

fiuba

algo3

# **Buenas prácticas de programación**

**(Código de calidad y prácticas  
de XP)**

Pablo Suárez  
psuarez@fi.uba.ar

# Buenas prácticas de programación



# ¿Verdadero o Falso?

- A. Siempre conviene que los programas se ejecuten lo más rápido posible
- B. La razón principal para no repetir código es el costo de escribir lo mismo dos veces
- C. El código se lee muchas más veces que las que se escribe
- D. Antes de agregar un comentario para aclarar, debería tratar de aclarar el código
- E. Al programar una clase, conviene prever todo lo que se vaya a necesitar, para incluirlo desde el principio



# Antes de empezar

- “Los necios obedecen las reglas; los sensatos, las usan como guía” (Douglas Bader)
- “El código es el único artefacto del desarrollo de software que siempre se va a construir” (Carlos Fontela)
- “I’m not a great programmer; I’m just a good programmer with great habits” (Kent Beck)



# Temario

Legibilidad

Guías para mejora de desempeño

Buenas prácticas de XP

# Legibilidad

Hecho:  
Dijeron 2 de los 2h  
20.000 flexaciones  
cada 12h x 10 días  
Beck

# Legibilidad: por qué

El código se escribe una vez y se lee muchas

Escribir código para humanos

En un texto usamos:

Títulos de distinto nivel

Sangrías

Signos de puntuación

Tipografías especiales

¡Y se aprende practicando!

=> ¿Por qué no hacer el código legible?



# Mentalidad que atrasa 60 años

“Now listen. We are paying \$600 an hour for this computer and \$2 an hour for you, and I want you to act accordingly.”

Según Barry Boehm, lo que su jefe le dijo en su primer trabajo (1950s)





# Principio Nº 1: no repetir

¿Se entiende por qué?

Extraer métodos

Extraer clases, por delegación o  
por generalización



# Principio Nº 2: estandarizar



Nombres

Formatos: líneas en blanco, sangrías, etc.

Comentarios

=> Código autodocumentado

# Principio Nº 3: no sorprender

Suponer igual nivel del desarrollador que va a necesitar mi código

Evitar lucirse

No explicar lo obvio



# Nombres (1)

## Descriptivos

lineasPorPagina es mejor que lpp

distanciaEnMetros es mejor que distancia

Poner antónimos en forma consistente: mayor, menor / min, max

No usar números: total1, total2

## Términos del problema, no de la solución

empleados en vez de arregloEmpleados

## Casos especiales (¿?)

Salvo en nombres muy largos: num, cant, long, max, min

i, j, k, para índices

tmp, temp, para temporales y auxiliares

# Nombres (2)

Evitar significados ocultos

“long” es la longitud del documento, o -1 si no se encontró

Legibles en algún idioma

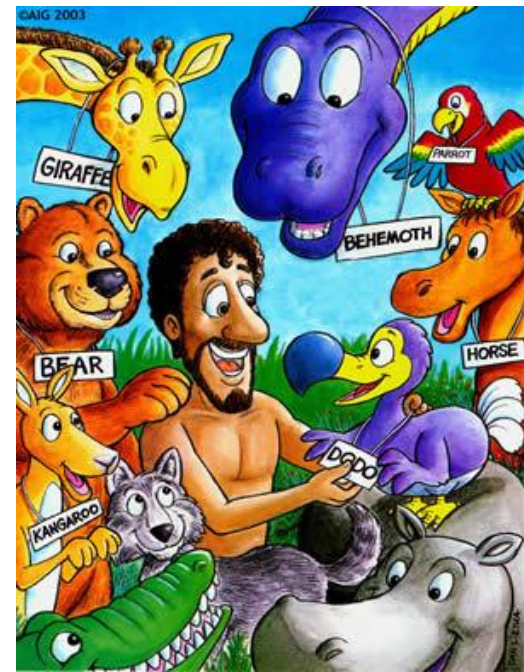
Variables booleanas

encontrado, terminado, error

No expresarlas en forma negativa

Que se lean como verdadera o falsa:

“sexo” no es buen nombre



# Nombres de métodos

Que describa todo lo que hace el método

Que describa la intención (el qué) y no detalles de implementación (el cómo)

empleados **borrar**: dni

Malos sucedáneos: procesarElemento,  
buscarDniEmpleadoBorrar, buscEmBor, beb,  
buscarBorrar, findEmpleadoByDniDelete

# Métodos y legibilidad

Tratar de que no dependan de un orden de ejecución

Si no, documentarlo adecuadamente

Recursividad sólo cuando ayuda a la legibilidad

Y sólo al nivel de un método

No:  $A \rightarrow B \rightarrow A$

Que afecte a la clase en la que está declarado

Si es así, suele tener el nombre de otra clase en su nombre

Si no, moverlo de clase



# Parámetros

Que no sean muchos

Ninguno es lo mejor

Que estén ordenados con algún criterio

Mantener el criterio en diferentes métodos

Chequear valores válidos a la entrada

Los valores de entrada no deben cambiarse

# Variables y métodos locales

Inicialización: siempre explícita

Un uso para cada variable

Vida de la variable lo más corta posible

Usar métodos booleanos para guardar partes de tests complicados

```
if ( (anio % 400 == 0) ||  
    ((anio % 4 == 0) && (anio % 100 != 0) ) ...
```

# Números mágicos

( 1 to: 14) do: ... “ 14 clientes”

(usuario id = 1234) ifTrue: [...]

Evitar todo “hardcodeo” o “números mágicos”

¿Por qué?

Salvo algunos 0, 1, “”, nil, y cosas que no cambian



# Sangrías y espacios

No tener miedo de usar espacios

```
(a=b)ifTrue:[Transcript show:'OK']
```

```
(a = b) ifTrue: [ Transcript show: 'OK' ]
```

Usar líneas en blanco

Usar paréntesis sobreabundantes para aclarar

Usar layouts del lenguaje



# Comentarios (1)

Son buenos para aclarar código

Aunque deberíamos tratar de aclarar el código antes

O como resumen de una secuencia de acciones

Aunque deberíamos tratar de separar en un método

No repetir en los comentarios lo que dice el código

“ asigno 1 a x: ”

$x := 1.$

Corolario: no deberían abundar

# Comentarios (2)

Que los comentarios sean fáciles de mantener

El comentario debe ir antes del código al que se refieren

Preparan mejor al lector

A veces conviene ponerlos al final de la línea

Casos especiales a documentar con comentarios

Efectos laterales

Restricciones de un método o clase

Nombres de algoritmos, fuentes (de dónde lo saqué)

# Condicionales y ciclos

Cuidar los  $\geq$ ,  $\leq$ ,  $>$ ,  $<$

Ojo con anidaciones profundas

No más de 3 niveles

Ordenar los “case” de los “switch”

Orden lógico: alfabético, numérico

Hacer obvio al lector la manera de salir del ciclo

Usar salidas explícitas de ciclos cuando se pueda

O incluso retornos de métodos



# Tabulaciones de resultados cuando sea más legible

Días de un mes

```
public int diasMes () {  
  
    int[] diasMes = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
  
    if (esAnioBisiesto() && mes == 2)  
        return 29;  
  
    else return diasMes[mes];  
}
```

¿Es mejor que un “switch”?

Depende de cada uno

# Refactorización: para mejorar legibilidad



# Recapitulación



# Recapitulación: preguntas

- ¿Por qué no conviene ser “ingenioso” al escribir código?
- ¿A qué llamamos “números mágicos”?
- ¿Cuáles son las dos teclas que más debemos usar del teclado?
- ¿Cuál es el número ideal de parámetros de un método?



# Desempeño



# Mejora de desempeño

## Resistir la tentación

No siempre el usuario percibe las mejoras en el código, sino que privilegia su propia experiencia

¿Mejorar el hardware?

## Cuando todo falle...

Mejoramos el código

Pero sólo en las partes críticas

(recordar a Pareto: 80-20)

Usar “profilers” si los tenemos





# Algunas posibilidades (1)

Evitar el uso de memoria externa cuando se puede usar memoria interna

Accesos innecesarios a disco

En las evaluaciones lógicas, utilizar la opción de cortocircuito

En un “(a & b) ifTrue:” no necesitamos evaluar b si a es falso

En un “(a | b) ifTrue:” no necesitamos evaluar b si a es verdadero

Verificar si el entorno no lo hace solo



# Algunas posibilidades (2)

Ordenar las preguntas en acciones *if* compuestas y *switch*

Conviene evaluar primero las más frecuentes y menos costosas de evaluar

```
switch (mes) {  
    case 1, 3, 5, 7, 8, 10, 12 : dias = 31; break;  
    case 4, 6, 9, 11 : dias = 30; break;  
    case 2    if (anio.bisiesto())  
                dias = 29;  
                else dias = 28;  
                break;  
    default: throw new MesInvalidoException();  
}
```

# Algunas posibilidades (3)

Calcular todo lo que se pueda antes de entrar a un ciclo

```
// mal ejemplo:
```

```
f1 leerFecha.
```

```
f2 leerFecha.
```

```
100 timesRepeat: [
```

```
    periodo f1 diasHasta: f2.
```

```
    periodoDoble := periodo * 2.
```

```
    f3 := Fecha hoy.
```

```
    i := i+1;
```

```
    Transcript show: i * periodo + periodoDoble ]
```

Usar tipos por valor siempre que se pueda

# Algunas posibilidades (4)

Usar tipos enteros en vez de punto flotante

int mejor que double

Tabular datos en vez de usar funciones trascendentes

Salvo que se requiera mucha precisión

En algunos casos es sencillo

Liberar memoria que ya no se necesita, si el entorno no lo hace solo

Usar lenguajes compilados o de menor nivel

Assembler >> C++ >> Java / C# / Smalltalk >> PHP / Python

Si la modularización es buena, podemos acotarlo

# Después de la mejora

Probar su efectividad

Muchas presuntas mejoras no resultan, porque los compiladores, entornos de ejecución y otros, las introducen en forma automática

No hay expertos generalistas

Hay cosas que han cambiado con el tiempo

P. ej., llamadas a funciones, arreglos multidimensionales

# Dichos

“Premature optimization is the root of all evil”  
(Donald Knuth)

“

1. Make it work.
  2. Make it right.
  3. Make it fast.”
- (Kent Beck)



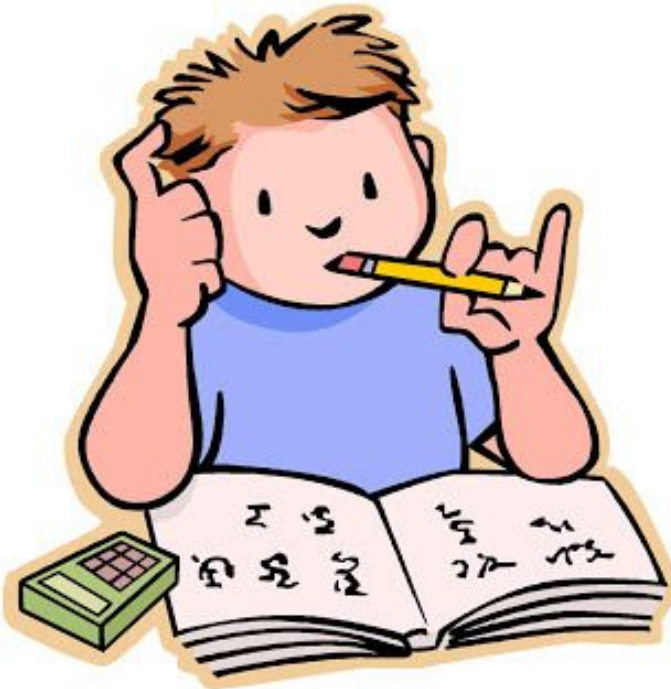
# Recapitulación



# Recapitulación: preguntas

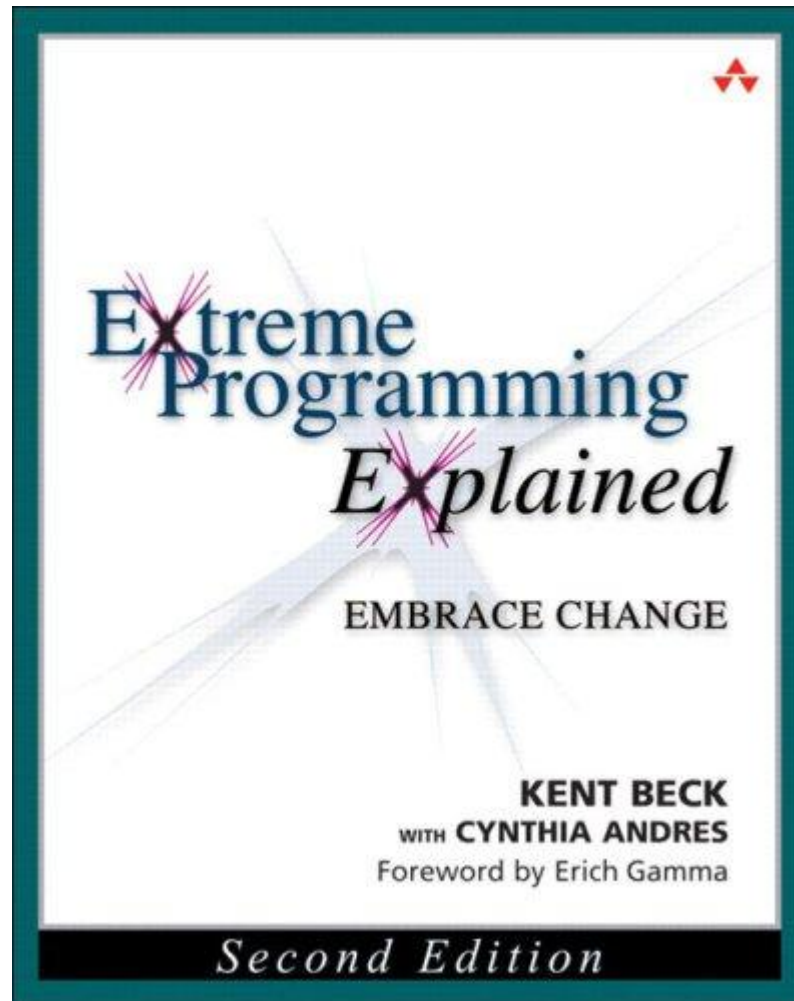
¿Por qué no conviene optimizar todo desde el comienzo?

¿Cómo sabemos que una optimización dio sus frutos?

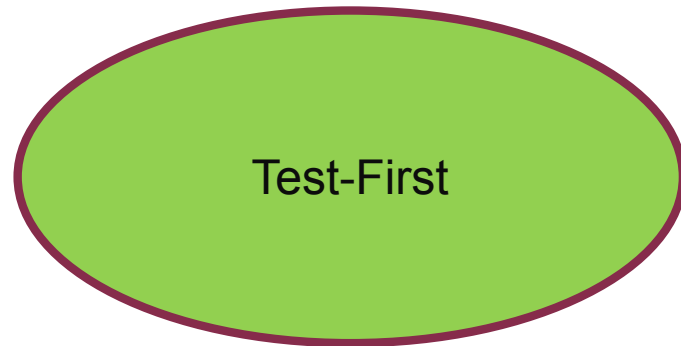




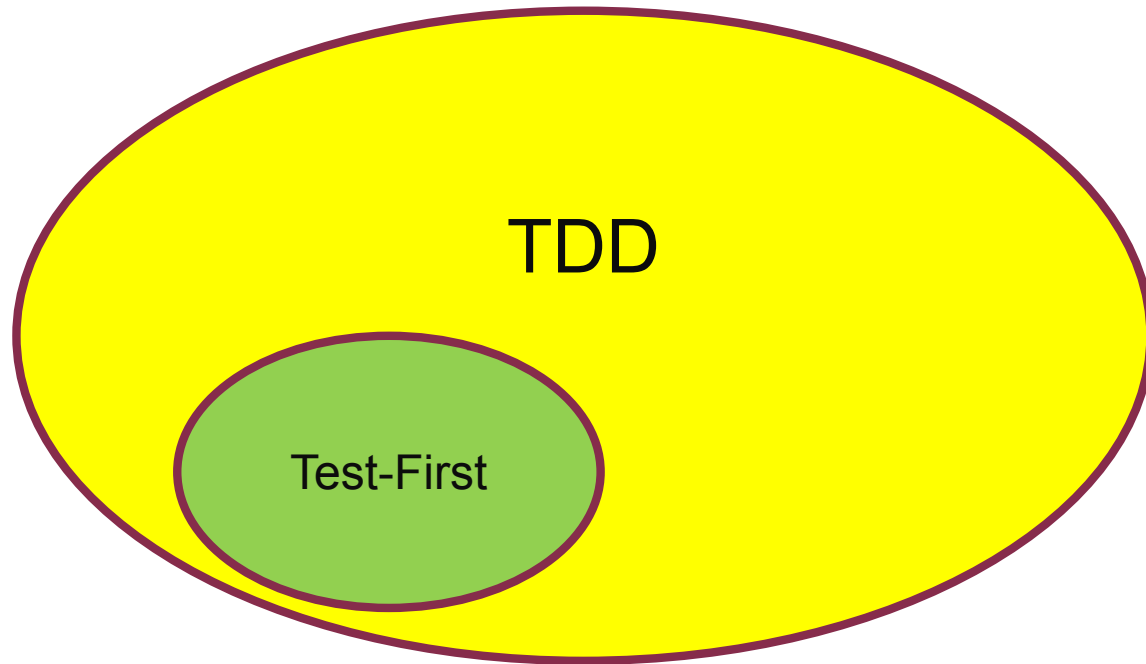
# Extreme Programming y sus recomendaciones



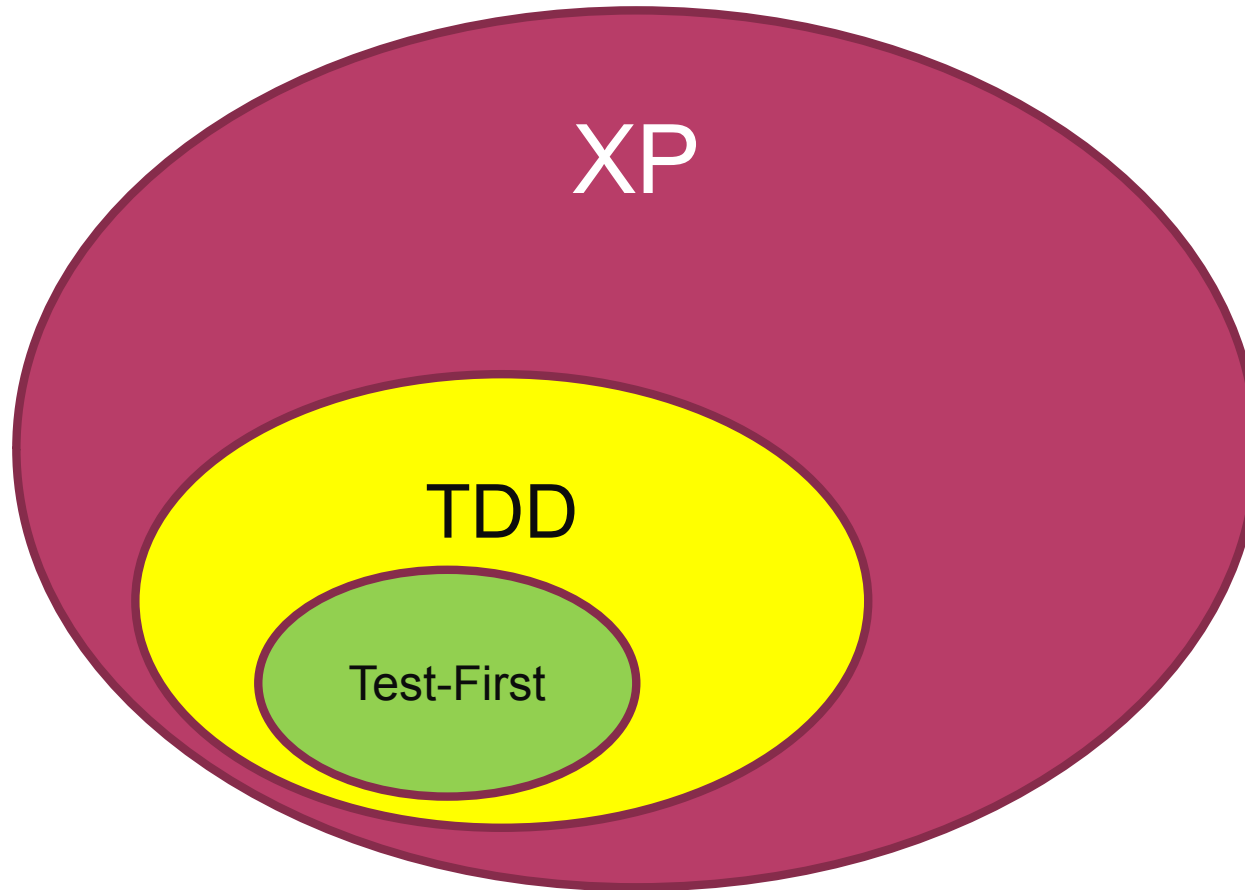
# Construcción metodológica incremental



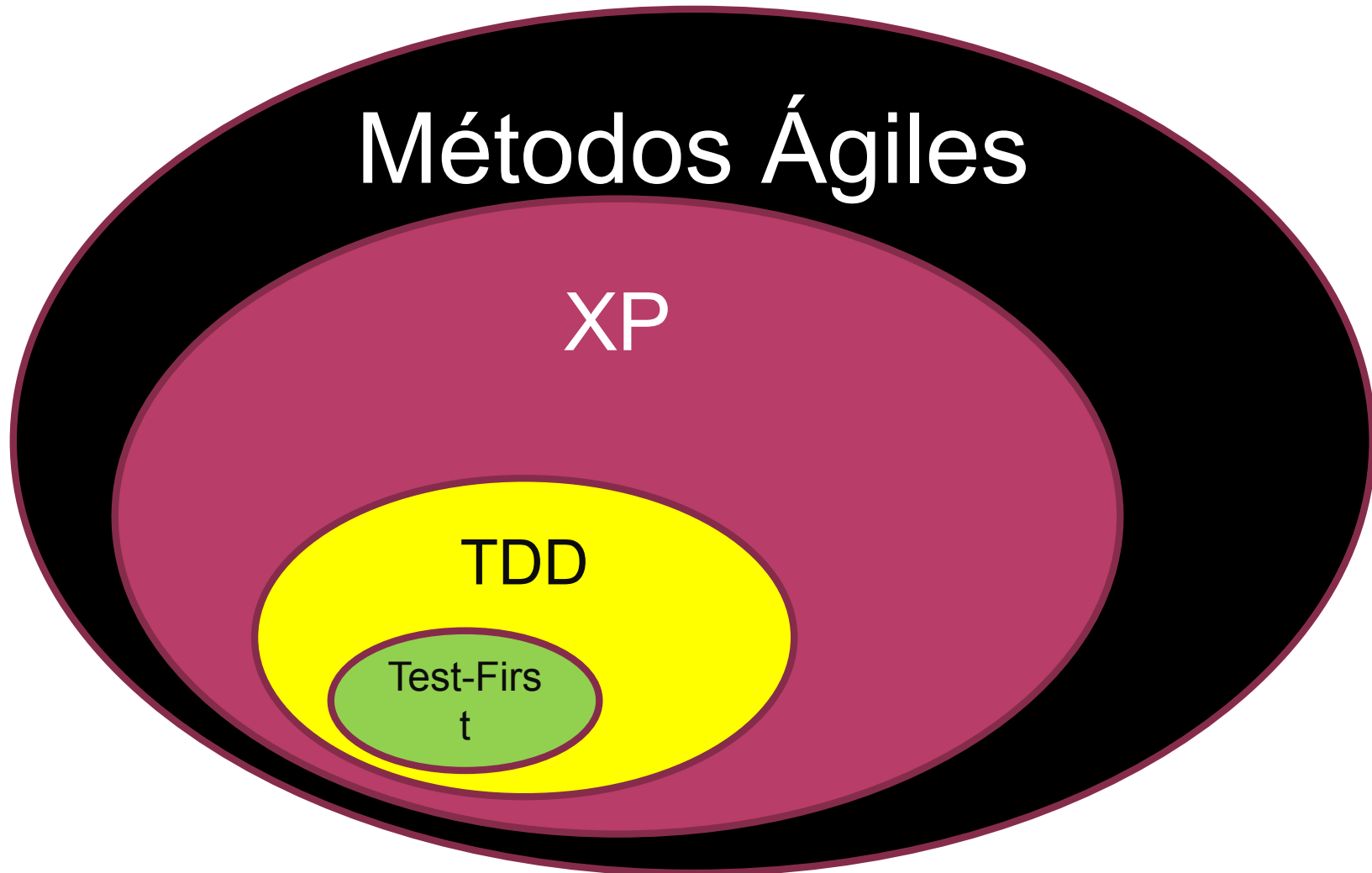
# Construcción metodológica incremental



# Construcción metodológica incremental



# Construcción metodológica incremental



# XP

“Extreme Programming” o “Programación Extrema”

Conjunto de prácticas de desarrollo centradas en la programación

Uno de los “métodos ágiles”

Más adelante volvemos...

“Lleva al extremo las buenas prácticas de programación y cuestiones de sentido común”

Veremos sólo un poco

Resto en

Técnicas de Diseño o  
MeMo Ing Soft 2



# Buenas prácticas XP (1)

¿Probar el código es bueno?

- Hacer pruebas todo el tiempo

- El desarrollo es dirigido por las pruebas: Test-Driven Development (TDD) y afines (BDD, ATDD, ...)

- Se diseñan antes de codificar

- Ojo: mucha disciplina

¿Y las pruebas de integración?

- Integración continua como práctica

- Evita que sea cada vez más complicado integrar código de varias fuentes

# Buenas prácticas XP (2)

¿Los diseños de software son cambiantes?

El diseño evoluciona junto con la programación

Lo hacen los propios programadores

Minimizar documentación que se guarda, si no se la va a mantener actualizada

¿Revisar la calidad del código es recomendable?

Revisar código todo el tiempo

Pair-programming

Se mantiene más el foco

Refactorizaciones



# Buenas prácticas XP (3)

¿Estándares de codificación permiten una mejor comunicación y reducen errores?

Fijar estándares precisos y estrictos

¿El desarrollo incremental es positivo?

Hacer micro-iteraciones

Diseñar una pequeña porción, codificarla y probarla

Preocuparse sólo por lo que se está haciendo

Nada por adelantado

Recordar: el cliente pide (requerimientos), no necesariamente lo que quiere (expectativas), ni mucho menos lo que necesita (necesidades)

# Buenas prácticas XP (4)

## Simplicidad

“The simplest thing that could possibly work”

Complejidad dificulta refactorizaciones,  
comunicación y depuraciones

No implementar lo que no se sabe si servirá,  
y en el 80% de los casos no sirve

Se mantiene baja la inversión inicial en el proyecto

El código ejecutable permanece más chico

YAGNI

La simplicidad cuesta trabajo y es fruto de un  
buen diseño

# XP y documentación

## Prioridades

1. Código autodocumentado
2. Embebida en el código (comentarios)
3. Pruebas unitarias
4. Otras pruebas automatizadas

## Menor valoración a

UML

Documentos externos

... aunque no se descartan

# Resumen prácticas XP

- TDD
- Integración continua
- Refactoring

• Pair programming



# Veamos...



fiuba  
algo3

# Claves: los mandamientos del alumno de Algoritmos III (1ra parte)

No repetirás código

No dejarás a tus semejantes lo que no quieres que te dejen a ti

Escribirás código legible

Usarás la barra espaciadora y el salto de línea de tu teclado

“La optimización prematura es la raíz de todos los males”

Donald Knuth, The Art of Computer Programming, 1968-1973

# Lecturas obligatorias

No hay  
¡Revisar las anteriores!



fiuba  
algo3

# ¿Verdadero o Falso?



- F Siempre conviene que los programas se ejecuten lo más rápido posible
- F La razón principal para no repetir código es evitar escribir lo mismo dos veces
- v El código se lee muchas más veces que las que se escribe
- v Antes de agregar un comentario para aclarar, debería tratar de aclarar el código
- F Al programar una clase, conviene prever todo lo que se vaya a necesitar, para incluirlo desde el principio



# Lecturas opcionales

Code Complete, Steve McConnell

Capítulos 6 a 8, 10 a 13, 14 a 16, 18 a 19: buena parte del libro

No está en la Web ni en biblioteca

# Qué sigue

Primer parcial

Temas de diseño

Temas avanzados derivados de P00

