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.

Introducción

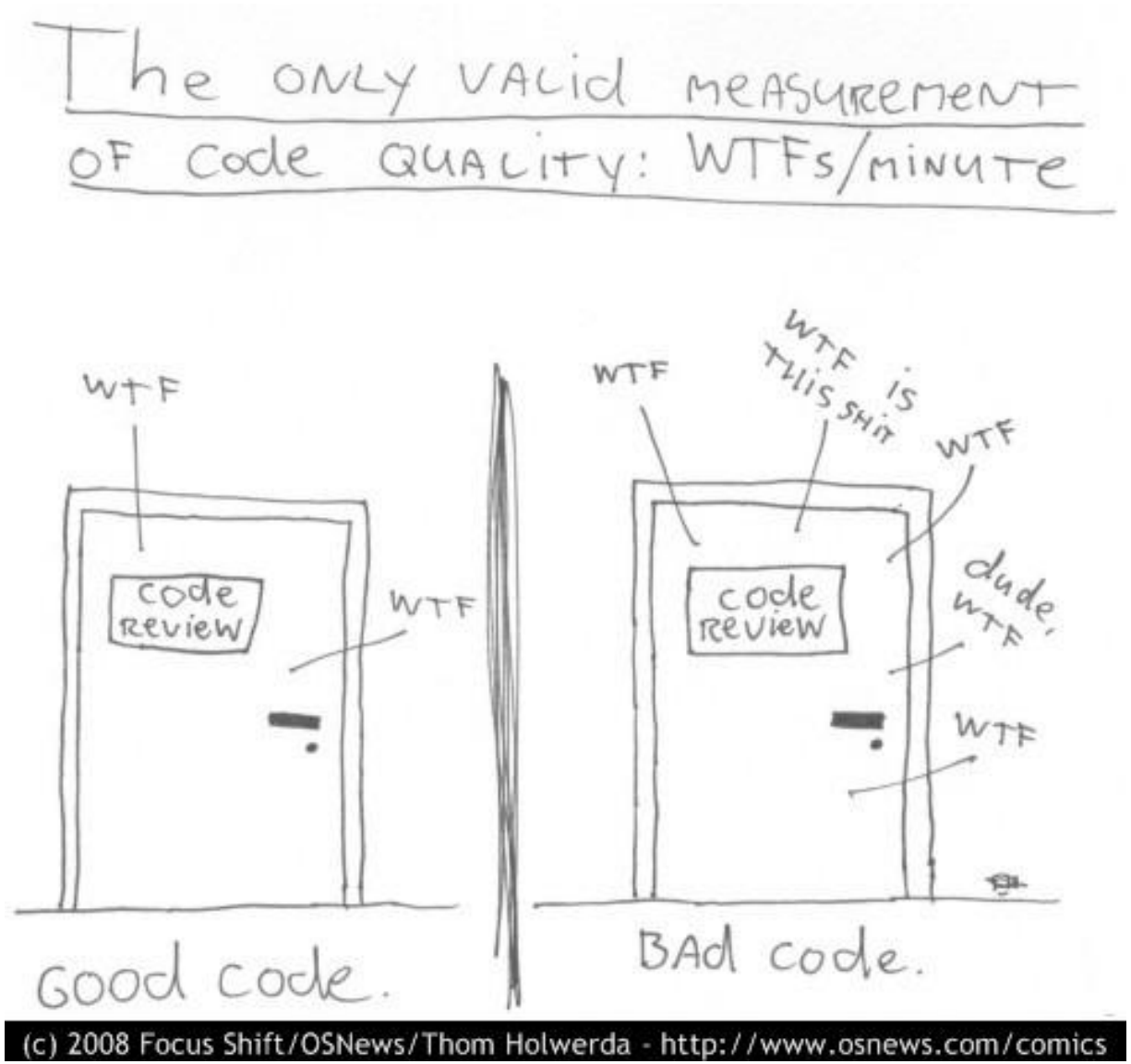
¿QUÉ ES? ¿PARA QUÉ?

Introducción

¿Cuál de las dos puertas representa tú código?

*Aprender a escribir **código limpio** es un trabajo duro. Requiere algo más que el conocimiento de principios y patrones. Se debe sudar por eso.*

Debes practicarlo y observar cómo fracasas. Debes observar a otros practicarlo y fracasar. Debes verlos tropezar y volver sobre sus pasos. Debes verlos agonizar por sus decisiones y ver el precio que pagan por tomar esas decisiones de manera incorrecta.



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.
- Facilita la **depuración**, la **ampliación** y la **colaboración** con otros desarrolladores.

> *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.*

Clean code: ¿para qué?

You are reading this book for two reasons.

First, you are a programmer.

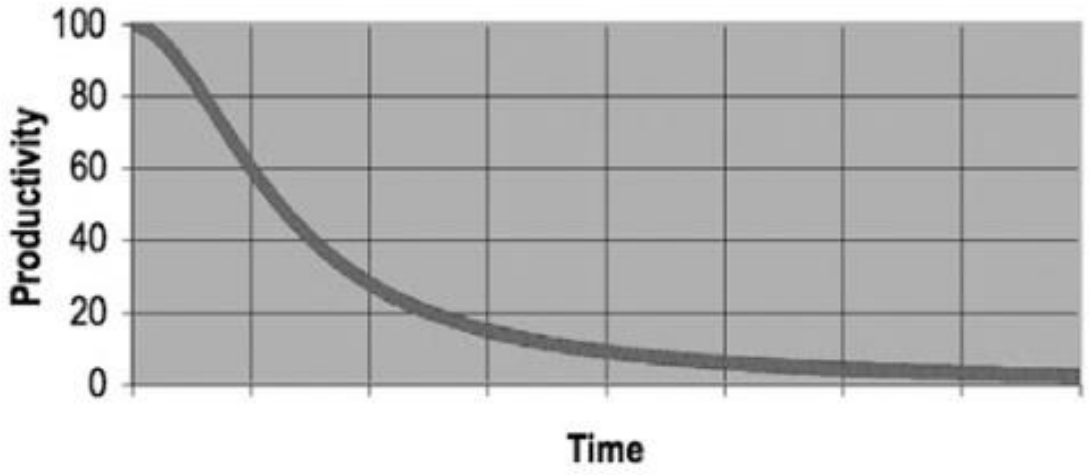
Second, you want to be a better programmer.

Good.

We need better programmers.

- Que el **código funcione**, ¿**alcanza**?
  - Luego de **dos o tres años como programador**, probablemente el **código desordenado** de otra persona **ralentice** su trabajo **significativamente**.
  - Con **más de tres años**, probablemente con este mismo código, su productividad **directamente se haya frenado**.

- Con el tiempo, el **desorden** en el código se vuelve tan grande, tan profundo y tan alto que **nadie puede limpiarlo**. No hay manera alguna.



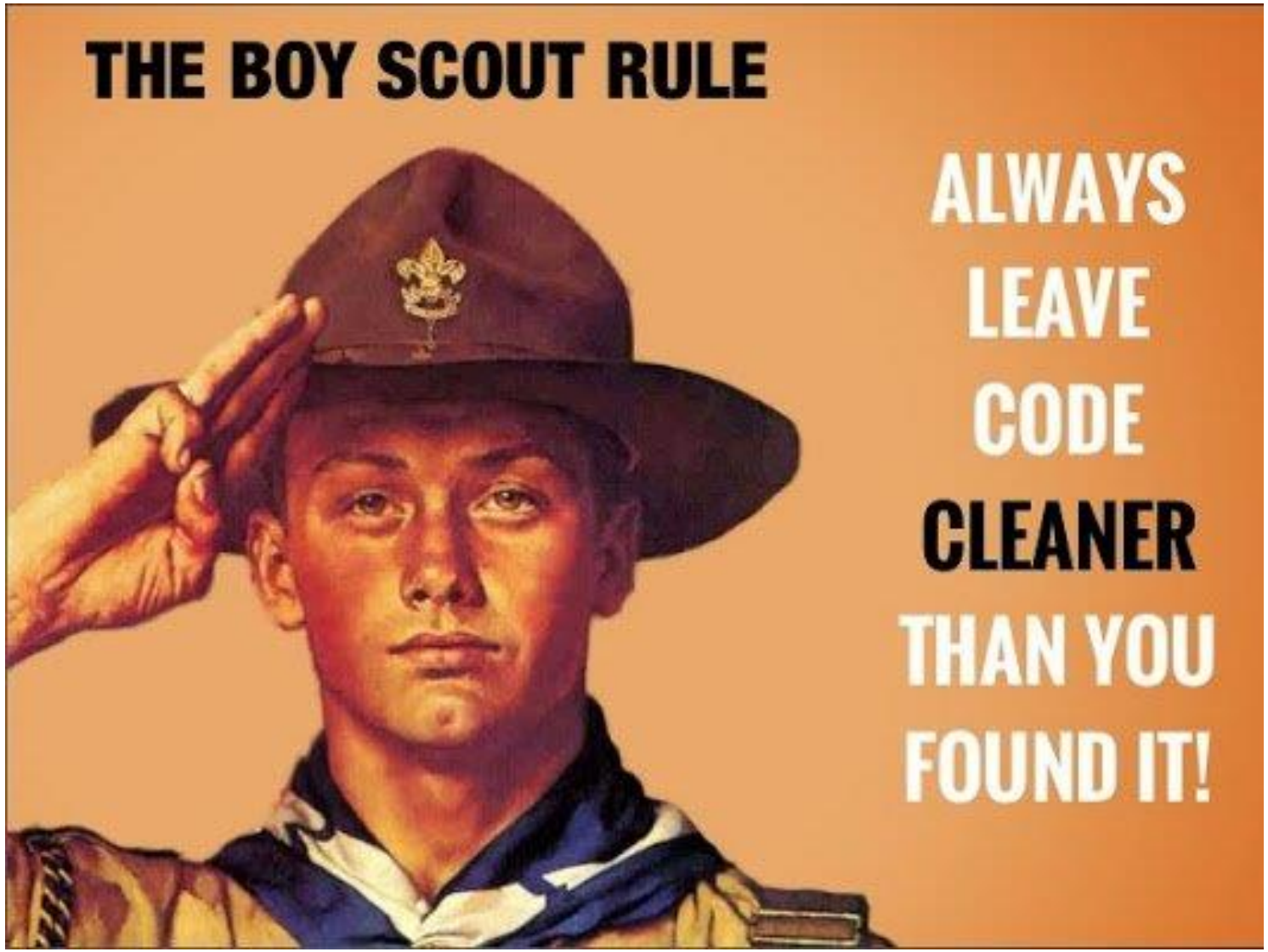
Clean code: el dilema primordial

- Los **programadores profesionales** se enfrentan frecuentemente ante un **dilema primordial**
  - avanzar para **cumplir** con los plazos aún produciendo **bad code.**
  - generar **clean code** aún cuando esto **demande dedicación**.
- Cualquier programador más o menos experimentado sabe que el **bad code**, con el tiempo, **ralentiza su trabajo**.
- Sin embargo, todos los desarrolladores sienten **la presión** de producir **bad code** en muchos casos para cumplir con los plazos.
- La **primera parte** del dilema es la **equivocada**.
  - **Nunca se cumplirán** los plazos generando **bad code**.
  - El **bad code** ralentizará el trabajo y obligará **a no cumplir** con los plazos.
  - La única manera de ir rápido, es **mantener el código** lo más **limpio** posible en **todo** momento.

Clean code: la Regla del Boy Scout

The cleanup doesn't have to be something big.

Change one variable name for the better, break up one function that's a little too large, eliminate one small bit of duplication, clean up one composite if statement.



Nombres Significativos

NOMBRAMOS TODO EN SOFTWARE: VARIABLES, FUNCIONES, PARÁMETROS, CLASES, PAQUETES, ETC. SI LO HACEMOS, HAGÁMOSLO BIEN.

Nombres que revelan la intención

- El nombre de una variable, función o clase debería responder: **por qué existe**, **qué hace** y **cómo se utiliza**.

```
int d; //elapsed time in days
```

- Si un nombre **requiere** un comentario, entonces el nombre **no revela** su **intención**.
- El nombre **d** no revela nada. Se **debe** elegir un nombre que **especifique** lo que se está midiendo y la unidad de esa medida:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Nombres que revelan la intención

- Elegir nombres que **revelen la intención** hace que sea **mucho más fáci**l **comprender** y **cambiar** el código.
- ¿Cuál es el **propósito** de este código?

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

- El **problema** con este código **no es que no sea simple**, sino que **no es explícito**.
  - ¿Qué elementos contiene **`theList`**?
  - ¿Qué significa la **`constante 4`**?
  - ¿Cómo se usa la lista de retorno **`list1`**?
- Las respuestas **deberían** hallarse en el código.

Nombres que revelan la intención

- Elegir nombres que **revelen la intención** hace que sea **mucho más fáci**l **comprender** y **cambiar** el código.
- ¿Cuál es el **propósito** de este código?

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

- Suponga que el **código anterior** implementa un `busca-minas`.
  - **`theList`** representa el tablero. Un mejor nombre sería **`gameBoard`**.
  - Cada **`celda`** se representa por un **`arreglo`**.
  - El primer elemento de ese **`arreglo`** indica el **`status`** de la **`celda`**.
  - El valor **`constante 4`** significa **`flagged`**.

Nombres que revelan la intención

- Un código mejor que el **anterior** podría ser el siguiente.

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
    return flaggedCells;
}
```

- ¿Cambió la **simplicidad** del código?
- Se podría **mejorar** abstrayendo el concepto de celda:

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Con estos simples cambios de nombre, no es difícil entender lo que está pasando.

Éste es el poder de elegir buenos nombres.

Nombres que evitan la desinformación

- Los programadores deben evitar dejar **pistas falsas** que **oscurezcan** el significado del código.
- Se **debe evitar** palabras cuyos significados **varíen** del significado deseado. Por ejemplo

```
int hp, aix, sco;
```

  serían nombres de variables **deficientes** porque son nombres de plataformas o variantes de Unix.
- Incluso si se está codificando una **hipotenusa** y **hp** parece una buena abreviatura, podría resultar desinformativo.

Nombres que evitan la desinformación

- **Se deben evitar** palabras que puedan llevar a una **mala interpretación.**
  - Nombrar como **`accountList`** un conjunto de **cuentas puede ser un problema** a menos que en realidad sea una **`List`**.
  - En este caso, **sería adecuado** simplemente utilizar **`accounts`**.
- **Se deben evitar** palabras que **varían ligeramente.**
  - ¿Cuál es la diferencia entre **`XYZControllerForEfficientHandlingOfStrings`** y **`XYZControllerForEfficientStorageOfStrings`**?
- Nombrar conceptos **similares** de **manera similar** es **información**. Ser **inconsistente** es **desinformación**.
  - Usar a veces **`accounts`** y otras **`listOfAccounts`** es inconsistente (complica la búsqueda y el auto-completado de una variable en particular).

Nombres que evitan la desinformación

- Un ejemplo realmente **terrible** de nombres desinformativos es el siguiente:

```
int a = l;
if ( O == l )
  a = O1;
else
  l = 01;
```

  - El uso de **L** minúscula o **O** mayúscula como nombres de variables, especialmente en combinación, es un **grave error**.
  - El problema es que se **parecen casi por completo** a las constantes **uno** y **cero**, respectivamente.

Nombres con distinciones significativas

- Los nombres de **elementos similares**, **deben ser** claramente **distinguibles**.
  - **No es suficiente** agregar **series numéricas** o **palabras irrelevantes**.

```
a1; a2; account1; account2; accountAux
```

  - Si los **nombres deben ser diferentes**, entonces **también** deberían significar algo diferente.
- Es importante **darle contexto** a los nombres.
- Por ejemplo, **`account1`** y **`account2`** podrían notarse como **`accountOrigin`** y **`accountDestination`** si se trata de una transacción.

Nombres con distinciones significativas

- Los nombres con **palabras irrelevantes** son otra distinción **sin sentido**.
  - Si se tiene una clase **`Product`**, **no ofrece distinción** la definición de clases como **`ProductInfo`** o **`ProductData`**.
- **`Info`** y **`Data`** son palabras irrelevantes y confusas como **a**, **an** y **`the`**.
- No tiene **nada de malo** usar **convenciones** de **prefijos**. Por ejemplo, los prefijos **a** y **`the`** para definir nombres de variables y funciones.
  - Sin embargo, siempre se debe **respetar la convención**.
  - Llamar a una variable **`theZork`** porque ya existe otra **`zork`** es un problema.

Nombres con distinciones significativas

- Los nombres con **palabras irrelevantes** son **redundantes**.
  - Las palabras **`variable`** y **`table`** no deberían aparecer jamás en la definición de una variable o tabla.
  - ¿Es mejor la definición de **`NameString`** que simplemente **`Name`**?
  - ¿Podría **`Name`** ser un número flotante?
- ¿**Cómo se supone** que los programadores de este proyecto sabrán **cuál de estas funciones invocar**?

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

- En **ausencia de convenciones**, la variables **`moneyAmount`**, **`customerInfo`** y **`accountData`** **no se pueden distinguir** de **`money`**, **`customer`** y **`account`**, respectivamente.

If you can't pronounce it, you can't discuss it without sounding like an idiot.

Nombres que sean pronunciables

- Los humanos **son buenos** con las **palabras**. Una parte importante de el **cerebro** humano está dedicada a las palabras. Y las **palabras son**, por definición, **pronunciables**.
- ¿**Cómo** pronunciaría esta forma de notar la generación de date, year, month, day, hour, minute, and second?

genymdhms

- ¿Qué código **preferiría** para **presentar** ante su equipo de trabajo?

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private String pszqint = "102";
  /* ... */
};
```

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;
  private String recordId = "102";
  /* ... */
};
```

Nombres fáciles de buscar

- Los identificadores de **una sola letra** y las **constantes numéricas** son **difíciles de buscar** a través del texto.
  - Estas deberían utilizarse **únicamente** como **variables locales** a métodos **cortos**.

```
for (i=1; i<10; i++){
   …
}
```

  - En general, **la longitud** de un identificador debería **corresponder** al tamaño del **scope** al que pertenece.

Nombres fáciles de buscar

- Si una variable o constante **se puede ver o usar** desde **múltiples** lugares del cuerpo del código, es **imperativo** darle un nombre *search-friendly*.
- Compare los siguientes fragmentos de código

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

```
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;
}
```

If you can't pronounce it, you can't discuss it without sounding like an idiot.

Evitar encodings

- Los programadores **cargan con suficientes** codificaciones **al operar un código** como para lidiar con *encodings* adicionales obligatorios.
- Si el código es limpio, no se requiere nada adicional.
  - **Hungarian Encoding**: fue exitoso su uso en lenguajes que no contaban con verificación de tipos.
  - **Prefijos de miembros** : m_attribute, mAttribute.
- ¿Cómo **distinguir** una **interfaz** de una **clase concreta**? Siempre es preferible mantener el nombre de la interfaz limpio.

```
interface ShapeFactory
class ShapeFactoryImpl implements ShapeFactory
```

Es preferible a

```
interface IShapeFactory
class ShapeFactory implements IShapeFactory
```

Evitar mapeo mental

- Quien lee el código **no debería** tener que **traducir** nombres mentalmente.
- Por ejemplo, una variable llamada **`cantAl`** que modela la **cantidad de alumnos** necesitaría un **mapeo mental** cada vez que se lee.

> *Los programadores son personas **bastante inteligentes**, y a veces les **gusta mostrar su inteligencia** demostrando sus habilidades de **malabarismo mental**. Por ejemplo, recordando que **r** es la versión en minúsculas de la **URL** sin el host y el esquema.*

*Una diferencia entre un **programador inteligente** y un **programador profesional** es que el **profesional entiende** que **la claridad es la clave**. Los **profesionales** usan sus poderes para el bien y escriben código que otros **pueden entender**.*

Nombres de clases y métodos

If you can't pronounce it, you can't discuss it without sounding like an idiot.

- Las **clases** y **objetos deben tener** nombres **sustantivos** o **frases sustantivas** como **`Costumer`**, **`WikiPage`**, **`Account`** o **`AddressParser`**.
  - Evitar nombres como **`Manager`**, **`Processor`**, **`Data`** o **`Info`** (suelen ser ambiguos, redundantes, o sin distinción significativa).
  - El nombre de una clase **nunca debería** ser un **verbo**.

- Las **métodos deben tener** nombres que son **verbos** o **frases verbales** como **`postPayment`**, **`WikiPage`**, **`deletePage`** o **`save`**.
- Métodos de **acceso**, **modificación** y **predicados** deberían tener como prefijo **`get`**, **`set`**, **`is`**, de acuerdo el standard **javabean.**

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

Don’t Be Cute

- Ser siempre **serio** con los nombres **que se eligen**.
- **No necesariamente** todos los posibles lectores del código **comparten el mismo humor**, y aún así, la elección de los nombres puede no declarar la intención.

```
private void imprimir_error(Error error){
  System.out.println(“Error: ” + error.getMessage());
}
```

```
imprimir_error(new MA2_old_man_excepcion());
```

If you can't pronounce it, you can't discuss it without sounding like an idiot.

Elegir una palabra por concepto

- Elegir **una palabra por concepto** abstracto y mantenerlo.
  - Es confuso tener **`fetch`**, **`retrieve`** y **`get`** como **métodos equivalentes** en **diferentes clases**.
  - **`Manager`**, **`Controller`**, **`Driver`** no deberían representar lo mismo.
- Este es un aspecto importante para **ponerse de acuerdo** con el equipo de trabajo.

Evitar igual palabra para conceptos diferentes

- Seguir **la regla anterior** puede llevar a que varias clases tengan por ejemplo, un mismo método **add**.
- Lo importante es mantener el sentido de **add** uniforme. Por ejemplo, **add** en una clase de manejo de **`Strings`** puede significar **concatenar**, y en una colección **agregar un elemento**.
- En el caso anterior, **sería mejor** renombrar el **add** de **`String`** como **append**.

Usar nombres del dominio: solución vs problema

If you can't pronounce it, you can't discuss it without sounding like an idiot.

- Siempre **es recomendable** utilizar nombres que se encuentran asociados al **dominio** en el que la solución se está dando.
- Como los **lectores** del código **son programadores**, es ventajoso, siempre que sea posible, usar términos de la Ciencias de la Computación.
  - Por ejemplo, nombres de **algoritmos** conocidos, **patrones**, términos **matemáticos**, etc.
- Cuando **no hay un concepto técnico** que declarar en el nombre, utilizar nombres del **dominio** del **problema**.
  - Por ejemplo, utilizar nombres definidos en los **casos de uso**.

Agregar contexto significativo

- Algunos nombres **son significativos por sí mismos**. Sin embargo, en general, los **nombres necesitan contexto**.
  - Por ejemplo, las variables **`firstName`**, **`lastName`**, **`street`**, **`houseNumber`**, **`city`**, **`state`** y **`zipcode`** tienen un significado claro **en conjunto**.
  - ¿Qué sucede cuando se utiliza el nombre **`state`** por sí solo?
  - No es claro que sea el **`state`** de una **dirección**, podría referirse al estado de un **objeto**.
- Se puede **agregar contexto** mediante el uso de **prefijos**: **`addressState`**.
- Una mejor opción, definir la clase **`Address`** (las clases dan contexto).

No agregar contexto no significativo

If you can't pronounce it, you can't discuss it without sounding like an idiot.

- Si se trabaja en una aplicación llamada **Gas Station Deluxe** es **una mala idea** usar el **prefijo GSD** en cada clase.
- De manera similar, **no abusar** del nombre de una **super clase** cuando el nombre de la **clase derivada** puede ser lo suficientemente claro.
  - Por ejemplo, cuando se requiere modelar la clase base **`Address`** y se requiere diferenciar direcciones **`MAC`** y **`Web`**.
  - **`MAC`** y **`URI`** son mejores elecciones que **`MACAddress`**, **`WebAddress`**.

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