Nombres con sentido

Nombres que revelen intenciones

El nombre de una variable, función o clase debe responder una serie de cuestiones básicas. Debe indicar por qué existe, qué hace y cómo se usa. Si un nombre requiere un comentario, significa que no revela su cometido.

```
int d; //tiempo transcurrido en días
int elapsedTimeInDays;
int daysSinceCreation;
int fileAgeInDays;
```

En este ejemplo se observa que se necesita un nombre que especifique lo que se mide y la unidad de dicha medida.

```
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

> Hay que evitar dejar pistas falsas que dificulten el significado del código.

1. Nombres de **variables de entorno del sistema**
2. Nombres de un **tipo de dato** cuando esa variable no es de ese tipo
3. **Nombres casi iguales con variaciones mínimas**. Por ejemplo: `cambiarPosicionCuadradoArriba` con `cambiarPosicionTrianguloArriba`.
4. **Nombre desinformativo**. Usar **l** minúcula o la **O** mayúscula como nombre de variable puede llevar a la confusión con constantes.

No usar la palabra `list` en una variable salvo que esa variable represente una colección de tipo `List`. Cambiar: `accountList` por `accounts` o `accountGroup`.

Realizar distinciones con sentido

No basta con añadir series de números o palabras adicionales con el único objetivo de que el código compile.

> Si los nombres tienen que ser distintos, también deben tener un significado diferente.

1. **Palabras sin información**. Series numéricas(a1, a2, ... aN).

2. **Palabras adicionales**. Clase `Product` con clase `ProductInfo` o `ProductData`. **Info** y **Data** son palabras adicionales, como **a**, **and** y **the**.

3. **Prefijos**. Cuando no siguen un convenio pueden desinformar. Usar por ejemplo `theZork` para nombrar una variable porque en ese ámbito ya está cogida la variable `zork`.

4. **Palabras adicionales redundantes**. La palabra `variable` no debe incluirse nunca en el nombre de una variable.

Sin convención:

- `moneyAmount` no se distingue de `money`
- `customerInfo` no se distingue de `customer`
- `accountData` no se distingue de `account`
- `theMessage` no se distingue de message

> Se deben diferenciar los nombres de forma que el lector del código aprecie las diferencias.

Nombres pronunciables

> Si no puedes pronunciar el nombre de una variable no podrás explicar lo que hace sin parecer tonto.

Nombres que se puedan buscar

Los nombres de una letra y las constantes numéricas tienen un problema: no son fáciles de localizar en el texto.

Localizar: `MAX_CLASSES_PER_STUDENT` > 7

1. **Número**. Un número puede resultar muy complicado de buscar y por ende de refactorizar.
2. **Nombres monosílabos**. Un nombre de variable como **e** puede ser muy complicado de encontrar. Sólo recomendables usar este tipo de nombres en variables locales.

La longitud de un nombre debe corresponderse al tamaño de su ámbito [N5]

> Si una variable o constante se usa en varios puntos del código, debe asignarle un nombre que se pueda buscar

Compara:

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

con

```
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 codificaciones

Al codificar información de tipos o ámbitos en un nombre se dificulta la descodificación.

Si alguien nuevo se incorpora tendrá que aprender cómo descifrar la convención que se sigue para nombres.

Notación húngara

> Evita usar el tipo de dato de la variable en el nombre.

Hacen más complicado cambiar el nombre o el tipo de una variable o clase. También dificulta la legibilidad del código y pueden hacer que el sistema de codificación confunda al lector: `PhoneNumber phoneString;`

Interfaces e implementaciones

El prefijo 'I', tan habitual en los archivos de legado actuales es, en el mejor de los casos, una distracción, y en el peor, un exceso de información.

Si hay que codificar la interfaz o la implementación, mejor esta última.

Interface: `ShapeFactory`

Implementation: `ShapeFactoryImpl`

Evitar asignaciones mentales

Nombres de variales de una sola letra. Son tradicionales en el contexto de un bucle(i, j o k) pero, en otros contextos, es una opción muy pobre: es como un marcador de posición que el lector debe asignar mentalmente a un concepto real.

Nombres de clase

> El nombre de una clase no deber ser un verbo

Nombres de métodos

> Los métodos deben tener nombres de verbo como `postPayment` , `deletePage` o `save`. Los métodos de acceso, de modificación y los predicados deben tener como nombre su valor y usar como prefijo `get`, `set` e `is`.

No se exceda con el atractivo

Si los nombres son demasiado inteligentes, sólo los recordará quién lo ha escrito.

```
Claridad > Entretenimiento
```

Una palabra por concepto

> Un léxico coherente es una gran ventaja para los programadores que tengan que usar su código.

Resulta confuso usar `fetch`, `retrieve` y `get` como métodos equivalentes de clases distintas. Es imposible recordar qué método equivale a cada clase.

> Los nombres de funciones deben ser independientes y coherentes para que pueda elegir el método correcto sin necesidad de búsquedas adicionales.

No haga juegos de palabras

> Evite usar la misma palabra con dos fines distintos.

Ejemplo: método `add()`

Si utilizas el siguiente método `add()` con un fin equivalente en varias clases:

```
public String add(String izquierda, String derecha) {
    return izquierda + derecha;
}
```

Tendrás un problema a la hora de implementar un método en otra clase para añadir un valor a una colección de tipo List.

Estaríamos ante una diferencia semántica, por lo que deberemos usar un nombre como `insert` o `append`.

Si llamamos `add` al nuevo método sería un **juego de palabras**.

> Nuestro objetivo es facilitar la comprensión del código. Tenemos que ser los responsables de transmitir el significado.

Nombres de dominios de soluciones

Usa términos informáticos, nombres de patrones y demás.

Nuestros colegas ya conocen lo que eso significa. Por ejemplo:

`AccountFactory` parece indicar que estamos ante una interfaz que implementa el patrón `Factory`.

Más ejemplos: `UserSingleton`, `StudentQuickSort`, `JobQueue`,...

Nombres de dominions de problemas

> Cuando no exista un término de programación para lo que está haciendo, use el nombre del dominio de problemas.

Añadir contexto con sentido

> Algunos nombres tienen significado por sí mismos, pero la mayoría no. Por ello, debe incluirlos en un contexto, en clases, funciones y espacios de nombres con nombres adecuados. Cuando todo lo demás falle, pueden usarse prefijos como último recurso.

Variables: `street`, `houseNumber`, `city`, `state` y `zipCode`.Parece que forman una dirección. Ahora bien, si `state` es usado en otro método... ¿cómo distinguirlo? Habría que añadir contexto.

Una forma de añadirlo sería con prefijos tales como: `addrState` , `addrHouseNumber`, etc. Aunque la mejor forma de dar contexto es creando la class `Address`.

Variables en un contexto ambiguo

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;

    if (count == 0) {
        number = "no";
        verb = "are";
        pluralModifier = "s";
    } else if (count == 1) {
        number = "1";
        verb = "is";
        pluralModifier = "";
    } else {
        number = Integer.toString(count);
        verb = "are";
        pluralModifier = "s";
    }

    String guessMessage = String.format("There %s %s %s%s", verb, number,
candidate, pluralModifier);

    print(guessMessage);
}
```

Para dividir la función en fragmentos más reducidos necesitamos crear una clase `GuessStatisticsMessage`. La mejora del contexto hace que el algoritmo sea más limpio y se divida en funciones más reducidas:

Variables con un contexto

```
    public class GuessStatisticsMessage {
        private String number;
        private String verb;
        private String pluralModifier;
```

```
        public Strin gmake(char candidate, int count) {
            return String.format("There %s %s %s%s", verb, number, candidate,
pluralModifier);
        }

        private void createPluralDependentMessageParts(int count) {
            if ( count == 0 ) {
                thereAreNoLetters();
            } else if (count == 1) {
                thereIsOneLetter();
            } else {
                thereAreManyLetters(count);
            }
        }

        private void thereAreManyLetters(int count) {
            number = Integer.toString(count);
            verb = "are";
            pluralModifier = "s";
        }

        private void thereIsOneLetter() {
            number = "1";
            verb = "is";
            pluralModifier = "";
        }

        private void thereAreNoLetters() {
            number = "no";
            verb = "are";
            pluralModifier = "s";
        }
    }
```

No añadir contextos innecesarios

> Los nombres breves suelen ser más adecuados que los extensos, siempre que sean claros.

> No añada más contexto del necesario a un nombre

Los nombres `accountAddress` y `customerAddress` son perfectos para instancias de la clase `Address` pero no sirven como nombres de clase.

Para distinguir entre direcciones mac, puertos y web, podría usar: `PostalAddress`, `MAC` y `URI`.

> El objetivo de cualquier nombre es ser más preciso