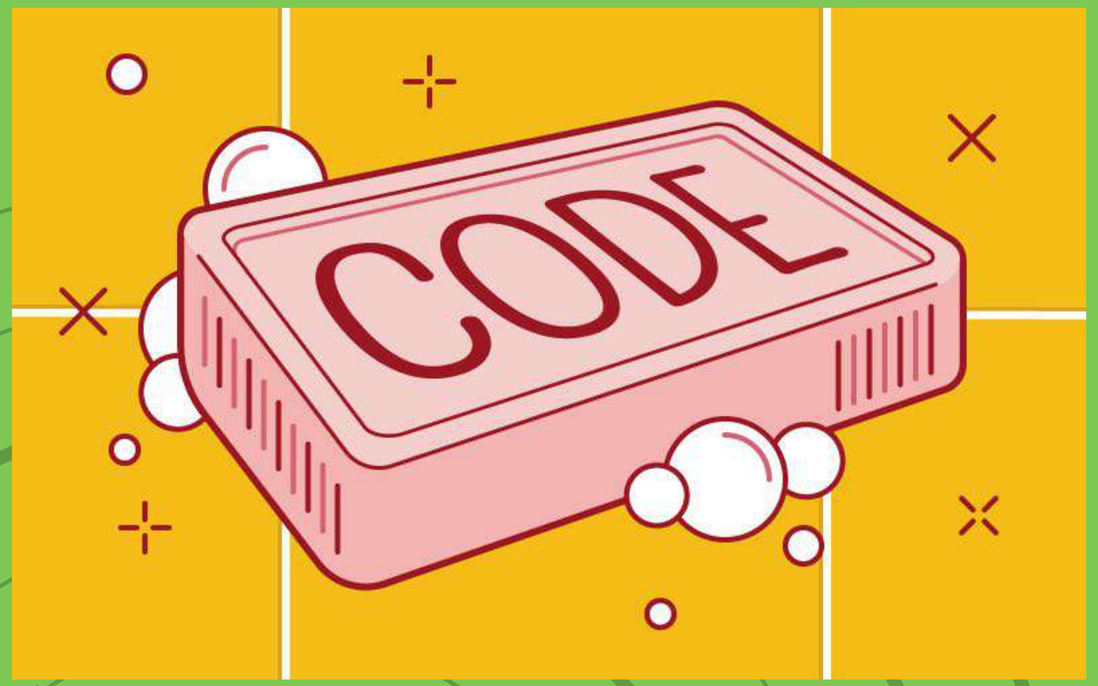
Clean Code



emmanuel.lagarrigue@cs.uns.edu.ar

Comentarios

“El único comentario bueno es el que no debería ser escrito”

- Nada mejor que un comentario bien puesto…
- Nada puede ser tan dañino como un comentario desinformante por no estar actualizado…
- **El uso de comentarios compensa la incapacidad de ser descriptivos con el código**. En este sentido, los comentarios son siempre una falla nuestra, el hecho de no poder expresarnos sin ellos.

Un mal código no se compensa con comentarios

- Una de las principales motivaciones para escribir comentarios es el código malo. Luego de escribir un módulo complejo, confuso, desorganizado, sentimos la necesidad de “mejorarlo” escribiendo un comentario que intente explicar lo que no es obvio.
- Si el código es limpio y expresivo, no hacen falta comentarios!
- En vez de perder tiempo escribiendo un comentario que intente explicar qué está pasando, invertirlo en limpiar el código

Sé expresivo

- ¿Cuál opción es más expresiva?

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

vs

```
if (employee.isEligibleForFullBenefits())
```

Buenos comentarios

- Algunos comentarios son necesarios o beneficiosos:
  - **Interfaces de módulos**: Javadocs.
  - **Comentarios Legales**: Copyright y autoría.
  - **Comentarios informativos**: A veces es necesario aclarar información básica (sólo cuando no es posible ser expresivo a través del código)

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
  "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

- **Explicación de la intención:** A veces hacen falta comentarios para explicar mejor la intención del código. Por ejemplo, casos particulares que no son obvios o pertenecen a la lógica de negocio.

```
public void testConcurrentAddWidgets() throws Exception {
  WidgetBuilder widgetBuilder =
    new WidgetBuilder(new Class[]{BoldWidget.class});

  String text = "'''bold text'''";
  ParentWidget parent =
    new BoldWidget(new MockWidgetRoot(), "'''bold text'''");
  AtomicBoolean failFlag = new AtomicBoolean();
  failFlag.set(false);

  //This is our best attempt to get a race condition
  //by creating large number of threads.
  for (int i = 0; i < 25000; i++) {
    WidgetBuilderThread widgetBuilderThread =
      new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
    Thread thread = new Thread(widgetBuilderThread);
    thread.start();
  }
  assertEquals(false, failFlag.get());
}
```

```
private fun selectAndTriggerUnSupportedCity(city: CityResult.CityLocation) =
    setSelectedCity(city)
        .andThen(
            // This is needed to position map to the unsupported location
            putMapCameraPosition.execute(city.location.toMapCameraPosition()).toCompletable())
        .andThen(
            // Save UnsupportedCitySelected flag, in order to show the proper alert.
            isUnsupportedCitySelectedRepository.setIsUnsupportedCitySelected(true))
        .andThen(Observable.fromCallable {
          unsupportedCitySelected.call(Unit)
        })
```

- **Clarificación:** A veces es bueno traducir el significado de algo. Por ejemplo, cuando se utilizan librerías externas.

```
public void testCompareTo() throws Exception
{
  WikiPagePath a = PathParser.parse("PageA");
  WikiPagePath ab = PathParser.parse("PageA.PageB");
  WikiPagePath b = PathParser.parse("PageB");
  WikiPagePath aa = PathParser.parse("PageA.PageA");
  WikiPagePath bb = PathParser.parse("PageB.PageB");
  WikiPagePath ba = PathParser.parse("PageB.PageA");

  assertTrue(a.compareTo(a) == 0);    // a == a
  assertTrue(a.compareTo(b) != 0);    // a != b
  assertTrue(ab.compareTo(ab) == 0);  // ab == ab
  assertTrue(a.compareTo(b) == -1);   // a < b
  assertTrue(aa.compareTo(ab) == -1); // aa < ab
  assertTrue(ba.compareTo(bb) == -1); // ba < bb
  assertTrue(b.compareTo(a) == 1);    // b > a
  assertTrue(ab.compareTo(aa) == 1);  // ab > aa
  assertTrue(bb.compareTo(ba) == 1);  // bb > ba
}
```

- **Advertir consecuencias:** A veces es bueno advertir a otros de posibles consecuencias.

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
   //SimpleDateFormat is not thread safe,
   //so we need to create each instance independently.
   SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM  yyyy HH:mm:ss z");
   df.setTimeZone(TimeZone.getTimeZone("GMT"));
   return df;
}
```

- **TODOs:** Los comentarios TODO a veces son necesarios. Sobre todo, cuando alguna funcionalidad no es posible de implementar por alguna dependencia por ejemplo.
- **Amplificación**: Un comentario se puede utilizar para amplificar la importancia de algo.

```
String listItemContent = match.group(3).trim();
// the trim is real important.  It removes the starting
// spaces that could cause the item to be recognized
// as another list.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

- **Javadocs en APIs públicas:** Este tipo de comentario sí es importante. Pero sólo aplican a las interfaces públicas de las apis.

Malos comentarios

- **Comentarios “porque sí”**: Escribir un comentario sólo porque nos “parece” que hace falta, o porque el proceso lo requiere es un hack. Si decidimos escribir un comentario tenemos que tomarnos el tiempo necesario para asegurarnos de que es el mejor comentario que se puede hacer. Por ejemplo, un comentario que nos obliga a estudiar el código para entenderlo.

- **Comentarios redundantes**: No sirve de nada destacar lo obvio.

```
// Utility method that returns when this.closed is true. Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
   if(!closed)
   {
      wait(timeoutMillis);
      if(!closed)
         throw new Exception("MockResponseSender could not be closed");
   }
}
```

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
}
```

```
/** The day of the month. */
    private int dayOfMonth;
```

```
/**
 * Returns the day of the month.
 *
 * @return the day of the month.
 */
public int getDayOfMonth() {
  return dayOfMonth;
}
```

```
/** The name. */
private String name;

/** The version. */
private String version;

/** The licenceName. */
private String licenceName;
```

- **Historia de cambios**: Una vieja práctica dictaba que cada cambio que se hacía en un archivo de código debía tener un comentario explicando el cambio. Esto ya no es necesario gracias a los controladores de versiones.

```
* Changes (from 11-Oct-2001)
* --------------------------
* 11-Oct-2001 : Re-organised the class and moved it to new package
*               com.jrefinery.date (DG);
* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
*               class (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
*               class is gone (DG);  Changed getPreviousDayOfWeek(),
*               getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
*               bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
* 29-May-2002 : Moved the month constants into a separate interface
*               (MonthConstants) (DG);
* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
* 13-Mar-2003 : Implemented Serializable (DG);
* 29-May-2003 : Fixed bug in addMonths method (DG);
* 04-Sep-2003 : Implemented Comparable.  Updated the isInRange javadocs (DG);
* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
```

- **No usar comentarios cuando se puede usar una variable o función**: De nuevo, la expresividad del código suplanta los comentarios.

```
// does the module from the global list <mod> depend on the
// subsystem we are part of?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

Podría refactorizarse a:

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

- **Atribuciones del código**: Generalmente los IDEs agregan un comentario en el encabezado de las clases que agregamos. Esto no es necesario gracias al controlador de versiones.
- **Código comentado**: A veces, comentamos código en nuestro proceso de desarrollo… Nunca subir estos cambios!! El código comentado debe ser eliminado **SIEMPRE**.

```
        InputStreamResponse response = new InputStreamResponse();
        response.setBody(formatter.getResultStream(), formatter.getByteCount());
//      InputStream resultsStream = formatter.getResultStream();
//      StreamReader reader = new StreamReader(resultsStream);
//      response.setContent(reader.read(formatter.getByteCount()));
```

- **Información no local**: Si hace falta escribir un comentario, asegurarse de que sea sobre el contexto al que pertenece. No ofrecer información sobre contextos externos, ya que es difícil de mantener.

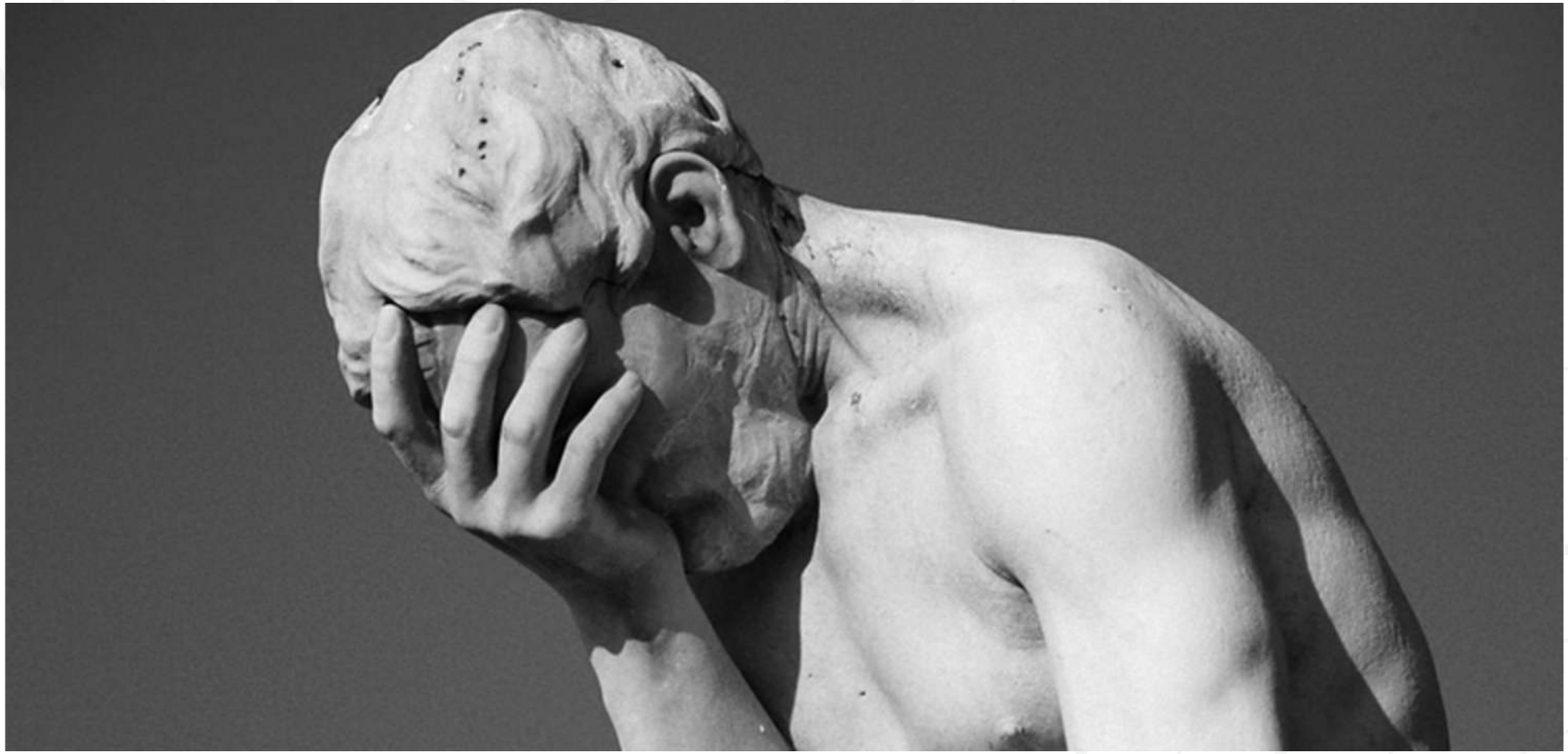
```
/**
 * Port on which fitnesse would run. Defaults to <b>8082</b>.
 *
 * @param fitnessePort
 */
public void setFitnessePort(int fitnessePort)
{
  this.fitnessePort = fitnessePort;
}
```

- **Demasiada información**: No agregar datos históricos o irrelevantes. Ser concisos y directos, y recordar que los comentarios deben ser minimizados al máximo! Nuestro código debe ser el mejor documento.

Formatting

- El código debería estar siempre correctamente **formateado**. El formato implica tanto la indentación como la disponibilidad de los elementos del código.
- Esto facilita la lectura y demuestra profesionalismo.
- **The Newspaper Metaphor**: Las funciones y subfunciones deben aparecer en el orden que se utilizan. Esto simplifica la legibilidad del código.
- Dejar siempre sólo un “enter” de separación entre conceptos diferentes (secciones de variables, métodos, definición de clase, etc).
- **Indentación**: Es fundamental y obligatoria!
- **Reglas del equipo**: Las reglas de formatting deben ser acordadas y utilizadas por TODO el equipo de manera uniforme.

Manejo de Errores



- **Usar excepciones en vez de códigos de error**: Viejos lenguajes de programación no tenían manejo de excepciones, por lo que se utilizaban variables o parámetros para indicar errores.
- **Escribir la sentencia try-catch-finally primero**: El método de nivel superior debería tener la sentencia `try-catch-finally` de manera que los errores de capturen a ese nivel y se simplifiquen las sub funciones.
- **Utilizar excepciones no checkeadas**: Las checked exceptions generan acoplamiento y no mejoran realmente el manejo de errores. En los lenguajes modernos (como Kotlin) todas las excepciones son no checkeadas.

- **Proveer contexto significativo con las Excepciones**: Las excepciones que se disparan deben tener la información suficiente para poder entender la razón y origen del error. Evitar sentencias como `throw new Exception();`
- **No retornar null ni pasar null como parámetro**: Los punteros nulos son siempre fuente de errores. Debemos tratar de evitarlos lo máximo posible. Por ejemplo, si manejamos listas es mejor retornar una lista vacía a una lista null. También, utilizar las herramientas que nos provee el framework de desarrollo (por ejemplo, markers de java `@NotNull`).

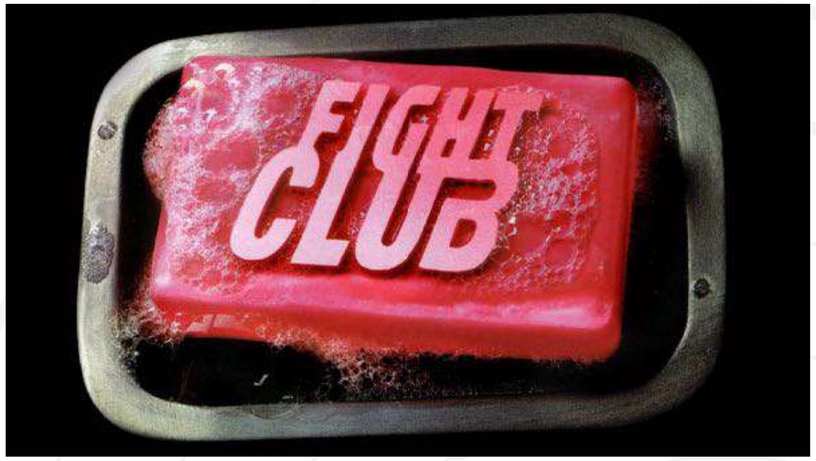
Clases

Organización de las clases

- Siguiendo las convenciones standard de Java, una clase debería seguir el siguiente orden:
  - Lista de variables:
    - Constantes estáticas públicas
    - Variables estáticas privadas
    - Variables de instancia privadas
- Luego de la lista de variables, de definen las funciones públicas. Es preferible definir las funciones privadas inmediatamente después de la función pública que la llama, para seguir la regla de lectura “newspaper article”.

Las Clases deben ser pequeñas!

- 1ra regla: las clases deberían ser pequeñas.
- 2da regla: las clases deberían ser aún más pequeñas!
- ¿Qué tan pequeñas? Lo mínimo indispensables para que las clases **tengan una sola responsabilidad**.



Principio de Responsabilidad Única

- **El Principio de Responsabilidad Única** (SRP) enuncia que una clase o módulo debe tener una única **razón de cambio**.
- El principio nos da la definición de **responsabilidad** como guía del tamaño de una clase.

Cohesión

- Las clases deberían manejar una lista pequeña de variables de instancia.
- Cada uno de los métodos de la clase debería manipular una o más de esas variables.
- En general, mientras más variables manipula un método, más cohesiva es a la clase.
- Una clase en donde todos los métodos manipulan todas las variables es altamente cohesivo.

```
public class Stack {
  private int topOfStack = 0;
  List<Integer> elements = new LinkedList<Integer>();

  public int size() {
    return topOfStack;
  }

  public void push(int element) {
    topOfStack++;
    elements.add(element);
  }

  public int pop() throws PoppedWhenEmpty {
    if (topOfStack == 0)
      throw new PoppedWhenEmpty();
    int element = elements.get(--topOfStack);
    elements.remove(topOfStack);
    return element;
  }
}
```

Mantener la Cohesión da como resultado muchas clases pequeñas

- Transformar una función grande en pequeñas funciones puede dar como resultado la proliferación de clases pequeñas cuando tratamos de maximizar la cohesión.
- Por ejemplo, partimos una función en varios pedazos. Eso implica enviar parámetros a las distintas funciones. Pero, intentamos minimizar el uso de parámetros, por lo que creamos variables de instancia que compartan esas funciones.
- Ahora, esas variables son usadas sólo por esas funciones, lo que compromete la cohesión. Pero, si sacamos esas funciones relacionadas a clases mas pequeñas, mantenemos la cohesión y el principio SRP.

Smells and Heuristics

- La siguiente es una lista de “**code smells**”.
- Son indicaciones de que el código no es limpio y debería ser limpiado (boy scout rule).

Comentarios

- **Información inapropiada**: Un comentario tiene información que debería mantenerse en otro lado, por ejemplo el control de versiones.
- **Comentario obsoleto**: Un comentario que es viejo, irrelevante o incorrecto.
- **Comentario redundante**: Describe lo mismo sin agregar valor informativo.
- **Comentario pobremente escrito**: Todo comentario que valga la pena debe estar bien escrito.
- **Código comentado!!**

Entorno

- **El build requiere más de un paso**: Hacer build del proyecto debería ser trivial. Si requiere de demasiados pasos (por ejemplo generar archivos de recursos, pre procesar data, todo de manera manual), puede ser que tenga una complejidad excesiva que puede conducir a errores (por ejemplo, omitir un paso por equivocación).
- **Los tests requieren más de un paso**: Los tests deberían ejecutarse con un solo comando.

Funciones

- **Demasiados argumentos**: Las funciones deberían tener un número pequeño de argumentos.
- **Argumentos de salida**: Los argumentos de salida no son intuitivos.
- **Argumentos bandera**: Indicación de que la función hace más de una cosa.
- **Función muerta**: Métodos que nunca son llamados deberían ser eliminados.

General

- **Múltiples lenguajes en un solo archivo**: Por ejemplo, un archivo de java con snippets de xml.
- **Comportamiento obvio no implementado**: Siguiendo el “Principio de la Menor Sorpresa”, cada función o clase debería implementar comportamiento esperado. Por ejemplo, una lista mutable debería implementar el método “add”.
- **Comportamiento incorrecto en los límites**: A veces no esta claro cuándo un algoritmo contempla todos los caminos, en especial los límites. Para este smell, la acción es implementar tests que cubran todos los caminos.

- **Duplicación**: DRY!
- **Código en el nivel de abstracción incorrecto**: Queremos que todos los conceptos de bajo nivel deriven de todos los conceptos de alto nivel. Inversión de Dependencias.
- **Demasiada información**: Segregación de Interfaces.
- **Código muerto**: Código que no se ejecuta, por ejemplo, un “else” que nunca se cumple.

- **Separación vertical**: Las funciones y variables que definimos deberían estar cerca del lugar donde se utilizan. No queremos buscar definiciones con cientos de líneas de separación.
- **Inconsistencia**: Si se hace algo de una forma, cuestiones similares deberían seguir esa forma. Por ejemplo, sufijos de nombres (`xRequest, yRequest, zSendMessage`).
- **Las funciones deben hacer una sola cosa**: SRP.

- **Envidia de comportamiento**: Los métodos de una clase deberían interesarse por variables y funciones de la clase a la que pertenecen, y no por variables y funciones otras clases.

```
public class HourlyPayCalculator {
  public Money calculateWeeklyPay(HourlyEmployee e) {
    int tenthRate = e.getTenthRate().getPennies();
    int tenthsWorked = e.getTenthsWorked();
    int straightTime = Math.min(400, tenthsWorked);
    int overTime = Math.max(0, tenthsWorked - straightTime);
    int straightPay = straightTime * tenthRate;
    int overtimePay = (int)Math.round(overTime*tenthRate*1.5);
    return new Money(straightPay + overtimePay);
  }
}
```

- **Argumentos selectores**: Argumentos que se utilizan para decidir la distintos caminos en un algoritmo son indicadores de que la función hace más de una cosa (banderas, enumerados, etc).
- **Intento oscuro**: El código debe ser lo mas expresivo posible, y “gritar” su intención.
- **Responsabilidad fuera de lugar**: Una de las desiciones más importante de los desarrolladores es decidir dónde ubicar el código. El principio de la mínima sorpresa aplica en este caso.

- **Estáticos inapropiados**: A veces escribimos funciones estáticas que no deberían serlo.

```
HourlyPayCalculator.calculatePay(employee, overtimeRate).
```

- **Uso de variables explicativas**: Ser conciso puede llevar a complicar la legibilidad.

```
Matcher match = headerPattern.matcher(line);
if(match.find())
{
  String key = match.group(1);
  String value = match.group(2);
  headers.put(key.toLowerCase(), value);
}
```

- **Los nombres de las funciones deben decir qué cosa hacen**.

```
Date newDate = date.add(5);
```

- **Polimorfismo sobre estructuras if-else o switch-case**: Open Closed Principle.
- **Seguir las convenciones standard**: Cada equipo debe definir sus propias reglas, y todos los miembros seguirlas.
- **Remplazar números mágicos por constantes.**
- **Ser preciso**: Todas las desiciones que se toman al desarrollar tiene que ser precisas y sin ambigüedades. Si retornamos null, tiene que haber una razón. Si se dispara una excepción, todos los caminos posibles, etc.

- **Encapsular condiciones**: La lógica boolean puede ser lo suficientemente compleja como para que también haya que leerla dentro de un if. Extraerlas a funciones que expliquen el intento de dicha condición. Por ejemplo, qué es mejor?

```
if (shouldBeDeleted(timer))
```

```
if (timer.hasExpired() && !timer.isRecurrent())
```

- **Evitar condiciones negativas**: Los negativos son un poco más complicados de entender que los positivos.

```
if (buffer.shouldCompact())
```

```
if (!buffer.shouldNotCompact())
```

- **Encapsular condiciones límite.**

```
if(level + 1 < tags.length)
{
  parts = new Parse(body, tags, level + 1, offset + endTag);
  body = null;
}
```

```
int nextLevel = level + 1;
if(nextLevel < tags.length)
{
  parts = new Parse(body, tags, nextLevel, offset + endTag);
  body = null;
}
```

- **Las funciones deberían descender solo un nivel de abstracción**: Las sentencias dentro de una función deberían estar todas al mismo nivel de abstracción.
- **Mantener los datos de configuración en los niveles altos**: Por ejemplo, constantes globales, settings, deberían estar en clases fáciles de encontrar, no metidas en funciones de bajo nivel.
- **Evitar la navegación transitiva**: Un módulo no debería conocer demasiado sobre sus colaboradores. Si `A` colabora con `B` y `B` colabora con `C`, no queremos que `A` conozca `C` (Por ejemplo, `a.getB().getC().doSomething()`).

Nombres

- **Usar nombres descriptivos.**
- **Usar nombres en el nivel de abstracción apropiado**: Por ejemplo, nombres con detalle de implementación a nivel alto.
- **Usar nomenclatura standard cuando sea posible**: Nombres de patrones, estilos, etc.
- **No usar nombres ambiguos.**
- **Usar nombres largos para scopes grandes.**
- **Evitar encodings.**
- **Si hay efectos secundarios, indicarlos en el nombre.**

Bibliografía