

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Clase 03: Clean Code (Cap. 1 y 2)

Francisco Gazitúa Requena (fco_gazitua_requena@uc.cl)

IIC2113 Diseño Detallado de Software

12 de Agosto, 2025

Anuncios

Anuncios

Hubo un cambio en la fecha de la interrogación:

- **Fecha**: 24/11 a las 17:30

No hay cambios en las fechas de otras evaluaciones ni en el formato de la interrogación

Clean Code



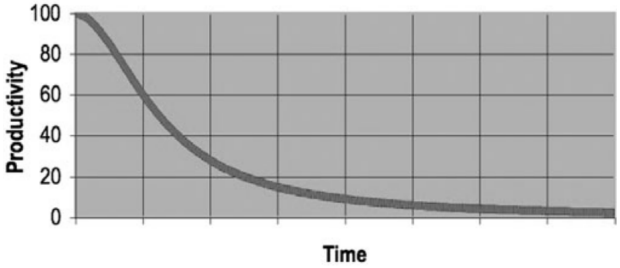
Clean code

"*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.*"

– Robert C. Martin.

Clean code

Motivación: "*good code matters.*"



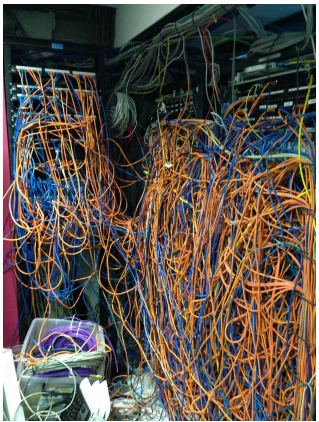
Clean code

Ser profesional implica preocuparse por la calidad del código que escribimos.

Clean code

Ser profesional implica preocuparse por la calidad del código que escribimos.





Clean code

Escribir código limpio es un arte que toma tiempo (y disciplina) perfeccionar.

Clean code

Escribir código limpio es un arte que toma tiempo (y disciplina) perfeccionar.

... y el primer paso es saber qué significa que un código esté limpio.

¿Qué es el código limpio?

¿Qué es el código limpio?

Bjarne Stroustrup, inventor of C++ and author of *The C++ Programming Language*

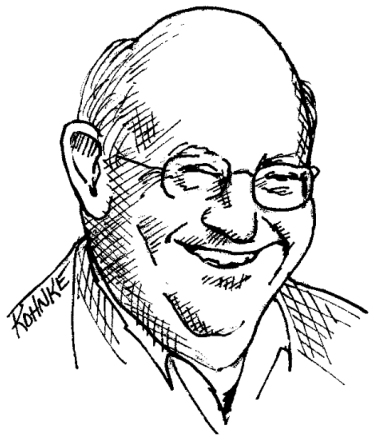
> *I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.*



¿Qué es el código limpio?

“Big” Dave Thomas, founder of OTI, godfather of the Eclipse strategy

Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.



¿Qué es el código limpio?

Michael Feathers, author of *Working Effectively with Legacy Code*

> *I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them. Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better. All of those things were thought about by the code's author, and if you try to imagine improvements, you're led back to where you are, sitting in appreciation of the code someone left for you—code left by someone who cares deeply about the craft.*



Clean code

No existe una definición única de código limpio, pero sí hay consenso en algunas cosas.

Clean code

No existe una definición única de código limpio, pero sí hay consenso en algunas cosas.

- Trabajar con código limpio aumenta la productividad (en el mediano/largo plazo).

Clean code

No existe una definición única de código limpio, pero sí hay consenso en algunas cosas.

- Trabajar con código limpio aumenta la productividad (en el mediano/largo plazo).
- Código sucio fomenta más código sucio.

Clean code

No existe una definición única de código limpio, pero sí hay consenso en algunas cosas.

- Trabajar con código limpio aumenta la productividad (en el mediano/largo plazo).
- Código sucio fomenta más código sucio.
- Un código está limpio cuando alguien más es capaz de entenderlo fácilmente.

Clean code

No existe una definición única de código limpio, pero sí hay consenso en algunas cosas.

- Trabajar con código limpio aumenta la productividad (en el mediano/largo plazo).
- Código sucio fomenta más código sucio.
- Un código está limpio cuando alguien más es capaz de entenderlo fácilmente.
- El primer paso para escribir código limpio es entender qué aspectos hacen que el código esté limpio (esto es fácil).

Clean code

No existe una definición única de código limpio, pero sí hay consenso en algunas cosas.

- Trabajar con código limpio aumenta la productividad (en el mediano/largo plazo).
- Código sucio fomenta más código sucio.
- Un código está limpio cuando alguien más es capaz de entenderlo fácilmente.
- El primer paso para escribir código limpio es entender qué aspectos hacen que el código esté limpio (esto es fácil).
- El segundo paso es aprender a programar códigos limpios (esto es difícil).

Meaningful Names

by Tim Ottinger



Names

"Selecting names for variables, methods, and other entities is one of the most underrated aspects of software design.

"

– John Ousterhout

Names

"Selecting names for variables, methods, and other entities is one of the most underrated aspects of software design. Good names are a form of documentation: they make code easier to understand. They reduce the need for other documentation and make it easier to detect errors.

"

– John Ousterhout

Names

"Selecting names for variables, methods, and other entities is one of the most underrated aspects of software design. Good names are a form of documentation: they make code easier to understand. They reduce the need for other documentation and make it easier to detect errors. Conversely, poor name choices increase the complexity of code and create ambiguities and misunderstandings that can result in bugs.

"

– John Ousterhout

Names

"Selecting names for variables, methods, and other entities is one of the most underrated aspects of software design. Good names are a form of documentation: they make code easier to understand. They reduce the need for other documentation and make it easier to detect errors. Conversely, poor name choices increase the complexity of code and create ambiguities and misunderstandings that can result in bugs. Name choice is an example of the principle that complexity is incremental.

"

– John Ousterhout

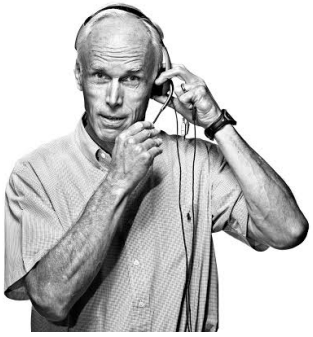
Names

"Selecting names for variables, methods, and other entities is one of the most underrated aspects of software design. Good names are a form of documentation: they make code easier to understand. They reduce the need for other documentation and make it easier to detect errors. Conversely, poor name choices increase the complexity of code and create ambiguities and misunderstandings that can result in bugs. Name choice is an example of the principle that complexity is incremental. Choosing a mediocre name for a particular variable, as opposed to the best possible name, probably won't have much impact on the overall complexity of a system. However, software systems have thousands of variables; choosing good names for all of these will have a significant impact on complexity and manageability."

– John Ousterhout

Malos nombres causan bugs

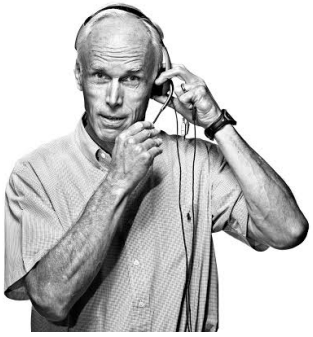
Names



John Ousterhout
Stanford University

"The most challenging bug I ever fixed came about because of a poor name choice. In the late 1980's and early 1990's my graduate students and I created a distributed operating system called Sprite. At some point we noticed that files would occasionally lose data: one of the data blocks suddenly became all zeroes, even though the file had not been modified by a user."

Names

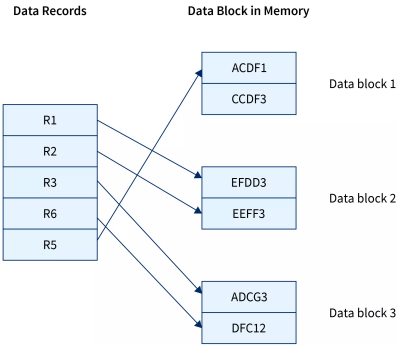


John Ousterhout
Stanford University

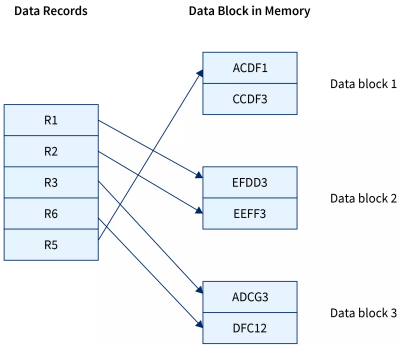
"The most challenging bug I ever fixed came about because of a poor name choice. In the late 1980's and early 1990's my graduate students and I created a distributed operating system called Sprite. At some point we noticed that files would occasionally lose data: one of the data blocks suddenly became all zeroes, even though the file had not been modified by a user."

"It took six months, but I eventually found and fixed the bug. The problem was actually quite simple (as are most bugs, once you figure them out)."

Names

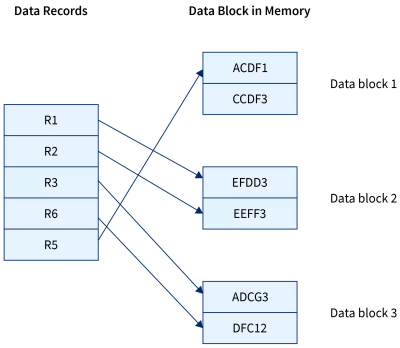

Data Records
Data Block in Memory
R1
R2
R3
R6
R5
ACDF1
CCDF3
Data block 1
EFDD3
EEFF3
Data block 2
ADCG3
DFC12
Data block 3

Names



"The file system code used the variable name `block` for two different purposes. In some situations, `block` referred to a physical block number on disk; in other situations, `block` referred to a logical block number within a file"

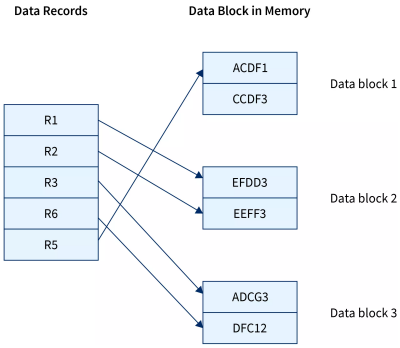
Names



"The file system code used the variable name `block` for two different purposes. In some situations, `block` referred to a physical block number on disk; in other situations, `block` referred to a logical block number within a file"

¡Seis meses costó encontrar el bug producto de un nombre poco descriptivo!

Names



"The file system code used the variable name `block` for two different purposes. In some situations, `block` referred to a physical block number on disk; in other situations, `block` referred to a logical block number within a file"

¡Seis meses costó encontrar el bug producto de un nombre poco descriptivo! **¡Seis meses!**

Names

“Unfortunately, most developers don’t spend much time thinking about names.

Names

"Unfortunately, most developers don't spend much time thinking about names. They tend to use the first name that comes to mind, as long as it's reasonably close to matching the thing it names.

Names

"Unfortunately, most developers don't spend much time thinking about names. They tend to use the first name that comes to mind, as long as it's reasonably close to matching the thing it names. For example, block is a pretty close match for both a physical block on disk and a logical block within a file; it's certainly not a horrible name.

Names

"Unfortunately, most developers don't spend much time thinking about names. They tend to use the first name that comes to mind, as long as it's reasonably close to matching the thing it names. For example, block is a pretty close match for both a physical block on disk and a logical block within a file; it's certainly not a horrible name. Even so, it resulted in a huge expenditure of time to track down a subtle bug.

Names

"Unfortunately, most developers don't spend much time thinking about names. They tend to use the first name that comes to mind, as long as it's reasonably close to matching the thing it names. For example, block is a pretty close match for both a physical block on disk and a logical block within a file; it's certainly not a horrible name. Even so, it resulted in a huge expenditure of time to track down a subtle bug. Thus, you shouldn't settle for names that are just "reasonably close." Take a bit of extra time to choose great names, which are precise, unambiguous, and intuitive.

Names

"Unfortunately, most developers don't spend much time thinking about names. They tend to use the first name that comes to mind, as long as it's reasonably close to matching the thing it names. For example, block is a pretty close match for both a physical block on disk and a logical block within a file; it's certainly not a horrible name. Even so, it resulted in a huge expenditure of time to track down a subtle bug. Thus, you shouldn't settle for names that are just "reasonably close." Take a bit of extra time to choose great names, which are precise, unambiguous, and intuitive. The extra attention will pay for itself quickly, and over time you'll learn to choose good names quickly."

Names

"Unfortunately, most developers don't spend much time thinking about names. They tend to use the first name that comes to mind, as long as it's reasonably close to matching the thing it names. For example, block is a pretty close match for both a physical block on disk and a logical block within a file; it's certainly not a horrible name. Even so, it resulted in a huge expenditure of time to track down a subtle bug. Thus, you shouldn't settle for names that are just "reasonably close." Take a bit of extra time to choose great names, which are precise, unambiguous, and intuitive. The extra attention will pay for itself quickly, and over time you'll learn to choose good names quickly."

Ahora veamos algunas reglas básicas para elegir buenos nombres.

Names

Regla 1: Use Intention-Revealing Names.

Names

Regla 1: Use Intention-Revealing Names.

El nombre de una variable/función/clase debe responder las 3 grandes preguntas:

1. ¿Por qué existe?
2. ¿Qué hace?
3. ¿Cómo se usa?

Names

Regla 1: Use Intention-Revealing Names.

El nombre de una variable/función/clase debe responder las 3 grandes preguntas:

1. ¿Por qué existe?
2. ¿Qué hace?
3. ¿Cómo se usa?

Ejemplo:

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

Names

Regla 1: Use Intention-Revealing Names.

El nombre de una variable/función/clase debe responder las 3 grandes preguntas:

1. ¿Por qué existe?
2. ¿Qué hace?
3. ¿Cómo se usa?

Ejemplo:

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

Si necesita un comentario entonces es un mal nombre. `d` es un mal nombre porque no revela nada sobre la variable. No hace referencia ni a tiempo ni a días.

Names

Regla 1: Use Intention-Revealing Names.

El nombre de una variable/función/clase debe responder las 3 grandes preguntas:

1. ¿Por qué existe?
2. ¿Qué hace?
3. ¿Cómo se usa?

Deberíamos usar un nombre que especifique qué se está midiendo y en qué unidades.

```
2 int elapsedTimeInDays;
3 int daysSinceCreation;
4 int daysSinceModification;
5 int fileAgeInDays;
```

Names

Regla 1: Use Intention-Revealing Names.

¿Cuál es el propósito de este código?

```
2 public List<int[]> getThem() {
3   List<int[]> list1 = new ArrayList<int[]>();
4   foreach (int[] x in theList)
5     if (x[0] == 4)
6       list1.add(x);
7   return list1;
8 }
```

Names

Regla 1: Use Intention-Revealing Names.

He aquí el mismo código pero con mejores nombres.

```
2 public List<int[]> getFlaggedCells() {
3   List<int[]> flaggedCells = new ArrayList<int[]>();
4   foreach (int[] cell in gameBoard)
5     if (cell[STATUS_VALUE] == FLAGGED)
6       flaggedCells.add(cell);
7   return flaggedCells;
8 }
```

¿Cuál es su propósito?

Names

Regla 1: Use Intention-Revealing Names.

El código lo podemos seguir mejorando si incluimos una clase `Cell`:

```
2 public List<Cell> getFlaggedCells() {
3   List<Cell> flaggedCells = new ArrayList<Cell>();
4   foreach (Cell cell in gameBoard)
5     if (cell.isFlagged())
6       flaggedCells.add(cell);
7   return flaggedCells;
8 }
```

Names

Regla 2: Avoid Disinformation.

Ten cuidado con los nombres que puedan ser mal interpretados.

Names

Regla 2: Avoid Disinformation.

Ten cuidado con los nombres que puedan ser mal interpretados.

Ejemplos:

Los nombres nos dan una idea de la naturaleza de lo programado. Por ejemplo, `accountList` sugieren que la variable es de tipo `List`.

```
1 public void StartRound(Player attackinPlayer)
2 {
3
4
5 }
```

Names

Regla 2: Avoid Disinformation.

Ten cuidado con los nombres que puedan ser mal interpretados.

Ejemplos:

Los nombres nos dan una idea de la naturaleza de lo programado. Por ejemplo, `accountList` sugieren que la variable es de tipo `List`.

```
1 public void StartRound(Player attackinPlayer)
2 {
3     _view.WriteLine(
4         $"Round {attackinPlayer.Round}: {attackinPlayer.ActiveUnit().
    Name} (Player {attackinPlayer.Id}) comienza");
5 }
```

Names

Regla 2: Avoid Disinformation.

Ten cuidado con los nombres que puedan ser mal interpretados.

Ejemplos:

Nombres que varían en detalles chicos son fáciles de confundir:

```
1 private List<string>
      GetAllHandCardsFromCurrentTurnPlayerInStringFormat(){...}
2 private List<string>
      GetAllRingAreaCardsFromCurrentTurnPlayerInStringFormat()(){...}
3 private List<string>
      GetAllRingAreaCardsFromOppositeTurnPlayerInStringFormat(){...}
4 private List<string>
      GetAllRingSideCardsFromOppositeTurnPlayerInStringFormat(){...}
```

Names

Regla 2: Avoid Disinformation.

Ten cuidado con los nombres que puedan ser mal interpretados.

Ejemplos:

Escribir conceptos similares de forma similar es información. Escribirlos inconsistentemente es desinformación.

```
1 public string name;
2 public string _lastName;
3 private int _accountBalance;
4 private string _encriptedPassword;
```

Names

Regla 3: Make Meaningful Distinctions.

Names

Regla 3: Make Meaningful Distinctions.

Muchas veces damos nombres distintos a variables solo porque el compilador nos obliga.

Names

Regla 3: Make Meaningful Distinctions.

Muchas veces damos nombres distintos a variables solo porque el compilador nos obliga. En esos momentos, recurrimos a las siguientes malas prácticas:

- **Números**: `a1`, `a2`, `a3`, `a4`, ...

Names

Regla 3: Make Meaningful Distinctions.

Muchas veces damos nombres distintos a variables solo porque el compilador nos obliga. En esos momentos, recurrimos a las siguientes malas prácticas:

- **Números**: a1, a2, a3, a4, ...

```
2 public static void copyChars(char a1[], char a2[]) {
3   for (int i = 0; i < a1.length; i++) {
4     a2[i] = a1[i];
5   }
6 }
```

Names

Regla 3: Make Meaningful Distinctions.

Muchas veces damos nombres distintos a variables solo porque el compilador nos obliga. En esos momentos, recurrimos a las siguientes malas prácticas:

- **Números**: a1, a2, a3, a4, ...

```
2 public static void copyChars(char source[], char destination[]) {
3   for (int i = 0; i < source.length; i++) {
4     destination[i] = source[i];
5   }
6 }
```

Names

Regla 3: Make Meaningful Distinctions.

Muchas veces damos nombres distintos a variables solo porque el compilador nos obliga. En esos momentos, recurrimos a las siguientes malas prácticas:

- **Números**: `a1`, `a2`, `a3`, `a4`, ...
- **Sinónimos**: Tenemos las clases `Product`, `ProductInfo` y `ProductData`.

Names

Regla 3: Make Meaningful Distinctions.

Muchas veces damos nombres distintos a variables solo porque el compilador nos obliga. En esos momentos, recurrimos a las siguientes malas prácticas:

- **Números**: `a1`, `a2`, `a3`, `a4`, ...
- **Sinónimos**: Tenemos las clases `Product`, `ProductInfo` y `ProductData`.

Esto también afecta a los nombres de métodos (este es un ejemplo real):

- `getActiveAccount()`, `getActiveAccounts()` y `getActiveAccountInfo()`.

Names

Regla 3: Make Meaningful Distinctions.

Muchas veces damos nombres distintos a variables solo porque el compilador nos obliga. En esos momentos, recurrimos a las siguientes malas prácticas:

- **Números**: `a1`, `a2`, `a3`, `a4`, ...
- **Sinónimos**: Tenemos las clases `Product`, `ProductInfo` y `ProductData`.

Esto también afecta a los nombres de métodos (este es un ejemplo real):

- `getActiveAccount()`, `getActiveAccounts()` y `getActiveAccountInfo()`.

En resumen, que las distinciones que hagas entre nombres sean significativas.

Names

Regla 4: Use Searchable Names

Un problema que surge cuando nombramos variables usando una letra o ponemos números en nuestro código es que cuesta encontrar sus apariciones en el código.

Ejemplo:

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

Names

Regla 4: Use Searchable Names

Un problema que surge cuando nombramos variables usando una letra o ponemos números en nuestro código es que cuesta encontrar sus apariciones en el código.

Ejemplo:

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

Names

Regla 4: Use Searchable Names

Un problema que surge cuando nombramos variables usando una letra o ponemos números en nuestro código es que cuesta encontrar sus apariciones en el código.

Ejemplo:

```
int realDaysPerIdealDay = 4;
const int WorkDaysPerWeek = 5;
int sum = 0;
for (int j=0; j < NumberOfTasks; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = realdays / WorkDaysPerWeek;
  sum += realTaskWeeks;
}
```

Aquí los nombres de variables y constantes son fáciles de buscar.

Names

Regla 5: Class Names and Method Names

Clases, variables, atributos y objetos deben ser nombrados sustantivos o frases sustantivas. No uses verbos

```
1 public static class PrintAttributes
2 {
3     public static void Print(Character character, View view) {...}
4 }
```

Names

Regla 5: Class Names and Method Names

Métodos deben ser nombrados usando verbos o frases verbales. No deben ser sustantivos

```
1 public void AttackResult(int damage, Unit attackinUnit, Unit
      defendingUnit) {...}
```

```
1 public void Winner(Player winner) {...}
```

```
1 public void DamageDealt(int damage) {...}
```

Names

Regla 6: Pick One Word per Concept.

Elige una palabra por concepto abstracto y úsala de manera consistente.

Names

Regla 6: Pick One Word per Concept.

Elige una palabra por concepto abstracto y úsala de manera consistente.

- Para obtener valores no uses `fetch`, `retrieve` y `get`. Cásate con uno y se le fiel.
- Lo mismo ocurre con `controller`, `manager` y `driver`.

Names

Regla 6: Pick One Word per Concept.

Elige una palabra por concepto abstracto y úsala de manera consistente.

- Para obtener valores no uses `fetch`, `retrieve` y `get`. Cásate con uno y se le fiel.
- Lo mismo ocurre con `controller`, `manager` y `driver`.

Regla 7: Don't Pun.

Pero **no** uses la misma palabra para dos propósitos distinto.

Names

Regla 6: Pick One Word per Concept.

Elige una palabra por concepto abstracto y úsala de manera consistente.

- Para obtener valores no uses `fetch`, `retrieve` y `get`. Cásate con uno y se le fiel.
- Lo mismo ocurre con `controller`, `manager` y `driver`.

Regla 7: Don't Pun.

Pero **no** uses la misma palabra para dos propósitos distinto.

- Usar `block` para bloques físicos y bloques lógicos causa bugs que toman 6 meses en ser descubiertos.

Names

Regla 8: Use Solution Domain Names.

De ser posible, usa términos computines estándares para nombrar algoritmos, patrones, estructuras de datos, etc. Recuerda que el código será leído por programadores.

- `BinaryTree`, `Dijkstra`, `JobQueue`, `AccountVisitor`, etc...

Names

Regla 8: Use Solution Domain Names.

De ser posible, usa términos computines estándares para nombrar algoritmos, patrones, estructuras de datos, etc. Recuerda que el código será leído por programadores.

- `BinaryTree`, `Dijkstra`, `JobQueue`, `AccountVisitor`, etc...

Regla 9: Use Problem Domain Names.

Cuando no exista un término computín para lo que está haciendo, usa nombres que hagan referencia al dominio del problema que estás resolviendo.

- `Units`, `Damage`, `Bonus`, `Penalty`, etc..

Names

Regla 10: Add Meaningful Context.

Names

Regla 10: Add Meaningful Context.

Por lo general los nombres no son suficientemente explicativos por sí solos – debemos situarlos en un contexto apropiado. Estos contextos son clases, funciones o namespaces que tienen buenos nombres.

Ejemplo:

- El significado de una variable llamada `state` es ambigua por si sola.

Names

Regla 10: Add Meaningful Context.

Por lo general los nombres no son suficientemente explicativos por sí solos – debemos situarlos en un contexto apropiado. Estos contextos son clases, funciones o namespaces que tienen buenos nombres.

Ejemplo:

- El significado de una variable llamada `state` es ambigua por si sola.
- Tal vez un mejor nombre sería `addrState` pues deja explícito que se refiere a una dirección.

Names

Regla 10: Add Meaningful Context.

Por lo general los nombres no son suficientemente explicativos por sí solos – debemos situarlos en un contexto apropiado. Estos contextos son clases, funciones o namespaces que tienen buenos nombres.

Ejemplo:

- El significado de una variable llamada `state` es ambigua por si sola.
- Tal vez un mejor nombre sería `addrState` pues deja explícito que se refiere a una dirección. Lo que implica que también existen las variables `addrFirstName`, `addrLastName`, `addrStreet`, `addrHouseNumber`, `addrCity` y `addrZipcode`.

Names

Regla 10: Add Meaningful Context.

Por lo general los nombres no son suficientemente explicativos por sí solos – debemos situarlos en un contexto apropiado. Estos contextos son clases, funciones o namespaces que tienen buenos nombres.

Ejemplo:

```
1  public class Address
2  {
3      private string _firstName;
4      private string _lastName;
5      private string _street;
6      private int _houseNumber;
7      private string _city;
8      private string _state;
9      private string _zipcode;
10 }
```

Names

Regla 10: Add Meaningful Context.

Por lo general los nombres no son suficientemente explicativos por sí solos – debemos situarlos en un contexto apropiado. Estos contextos son clases, funciones o namespaces que tienen buenos nombres.

Regla 11: Don't Add Gratuitous Context.

Names

Regla 10: Add Meaningful Context.

Por lo general los nombres no son suficientemente explicativos por sí solos – debemos situarlos en un contexto apropiado. Estos contextos son clases, funciones o namespaces que tienen buenos nombres.

Regla 11: Don't Add Gratuitous Context.

Nombres cortos son mejores que nombres largos, siempre y cuando sus significados sean claros. No agregues contexto a nombres si no lo necesita.

Names

Regla 10: Add Meaningful Context.

Por lo general los nombres no son suficientemente explicativos por sí solos – debemos situarlos en un contexto apropiado. Estos contextos son clases, funciones o namespaces que tienen buenos nombres.

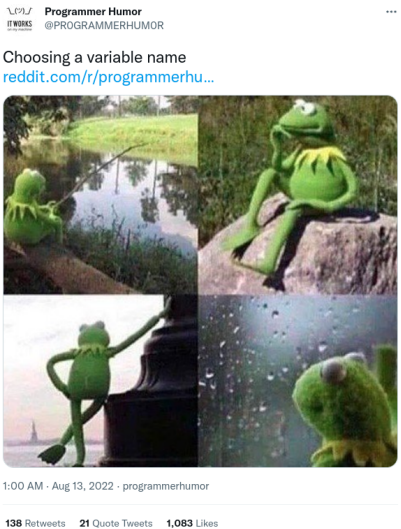
Regla 11: Don't Add Gratuitous Context.

Nombres cortos son mejores que nombres largos, siempre y cuando sus significados sean claros. No agregues contexto a nombres si no lo necesita.

Ejemplo:

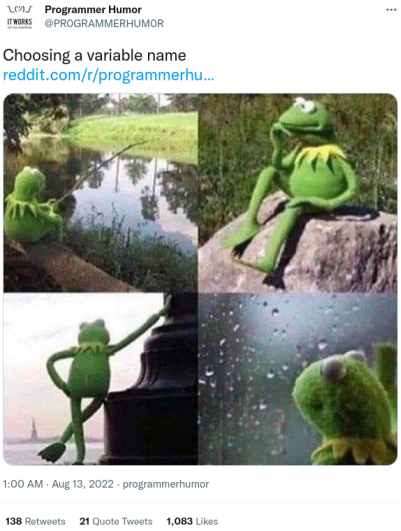
- Dentro de un clase `Address` llamar a un atributo `addrState` es innecesario. Basta con llamarlo `state` – que es igual de informativo y más corto.

Names



Resumen:

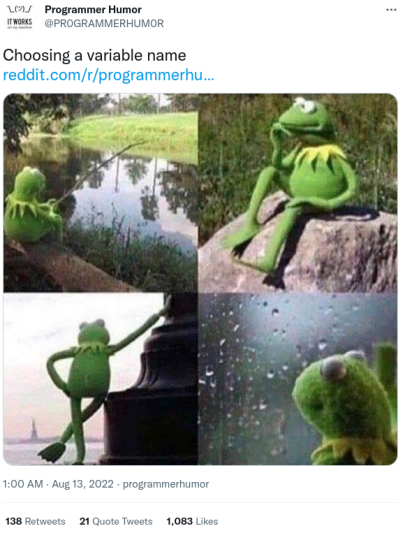
Names



Resumen:

Dedica tiempo y cariño a elegir buenos nombres para tus clases, métodos y variables.

Names

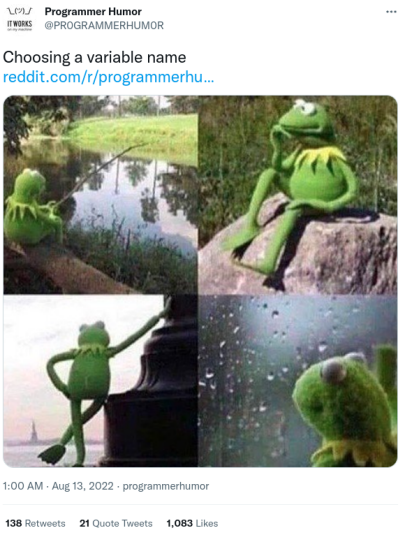


Resumen:

Dedica tiempo y cariño a elegir buenos nombres para tus clases, métodos y variables.

Un nombre es bueno si hace que entender tu código sea más fácil.

Names



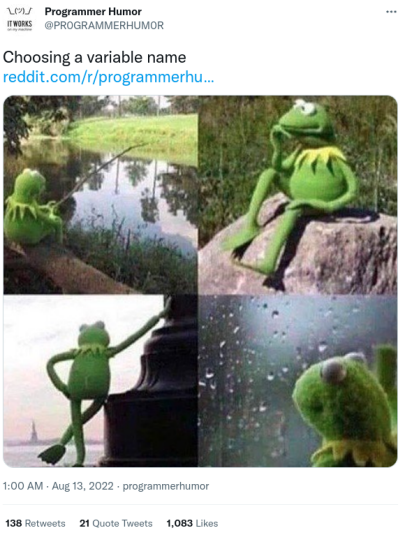
Resumen:

Dedica tiempo y cariño a elegir buenos nombres para tus clases, métodos y variables.

Un nombre es bueno si hace que entender tu código sea más fácil.

Un nombre es malo si no te dice nada.

Names



Resumen:

Dedica tiempo y cariño a elegir buenos nombres para tus clases, métodos y variables.

Un nombre es bueno si hace que entender tu código sea más fácil.

Un nombre es malo si no te dice nada.

Y un nombre es muy malo si te hace pensar que el código está haciendo algo distinto.

¿Cuáles son los descuentos típicos en el proyecto debido al cap. 2?

Names

Descuentos más comunes:

- Nombres poco descriptivos.
- Métodos con nombres muy similares que hacen cosas distintas.
- Nombres de métodos que no son verbos o frases verbales (acciones).
- Nombres de clases/variables que no son sustantivos.
- No encapsular constantes (i.e., tener números o **strings** sueltos en el código).
- Ser inconsistente en cómo se nombran las cosas:
  - Usar distintos nombres para el mismo concepto (e.j., `get`, `fetch`, `retrieve`).
  - Escribir cosas similares de forma distinta (e.j., atributos publicos con y sin _)
  - Recomendamos seguir las convenciones de nombres en C#.

Names

Kind	Rule
Private field	`_lowerCamelCase`
Public field	`UpperCamelCase`
Protected field	`UpperCamelCase`
Internal field	`UpperCamelCase`
Property	`UpperCamelCase`
Method	`UpperCamelCase`
Class	`UpperCamelCase`
Interface	`IUpperCamelCase`
Local variable	`lowerCamelCase`
Parameter	`lowerCamelCase`

Actividad