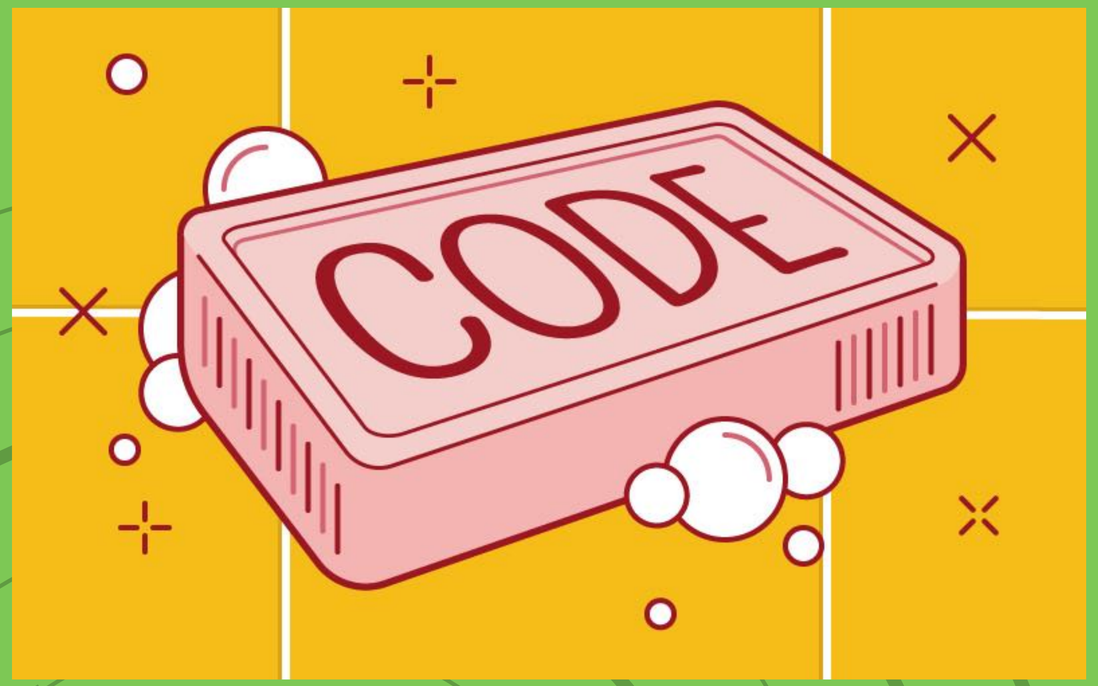
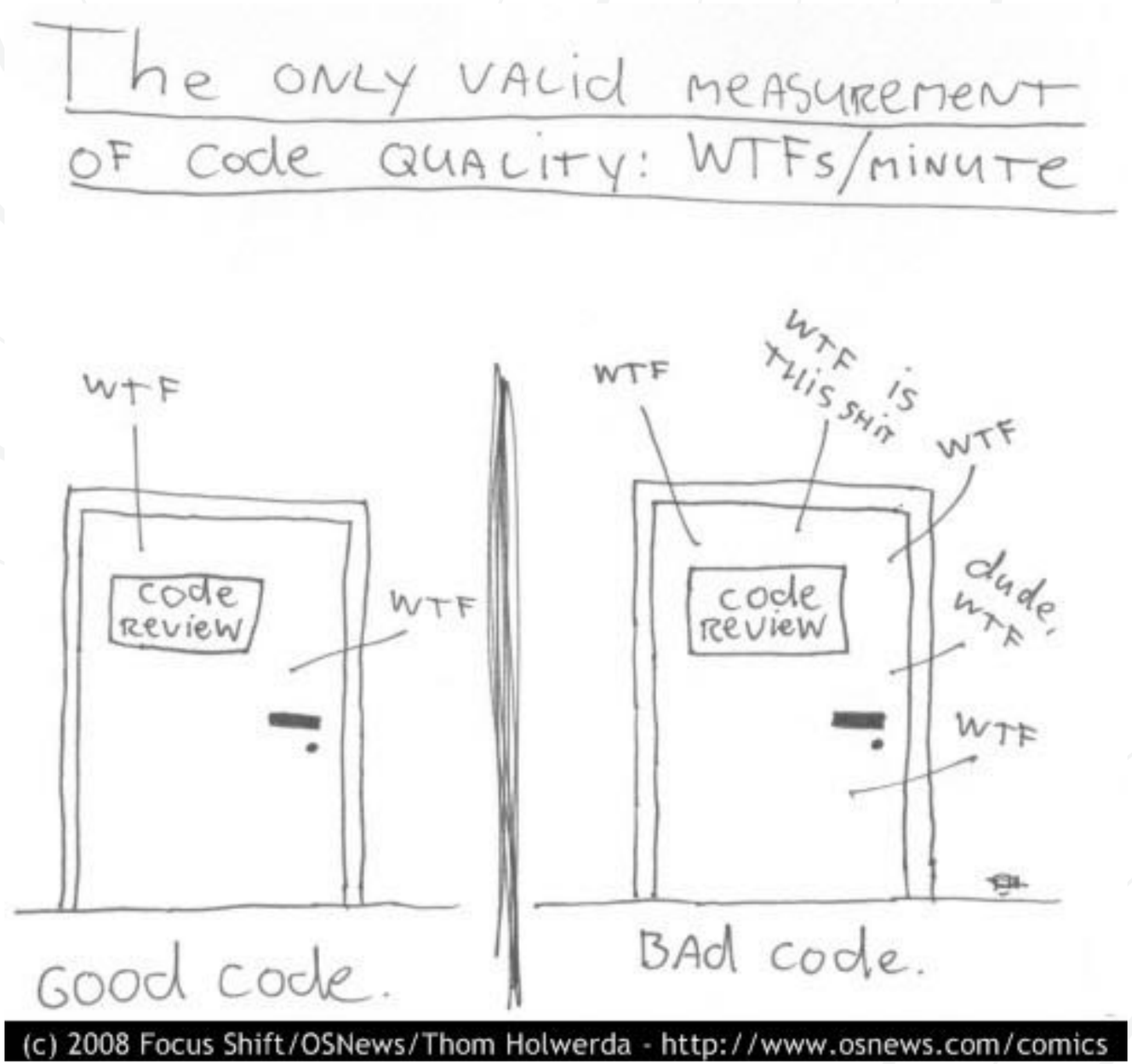
# Clean Code




emmanuel.lagarrigue@cs.uns.edu.ar

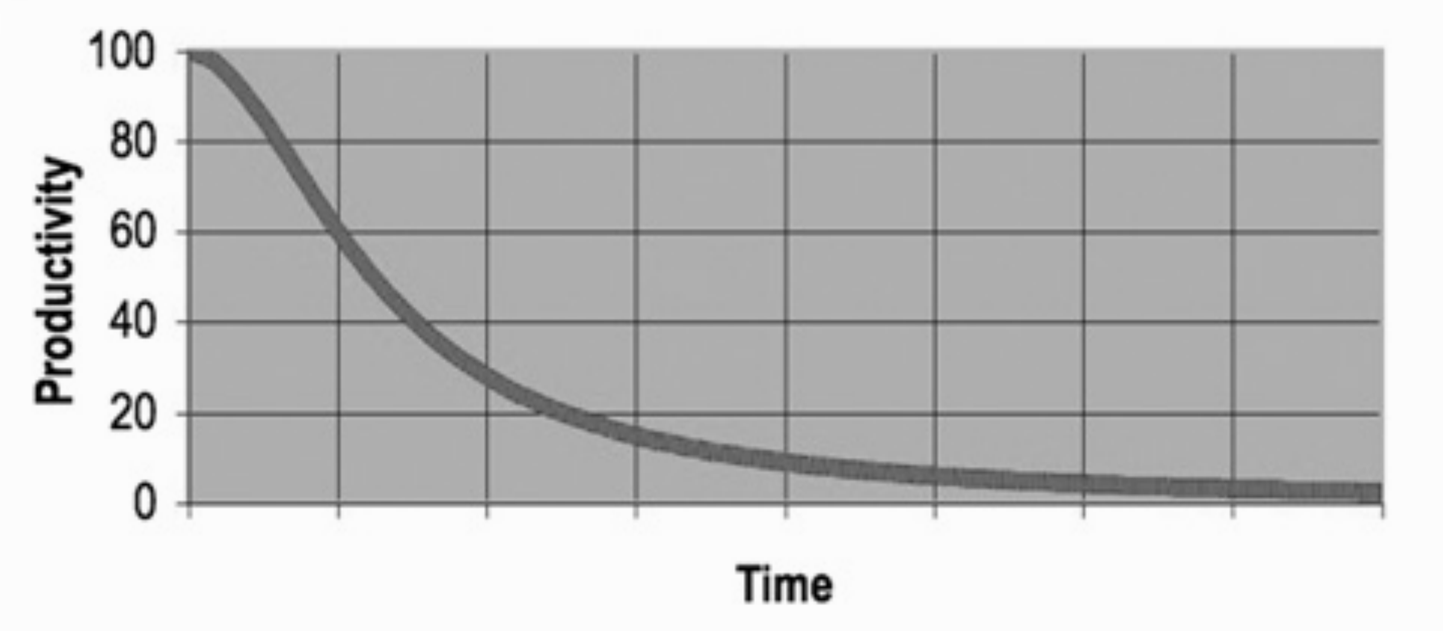

The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF is THIS SHIT
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

# Clean Code

- Es una serie de recomendaciones y buenas prácticas sobre cómo debería escribirse el código para que sea **fácil de interpretar y modificar**.
- “ Clean code always looks like it was written by **someone who cares**.”
- Es el resultado de un trabajo **profesional**.

# ¿Es necesario?

- Que el código funcione, es suficiente?
- El costo de mantener un código fuente de mala calidad puede ser muy alto!




THE BOY SCOUT RULE
ALWAYS
LEAVE
CODE
CLEANER
THAN YOU
FOUND IT!

# Nombres Significativos


- ¿Dónde usamos nombres en el software?

# Use Intention-Revealing Names

- El nombre de una variable debería decirnos porqué existe, qué hace, y cómo se utiliza.
- Si un nombre requiere de un comentario, el nombre no revela la intención.

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

- El nombre `d` no revela nada. No evoca el sentido de tiempo transcurrido, ni de días.
- El nombre debería especificar qué se esta midiendo, y en qué unidad de medida:

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

- Elegir nombres que revelan intención puede hacer el código mucho mas sencillo de entender y cambiar.
- ¿Cuál es el propósito del siguiente 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 no es que no sea **simple**, sino que no es **explícito**.
- Entender este código requiere que sepamos:
  - Qué tipo de elementos contiene `theList`
  - Cuál es el significado del elemento 0 de un item en `theList`.
  - Cuál es el significado de la constante `4`.
  - Cómo se usa la lista de retorno.

- Las respuestas a las preguntas podrían haber estado presentes en el código.
- Supongamos que que el código pertenece a una implementación del “busca minas”.
- El tablero está representado por una lista de celdas, llamada `theList`. Un nombre mejor sería `gameBoard`.
- Cada celda se representa por un arreglo. El 1er elemento de ese arreglo indica el status de la celda, y el valor 4 significa “flagged”.

- Sólo con indicar esos conceptos en los nombres, la legibilidad del código mejora considerablemente:

```
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 aún más abstrayendo el concepto de celda a una clase, y haciendo más declarativa la condición del if mediante un método en celda:

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

# Evitar la desinformación

- Evitar palabras que puedan llevar a una mala interpretación de su significado.
- Por ejemplo, no nombrar a un grupo de “cuentas” `accountList`, a menos que efectivamente sea de tipo `List`.
- En ese caso, es mejor llamar a la variable simplemente `accounts`.

- Nombrar conceptos similares de manera similar es información.
- Ser inconsistente es desinformación.
- Por ejemplo, usar a veces `accounts` y otras `listOfAccounts` (complica la búsqueda de una variable en particular).

# Distinciones significativas

- Cuando tenemos elementos similares debemos indicar significativamente esa distinción.
- Por ejemplo, cuando tenemos las variables a las que se la agrega un número al final: `account1`, `account2`.
- Es importante darle con contexto, y por ejemplo llamarlas `accountOrigin`, `accountDestination` si se trata de una transacción.

# Identificadores que sean fáciles de buscar

- Los identificadores de una sola letra y las constantes numéricas son difíciles de buscar a través del texto.
- En general, la longitud de un identificador debería corresponder al tamaño del scope al que pertenece.

- Si una variable o constante se puede ver o usar desde múltiples lugares, es imperativo darle un nombre “search-friendly”

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

vs

```
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;
}
```

# Evitar encodings

- Si el código es limpio, no hace faltan encodings.
- Ejemplos:
  - Hungarian Encoding
  - Prefijos de miembros (`m_attribute, mAttribute`)

- En el caso de interfaces y sus implementaciones, es preferible mantener el nombre de la interfaz limpio.
- Por ejemplo:

```
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 contenga “cantidad de alumnos” necesitaría un mapeo mental cada vez que se lee.

# Nombres de Clases

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

# Nombres de Métodos

- Las métodos deben tener como nombre verbos o frases verbales como `postPayment, 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.

```
public void yoNoTePoCree(Error e) {
    System.out.println(“Error ” +  e.getMessage());
}
```

# Elegir una palabra por concepto

- Elegir una palabra por concepto abstracto y mantenerlo. Por ejemplo, es confuso tener `fetch`, `retrieve` y `get` como métodos equivalentes en diferentes clases. `Manager`, `Controller`, `Driver` es otro ejemplo.
- Es importante ponerse de acuerdo con el equipo.

# Evitar usar la misma palabra para dos 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 a “`append`” por ejemplo.

# Usar Nombres del Dominio de la Solución

- Los lectores del código van a ser programadores, por lo que es ventajoso usar términos de la Ciencia de la Computación.
- Por ejemplo, nombres de algoritmos conocidos, patrones, términos matemáticos, etc.

# Usar Nombres del Dominio del Problema

- 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. En general, los nombres necesitan contexto.
- Por ejemplo, las variables `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` y `zip` code tienen un significado claro en conjunto.
- ¿Qué pasaría si viéramos el nombre `state` por sí solo? No es claro que sea el state de una dirección, podría referirse al estado del objeto.

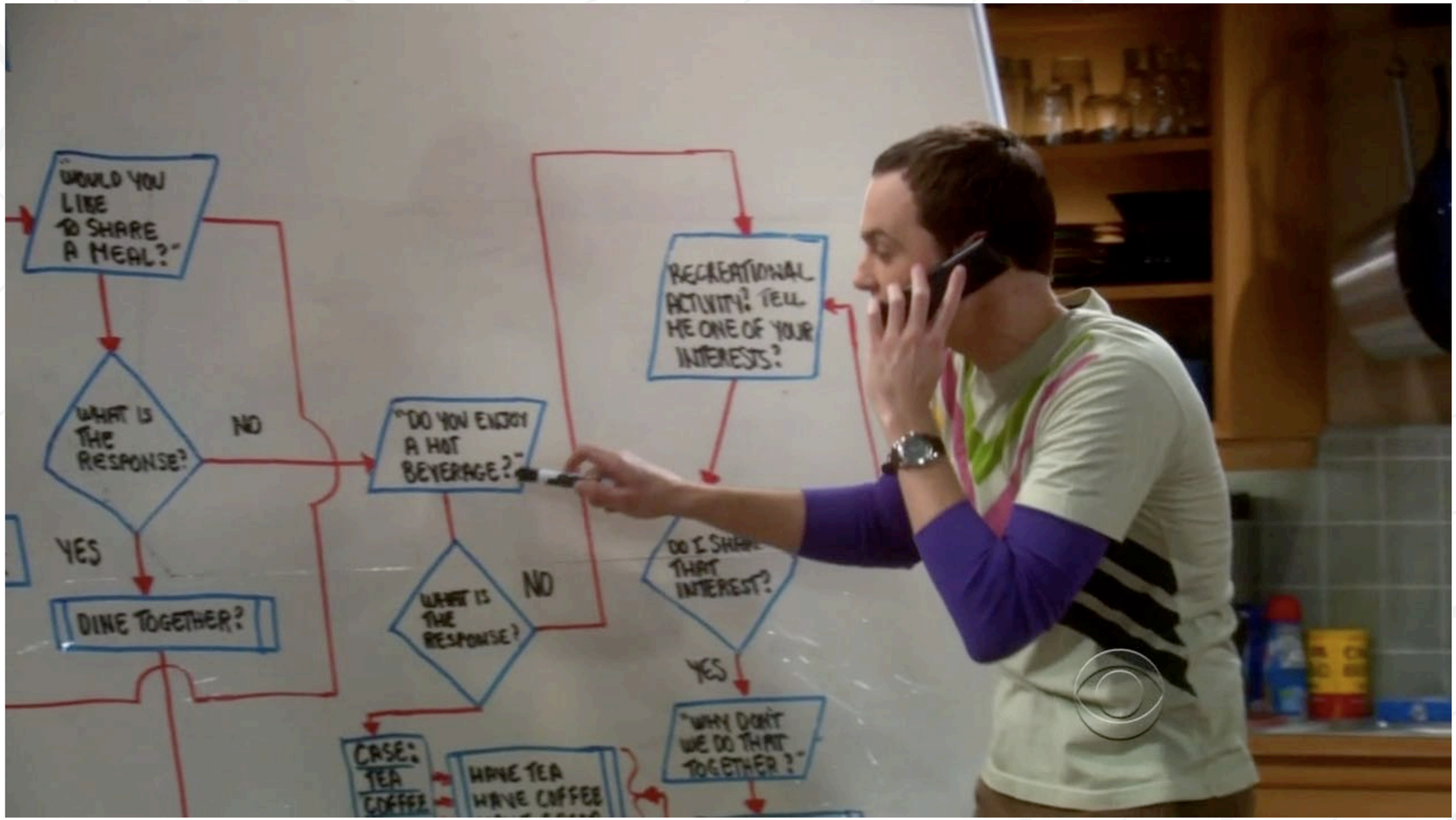
- Se puede agregar contexto mediante el uso de prefijos: `addressState`.
- Una mejor opción, definir la clase `Address` (las clases dan contexto).

# No Agregar Contexto Gratis

- Si trabajamos en una aplicación llamada “Gas Station Deluxe” es una mala idea usar el prefijo `GSD` en cada clase.
- No aporta información y complica las búsquedas y el autocomplete.

- 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, supongamos la clase base Address y la necesidad de diferenciar direcciones MAC y Web
- MAC y URI son mejores elecciones que MACAddress, WebAddress

# Funciones


"WOULD YOU LIKE TO SHARE A MEAL?"
WHAT IS THE RESPONSE?
NO
YES
DINE TOGETHER?
"DO YOU ENJOY A HOT BEVERAGE?"
WHAT IS THE RESPONSE?
NO
CASE: TEA
HAVE TEA
RECREATIONAL ACTIVITY? TELL ME ONE OF YOUR INTERESTS?
DO I SHARE THAT INTEREST?
YES
"WHY DON'T WE DO THAT TOGETHER?"

# Small!

- 1ra regla: las funciones deberían ser pequeñas.
- 2da regla: las funciones deberían ser aún más pequeñas!
- ¿Qué tan pequeñas? Lo mínimo indispensables para las funciones **hagan una sola cosa**.

# Do One Thing

- Las funciones son grandes cuando hacen más de una cosa.

***FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY.***

# Un Solo Nivel de Abstracción por Función

- Para asegurarnos que nuestras funciones hacen una sola cosa, tenemos que asegurarnos que de las sentencias dentro de nuestra función están al mismo nivel de abstracción.
- Por ejemplo:
  - `getHtml()`
  - `String pagePathName = PathParser.render(pagePath);`
  - `.append("\n")`

- No es una tarea trivial mantener el mismo nivel de abstracción, a veces es confuso y sutil.
- Hay que enfocarse en los conceptos y no mezclarlos.
- El objetivo es que el código sea simple y fácil de leer. Queremos que el código se lea como una narrativa de arriba hacia abajo: **The Stepdown Rule**.
- Queremos que cada función sea seguido por las funciones del siguiente nivel de abstracción.

# Ejemplo

# Usar nombres descriptivos

- El nombre de una función debe indicar su intención! El nombre debe indicar qué es lo que hace (una sola cosa).
- A mayor nivel de abstracción, el nombre parece indicar que hace más de una cosa en algunos casos, pero conceptualmente debe ser una.
- No tener miedo de usar nombres largos.
- Ser consistentes, usar el mismo tipo de frases, nombres, verbos, etc.

# Argumentos

- La cantidad ideal de argumentos es **0**!
- La siguiente mejor, 1, seguido de 2…
- 3 argumentos deben ser evitados…
- La razón es la **legibilidad**. Mientras mayor es la cantidad de argumentos, menos clara es la intención de la función.
- Por ejemplo,
  - `includeSetupPage()` es más claro que
  - `includeSetupPageInto(newPage Content)`
- Además, la complejidad de los tests unitarios crece exponencialmente junto con el número de parámetros.
- Argumentos output deben ser evitados.

- Razones para utilizar funciones de un solo argumento:
  - Cuando se hace una pregunta sobre el argumento, por ejemplo `boolean fileExists(“MyFile”)`
  - Cuando se realiza una operación sobre el argumento, y se espera un valor de resultado, por ejemplo `InputStream fileOpen(“MyFile”)`
- Debemos evitar funciones monádicas que no sigan ese formato.
- Si una función va a trasformar un input, es mejor retornarlo explícitamente en vez de modificar el input. Por ejemplo `StringBuffer transform(StringBuffer in)` es mucho más claro que `void transform(StringBuffer out)`

- **Banderas como Argumentos**: Evitarlos!! Es una clara indicación de que la función hace al menos dos cosas diferentes.

```
float getFinalSaleAmount(Product product, int cant, boolean applyIVA)
{
   float amount = product.getPrice() * cant;
   if(applyIVA) amount *= 1.21;
   return amount;
}
```

```
        ******   vs   ******
float getSaleAmount(Product product, int cant){
   return product.getPrice() * cant;
}
float getSaleAmountWithIVA(Product product, int cant){
   return getSaleAmount(product, cant) * 1.21;
}
```

- **Objeto Argumento**: Cuando indefectiblemente se necesitan varios argumentos, si la función hace una sola cosa (como debería), los argumentos deberían estar relacionados. En este caso, es preferible abstraer los argumentos en un clase. Por ejemplo:

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

# Evitar los efectos secundarios

- Una función debe hacer **una sola cosa**. Los efectos secundarios no solo implican una segunda acción, sino que además atentan contra la legibilidad e intención de la función.
- Los efectos secundarios son semilleros de bugs.

```
public class UserValidator {
  private Cryptographer cryptographer;

  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, no ambas cosas.
- Una función debería cambiar el estado de un objeto, o (xor) 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");
   ...
}
```

# DRY: Don’t Repeat Yourself

- Nunca hay que duplicar código!!! **Siempre** tenemos que abstraer el comportamiento general.
- A veces la repetición no es tan obvia. Los patrones de diseño ayudan a solucionar este problema.

# ¿Cómo hacemos para escribir código siguiendo estas prácticas?

- Es muy difícil seguir todas estas reglas cuando se empieza a escribir código.
- El truco está en ir mejorando el código en etapas. Escribir, hacer funcionar, refactorizar para dejarlo mejor (regla del boy scout sobre nuestro propio código).