FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Código de calidad y buenas prácticas ágiles

Carlos Fontela
cfontela@fi.uba.ar

A3F

Antes de empezar

Todo es relativo…

Lleve lo que necesite…

Recordemos:

"El código es el único artefacto del desarrollo de software que siempre se va a construir" (Carlos Fontela)

```
public static void EjecutarAccion (int cant, Fecha fecha, ArrayList v, ArrayList w) {
      for (int j = 0; j < v.size(); j++) w.add(v.get(i));
      int j = 0;
      while (j < w.size()) {
        System.out.println(w.get(j)); i++; }
      Iterator i = v.iterator();
      while (i.hasNext()) {
        System.out.println(i.next());
      }
      if ((fecha.mes == 2) && (fecha.dia < 30)) return;
      if (((fecha.mes == 4) || (fecha.mes == 6) || (fecha.mes == 9) || (fecha.mes == 11)
        && (fecha.dia < 31)) return;
      if (fecha.dia < 31) return;
      mensaje();
      throw new RuntimeException();
}
public static void mensaje() {
      System.out.println ("Fecha inválida. Por favor, ingrese otra.");
      fecha.leer();
}
```

Agenda

Calidad en general

Temas de legibilidad

Guías para mejora de desempeño

Otros temas

Buenas prácticas ágiles

Calidad

Calidad del software

No es algo objetivo

¿Qué importa más?

- Código claro y mantenible
- Desempeño (performance) de la aplicación
- Performance del usuario
- Portabilidad a distintos ambientes

Todo depende del cliente y de la aplicación (criticidad, longevidad…)

Principio Nº 1: no repetir

Principio básico: ¿se entiende por qué?

Extraer métodos: ¡privados!

Extraer clases, por delegación o por generalización

Extraer y evidenciar patrones

Principio Nº 2: estandarizar

Nombres

Formatos: líneas en blanco, sangrías, etc.

Comentarios

Arquitecturas, manejo de seguridad, herramientas

patrones de diseño

Modularización: cuándo construimos clases, paquetes y métodos

=> Código autodocumentado

Principio Nº 3: no sorprender

Suponer igual nivel del desarrollador que va a necesitar mi código

- Evitar lucirse
- No explicar lo obvio

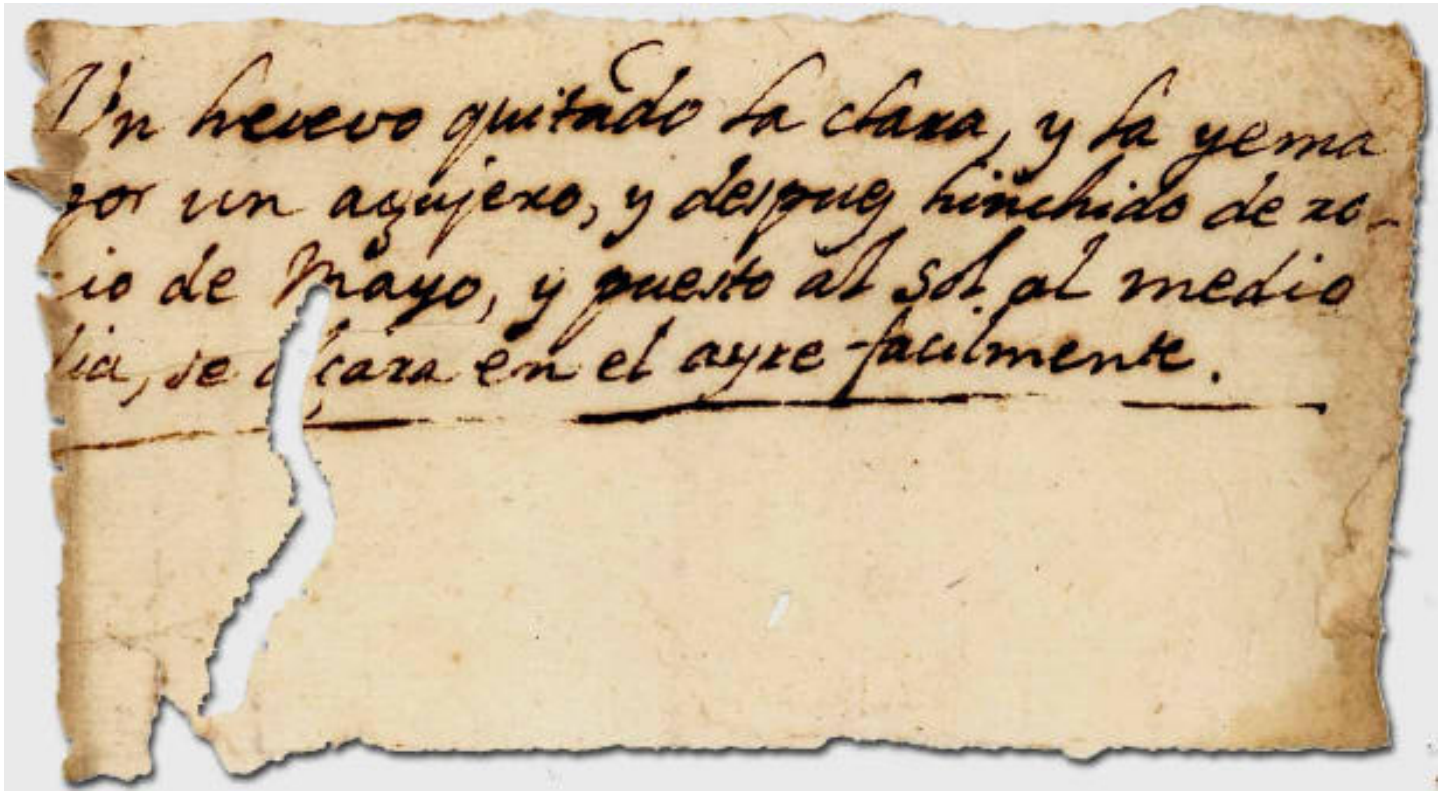
Reflexión

"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)

¿Hasta qué punto sigue siendo nuestra mentalidad?

Legibilidad



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

¿Por qué no hacer el código legible?

Nombres (1)

- Ser descriptivo
  - lineasPorPagina es mejor que lpp
  - DistanciaEnMetros es mejor que Distancia
  - Poner antónimos en forma consistente: mayor, menor / min, max
  - No usar números: total1, total2
- Hablar en términos del problema y no de la solución
  - datosEmpleado en vez de registroEmpleado
- Usar pocas abreviaturas
  - Salvo en nombres muy largos: num, cant, long, max, min
- Nombres cortos son buenos en ámbitos restringidos

Nombres (2)

Evitar significados ocultos

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

Que se puedan leer en algún idioma

Casos especiales , cuando no hay nombres mejores

- i, i, k, para índices
- Por qué no canales[numero]
- Tmp, temp, para temporales y auxiliares

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)

Mal ejemplo en Java: Arrays.binarySearch()

A lo sumo, se podría llamar busquedaRapida()

eliminarEmpleado()

Bien

Malos sucedáneos

buscarEmpleado(), procesarEmpleado() , buscarEmpleadoEliminar(), buscEmpEl(), bee(), buscarEliminar(), findEmpleadoDelete()

Nombres e interoperabilidad

Ojo con el case-sensitive

En C#, puedo escribir:

```
public int Saldo {
    get { return saldo; }
}
```

En VB.NET no puedo escribirlo

Y hasta tener problemas de interoperabilidad

No usen declaraciones implícitas y declaren todas las variables (permitidas en VB)

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 -> B -> 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

- No más de 7, y si es menos, mejor

Que estén ordenados con algún criterio

- Mantener el criterio en diferentes métodos

Chequear valores válidos a la entrada

- En las referencias, ver que no estén en null

Los valores de entrada no deben cambiarse

Variables locales

Inicialización: siempre explícita

Un uso para cada variable

Que la vida de la variable sea lo más corta posible

Usar variables booleanas para guardar partes de tests complicados

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

O métodos booleanos

Números mágicos

```
Collection c = new ArrayList(14);          // 14 clientes
for (int i = 1; i <= 12; i++) …            // meses del año
int[ ] horasTrabajadas = new int [5];      // de lunes a viernes
if (usuario == “Juan Gómez”) …
```

Evitar todo “hardcodeo” o “números mágicos”

¿Por qué?

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

Sangrías y espacios

No tener miedo de usar espacios

```
x=f(a,b,c);      ->      x = f (a, b, c);
x=v[i+j*k];      ->      x = v [i + j*k];
```

Separar al menos con una línea los métodos

Separar clases si van en el mismo archivo

Usar paréntesis y llaves sobreabundantes para aclarar

Usar layouts del lenguaje

- O al menos siendo consistentes

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
- Usar javadoc o NDoc siempre que sea posible
  - Los comentarios tienen doble utilidad
- 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 (if, switch)

- Cuidar los >=, <=, >, <
- Ojo con anidaciones profundas
  - No más de 3 niveles
- Si hay muchos if-then-else, usar switch o case
- Ordenar los “case” de los “switch”
  - Orden lógico: alfabético, numérico
- Lo que se hace en cada “case” debe ser simple
  - Si no, separar en un método
- Usar el “default” sólo para casos default, errores y valores atípicos

Ciclos (while, for)

- Usar los “for” como “for”
- Cuidar los >=, <=, >, <
- Ojo con anidaciones profundas
  - No más de 3 niveles
- Evitar ciclos vacíos, con todo el código en el encabezado
- Hacer obvio al lector la manera de salir del ciclo
- Usar “break” en vez de complicar un while con un chequeo booleano
  - O incluso “return” o una excepción puede clarificar a veces
  - Pero si hay muchos “break”, sospechar

Tabulaciones de resultados

Días de un mes

```
int [ ] diasMes = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
if (mes > 0 && mes <= 12)
    dias := diasMes[mes];
else throw new MesInvalidoException();
if (mes == 2 && anio.bisiesto() )
    dias := 29;
```

¿Es mejor que un “switch”?

Depende de cada uno

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 Paretto: 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 “if (a && b)” no necesitamos evaluar b si a es falso
  - En un “if (a || b)” no necesitamos evaluar b si a es verdadero
  - En Java y C# esto ocurre automáticamente
  - Se puede implementar con if anidados.

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:
Fecha f1 = Utilidades.ingresarFecha();
Fecha f2 = Utilidades.ingresarFecha();
for (int i = 0; i < x.size(); i++) {
    int periodo = f1.diasEntre(f2);
    int periodoDoble = periodo * 2;
    Fecha f3 = Fecha.getHoy();
    int periodo2 = f1.diasEntre(f3);
    x[i] = (periodo + periodoDoble * i + periodo2 / i)
}
```

Usar tipos por valor siempre que se pueda

int mejor que Integer

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# >> 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

Otras ideas

Warnings

Mirar los “warnings” de los compiladores

- Evita errores posteriores
- Aunque algunos son muy molestos

Referencias

Cuando ya no quiero una referencia, recordar poner el null/nil

- Facilita tareas al recolector de basura

Inicializar siempre con null/nil si no sé qué hacer

Herramientas

Analizadores de código

Profilers: buscan dónde pasa más tiempo la aplicación

Extractores de código muerto

Buscadores de código repetido

Compiladores que mejoran el desempeño

Buenas prácticas ágiles

Buenas prácticas XP (I)

¿Probar programas es bueno?

- Hacer pruebas unitarias y funcionales todo el tiempo
- El desarrollo es dirigido por las pruebas: Test-Driven Development (TDD)
- Se diseñan antes de codificar
- Ojo: mucha disciplina

¿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

Buenas prácticas XP (II)

- ¿Hacer pruebas de integración es importante?
  - Integración sigue inmediatamente a las pruebas unitarias
  - Evita que sea cada vez más complicado integrar código de varias fuentes
- ¿Revisar la calidad del código es recomendable?
  - Revisar código todo el tiempo
  - Pair-programming
    - Se mantiene más el foco
  - Refactorizaciones (cambio de diseño para hacerlo simple, sin cambiar funcionalidad)

Buenas prácticas XP (III)

- ¿Estándares de codificación permiten una mejor comunicación y reducen errores?
  - Fijar estándares precisos y estrictos
- ¿El desarrollo incremental es positivo?
  - Hacer iteraciones muy cortas
  - 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 (necesidad)

Buenas prácticas XP (IV)

¿Es importante tener una comunicación frecuente con el cliente?

- Un cliente acompañando el desarrollo
- Fundamental para escribir pruebas de aceptación (UAT) y priorizar tareas

¿Es frecuente que los proyectos se suspendan por falta de presupuesto?

- Usar el alcance como variable de ajuste
- Implementar primero lo que tiene mayor valor para el cliente
- La primera iteración debe implementar un esqueleto, con funcionalidades básicas
- En la mayoría de los sistemas, el 20% del desarrollo genera el 80% del beneficio

Buenas prácticas XP (V)

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

Las prácticas de XP se realimentan: ¿somos ágiles realmente?

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

No repetirás código

No dejarás a tu prójimo lo que no quieres que te dejen a ti

Escribirás código legible

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

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

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

Segundo parcial

Información de tipos en tiempo de ejecución

Modelos de datos y memoria