Tecnología de Programación

Clean Code

Basado en el libro ***Clean Code – A handbook of agile software craftsmanship*** del autor ***Robert C. Martin***

Dr. Federico Joaquín
federico.joaquin@cs.uns.edu.ar

Algunos derechos reservados

Clean Code.

Repaso: Clean code: ¿qué es?

- Es un **concepto** que se refiere a **escribir código** de programación de **alta calidad** que sea fácil de **entender**, **mantener**, y **extender**.
- Fue popularizado por el autor **Robert C. Martin** en su libro *Clean Code: A Handbook of Agile Software Craftsmanship*.
- El **objetivo** principal es **producir software** que sea **legible**, **eficiente** y **libre** de errores.

> *Clean code always looks like it was written by someone who cares.*
>
> *Writing clean code is what you must do in order to call yourself a professional. There is no reasonable excuse for doing anything less than your best.*

Funciones

LAS FUNCIONES SON LA PRIMERA LÍNEA DE ORGANIZACIÓN EN CUALQUIER PROGRAMA. ESCRIBIRLAS BIEN ES EL TEMA DE ESTA SECCIÓN.

Funciones pequeñas

- La **primera regla** para definir funciones es que **deben ser pequeñas**.
- La **segunda regla** de las funciones es que **deben ser aún más pequeñas**.
  - No existe **investigación** que demuestren que las **funciones pequeñas son mejores**. Lo que si se observa es que los **programadores profesionales** con gran experiencia, indican que cuanto más pequeñas, mejor les han resultado.
- ¿Qué tan **pequeñas** deberían ser?
  - Lo **mínimo** indispensable.
  - Parafraseando uno de los Principios de Diseño, tan pequeñas como para **realizar una sola cosa** (principio de **responsabilidad única - SOLID**).
  - Las líneas **no deberían** tener 150 caracteres. Las funciones **no deben** tener 100 líneas. Las funciones **casi nunca deberían** tener más de 20 líneas.

Funciones pequeñas

- ¿Qué tan **pequeñas** deberían ser?
  - Considere la siguiente **refactorización**

Listing 3-2
HtmlUtil.java (refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }

  return pageData.getHtml();
}
```

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

Funciones pequeñas

- ¿Qué tan **pequeñas** deberían ser?
  - Considere la siguiente **refactorización**
- **Bloques** e **indentación**
  - Los **bloques** dentro de las declaraciones **`if`**, **`else`**, **`while`**, **`etc`**, deberían tener una longitud de una línea.
  - Probablemente esa **línea** debería ser una **llamada** de **función**.
  - En consecuencia, las **funciones no deberían** ser lo suficientemente grandes como para contener **estructuras anidadas**.
  - El nivel de **sangría** (**indentación**) no debería ser mayor que **uno** o **dos**.

Listing 3-3
`HtmlUtil.java (re-refactored)`

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

Do one thing

- El problema asociado a esta guía es **reconocer qué es una sola cosa**.

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

- ¿Qué hace el código anterior? Es **fácil indicar** que hace tres cosas.
  - Determinar cuándo la página es un test.
  - Si es así, incluir setups y teardowns.
  - Renderizar la página en HTML.

> *Functions should do one thing. They should do it well. They should do it only.*

Do one thing

- El problema asociado a esta guía es **reconocer qué es una sola cosa**.

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

- Observe que los **tres pasos** de la función están un **nivel de abstracción por debajo** del nombre indicado de la función.
- Una función **está haciendo más de una cosa si se puede extraer** otra función de ella con un nombre que no sea una reformulación de su implementación.

> *Functions should do one thing. They should do it well. They should do it only.*

Nivel de abstracción y The stepdown rule

- Para asegurar que una función **hace una sola cosa**, se debe asegurar que las sentencias **están al mismo nivel** de abstracción.
  - `getHtml();` // alto
  - `String pagePathName = PathParser.render(pagePath);` // medio
  - `.append("\n");` // bajo
- **Mezclar** niveles de abstracción **dentro de una función** siempre **resulta confuso**.
- El código **debe leerse** como una narrativa de **arriba hacia abajo**.
  - Se espera que **cada función** sea **seguida** por aquellas en el **siguiente nivel** de abstracción.
  - Leer el programa, descendiendo de a un nivel de abstracción a la vez, permite **comprender fácilmente** el conjunto de funciones en el código.

Nombres descriptivos

- Para lograr **este principio siempre** se debe elegir **buenos nombres** de, funciones **pequeñas** que, hagan **una sola cosa**.
- Cuanto más **pequeña** y **enfocada** sea una función, más fácil será elegir un nombre descriptivo.
  - **No temer** por el uso de nombres **largos**.
    - Son mejores que los **nombres enigmáticos y cortos**
    - Son mejores que un **comentario descriptivo largo**.
  - Ser **consistentes** en el uso del mismo tipo de frases, nombres, verbos, etc.

> *You know you are working on clean code when each routine turns out to be pretty much what you expected.*
>
> **Ward principle.**

*Ward **Cunningham**, inventor of Wiki, inventor of Fit, coinventor of eXtreme Programming.*

Motive force behind Design Patterns.

Smalltalk and OO thought leader.

The godfather of all those who care about code.

Argumentos

- El número **ideal** de argumentos para una función es **cero**.
  - El siguiente número ideal es **dos**.
  - Siempre que sea posible, **deben evitarse tres** argumentos.
  - Más de tres requieren una **justificación muy especial**, y entonces **no debería usarse**.
- Los argumentos son **difíciles** de nombrar. **Requieren** mucho **poder conceptual**.
  - La razón es la **legibilidad**: mayor cantidad de argumentos, menos clara es la intención de la función.
  - Por ejemplo, `includeSetupPage()` es más claro de comprender que `includeSetupPageInto(newPageContent).`
- Además, la **complejidad** de los tests unitarios crece rápidamente junto con el número de parámetros.

Argumentos

- Los argumentos definidos como **parámetros** de **salida** **deben ser evitados**.
  - Es ampliamente aceptado que la **información ingresa** a la función a través de **argumentos** y **sale** a través del **valor de retorno**.
  - Por el contrario, los **argumentos** utilizados como parámetros de **salida** requieren de una **atención adicional**.
- Razones **esperadas** para utilizar una función con **un** argumento:
  - Cuando se realiza una **consulta** por sobre el **argumento**, por ejemplo `boolean fileExists(MyFile)`
  - Cuando se realiza una **operación** por sobre el **argumento**, por ejemplo `InputStream fileOpen(MyFile)`
  - Se deben evitar funciones de un argumento que **no sigan** este formato.
- Si una función va a **trasformar** un **input**, es mejor **retornarlo explícitamente** en vez de modificar el input.
  - `StringBuffer transform(StringBuffer in)` es más claro que
  - `void transform(StringBuffer out)`

Argumentos

- Los argumentos definidos como **banderas deben ser evitados**.
  - Un parámetro **booleano complejiza** encontrar un identificador adecuado para la función.
  - El **booleano** proclama que la función **hace más de una cosa**.
  - Hace una cosa si el **booleano** es **true** y otra si es **false**.
- Es recomendable realizar una **refactorización** de la función de manera tal de **evitar las banderas**.

Argumentos

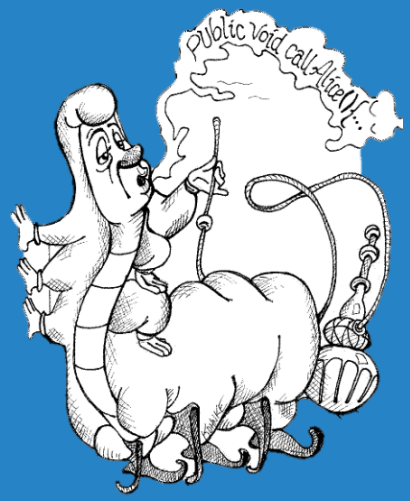
- Si una función **requiere más de dos** argumentos, es probable que estos deban incluirse en una **clase** que los **agrupe**.

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

Evitar efectos secundarios

- Los **efectos secundarios** de una función **no deben ocurrir**.
- Esto es así ya que los efectos secundarios se vuelven **mentiras**.
  - Se espera que una **función haga una cosa**, pero **también hace otras**.
- Los efectos secundarios **no solo** implican una segunda acción, sino que además **atentan** contra la **legibilidad**.
  - Dan lugar a nuevos bugs.

```
public boolean checkPassword(String userName, String password) {
  User user = UserGateway.findByName(userName);
  if (user != User.NULL) {
    String codedPhrase = user.getPhraseEncodedByPassword();
    String phrase = cryptographer.decrypt(codedPhrase, password);
    if ("Valid Password".equals(phrase)) {
      Session.initialize();
      return true;
    }
  }
  return false;
}
```



Command Query Separation

- Una función debería **hacer** algo, **o contestar** algo, pero **nunca ambas** cosas al mismo tiempo.
- Una función debería **cambiar** el **estado** de un objeto, o **retornar información** sobre el objeto.
  - Hacer ambas cosas puede resultar confuso.

```
public boolean set(String attribute, String value);

if (set("username", "unclebob"))...

if (attributeExists("username")) {
  setAttribute("username", "unclebob");
  ...
}
```

Comentarios

*The proper use of comments is to compensate for our **failure** to express ourself in code.*

*Note that I used the word **failure**. I meant it. Comments are always **failures**.*

We must have them because we cannot always figure out how to express ourselves without them, but their use is not a cause for celebration.

> *El único buen comentario es el que no debería ser escrito.*

Comentarios no compensan el *bad code*

- Una de las **motivaciones** más comunes para **escribir comentarios** es el **bad code**.
- Se escribe un módulo **confuso** y **desorganizado:** un **desastre** de código. Entonces nos decimos a nosotros mismos: *¡Oh, será mejor que comente eso!*
  - La respuesta es **¡no!** **¡Será mejor que lo limpies!**
  - El **clean code**, **expresivo**, con **pocos comentarios** es muy **superior** al código **complejo** y **desordenado** con muchos comentarios.
- En lugar de **dedicar** tiempo a **escribir comentarios** que explican algo desastroso, mejor **dedicarlo** a **realizar clean code**.

Explain Yourself in Code

- Existen **situaciones** en las que el **código** no es el medio adecuado para una **explicación**.
- Desafortunadamente, muchos programadores han **interpretado** que esto significa que el código **rara vez**, o **nunca**, es un **buen medio** de explicación.
  - Esta sensación es **falsa**.
  - Observe el siguiente ejemplo, e indique qué versión prefiere:

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

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

Buenos comentarios

- Algunos comentarios son **necesarios** o **beneficiosos**.
  - Comentarios **legales**: copyright y autoría.

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc.
// All rights reserved.
// Released under the terms of the GNU General Public License
// version 2 or later.
```

  - Comentarios **informativos**: aclarar información básica 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*");
```

Buenos comentarios

- Algunos comentarios son **necesarios** o **beneficiosos**.
  - Explicación de la **intención**: para explicar mejor la intención del código en casos que **no son obvios** o que pertenecen a la lógica de negocio.

```
public int compareTo(Object o) {
  if(o instanceof WikiPagePath){
    WikiPagePath p = (WikiPagePath) o;
    String compressedName = StringUtil.join(names, "");
    String compressedArgumentName = StringUtil.join(p.names, "");
    return compressedName.compareTo(compressedArgumentName);
  }
  return 1; // we are greater because we are the right type.
}
```

Buenos comentarios

- Algunos comentarios son **necesarios** o **beneficiosos**.
  - Para **clarificar**: para **traducir** el significado de un parámetro o condición de retorno oscura. 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
}
```

Buenos comentarios

- Algunos comentarios son **necesarios** o **beneficiosos**.
  - Para **advertir consecuencias**.

```
// Don't run unless you have some time to kill.
public void _testWithReallyBigFile() {
  writeLinesToFile(10000000);
  response.setBody(testFile);
  response.readyToSend(this);
  String responseString = output.toString();
  assertSubString("Content-Length: 1000000000", responseString);
  assertTrue(bytesSent > 1000000000);
}
```

Buenos comentarios

- Algunos comentarios son **necesarios** o **beneficiosos**.
  - Comentarios **TO-DO**. cuando alguna funcionalidad no es posible de implementar por alguna dependencia.

```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout model
protected VersionInfo makeVersion() throws Exception {
  return null;
}
```

  - **Amplificación**: 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()));
```

Malos comentarios

- La **mayoría** de los comentarios **entran** en esta categoría. Por lo general, son **excusas** para un código **deficiente** o **justificaciones** para decisiones **insuficientes**.
  - Comentarios **porque sí**: deben ser evitados **siempre**.
  - Si se escribe un comentario se **debe tomar** el tiempo necesario para que sea el **mejor comentario** que se puede hacer.
    - Por ejemplo, no es un buen comentario aquel que **nos obliga a estudiar** el código para entenderlo.

Malos comentarios

- La **mayoría** de los comentarios **entran** en esta categoría. Por lo general, son **excusas** para un código **deficiente** o **justificaciones** para decisiones **insuficientes**.
  - Comentarios **redundantes**: no resulta de utilidad destacar lo que es **obvio** en el propio **código fuente**.

```
// 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");
  }
}
```

Malos comentarios

- La **mayoría** de los comentarios **entran** en esta categoría. Por lo general, son **excusas** para un código **deficiente** o **justificaciones** para decisiones **insuficientes**.
  - Comentarios **engañosos**: la declaración de los comentarios deben ser lo suficientemente **precisas** como para ser **exactas**.

```
// 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");
  }
}
```

  - Note que en el ejemplo anterior, el comentario **no es preciso**: la función **no retorna** cuando **`this.close == true`**, sino que, en algún caso, espera **`timeOutMillis`**.

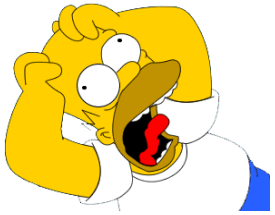
Malos comentarios

- La **mayoría** de los comentarios **entran** en esta categoría. Por lo general, son **excusas** para un código **deficiente** o **justificaciones** para decisiones **insuficientes**.
  - Comentarios **molestos**: aquellos comentarios que no son más que ruido **inútil** para el código.

```
/** 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;

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

Malos comentarios

- La **mayoría** de los comentarios **entran** en esta categoría. Por lo general, son **excusas** para un código **deficiente** o **justificaciones** para decisiones **insuficientes**.
  - Comentarios que podrían ser una **función o variable**.

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

  - Podría **refactorizarse** de la siguiente forma

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

Malos comentarios

- La **mayoría** de los comentarios **entran** en esta categoría. Por lo general, son **excusas** para un código **deficiente** o **justificaciones** para decisiones **insuficientes**.
  - Código descartado mediante **comentarios**. No debería ocurrir **nunca**, y tampoco se debería compartir código con **código comentado**.

```
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()));
```

Formateo del código

LA FALTA DE ATENCIÓN AL DETALLE IMPREGNA TODOS LOS DEMÁS ASPECTOS DEL PROYECTO.

*We want them to be **struck** by the **orderliness**.*

*We want their **eyebrows to rise** as they **scroll** through the **modules**.*

*We want them to **perceive** that **professionals** have been at work.*

*“Cuando las personas miran debajo del capó, queremos que queden impresionadas con la **pulcritud**, la **coherencia** y la **atención** al **detalle** que perciben.*

El propósito del formato

- Seamos claros: ¡el **formato del código** es **muy importante**!
  - Tiene que ver con la comunicación, y la comunicación es la **primera tarea** del **desarrollador profesional**.
- Es posible que como desarrollador **haya pensado** en que ***hacerlo funcionar*** sea la primera orden del día para un profesional.
  - **Clean clode** permite **desengañarse** de esa **muy mala premisa**.
- La **legibilidad** tendrá un efecto **profundo** en todos los cambios futuros.
- El **estilo** de **codificación** y la **legibilidad** sientan precedentes que continúan afectando la capacidad de **mantenimiento** y la **extensibilidad**.

Formateo vertical

*What does that mean to us? It appears to be possible to **build significant systems** out of files that are typically **200 lines long**, with an **upper limit of 500**.*

*Although this should **not be** a **hard** and **fast rule**, it should be considered very **desirable**.*

- ¿Qué **tamaño debe tener** un archivo **fuente**?
  - En Java, este tamaño **está relacionado** con el tamaño de la **clase**.

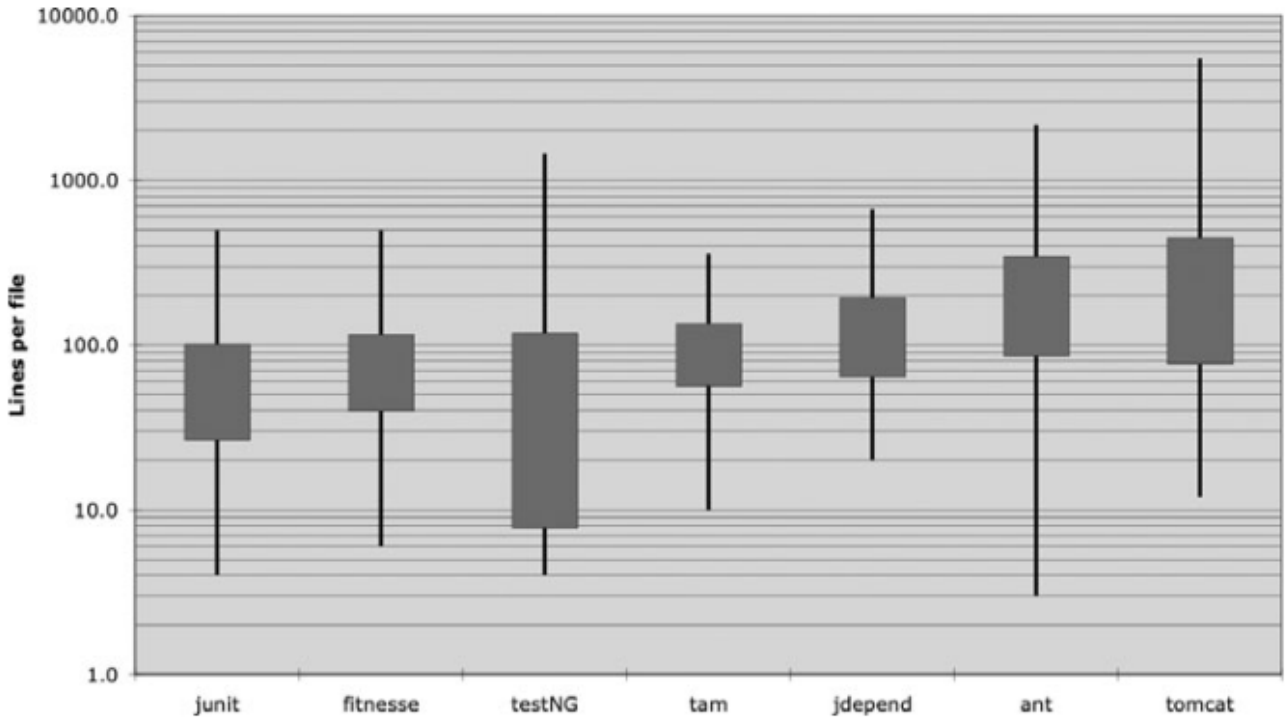


Figure 5-1
File length distributions LOG scale (box height = sigma)

Las líneas horizontales muestran la longitud mínima y máxima. El medio del cuadro es la media. El tamaño promedio de los archivos en FitNesse es de 65 líneas, y un tercio de los archivos tienen entre 40 y más de 100 líneas. El archivo más grande de FitNesse tiene 400 líneas y el más pequeño tiene 6.

Formateo vertical

- La **metáfora** del **periódico**
  - Un diario se lee verticalmente: **titulo** + primer párrafo (**sinopsis**) + siguientes párrafos **aumentan** los **detalles**.
- Un **archivo fuente** debe **asemejarse** a un artículo de periódico.
  - El nombre debe ser sencillo pero **explicativo**.
  - Las partes superiores del archivo **deben proporcionar** información de **alto nivel**.
  - Los detalles deberían **aumentar** a medida que avanzamos hacia abajo.

Formateo vertical

- **Apertura** vertical entre **conceptos**
- Casi todo el código se lee de **izquierda** a **derecha** y de **arriba** a **abajo**.
  - Cada **línea** representa una **expresión** o una **cláusula**.
  - Cada **grupo** de **líneas** representa un **pensamiento completo**.
  - Esos **pensamientos** **deben estar separados** unos de otros con **líneas en blanco**.
- Esta **regla** extremadamente **simple** **tiene un profundo efecto** en el diseño visual del código.
  - Cada línea en blanco es una **señal visual** que identifica un **concepto nuevo** y **separado**.

Formateo vertical

- **Distancia** vertical entre **conceptos**
- Los **conceptos estrechamente relacionados** **deben mantenerse** verticalmente **cerca** unos de otros.
  - **Funciones** y **subfunciones** (deben encontrarse **próximas entre sí**, una debajo de las otras, evitando la navegación excesiva en los archivos).
  - Declaración de **variables** y su **utilización**.
    - Las variables deben declararse **lo más cerca posible** de su uso. Esto **no implica** que deban declararse **dentro** de bloques **`then`** o **`else`** en un **`if`**.
    - **Clean code** comulga con funciones que **hacen una sola tarea** y son **muy cortas**. Luego, las variables deberían aparecer en la **parte superior** de cada función.
  - Funciones con **cierta afinidad**, **deben agruparse** y **ubicarse** juntas o próximas. Por ejemplo, **`getters`**, **`setters`**.

Formateo horizontal

- ¿Qué **tamaño debe tener** cada línea de un archivo **fuente**?

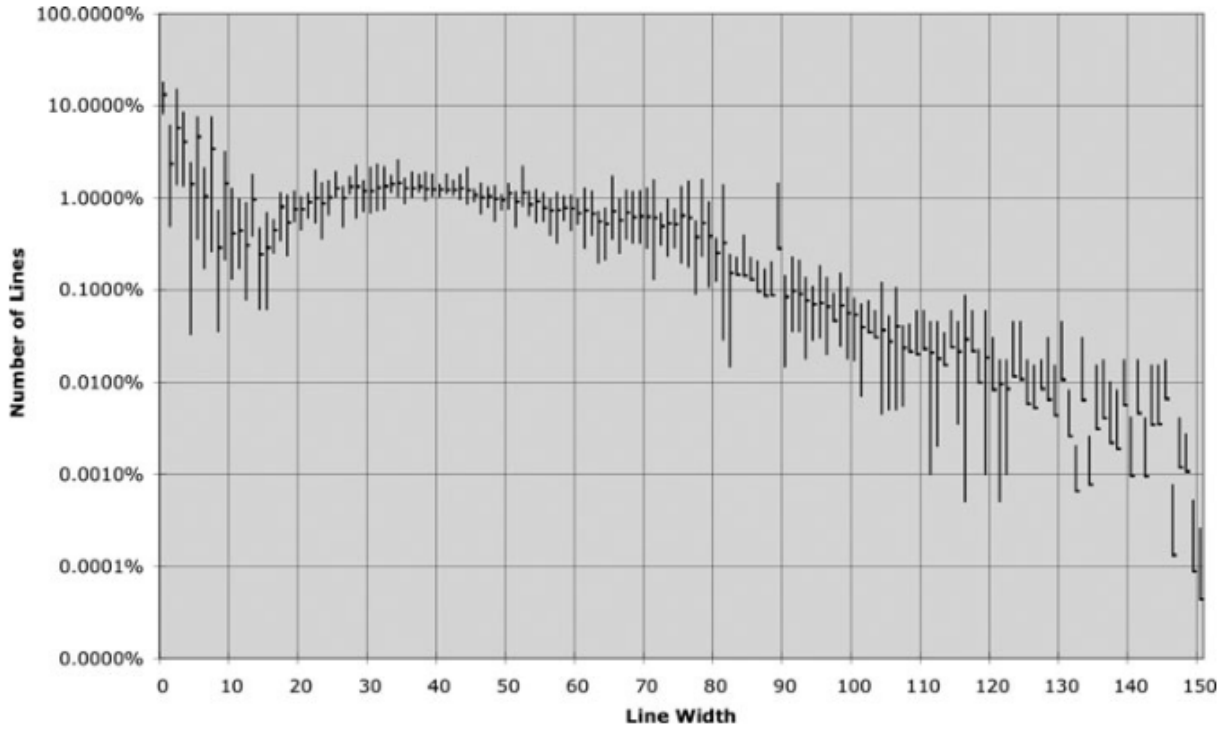


Figure 5-2
Java line width distribution

La Figura muestra la distribución de las longitudes de línea de siete proyectos. La regularidad es impresionante, especialmente alrededor de los 45 caracteres. De hecho, cada tamaño del 20 al 60 representa aproximadamente el 1 por ciento del número total de líneas. ¡Eso es el 40 por ciento! Claramente, los programadores ***prefieren las líneas cortas****.*

Formateo horizontal

- **Apertura** horizontal entre **conceptos**
  - **Se deben** utilizar espacios en blanco horizontales (o tabuladores) para asociar cosas que **están** fuertemente **relacionadas** y **disociar** cosas que están menos relacionadas.

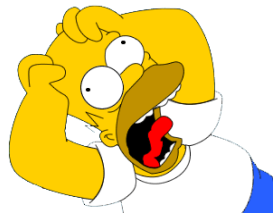
```
private void measureLine(String line) {
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  recordWidestLine(lineSize);
}
```

  - **Operadores** de **asignación con** espacios para acentuarlos.
  - **Declaraciones** de **asignación con** espacios hacen obvia la **diferencia** entre el lado izquierdo y derecho de la asignación.
  - **No** se usan espacios entre los **nombres de las funciones y el paréntesis** de apertura (estos conceptos están estrechamente relacionados).
  - Los **argumentos** de la función están **separados**, para acentuar su diferencia.
  - Se **utilizan espacios** para acentuar la **precedencia** de los operadores.

Formateo horizontal

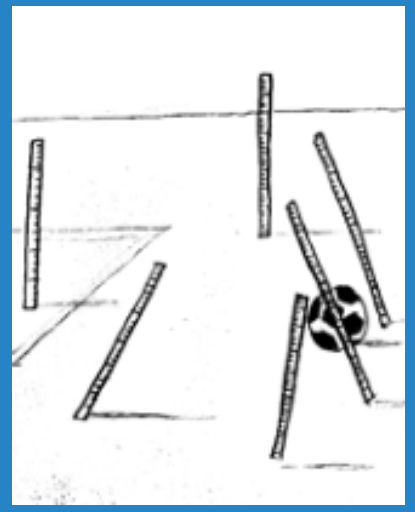
- **Indentación**
  - Un archivo fuente es una **jerarquía** parecida a un **esquema**.
  - Hay **información** que pertenece al archivo en su **conjunto**, a las **clases individuales** dentro del archivo, a los **métodos** dentro de las **clases**, a los **bloques dentro** de los **métodos** y, recursivamente, a los **bloques dentro** de los **bloques**.
  - Para **hacer visible** esta **jerarquía** de **ámbitos**, **indentamos** las líneas del código fuente en proporción a su posición en la jerarquía.

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }
```



  - El código **debe indentarse siempre**, **sin excepción ninguna**.

Team rules



We want the software to have a consistent style. We don't want it to appear to have been written by a bunch of disagreeing individuals.

- Cada **programador** tiene **sus propias reglas** de formato favoritas, **sin embargo**, si trabaja en equipo, **entonces el equipo manda**.
- Un **equipo de desarrolladores** **debe acordar** un estilo de **formato** único y luego cada miembro de ese equipo **debe usar ese estilo**. Se debe decidir
  - **dónde** colocar las **llaves**
  - **cuál** es el tamaño de **sangría**
  - **cómo** **nombrar** las **clases**, **variables** y **métodos**, etc.

Formateo de código

*“Un **buen sistema** de **software** se compone de un conjunto de documentos que se **leen bien**. Deben tener un **estilo consistente** y **suave**. El lector debe poder **confiar** en que los gestos de **formato** que ha visto en un archivo significarán **lo mismo** en otros.*

Moraleja


I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code
I will not write any more bad code

Fin de la presentación.

Referencias

- ***Clean Code: A Handbook of Agile Software Craftsmanship*** (1st. ed.). 2008. Robert C. Martin. Prentice Hall PTR, USA.