



“UNIVERSIDAD DE LAS FUERZAS ARMADAS”

ESPE

SOFTWARE ENGINEERING

OBJECT - ORIENTED PROGRAMMING

TOPIC:

WORKSHOP 13 - Teamwork on project code

TEACHER:

PHD. EDISON LASCANO

NRC:

4680

DATE:

June 12, 2022

SEDE MATRIZ SANGOLQUÍ

QUITO-ECUADOR

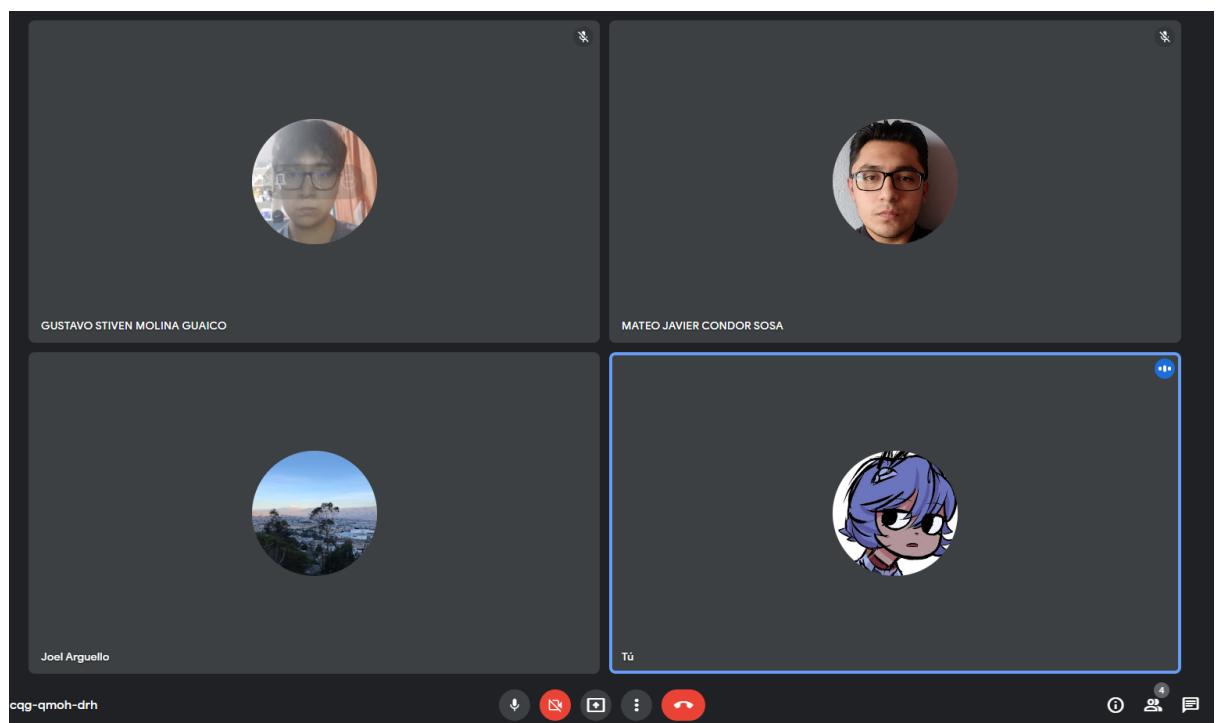
2022

Arguello Espinosa Joel Daniel 1
Bedon Guevara Roberto Alexander 2
Burbano Pacheco Luis Ariel 3
Caiza Pullotasig Widinson Danilo 4
Chavez Escudero Genaro David 5
Chicaiza Ortiz Frank Sebastian 6
Chiriboga Zurita Daniel Isai 7
Condor Sosa Mateo Javier 1
Granda Velasquez Carlos Javier 2
Haro Cachumba Alexis Alvaro 3
Ibarra Gaona Ronny Joel 4
Imbaquinga Guaña Jose Ricardo 5
Loor Mercado Cesar Ignacio 6
Mendoza Arteaga Aldric Mateo 7
Molina Guaico Gustavo Stiven 1
Montufar Saltos David Ariel 2
Moreira Vinueza Erick Patricio 3
Pabon Gonzalez Elkin Andres 4
Ponce Arguello Diego Armando 5
Pozo Analuisa Steven Jefferson 6
Quilumbaquin Lanchimba Jauro Smith 7
Quimbiulco Juan Diego 1
Rivera Espin Carlos Sebastian 2
Silva Velasquez Raul Andres 3
Sosa Romero Diana Carolina 4
Tituaña Moreno Daniela Lissette 5
Toapanta Vásquez Martín Honorio 6
Villavicencio Campoverde Bolivar Josue 7

TEAM#1: Type embedded in name

9

Arguello Espinosa Joel Daniel 1
Condor Sosa Mateo Javier 1
Molina Guaico Gustavo Stiven 1
Quimbiulco Juan Diego 1



Embedded Types in Names

- Problem Description:
 - The identifiers in the implementation include type information, e.g. `firstNameString` instead of just `firstName` for a first name property
- Consequences:
 - Poor understandability and readability → lower maintainability and reuse
 - Can break encapsulation, because it exposes the implementation type
- Causes:
 - Misguided efforts to provide more information in the code
 - Poor development environments that don't provide type-lookup tools

Embedded Types in Names

- Avoidance:
 - Don't include type names in identifiers
- Recognition:
 - Design / Code walkthroughs and inspects
- Extrication:
 - Refactor names

Embedded Type in names (Tipos integrados en el nombre)

Descripción:

El problema surge cuando en la implementación de los identificadores, en el nombre se incluye información del tipo de dato que va a tener la variable.

Consecuencias:

El entendimiento y la lectura van a ser ineficientes además de tener un nivel de mantenimiento bajo, junto con poca reusabilidad, esto puede provocar que el encapsulamiento se rompa debido a que se está exponiendo el tipo de implementación que se está realizando.

Causa:

- IDE's que no proporcionan herramientas para la búsqueda de variables.
- Intentar explicar con más palabras el tipo de variable, lo cual nos lleva a poner un nombre más largo de lo necesario.

Cómo evitarlo:

- Usar un buen IDE que nos proporcione suficientes herramientas.
- Intentar usar un buen lenguaje de clean code para poder tener un buen entendimiento sin exagerar las palabras

Como reconocerlo:

- Podemos reconocerlo a través del modelado o el diseño
- Revisando e inspeccionando el código

Cómo solucionarlo:

- Podemos renombrar los identificadores
- Usar nombres adecuados que no incluyan los tipos de datos de los identificadores facilitando su comprensión.

Código:

```
1 type Encoder interface {Encode([]byte) []byte}
2 type Person struct {name string; age int}
3 type Alias = struct {name string; age int}
4 type AliasPtr = *struct {name string; age int}
5 type IntPtr *int
6 type AliasPP = *IntPtr
7
8 // These types and aliases can be embedded.
9 Encoder
10 Person
11 *Person
12 Alias
13 *Alias
14 AliasPtr
15 IntPtr
16 *int
17 *IntPtr
18
19 // These types and aliases can't be embedded.
20 AliasPP      // base type is a pointer type
21 *Encoder     // base type is an interface type
22 *AliasPtr    // base type is a pointer type
23 IntPtr       // named pointer type
24 *IntPtr      // base type is a pointer type
25 *chan int    // base type is an unnamed type
26 struct {age int} // unnamed non-pointer type
27 map[string]int // unnamed non-pointer type
28 []int64      // unnamed non-pointer type
29 func()        // unnamed non-pointer type
```

TEAM#2: **3/10**

Topic: Long Methods (métodos largos)

Bedon Guevara Roberto Alexander

Granda Velasquez Carlos Javier

Montufar Saltos David Ariel

Rivera Espin Carlos Sebastian

Problema

Un método que contiene demasiadas líneas de código.

Descripción

Los métodos largos entran en la clasificación de los pitfalls(errores al momento de codificar), son métodos con muchas líneas de código y baja entendibilidad.

Causas

- No se realizó un refactor en la evolución del código.
- No tener claro el propósito de la codificación del método.
- Tratar de solucionar varios problemas en un solo método.

Consecuencias

- Una menor mantenibilidad del código se puede producir al agregar un método largo.
- Al hacer un método largo se volverá más difícil de entenderlo.
- Mayor dificultad al momento de probar el código.

Solución

- Extraer el código duplicado en un nuevo método.
- Separar el código en métodos más pequeños.
- Realizar un refactor del método que se duplica en el código para acortar líneas y se vea más limpio.
- Ejecutar las pruebas en cada pequeño avance del programa.

Ejemplo de código:

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";
    while(rentals.hasMoreElements()) {
        Rental each = (Rental)rentals.nextElement();

        //determine amounts for each line
        double thisAmount = amountFor(each);

        //add frequent renter points
        if(each.getMovie().getPriceCode() == Movie.NEW_RELEASE &&
           each.getDaysRented() > 1)
            frequentRenterPoints += 2;
        else
            frequentRenterPoints++;

        //show figures for this rental
        result += "\t" + each.getMovie().getTitle() + "\t" +
        String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }

    //add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
    " frequent renter points";
    return result;
}
```

código 2:

```
public class LongMethod
{
    private IEnumerable<Order> _orders = new List<Order>();
    private string _name = "ABC Order Detail";

    void printOwing()
    {

        decimal outstanding = 0;
        //print banner
        Console.WriteLine("*****");
        Console.WriteLine("**** Customer Owes ****");
        Console.WriteLine("*****");

        //calculate outstanding
        foreach (Order order in _orders)
        {
            outstanding += order.Amount;
        }

        //print details
        Console.Write("name: " + _name);
        Console.WriteLine("amount: " + outstanding);
    }
}
```

código 2 resuelto:

```
void PrintOwing()
{
    printBanner();

    var outstanding = GetOutstanding();

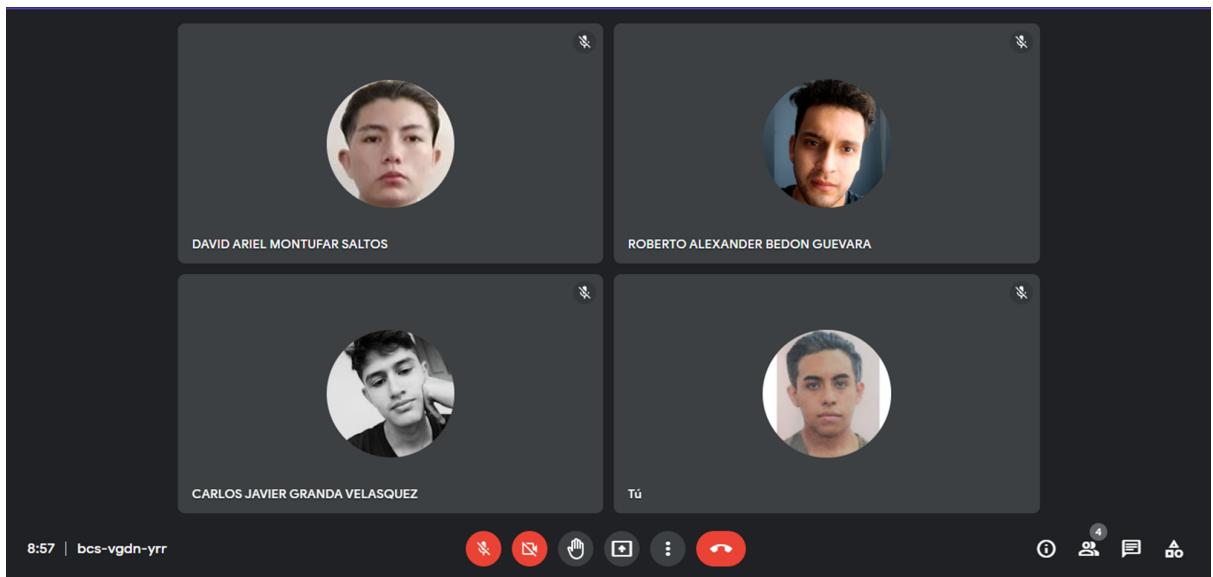
    PrintDetails(outstanding);
}

private decimal GetOutstanding()
{
    decimal outstanding = 0;
    foreach (Order order in _orders)
    {
        outstanding += order.Amount;
    }
    return outstanding;
}

private void PrintDetails(decimal outstanding)
{
    Console.WriteLine("name: " + _name);
    Console.WriteLine("amount: " + outstanding);
}

private static void printBanner()
{
    Console.WriteLine("*****");
    Console.WriteLine("**** Customer Owes ****");
    Console.WriteLine("*****");
}
```

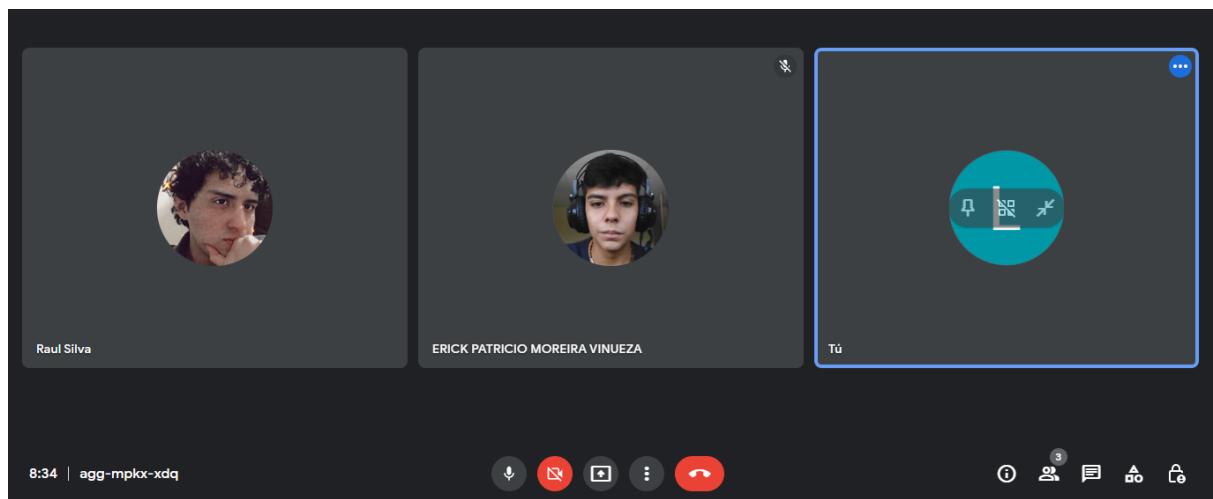
MEET:



Team #3: Duplicate Code 7/10

Burbano Pacheco Luis Ariel
Haro Cachumba Alexis Alvaro
Moreira Vinueza Erick Patricio
Silva Velasquez Raul Andres

Meet:



¿Qué es un pitfall?

Son malas prácticas realizadas por el desarrollador, que provocan que su código no sea legible y sea necesario leer varias veces, provocando que sea complicado e innecesariamente largo darle un mantenimiento adecuado.

Tema: Duplicate Code

¿Qué es un código duplicado?

Básicamente es alargar el código innecesariamente volviendo su comprensión y mantenimiento algo complejo, usando código repetido que tranquilamente se podría usar con métodos o funciones.

Consecuencias

Alarga innecesariamente las líneas de código, dificultando la lectura y comprensión del mismo para futuros mantenimientos.

Causas

Puede ser la falta de creatividad al crear el código o no conocer cómo encapsular ese código y llamarlo como método, también puede ser la falta de comunicación entre los desarrolladores del código.

Cómo evitarlo

Usando funciones o métodos para llamar al código, haciéndolo más legible y fácil de comprender .

Ejemplo de código duplicado

```
if (i == 1) {
    return "One";
}
if (i == 2) {
    return "Two";
}
if (i == 3) {
    return "Three";
}
if (i == 4) {
    return "Four";
}
if (i == 5) {
    return "Five";
}
if (i == 6) {
    return "Six";
}
if (i > 6) {
    throw new NotImplementedException();
}
```

Código resuelto

```
/*
 *
 * @author Luis Burbano, DCCO- ESPE, BettaCoders
 */
public class ComplexMetod {

    public String intToEnglishValue(int number){

        String[] arrayNumbersString = {"one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten"};
        if(number>=1 && number <= arrayNumbersString.length){
            return arrayNumbersString[number-1];
        }else{
            return "Enter numbers from 1 to 10 :)";
        }
    }
}
```

TEAM#4: **8/10**

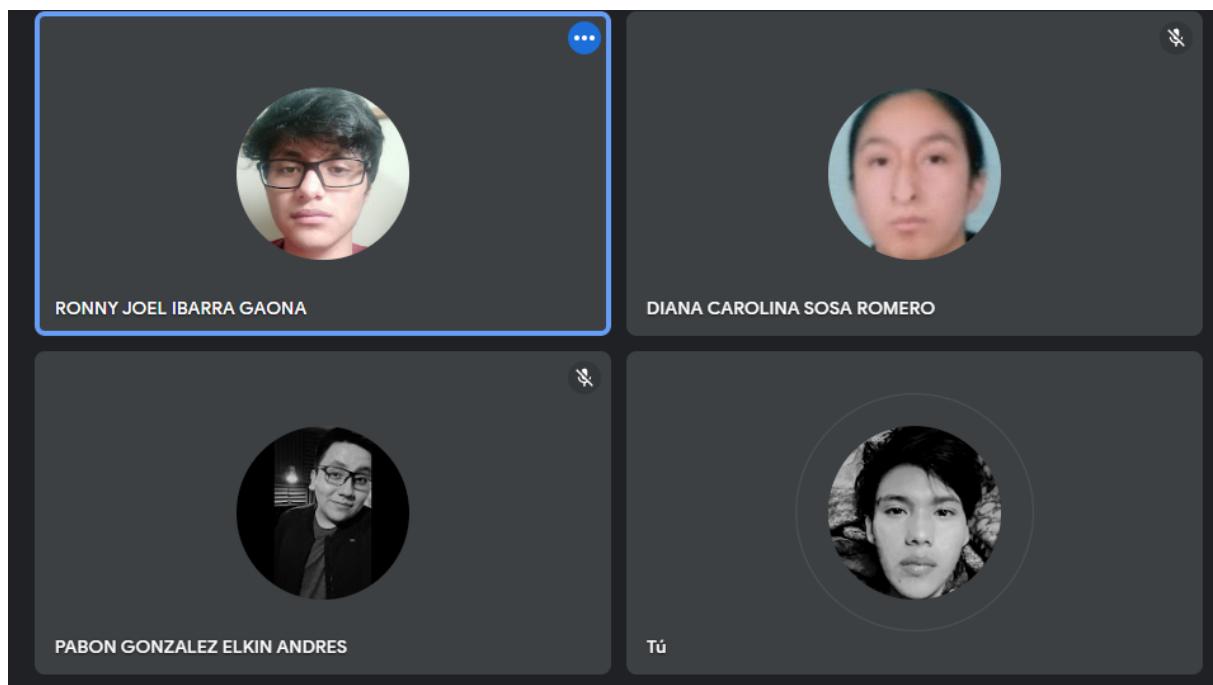
Caiza Pullotasig Widinson Danilo 4

Ibarra Gaona Ronny Joel 4

Pabon Gonzalez Elkin Andres 4

Sosa Romero Diana Carolina 4

Meet:



¿Qué es Pitfall?

"Pitfall" es un conjunto de malos hábitos o prácticas realizadas por un programador al momento de escribir su código, causando que sea confuso o difícil de comprender y que a la larga cause problemas en su mantenimiento o revisión por terceros.

Description

-Long Message Chain

Descripción del problema:

Un objeto tiene que delegar una llamada de método a un objeto contenido es decir es redundante, que a su vez tiene que delegar la llamada de método a un objeto que contiene, y así sucesivamente. Esto puede ocurrir

cuando el patrón decorador o una relación de composición es recursiva y se usa incorrectamente.

Consecuencias:

Bajo rendimiento gasto innecesario

Al momento de ejecutar el código

Complejidad accidental en el flujo de control en tiempo de ejecución.

Causas:

Uso inapropiado del patrón decorador o abuso de cualquier relación de composición recursiva en el código.

Evitar:

Lo primero que debemos hacer es razonar exhaustivamente en cómo se van a utilizar los decoradores o las composiciones recursivas y si estas se van a producir con largas frecuencias de cadenas de mensajes.

Reconocimiento:

Para hacer un buen reconocimiento debemos tener en cuenta los siguientes:

- Recorridos en inspecciones de diseño/código
- Prueba de rendimiento en el tiempo de ejecución y sus puntos de referencia.

Liberación:

Para la liberación lo que debemos de hacer es refactorizar(hacer un cambio en la estructura interna del software para hacerlo más fácil), también se puede utilizar estrategias, métodos de plantilla u otros patrones de diseño que no darán como resultado largas cadenas de mensajes y limitar la profundidad de la composición para los decoradores o relaciones de composición recursiva

Ejemplo

```
payrollService.GetDataForMonth(2022,  
5).GenerateReport().ConvertToSpreadsheet().Print()  
;
```

El ejemplo anterior contiene cuatro llamadas encadenadas. La solución es aplicar una refactorización llamada ocultar al delegado. Esta refactorización consiste en ocultar todas las llamadas bajo un único método. Por lo tanto, al aplicar esta refactorización, podría convertir nuestro primer ejemplo de C# en lo siguiente:

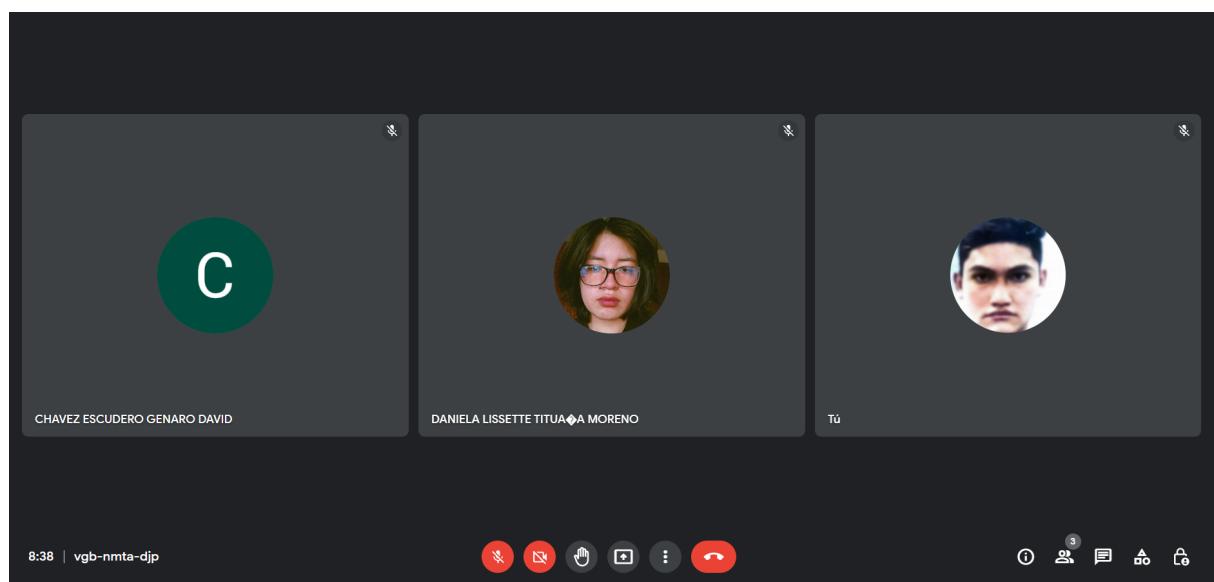
```
payrollService.PrintSpreadsheetForMonth(2022, 5);
```

TEAM#5 : Code Noob 8/10

Topic: DEAD CODE

Chavez Escudero Genaro David 5
Imbaquinga Guaña Jose Ricardo 5
Ponce Arguello Diego Armando 5
Tituaña Moreno Daniela Lissette 5

Meet:



Descripción del Problema: Código Muerto en programación significa, que una parte de código es ejecutable pero no se vuelve a reutilizar en ningún otro cálculo, lo cual también es un desperdicio de tiempo y memoria.

Consecuencias:

- Desperdicio de memoria
- Desperdicio de tiempo de ejecución
- Agrega complejidad al entendimiento del código.
- El Código muerto es engañoso si se lo comenta.
- Alguien puede volver a llamar al código muerto lo cual puede generar errores desastrosos.

Causas:

- En la mayoría de los casos donde existe Dead Code es debido al temor a perder código o malograr el sistema construido.
- Esto tiene relación directa con el desconocimiento de que los gestores de versiones permiten volver al pasado si es que se necesita ver algo que ha sido borrado.
- Otras causa muy común es el envejecimiento del código que debido a características obsoletas es una razón común a producir un Dead Code.

Cómo evitarlo:

- Inspecciones visuales
- Uso de un buen IDE
- Inspeccionar el código que no se usa durante meses o años.
- Si es un código heredado ejemplo de otra empresa, estar pendiente a comentarios de los anteriores desarrolladores

Como reconocerlo:

Instalar el sistema en ejecución para mantener registros del uso del código

Resultados jamás se miran o se utilizan

El código muerto hace que la base de código sea sustancialmente más difícil de mantener a escala

Código:

```
13
14     public static void main(String[] args) {
15         int addend1=10;
16         int addend2=20;
17         Variable restart1 is never read
18         ****
19         (Alt-Enter shows hints)
20
21         int restart1=12;
22         int restart2=20;
23         int sumResult;
24         int valor1=10;
25         int result;
26
27         sumResult=addend1+addend2;
28         result=valor1*sumResult;
29     }

```

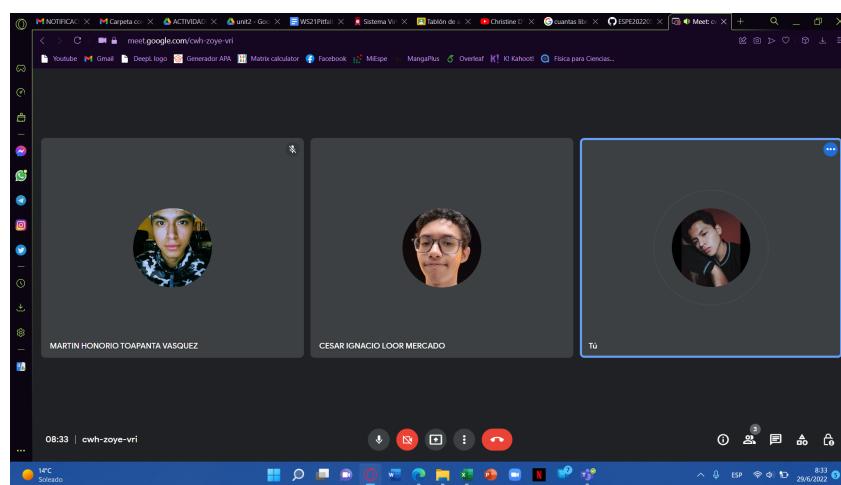
Código arreglado

```
public class ws21Pitfalls {

    public static void main(String[] args) {
        int addend1=10;
        int addend2=20;
        int sumResult;
        int valor1=10;
        int result;
        sumResult=addend1+addend2;
        result=valor1*sumResult;
        System.out.println("The result is"+result);
    }
}
```

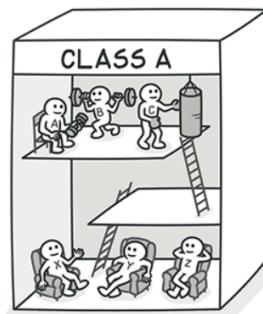
- a. Chicaiza Ortiz Frank Sebastian
- b. Loor Mercado Cesar Ignacio
- c. Pozo Analuisa Steven Jefferson
- d. Toapanta Vásquez Martín Honorio

REUNIÓN DE MEET:



TEMA: CAMPO TEMPORAL.

Los campos temporales son aquellos que obtienen sus valores en determinadas circunstancias y fuera de ellos están vacías.



Razones del problema:

Estos campos temporales a menudo son creados para tener una cantidad y uso único en el algoritmo y por ello no se usan el resto del tiempo.

Resolución del problema:

A partir de “Extraer clase” todos los campos temporales y el código en sí se pueden colocar en una clase separada, con esto se busca realizar lo mismo mediante la creación de un objeto a partir del método.

Ejemplo de campo temporal

Basic Examples of Temporary Fields

```
function nameToObject (name) {  
  var fullName = name.split(' ');\n  var firstName = fullName[0];\n  var lastName = lastName[1];\n\n  var name = {\n    firstName: firstName,\n    lastName: lastName\n  };\n\n  return name;\n}
```

Extraer las variables que no se relacionan con la acción y así convertirlas en un método

En estos casos muy específicos en lugar de pasar a otros métodos, el método se extiende para establecer o leer su valor

```
function sum (a, b) {  
  var total = a + b;\n  return total;\n}
```

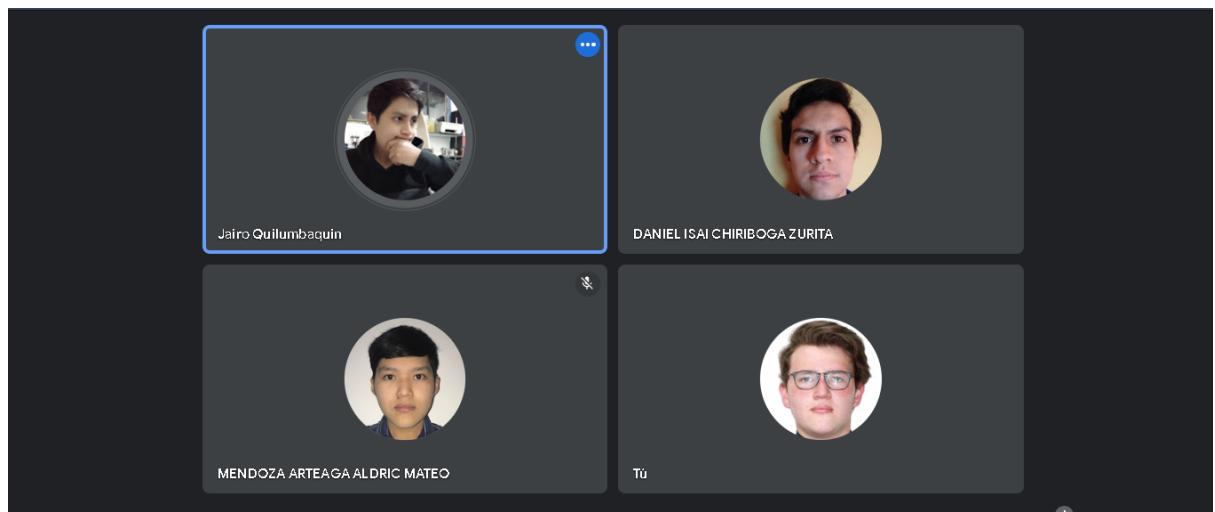
Example Solution

```
function nameToObject (name) {  
  var fullName = name.split(' ');\n\n  return {\n    firstName: fullName[0],\n    lastName: fullName[1]\n  };\n}
```

```
function sum (a, b) {  
  return a + b;\n}
```

TEAM#7 : 9/10

Chiriboga Zurita Daniel isai
Mendoza Arteaga Aldric Mateo
Quilumbaquin Lanchimba Jairo Smith
Villavicencio Campoverde Bolivar Josue
Meet:



Tema 7 Complejidad Condicional :

La complejidad condicional hace que el código sea más complicado de entender y, por lo tanto, de mantener. Además, la complejidad condicional suele ser un indicador de que el código está acoplado.

Problema:

La complejidad condicional se presenta cuando en un caso inicial se crea un código complejo y/o poco entendido para resolver un problema que se da en un momento, pero con el paso del tiempo y al querer actualizar el código este presenta un gran problema ya que por su complejidad no se puede actualizar o reemplazar partes del mismo haciendo que pierda calidad.

Consecuencias:

- El código se vuelve complicado de entender y mantener.
- La corrección de errores del código se vuelve casi imposible.
- El código corre el riesgo de volverse obsoleto.

Causas:

- Sobre analizar las formas de resolver un problema.
- Mal análisis e implementación del código por falta de tiempo.
- Falta de ética del programador

Ejemplo:

```
1  function getPayAmount() {  
2    let result;  
3    if (isDead){  
4      result = deadAmount();  
5    }else {  
6      if (isSeparated){  
7        result = separatedAmount();  
8      } else {  
9        if (isRetired){  
10          result = retiredAmount();  
11        }else{  
12          result = normalPayAmount();  
13        }  
14      }  
15    }  
16    return result;  
17 }
```

Como podemos observar en el primer ejemplo el código contiene una gran cantidad de condicionales “if” anidados dificultando la lectura y comprensión del código en cuestión.

```
1  function getPayAmount() {  
2    if (isDead) return deadAmount();  
3    if (isSeparated) return separatedAmount();  
4    if (isRetired) return retiredAmount();  
5    return normalPayAmount();  
6  }
```

Al limpiar el código y retirar los condicionales “else” resulta más fácil entender el código y por ende en un futuro mantenerlo será fácil.

Code with Conditional Complex:

```
input= new Scanner(System.in);

System.out.println("Seleccione el tipo de archivo que desea crear");
System.out.println("1.PDF\n");
System.out.println("2.DOC\n");
System.out.println("3.CSV\n");
System.out.println("4.TXT\n");
type=input.nextInt();

if(type==1){
    System.out.println("PDF was created");
}else{
    if(type==2){
        System.out.println("DOC was created");
    }else{
        if(type==3){
            System.out.println("CSV was created");
        }else{
            if(type==4){
                System.out.println("TXT was created");
            }else{
                }
            }
        }
    }
}
```

Code Refactored:

```
public static void main(String[] args) {  
    Scanner input;  
    int type;  
  
    input= new Scanner(System.in);  
  
    System.out.println("Seleccione el tipo de archivo que desea crear");  
    System.out.println("1.PDF\n");  
    System.out.println("2.DOC\n");  
    System.out.println("3.CSV\n");  
    System.out.println("4.TXT\n");  
    type=input.nextInt();  
  
    switch(type){  
  
        case 1:  
            System.out.println("PDF was created");break;  
        case 2:  
            System.out.println("DOC was created");break;  
        case 3:  
            System.out.println("CSV was created");break;  
        case 4:  
            System.out.println("TXT was created");break;  
    }  
  
}
```