en las elecciones legislativas paquistaníes de la semana pasada, ha muerto a tiros este sábado en Karachi a pocas horas de la repetición de los comicios en esta localidad paquistaní por las irregularidades de la primera ronda.

Zahra Shahid Hussain, vicepresidenta del partido para la región central, ha muerto esta tarde tras ser emboscada por dos hombres en motocicleta que la aguardaban a la salida de su domicilio.

"Abrieron fuego sobre ella tan pronto como llegó a la puerta de su domicilio", declaró una fuente policial. Poco después, el superintendente Nasir Aftab indicó que se trató de un robo fallido. "Ella murió cuando se resistía a un atraco a mano armada", dijo el oficial.

Un testigo indicó que los ladrones abrieron fuego sobre ella después de llevarse sus pertenencias. Recibió dos disparos en la cabeza a quemarropa, de acuerdo con fuentes médicas de un hospital privado donde fue ingresada. "Una bala entró por su barbilla, subió hasta la cabeza

y salió por la nuca", indicaron estas fuentes a la cadena GEO TV.

Imran Khan, excapitán del combinado nacional de cricket, condenó el asesinato y exigió justicia inmediata a través de su cuenta de Twitter, e hizo responsable del ataque al líder del Movimiento Muttahida Qaumi, Altaf Husain, quien hace unos días amenazó a los responsables del PTI por organizar sentadas contra el fraude electoral en Karachi. Husain, no obstante, también ha condenado el atentado.

**cuerpo de la noticia**

Ejemplo partes de una noticia

# Formato

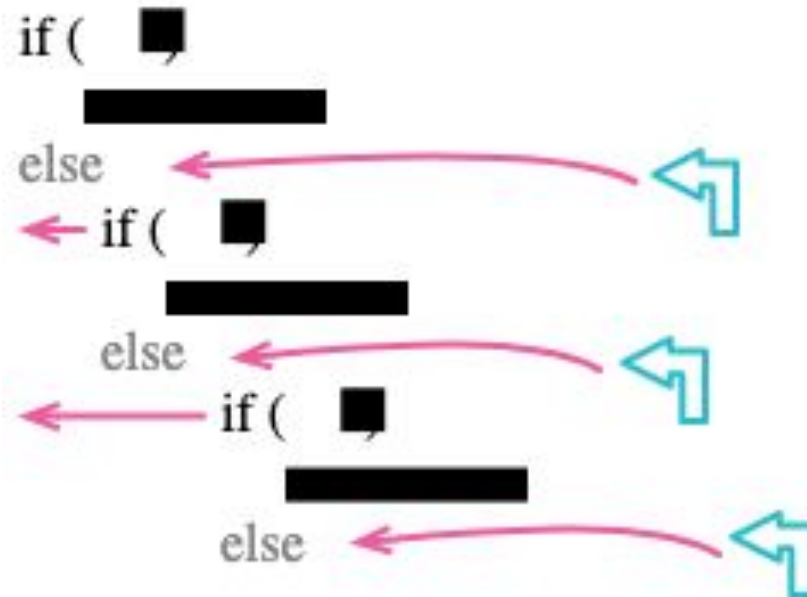
- **Formato vertical.**
  - Tamaño.
  - Distancia: conceptos relacionados deben estar tan juntos como sea posible (variables protected, funciones, variables).
  - Apertura: separación de conceptos.
  - Densidad: asociación de conceptos.
- **Formato horizontal:**
  - Tamaño
  - Distancia: juntar cosas relacionadas (paréntesis de una llamada), separar no relacionadas (asignaciones, estructuras de control).
  - Identación



**Cláusulas de guarda**



# Cláusulas de guarda



# Cláusulas de guarda

```
function getPayAmount() {  
  let result;  
  if (isDead)  
    result = deadAmount();  
  else {  
    if (isSeparated)  
      result = separatedAmount();  
    else {  
      if (isRetired)  
        result = retiredAmount();  
      else  
        result = normalPayAmount();  
    }  
  }  
  return result;  
}
```



# Cláusulas de guarda

```
function getPayAmount() {  
  if (isDead) return deadAmount();  
  if (isSeparated) return separatedAmount();  
  if (isRetired) return retiredAmount();  
  return normalPayAmount();  
}
```



**Pensar en verde**





**Baby steps**



# Tennis Refactoring Kata

- <https://github.com/540/tennis-refactoring-kata-php>
- Tennis game 1.
- Entender el código.
- ¿Veis la necesidad o posibilidad de mejorarlo?
- Trabajar en parejas.

# **Revisión y puesta en común de problemas**



# Problemas detectados



# Pair programming



# Pair programming

- Dos personas trabajando en un mismo ordenador.
- Asigna funciones y responsabilidades específicas a cada miembro de la pareja.
- Define estrategias para cambiar de posición.
- Promueve descansos para evitar que las parejas se agoten.



# Pair programming

## ¿Qué aporta?

- El software es más fácil de cambiar cuando lo estamos escribiendo. Si hacemos pair programming, podemos discutir las decisiones en tiempo real con otra persona.
- Acorta el ciclo de feedback a unos pocos segundos. Cuanto más rápido es el feedback, más fácil es converger en una solución de calidad.
- Aumenta el conocimiento compartido en el equipo.
- Mejora la satisfacción del equipo al tener un resultado consensuado.
- Mejora la comunicación al promover la argumentación de los diferentes puntos de vista.
- Mejora las habilidades del equipo al aprender de las diferentes formas de hacer.

# Pair programming

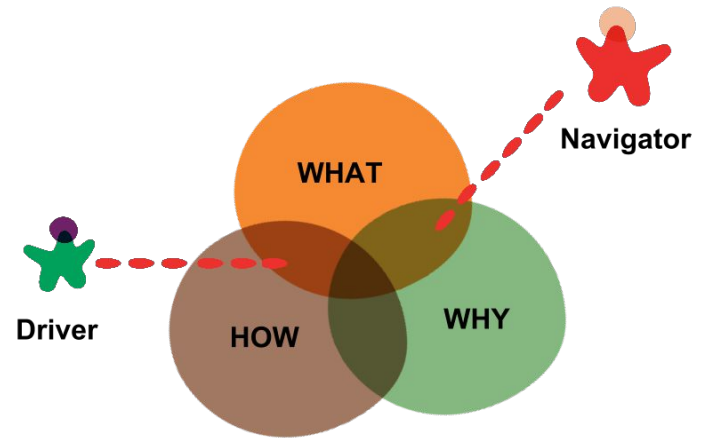
## ¿Merece la pena hacer pair programming?

- Valorar la complejidad de la tarea.
- 15% más de tiempo al desarrollar nuevas funcionalidades.
- 15% menos de bugs.

# Pair programming

## Estilos

- Driver and Navigator.
- Ping Pong.
- Strong-Style pairing
- Pair development



# **Iteración I: Mejorar el código como creáis**



# **Iteración I: Puesta en común**

# **Iteración II: A volver a mejorar el código con tests**



# **Iteración II: Puesta en común**



**DRY**





# Don't Repeat Yourself

- Cada pieza de conocimiento debe tener una única representación dentro de un sistema
- Si se aplica correctamente, una modificación en un elemento del sistema no requerirá un cambio en otros elementos no relacionados lógicamente -> Ortogonalidad



**KISS**



# Keep It Simple Stupid

- Mantener la simplicidad como un requisito u objetivo del diseño, es decir, no complicar la solución de forma innecesaria.
- Manteniendo las cosas sencillas, reducimos el tiempo de desarrollo y facilitamos su mantenimiento.



**Less surprise principle**



# Less surprise principle

- Una función, clase o variable debería hacer la cosa más obvia que se espera por su nombre.
- También conocido como Principle of least astonishment.



**Ley de Demeter**



# Ley de demeter

- Habla solo con tus amigos cercanos. No hables con extraños
- Evitar hacer: `ObjetoA.metodoDeA().metodoDeB().metodoDeC();`
- ¿Qué pasa si cambia la implementación de C?
- Beneficios:
  - Se reducen las dependencias entre clases y el acoplamiento.
  - Se vuelve más sencillo reutilizar las clases.
  - El código es más fácil de probar.
  - El código es más mantenible, más flexible a los cambios



**Tell don't ask**





# Tell don't ask

- No usar los objetos para pedirles cosas y según la información que nos devuelven tomar decisiones.
- Pedir a los objetos que hagan cosas y estos objetos internamente tomarán sus propias decisiones según su estado y reglas.
- Los objetos sólo toman decisiones sobre sus datos internos o sobre los que reciben como parámetros.
- Permite el principio Open/Closed por el cual un sistema debe estar abierto a ampliaciones y cerrado a modificaciones.



**SRP**



# Single Responsibility Principle

- Una clase debería tener una única razón para cambiar
- Nos lleva a separar los comportamientos basándonos en los motivos del cambio
- Reúne las cosas que cambian por las mismas razones. Separa las cosas que cambian por diferentes razones.
- Ayuda a evitar diseños frágiles que luego producen errores y problemas inesperados cuando se realizan cambios en el código



# **Cohesión y acoplamiento**



# Cohesión y acoplamiento

- Son dos conceptos distintos de gran importancia en Clean code.
- Son conceptos diferentes pero al hablar de un concepto normalmente acabamos hablando del otro.
- Están íntimamente ligados con el diseño del software y con la programación orientada a objetos.

# Cohesión

- La cohesión nos indica el grado de relación existente entre los distintos elementos de una clase.
- La cohesión será alta cuando:
  - Los métodos de una clase estén relacionados entre sí mediante llamadas a métodos.
  - Cuando se usan variables de ámbito interno de la clase.
- Cuanto mayor sea la cohesión, mejor.
- Mayor cohesión suele significar una única responsabilidad.

# Acoplamiento

- El acoplamiento hace referencia al nivel de dependencia con elementos externos.
- Cuantos más elementos externos usemos, mayor el acoplamiento.
- Cuanto menos acoplamiento mejor.

# Alta cohesión y bajo acoplamiento, beneficios

## Alta cohesión

- Mejora la mantenibilidad del código.
- Mejora lectura y comprensión.

## Acoplamiento bajo

- Mejora la reutilización.
- Mejora mantenibilidad.
- Mayor facilidad de comprensión.



# Recursos

- Pairing
  - <https://martinfowler.com/articles/on-pair-programming.html#Strong-stylePairing>
- Bad smell:
  - <https://refactoring.guru/es/refactoring/smells>
  - <https://codely.tv/blog/screencasts/code-smells-refactoring/>
  - <https://martinfowler.com/bliki/CodeSmell.html>
- Deuda técnica:
  - [https://www.youtube.com/watch?v=pqeJFYwnkJE&ab\\_channel=WardCunningham](https://www.youtube.com/watch?v=pqeJFYwnkJE&ab_channel=WardCunningham)
  - <https://martinfowler.com/bliki/TechnicalDebt.html>
  - <https://www.javiergarzas.com/2012/11/deuda-tecnica-2.html>
- DRY
  - <https://medium.com/@lalimijoro/the-dry-principle-and-why-you-should-use-it-f02435ae9449>
  - <https://www.kroatoan.es/articulos/principio-software-dry>
- Lenguaje ubicuo
  - <https://martinfowler.com/bliki/UbiquitousLanguage.html>
  - <https://medium.com/@felipefreitasbatista/developing-the-ubiquitous-language-1382b720bb8c>
-

# Recursos

- Ley de demeter:
  - <https://betterprogramming.pub/demeters-law-don-t-talk-to-strangers-87bb4af11694>
  - <https://www.javiergarzas.com/2014/05/beneficios-ley-de-demeter.html>
  - <https://mpijierro.medium.com/ley-de-demeter-en-la-programaci%C3%B3n-orientada-a-objetos-231106cb89a9>
- SRP
  - [http://en.wikipedia.org/wiki/Single\\_responsibility\\_principle](http://en.wikipedia.org/wiki/Single_responsibility_principle)
  - <https://mpijierro.medium.com/principios-solid-principio-de-responsabilidad-%C3%BAnica-13eb4d5537c1>
- Open/Close
  - [http://en.wikipedia.org/wiki/Open/closed\\_principle](http://en.wikipedia.org/wiki/Open/closed_principle)
  - <http://www.objectmentor.com/resources/articles/ocp.pdf>
- Feature envy:
  - <https://refactoring.guru/smells/feature-envy>
  - <https://moderatemisbehaviour.github.io/clean-code-smells-and-heuristics/general/g14-feature-envy.html>
- Less surprise principle
  - <https://www.code4it.dev/cleancodetips/principle-of-least-surprise>
  - <https://bognov.tech/software-engineering-oop-principles-and-good-practices-to-avoid-spaghetti-code#heading-principle-of-least-astonishment>

# Recursos

- Tell don't ask
  - <https://www.carlosble.com/2013/09/push-the-query-down-tell-dont-ask/>
  - <https://martinfowler.com/bliki/TellDontAsk.html>
- Cohesión y acoplamiento:
  - <https://www.baeldung.com/cs/cohesion-vs-coupling>
  - <https://blog.ahierro.es/cohesion-y-acoplamiento/>

**Iteración III:  
Última oportunidad  
de hacer código  
crema**



# **Iteración III: Puesta en común**



<https://540deg.com/>

@540deg