

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

Clase 04: Clean Code (Cap. 3)

Francisco Gazitúa Requena (fco_gazitua_requena@uc.cl)

IIC2113 Diseño Detallado de Software

14 de Agosto, 2025

Cuestionario cap 1 y 2

Cuestionario cap 1 y 2

- Es más fácil entender código limpio que código sucio.

Cuestionario cap 1 y 2

- Es más fácil entender código limpio que código sucio.
- Para la entrega 1 del proyecto, sería más rápido escribir un código limpio que pase todos los test cases que escribir un código sucio que pase todos los test cases.

Cuestionario cap 1 y 2

- Es más fácil entender código limpio que código sucio.
- Para la entrega 1 del proyecto, sería más rápido escribir un código limpio que pase todos los test cases que escribir un código sucio que pase todos los test cases.
- Limpiar tu código suele resultar en códigos más cortos.

Cuestionario cap 1 y 2

- Es más fácil entender código limpio que código sucio.
- Para la entrega 1 del proyecto, sería más rápido escribir un código limpio que pase todos los test cases que escribir un código sucio que pase todos los test cases.
- Limpiar tu código suele resultar en códigos más cortos.
- Si un equipo de desarrollo no destina tiempo a limpiar su código es esperable que su productividad decrezca a medida que pasa el tiempo.

Cuestionario cap 1 y 2

- Si el nombre de una variable necesita un comentario para ser entendido, entonces es un mal nombre.

Cuestionario cap 1 y 2

- Si el nombre de una variable necesita un comentario para ser entendido, entonces es un mal nombre.
- El problema de poner números directo en tu código es que en el futuro te será difícil encontrarlos.

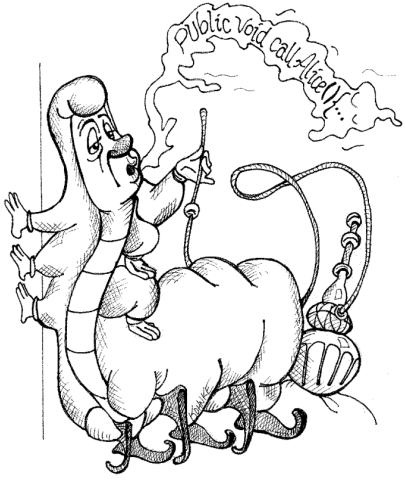
Cuestionario cap 1 y 2

- Si el nombre de una variable necesita un comentario para ser entendido, entonces es un mal nombre.
- El problema de poner números directo en tu código es que en el futuro te será difícil encontrarlos.
- El nombre de una variable debe ser tal que su propósito se entienda independiente del contexto en que aparece. En este sentido, el nombre “addrState” es siempre mejor que el nombre “state” porque “addrState” deja claro que se refiere al estado de una dirección (y no a cualquier otro tipo de estado).

Cuestionario cap 1 y 2

- Si el nombre de una variable necesita un comentario para ser entendido, entonces es un mal nombre.
- El problema de poner números directo en tu código es que en el futuro te será difícil encontrarlos.
- El nombre de una variable debe ser tal que su propósito se entienda independiente del contexto en que aparece. En este sentido, el nombre “addrState” es siempre mejor que el nombre “state” porque “addrState” deja claro que se refiere al estado de una dirección (y no a cualquier otro tipo de estado).
- Los nombres de interfaces deben ser precedidos por una ”i” mayúscula. Por ejemplo, si en el proyecto definimos una interfaz para modelar los efectos de las cartas, un buen nombre sería IEffect.

Functions



Functions

¿Por qué usamos funciones mientras programamos?

Functions

¿Por qué usamos funciones mientras programamos?

IIC1103: Para evitar copiar y pegar código.

Functions

¿Por qué usamos funciones mientras programamos?

IIC1103: Para evitar copiar y pegar código.

IIC2113: Para que nuestro código sea más fácil de entender.

Functions

¿Por qué usamos funciones mientras programamos?

IIC1103: Para evitar copiar y pegar código.

IIC2113: Para que nuestro código sea más fácil de entender.

Observación brillante

Códigos cortos son más fáciles de entender que códigos largos. Por lo tanto, usar funciones cortas hace que tu código sea más fácil de entender.

Functions

¿Qué tan pequeña debería ser una función?

- Ojalá entre 4 y 5 líneas.
- A lo más uno o dos niveles de indentación (e.j., un `if` dentro de un `while`).
- Que tu función tenga un nombre apropiado.

Functions

¿Qué tan pequeña debería ser una función?

- Ojalá entre 4 y 5 líneas.
- A lo más uno o dos niveles de indentación (e.j., un `if` dentro de un `while`).
- Que tu función tenga un nombre apropiado.

Ejemplo:

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

Functions

Programar funciones pequeñas nos ayuda a cumplir con uno de los mandamientos más importantes de clean code:

Functions

Programar funciones pequeñas nos ayuda a cumplir con uno de los mandamientos más importantes de clean code:

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

Functions

Programar funciones pequeñas nos ayuda a cumplir con uno de los mandamientos más importantes de clean code:

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

¿Por qué es recomendable que una función haga una sola cosa?

Functions

Programar funciones pequeñas nos ayuda a cumplir con uno de los mandamientos más importantes de clean code:

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

¿Por qué es recomendable que una función haga una sola cosa?

- Entender una cosa es más fácil que entender dos cosas.
- Es más fácil reutilizar funciones que hacen una sola cosa.
- Previene la existencia de efectos secundarios inesperados.

Functions

Ejemplo:

```
1  public bool CheckPassword(String userName, String password) {
2    User user = UserGateway.findByName(userName);
3    if (user != User.NULL) {
4      String codedPhrase = user.getPhraseEncodedByPassword();
5      String phrase = cryptographer.decrypt(codedPhrase, password);
6      if ("Valid Password".equals(phrase)) {
7        Session.initialize();
8        return true;
9      }
10   }
11   return false;
12 }
```

¿Alguien ve algún problema con esta función?

Functions

Command Query Separation

"*Functions should either do something or answer something, but not both.*"

Command Query Separation

"*Functions should either do something or answer something, but not both.*"

Ejemplo: Digamos que tenemos el siguiente método:

```
1 public bool Set(string attribute, string value);
```

Command Query Separation

"*Functions should either do something or answer something, but not both.*"

Ejemplo: Digamos que tenemos el siguiente método:

```
1 public bool Set(string attribute, string value);
```

Luego, usar el método se verá más o menos así:

```
3 if(Set("username", "unclebob"))
4 {
5     // ... etc
6 }
```

¿Qué significa esto?

Command Query Separation

"*Functions should either do something or answer something, but not both.*"

Solución: Separar en dos métodos distintos (donde cada uno haga una cosa).

```
8 public bool AttributeExists(string attribute);
9 public void SetAttribute(string attribute, string value);
```

Command Query Separation

"*Functions should either do something or answer something, but not both.*"

Solución: Separar en dos métodos distintos (donde cada uno haga una cosa).

```
8 public bool AttributeExists(string attribute);
9 public void SetAttribute(string attribute, string value);
```

Ahora es trivial entender el código que usa estos métodos:

```
11 if(AttributeExists("username")) {
12     SetAttribute("username", "unclebob");
13     // ... etc
14 }
```

Functions

¿Cómo logro que mis funciones hagan una sola cosa?

Functions

¿Cómo logro que mis funciones hagan una sola cosa?

- Extrae funciones de tu función hasta que ya no tenga sentido seguir extrayendo.

Functions

¿Cómo logro que mis funciones hagan una sola cosa?

- Extrae funciones de tu función hasta que ya no tenga sentido seguir extrayendo.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Mini-Actividad: Piensa como limpiar la función `DoTheThing(...)`

Functions

¿Cómo logro que mis funciones hagan una sola cosa?

- Extrae funciones de tu función hasta que ya no tenga sentido seguir extrayendo.

X	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Mini-Actividad: Piensa como limpiar la función `DoTheThing(...)`

Functions

¿Cómo logro que mis funciones hagan una sola cosa?

- Extrae funciones de tu función hasta que ya no tenga sentido seguir extrayendo.



Mini-Actividad: Piensa como limpiar la función `DoTheThing(...)`

Functions

¿Cómo logro que mis funciones hagan una sola cosa?

- Extrae funciones de tu función hasta que ya no tenga sentido seguir extrayendo.



Mini-Actividad: Piensa como limpiar la función `DoTheThing(...)`

Functions

¿Cómo logro que mis funciones hagan una sola cosa?

- Extrae funciones de tu función hasta que ya no tenga sentido seguir extrayendo.



Mini-Actividad: Piensa como limpiar la función `DoTheThing(...)`

Functions

¿Cómo logro que mis funciones hagan una sola cosa?

- Extrae funciones de tu función hasta que ya no tenga sentido seguir extrayendo.



Mini-Actividad: Piensa como limpiar la función `DoTheThing(...)`

```
public int[] DoTheThing(int mv) { // MaxValue
    if (mv >= 2) { // the only valid case
        // declarations
        int s = mv + 1; // size of array
        bool[] f = new bool[s];
        int i;
        // initialize array to true.
        for (i = 0; i < s; i++)
            f[i] = true;
        // get rid of known non-primes
        f[0] = f[1] = false;
        // sieve
        int j;
        for (i = 2; i < Math.Sqrt(s) + 1; i++) {
            if (f[i]) { // if i is uncrossed, cross its multiples.
                for (j = 2 * i; j < s; j += i)
                    f[j] = false; // multiple is not prime
            }
        }
        int c = 0; // how many primes are there?
        for (i = 0; i < s; i++) {
            if (f[i])
                c++; // bump count.
        }
        int[] ret = new int[c];
        // move the primes into the result
        for (i = 0, j = 0; i < s; i++) {
            if (f[i])   // if prime
                ret[j++] = i;
        }
        return ret; // return the primes
    }
    else // maxValue < 2
        return new int[0]; // return null array if bad input.
}
```

Functions

¿Cómo logro que mis funciones hagan una sola cosa?

- Extrae funciones de tu función hasta que ya no tenga sentido seguir extrayendo.

```
2  public int[] GeneratePrimes(int maxValue){
3      if (maxValue >= 2){ // the only valid case
4          // declarations
5          int s = maxValue + 1; // size of array
6          bool[] f = new bool[s];
7          int i;
8          // initialize array to true.
9          for (i = 0; i < s; i++)
10             f[i] = true;
11         // get rid of known non-primes
12         f[0] = f[1] = false;
13         /* ... */
14     }
15     else // maxValue < 2
16         return new int[0]; // return null array if bad input.
17 }
```

Functions

¿Cómo logro que mis funciones hagan una sola cosa?

- Extrae funciones de tu función hasta que ya no tenga sentido seguir extrayendo.

```
1  public int[] GeneratePrimes(int maxValue){
2      if (maxValue < 2)
3          return new int[0];
4      else {
5          UncrossIntegersUpTo(maxValue);
6          CrossOutMultiples();
7          PutUncrossedIntegersIntoResult();
8          return _result;
9      }
10 }
```

¿Cómo logro que mis funciones hagan una sola cosa?

- Extrae funciones de tu función hasta que ya no tenga sentido seguir extrayendo.

```
public int[] GeneratePrimes(int maxValue){
    if (maxValue < 2)
        return new int[0];
    else {
        UncrossIntegersUpTo(maxValue);
        CrossOutMultiples();
        PutUncrossedIntegersIntoResult();
        return _result;
    }
}
```

¡Momento! En este caso la función nos quedó más corta, pero técnicamente ambas funciones están haciendo exactamente lo mismo (generar números primos). ¿Por qué la segunda versión cumple con *hacer una cosa* y la primera no?

Functions

"If a function does only those steps that are one level below the stated name of the function, then the function is doing one thing. After all, the reason we write functions is to decompose a larger concept (in other words, the name of the function) into a set of steps at the next level of abstraction."

Functions

"If a function does only those steps that are one level below the stated name of the function, then the function is doing one thing. After all, the reason we write functions is to decompose a larger concept (in other words, the name of the function) into a set of steps at the next level of abstraction."

```
public int[] GeneratePrimes(int maxValue){
    if (maxValue < 2)
        return new int[0];
    else {
        UncrossIntegersUpTo(maxValue);
        CrossOutMultiples();
        PutUncrossedIntegersIntoResult();
        return _result;
    }
}
```

Functions

"If a function does only those steps that are one level below the stated name of the function, then the function is doing one thing. After all, the reason we write functions is to decompose a larger concept (in other words, the name of the function) into a set of steps at the next level of abstraction."

```
1  public int[] GeneratePrimes(int maxValue){
2      if (maxValue < 2)
3          return new int[0];
4      else {
5          UncrossIntegersUpTo(maxValue);
6          CrossOutMultiples();
7          PutUncrossedIntegersIntoResult();
8          return _result;
9      }
10 }
```

Spoiler del cap. 5: Las definiciones de las funciones llamadas deberían aparecer abajo de `GeneratePrimes` para que podamos leer el código de arriba hacia abajo.

Functions

Lo otro importante es minimizar la cantidad de argumentos en tus funciones.

- Cero argumentos es lo ideal, luego uno, luego dos, casi nunca 3. Nunca más de 3.

¿Por qué será buena idea usar funciones con pocos argumentos?

Functions

Lo otro importante es minimizar la cantidad de argumentos en tus funciones.

- Cero argumentos es lo ideal, luego uno, luego dos, casi nunca 3. Nunca más de 3.

¿Por qué será buena idea usar funciones con pocos argumentos?

Ejemplo: Si el retorno de la función es verdadero

```
1 if(HasDamageModifierOverRival(attacker, defender))
2   // ...
```

... ¿Quién tiene el modificador?

Functions

Lo otro importante es minimizar la cantidad de argumentos en tus funciones.

- Cero argumentos es lo ideal, luego uno, luego dos, casi nunca 3. Nunca más de 3.

¿Por qué será buena idea usar funciones con pocos argumentos?

Ejemplo: Si el retorno de la función es verdadero

```
1 if(HasDamageModifierOverRival(attacker, defender))
2   // ...
```

... ¿Quién tiene el modificador?

Pero si eliminamos un argumento entender (y usar) la función se vuelve más fácil:

```
1 if(attacker.HasDamageModifierOverRival(defender))
2   // ...
```

Functions

Lo otro importante es minimizar la cantidad de argumentos en tus funciones.

- Cero argumentos es lo ideal, luego uno, luego dos, casi nunca 3. Nunca más de 3.

¿Por qué será buena idea usar funciones con pocos argumentos?

Ejemplo: Si el retorno de la función es verdadero

```
1 if(HasDamageModifierOverRival(attacker, defender))
2   // ...
```

... ¿Quién tiene el modificador?

Pero si eliminamos un argumento entender (y usar) la función se vuelve más fácil:

```
1 if(attacker.HasDamageModifierOverRival(defender))
2   // ...
```

En general, mientras menos argumentos tiene una función es más fácil de entender.

Functions

Lo otro importante es minimizar la cantidad de argumentos en tus funciones.

- Cero argumentos es lo ideal, luego uno, luego dos, casi nunca 3. Nunca más de 3.

¿Cómo podemos minimizar la cantidad de argumentos que reciben nuestras funciones?

Functions

Lo otro importante es minimizar la cantidad de argumentos en tus funciones.

- Cero argumentos es lo ideal, luego uno, luego dos, casi nunca 3. Nunca más de 3.

¿Cómo podemos minimizar la cantidad de argumentos que reciben nuestras funciones?

Opción 1: Transformar argumentos en atributos.

```
2  public class GeneratePrimes{
3      /* ... */
4      private void CrossOutMultiplesOf(bool[] crossedOut, int i)
5      {
6          for (int multiple = 2*i;
7               multiple < crossedOut.Length;
8               multiple += i)
9              crossedOut[multiple] = true;
10     }
11     /* ... */
12 }
```

Functions

Lo otro importante es minimizar la cantidad de argumentos en tus funciones.

- Cero argumentos es lo ideal, luego uno, luego dos, casi nunca 3. Nunca más de 3.

¿Cómo podemos minimizar la cantidad de argumentos que reciben nuestras funciones?

Opción 1: Transformar argumentos en atributos.

```
2  public class GeneratePrimes{
3      private bool[] _crossedOut;
4      /* ... */
5      private void CrossOutMultiplesOf(int i)
6      {
7          for (int multiple = 2*i;
8               multiple < crossedOut.Length;
9               multiple += i)
10             _crossedOut[multiple] = true;
11     }
12 }
```

Functions

Lo otro importante es minimizar la cantidad de argumentos en tus funciones.

- Cero argumentos es lo ideal, luego uno, luego dos, casi nunca 3. Nunca más de 3.

¿Cómo podemos minimizar la cantidad de argumentos que reciben nuestras funciones?

Opción 2: Agrupar argumentos en un objeto apropiado.

- `Circle makeCircle(double x, double y, double radius)`

Functions

Lo otro importante es minimizar la cantidad de argumentos en tus funciones.

- Cero argumentos es lo ideal, luego uno, luego dos, casi nunca 3. Nunca más de 3.

¿Cómo podemos minimizar la cantidad de argumentos que reciben nuestras funciones?

Opción 2: Agrupar argumentos en un objeto apropiado.

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

Functions

Lo otro importante es minimizar la cantidad de argumentos en tus funciones.

- Cero argumentos es lo ideal, luego uno, luego dos, casi nunca 3. Nunca más de 3.

¿Cómo podemos minimizar la cantidad de argumentos que reciben nuestras funciones?

Opción 2: Agrupar argumentos en un objeto apropiado.

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

Opción 3: Si un argumento es un flag booleano, lo deberías separar en dos métodos.

- `render(bool isSuite)`

Functions

Lo otro importante es minimizar la cantidad de argumentos en tus funciones.

- Cero argumentos es lo ideal, luego uno, luego dos, casi nunca 3. Nunca más de 3.

¿Cómo podemos minimizar la cantidad de argumentos que reciben nuestras funciones?

Opción 2: Agrupar argumentos en un objeto apropiado.

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

Opción 3: Si un argumento es un flag booleano, lo deberías separar en dos métodos.

- `render(bool isSuite)`
- `renderForSuite()` y `renderForSingleTest()`.

Functions

Finalmente, evita a toda costa tener código duplicado.

Functions

Finalmente, evita a toda costa tener código duplicado.

Ejemplo: ¿Cual es el problema aquí?

```
public bool CheckIfChildrenSumMoreThanLimit(Node tree, int limit)
{
    int sum = 0;
    foreach (var node in tree.Children)
        sum += node.Value;
    return sum > limit;
}
```

```
public bool CheckIfChildrenSumLessThanLimit(Node tree, int limit)
{
    int sum = 0;
    foreach (var node in tree.Children)
        sum += node.Value;
    return sum < limit;
}
```

Functions

No repetir código es mucho más fácil de mantener

```
1  public bool CheckIfChildrenSumMoreThanLimit(Node tree, int limit)
2      => return GetChildrenSum(tree) > limit;
3
4  public bool CheckIfChildrenSumLessThanLimit(Node tree, int limit)
5      => return GetChildrenSum(tree) < limit;
6
7  private bool GetChildrenSum(Node tree)
8  {
9      int sum = 0;
10     foreach (var node in tree.Children)
11         sum += node.Value;
12     return sum;
13 }
```

Functions

Algunas otras reglas:

Functions

Algunas otras reglas:

- No deberíamos tener argumentos que funcionen como *output*.

Functions

Algunas otras reglas:

- No deberíamos tener argumentos que funcionen como *output*.
- Las funciones deben cambiar el estado interno de un objeto o retornar información sobre un objeto, **no ambas**.

Functions

Algunas otras reglas:

- No deberíamos tener argumentos que funcionen como *output*.
- Las funciones deben cambiar el estado interno de un objeto o retornar información sobre un objeto, **no ambas**.
- Lancen excepciones en vez de retornar código de error.

Functions

Algunas otras reglas:

- No deberíamos tener argumentos que funcionen como *output*.
- Las funciones deben cambiar el estado interno de un objeto o retornar información sobre un objeto, **no ambas**.
- Lancen excepciones en vez de retornar código de error.
- Manejo de error es una tarea.

Functions

Algunas otras reglas:

- No deberíamos tener argumentos que funcionen como *output*.
- Las funciones deben cambiar el estado interno de un objeto o retornar información sobre un objeto, **no ambas**.
- Lancen excepciones en vez de retornar código de error.
- Manejo de error es una tarea.
- Control de flujo es una tarea.

Functions

Resumen:

"When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code. But I also have a suite of unit tests that cover every one of those clumsy lines of code."

Functions

Resumen:

"When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code. But I also have a suite of unit tests that cover every one of those clumsy lines of code."

"So then I massage and refine that code, splitting out functions, changing names, eliminating duplication. I shrink the methods and reorder them. Sometimes I break out whole classes, all the while keeping the tests passing."

Functions

Resumen:

"When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code. But I also have a suite of unit tests that cover every one of those clumsy lines of code."

"So then I massage and refine that code, splitting out functions, changing names, eliminating duplication. I shrink the methods and reorder them. Sometimes I break out whole classes, all the while keeping the tests passing."

"In the end, I wind up with functions that follow the rules I've laid down in this chapter. I don't write them that way to start. I don't think anyone could."

¿Cuáles son los descuentos típicos en el proyecto debido al cap. 3?

Functions

Descuentos más comunes:

- Funciones largas (o que hacen más de una cosa).
- Funciones con más de 3 argumentos.
- Funciones con output arguments.
- Bloques de código duplicado (que pudieron ser una función).
- Funciones que reciben booleanos (o similar).
- Funciones con más de dos niveles de indentación.
- Condiciones complejas en ifs/whiles que debieron ser pasados a una función (con un buen nombre) que retorne un booleano.

Actividad