CLEAN CODE

CODE SMELLS GENERALES

Daniel Blanco Calviño

G1: MÚLTIPLES LENGUAJES EN UN FICHERO

- No mezclar dos lenguajes en un mismo fichero. Ej: Java + HTML.

```
public String getUserProfilePage(User user) {
    return "<html>"
            + "<body>"
            +   "<p>" + user.getFirstName() + "</p>"
            +   "<p>" + user.getLastName() + "</p>"
            + "</body>"
            + "</html>";
}
```

G2: COMPORTAMIENTO OBVIO NO IMPLEMENTADO

Month month = MonthDate.StringToMonth(String monthName);

- Esperamos que nos convierta "July" a Month.JULY. También esperamos que haga lo mismo con "JULY" y "july".
- Si el código no hace lo que se espera, **se pierde la confianza en él** y se debe estar revisando lo que hace cada línea.

G3: COMPORTAMIENTO INCORRECTO EN LOS LÍMITES

```
private final List<String> thirtyOneDayMonths = Arrays.asList(new String[]{"january", "march",
        "may", "july", "august", "october", "december"});
private final List<String> thirtyDayMonths = Arrays.asList(new String[]{"april", "june",
        "september", "november"});

public int getNumberOfDays(String monthName) {
    if(thirtyOneDayMonths.contains(monthName.toLowerCase()) {
        return 31;
    }else if(thirtyDayMonths.contains(monthName.toLowerCase())) {
        return 30;
    }else {
        return 28;
    }
}
```

G4: ANULAR MECANISMOS DE SEGURIDAD

- Deshabilitar warnings del compilador.
- Comentar o ignorar tests que están fallando.
- Ignorar los reportes de plataformas de análisis de código (SonarQube)

G5: DUPLICIDAD

- La duplicidad en el código representa una **oportunidad perdida para crear una nueva abstracción.**
- El código repetido se puede **abstraer en un método o incluso una clase.**
- La duplicidad en varias clases puede indicar la necesidad de una jerarquía.

G6: CÓDIGO EN EL NIVEL DE ABSTRACCIÓN INCORRECTO

```
public class Vehicle() {
    public void turnOn();
    public void turnOff();
    public Integer getLeftAutonomyInKm();
    public void changeRadioChannel(RadioChannel selectedChannel);
}
```

G7: CLASES BASE DEPENDEN DE LAS DERIVADAS

- Las clases base no deben saber nada de las derivadas.

```
public class PlaneGeometry {
    /* ... */

    public int getNumberOfSides() {
        if(this instanceof Rectangle) return 4;
        if(this instanceof Triangle) return 3;
    }
}

public class Rectangle extends PlaneGeometry {}
public class Triangle extends PlaneGeometry {}
```

G8: DEMASIADA INFORMACIÓN

- Los módulos bien definidos tienen **interfaces muy pequeñas** que permiten hacer mucho con pocos métodos.
- Expón únicamente lo estrictamente necesario.
- En OOP, esconde tus datos, ofrece operaciones.

G9: CÓDIGO MUERTO

- El código que no se ejecuta **se debe eliminar inmediatamente.**
  - Funciones que no se llaman
  - Condiciones en *switch/case* o *if* que no se dan nunca.
  - *try/catch* con excepciones que nunca se lanzan.

- El software de control de versiones recordará todo!

G10: DISTANCIA VERTICAL

- Las variables y funciones **deben estar cerca de donde se usen.**
  - Las variables locales se deben declarar justo antes de usarlas.
  - Las funciones privadas deben estar justo debajo de la primera función que las usa.

- No queremos que código local esté situado a cientos de líneas de donde se usa.

G11: INCONSISTENCIA

- Si haces algo de cierta manera, **hazlo siempre de la misma forma.**
- Por ejemplo, si utilizas la palabra *Processor* en User*RequestProcessor,* no crees la clase *AdminRequestHandler*.
- Si eliges la palabra *delete* para las eliminaciones, no uses *remove*.

- La consistencia hace que **nuestro código sea mucho más fácil de leer!**

G12: BASURA

- Constructores por defecto sin implementación.
- getters y setters de absolutamente todas las variables privadas de una clase.
- Variables que no se usan.
- Funciones que nunca se llaman.

G13: ACOPLAMIENTO ARTIFICIAL

- Los elementos que no dependan uno del otro no deben estar acoplados.

```java
public class Circle {
    public static final Double PI = 3.14159265359;

    private Double radius;

    public Circle(Double radius) {
        this.radius = radius;
    }

    public Double getArea() {
        return Circle.PI * radius * radius;
    }
}
```

G14: ENVIDIA DEL ÁMBITO DE OTRA CLASE

- Los métodos de una clase deben estar interesados en sus propias variables y funciones.

```
public class GameRankCalculator {

    public Rank calculateRank(User user) {
        Double hoursPlayed = user.getHoursPlayed();
        int gamesPlayed = user.getTotalGames();
        int gamesWon = user.getVictories();

        Double winPercentage = gamesPlayed / gamesWon;
        int rankFromOneToTen = (hoursPlayed * winPercentage) % 10 +1;

        return new Rank(rankFromOneToTen);
    }
}
```

G15: FLAGS

Un flag como argumento suele indicar que la función hace más de una cosa

```
private static final Double PREMIUM_DISCOUNT_FACTOR = 0.2;
private static final Double REGULAR_DISCOUNT_FACTOR = 0.1;

public Double calculateDiscount(Item item, boolean isPremium) {
    if(isPremium) {
        return item.getPrice() * PREMIUM_DISCOUNT_FACTOR;
    }

    return item.getPrice() * REGULAR_DISCOUNT_FACTOR;
}
```

G16: INTENCIONES OCULTAS O DIFÍCILES DE APRECIAR

- El código debe ser lo **más expresivo posible.**
- De nada vale un código que ocupe poco espacio si no lo entendemos.

```
public int otCalc() {
    return isWkn * wkRte + (int) Math.round(0.5 * wkRte *
                    Math.max(0, wkRte - 400));
}
```

G17: RESPONSABILIDAD FUERA DE LUGAR

- El código debe ser escrito en el **lugar más natural para un lector.**
- Dónde ponemos *PI?* En *Math,* en la clase *Trigonometry* o en la clase *Circle*?
- No escribir el código donde más nos convenga en un momento, si no en donde se esperaría leer.

G18: MÉTODOS ESTÁTICOS INAPROPIADOS

- Los métodos estáticos **no operan con ninguna instancia.**
- *Math.abs(double number)* es un buen ejemplo

```
public class HourlyPayCalculator {
    public static Double calculatePay(Employee employee, Double overtimeRate) {
```

- ¿Queremos que sea polimórfica? En ese caso debería ser una función de Employee.

G19: NO USAR VARIABLES EXPLICATORIAS

- Las variables intermedias pueden dejar más clara una operación.
- Ejemplo: cálculo del área de un triángulo rectángulo dada la hipotenusa y un cateto.

```
public Double calcularArea(Double hipotenusa, Double cateto) {
    return Math.sqrt(hipotenusa*hipotenusa - cateto*cateto) * cateto / 2;
}

public Double calcularArea(Double hipotenusa, Double cateto) {
    Double cateto_2 = Math.sqrt(hipotenusa*hipotenusa - cateto*cateto);

    return cateto * cateto_2 / 2;
}
```

G20: FUNCIONES QUE NO DICEN LO QUE HACEN

- Los nombres de las funciones son explicatorios.

```
Date newDate = oldDate.add(5);
```

- ¿Días, horas, minutos, segundos...?

G21: NO CONOCER EL ALGORITMO

- Debemos entender perfectamente los algoritmos complicados.
- Si no los entendemos, nuestro código no será todo lo limpio que podría ser.

G22: TENER DEPENDENCIAS LÓGICAS EN LUGAR DE FÍSICAS

- Si una clase depende de otra, debe depender de forma física.

```
public class ShoppingController {
    private Integer MAXIMUM_NUMBER_OF_ITEMS = 10;

    public void confirmPurchase(ShoppingCart cart) {
        if(cart.getItems().size() <= MAXIMUM_NUMBER_OF_ITEMS) {
            purchase(cart.getItems());
        }
    }
}
```

```
public class ShoppingController {
    private Integer MAXIMUM_NUMBER_OF_ITEMS = 10;

    public void confirmPurchase(ShoppingCart cart) {
        if(cart.getItems().size() <= cart.getMaximumNumberOfItems()) {
            purchase(cart.getItems());
        }
    }
}
```

G23: NO USAR POLIMORFISMO EN LUGAR DE IF/ELSE

- Se debe preferir polimorfismo en lugar de if/else o switch/case

```
public void speak(Animal animal) {
    if(animal.getType == CAT) {
        System.out.println("Miauu!");
    }else if(animal.getType == DOG) {
        System.out.println("Guau!");
    }else if(animal.getType == COW) {
        System.out.println("MUUUU!");
    }
}
```

G23: NO USAR POLIMORFISMO EN LUGAR DE IF/ELSE

```java
public class Cat extends Animal {
    public void speak() {
        System.out.println("Miauu!");
    }
}

public class Dog extends Animal {
    public void speak() {
        System.out.println("Guau!");
    }
}

public class Cow extends Animal {
    public void speak() {
        System.out.println("MUUUU!");
    }
}
```

- **Apunte**: usar **switchs o if/elses** en partes del programa **donde sea más probable añadir funcionalidad que añadir tipos.**

G24: NO SEGUIR CONVENCIONES

- Seguir las normas sobre nombres de clases, variables, tamaño de línea, donde poner las llaves etc.
- Todo **el equipo** de desarrollo **debe seguir las mismas normas.**

G25: USAR NÚMEROS MÁGICOS EN LUGAR DE CONSTANTES

- Usar variables para las constantes que se usen en el código.

```
public Double getAnnualSalary() {
    return 200.0 * 6.0 * 12.0;
}

public Double getAnnualSalary() {
    return WORKING_DAYS * WORKING_HOURS_PER_DAY * SALARY_PER_HOUR;
}
```

G26: NO SER PRECISOS

- **Debes ser preciso** en las decisiones que tomes sobre tu código.
- No ser vago con tus decisiones:
  - Si vas a tratar con dinero, usa enteros y trata bien el redondeo.
  - Si tratas con concurrencia, asegúrate de que no hay carreras críticas.
  - Si hay métodos que pueden lanzar excepciones que romperán tu programa, trátalas.
  - ...

G27: DARLE MÁS PESO A CONVENCIONES QUE A LA ESTRUCTURA

- Las convenciones son importantes, pero **el diseño de tu software es más importante.**
- Por ejemplo, no uses directamente la estructura controller-service-repository de Spring si no es la apropiada para tu código.

G28: NO ENCAPSULAR CONDICIONALES

- Las condiciones encapsuladas en una función son **mucho más fáciles de leer.**

```
if(user.getRole != Role.ADMIN && user.isInactive())

if(shouldBeDeleted(user))
```

G29: USAR CONDICIONALES POSITIVOS

- Si es posible, usar condicionales positivos.

```
if(!user.shouldNotBeDeleted())

if(user.shouldBeDeleted)
```

G30: LAS FUNCIONES HACEN MÁS DE UNA COSA

- Las funciones **deben hacer una única cosa.**
- Las funciones que hacen una cosa son **mucho más legibles.**
- Si una función hace más de una cosa, se debe refactorizar en dos o más funciones.

G31: ACOPLAMIENTOS TEMPORALES ESCONDIDOS

```
public class FileReader {
    private ReaderHandle handle;
    private FileBuffer fileBuffer;

    public void read(String filePath) {
        handle = getHandle(filePath);
        readFile();
        closeFile();
    }
}
```

```
public class FileReader {
    private ReaderHandle handle;
    private FileBuffer fileBuffer;

    public void read(String filePath) {
        handle = getHandle(filePath);
        readFile(handle);
        closeFile(handle);
    }
}
```

G32: SER ARBITRARIO

- **No seas arbitrario.**
- Si la estructura de tu código es arbitraria, otros harán lo mismo sobre él.
- Si tus decisiones sobre todo el sistema son consistentes, otros compañeros la seguirán, manteniendo el código limpio.

G33: NO ENCAPSULAR LAS CONDICIONES LÍMITE

```
if(level + 1 == game.getMaximumLevel()) {
    loadFinalBoss();
}

Integer nextLevel = level +1;
if(nextLevel == game.getMaximumLevel()) {
    loadFinalBoss();
}
```

G34: FUNCIONES CON MÁS DE UN NIVEL DE ABSTRACCIÓN

- Las funciones deben tratar con un único nivel de abstracción.

```
public class Person {

    public void drive(Car car) {
        car.openDoor();
        car.setDriver(this);
        car.getBattery().connect();
        car.getEngine().start();
    }
}
```

```
public class Car {
    Battery battery;
    Engine engine;

    public void start() {
        battery.connect();
        engine.start();
    }
}
```

```
public class Person {

    public void drive(Car car) {
        car.openDoor();
        car.setDriver(this);
        car.start();
    }
}
```

G35: CONFIGURACIONES NO MODIFICABLES A ALTO NIVEL

- El software debe ser configurable a alto nivel.
- Evitar que los parámetros de configuración estén mezclados con la lógica de negocio, a bajo nivel.

```
public static void main(String[] args) throws Exception {
    Arguments arguments = parseCommandLineArguments(args);
}

public class Arguments {
    public static final int DEFAULT_PORT = 80;
    public static final String DEFAULT_PATH = "/usr/exampleprogram";
    public static final String DEFAULT_DATA_PATH = "~/programdata"
}
```

G36: NAVEGACIÓN TRANSITIVA

- Un módulo debe **saber lo menos posible sobre los demás.**
- Si *A* usa *B* y *B* usa *C*, **evitar *a.getB().getC().exampleMethod();***
- Esta es la **Ley de Demeter**. *"Escribir código tímido"*.
- Si queremos intercalar un *D* entre *B* y *C*, tendrías que buscar todos los *a.getB().getC()* para cambiarlos a *a.getB().getD().getC()*.
- Así se forman las arquitecturas rígidas.