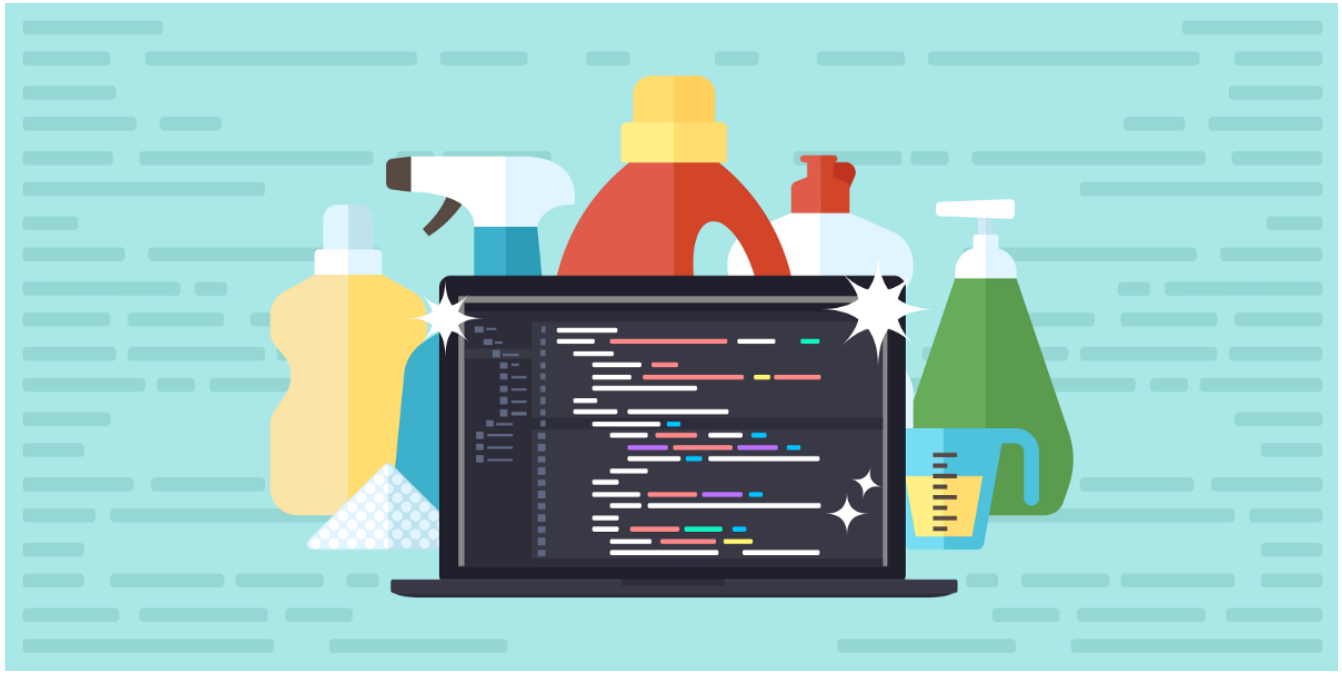
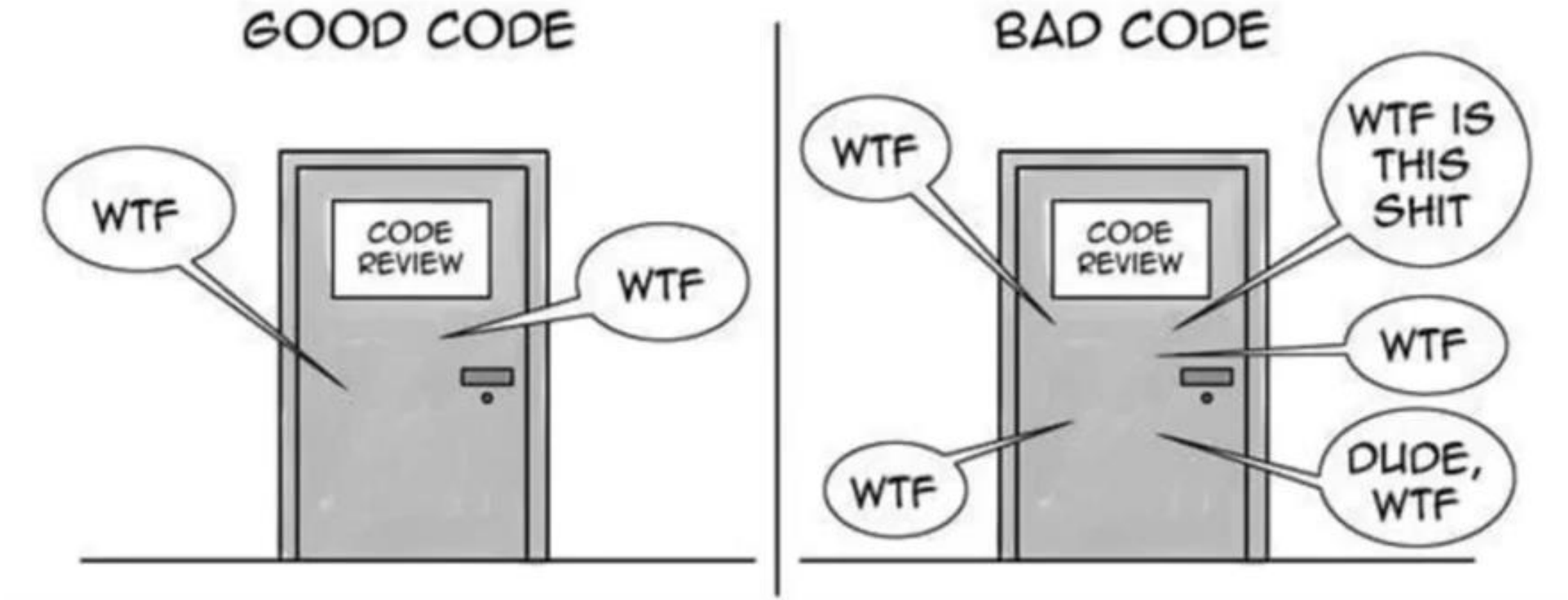


Producción del Software

Clean Code

Nelson Monzón López
nelson.monzon@ulpgc.es
Agustin Salgado de la Nuez
agustin.salgado@ulpgc.es
Daniel Santana Cedrés
daniel.santanacedres@ulpgc.es


GOOD CODE
WTF
CODE REVIEW
WTF
BAD CODE
WTF
WTF IS THIS SHIT
CODE REVIEW
WTF
WTF
DUDE, WTF
THE ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE

Una serie de síntomas en el código que nos vienen a indicar que tal vez no se están haciendo las cosas de una forma del todo correcta, lo que puede llevar a que haya algún problema a futuro y un problema de trasfondo.

- **No tienen por qué ser errores o bugs de programación**, ya que pueden no ser técnicamente incorrectos y el programa funcione correctamente.
- **Indican deficiencias en el diseño** y pueden hacer que se realice un desarrollo más lento.
- Aumentan el **riesgo de errores o fallos** en el futuro.

Deuda Técnica

Esfuerzo adicional por realizar un desarrollo rápido y sencillo en lugar del "mejor enfoque".

Tener presente que… **mejor enfoque, normalmente más tiempo**

Las **prisas** nos llevan a tener deuda técnica

Supondrá esfuerzo extra ya que a futuro vamos a tener que pagar esa deuda

Multiplicará el tiempo de desarrollo del proyecto inicial

Deuda Técnica

Deuda Técnica

- **Deliberada**: Dejar de lado buenas prácticas para obtener un resultado anticipado o TTM (time to market). Son intencionales.

- **Inadvertida**: Son las que aparecen cuando nuestro código necesita mejoras con el tiempo (mal diseño, degradación, nuevas funcionalidades)

Consecuencias de la deuda técnica

Con el tiempo genera más y más intereses.

- **Menos tiempo para liberar nuevas funcionalidades**
- **Más tiempo corrigiendo** problemas recurrentes.
- **La falta de documentación es deuda técnica** (perder tiempo de desarrollo por preguntas que se resuelven con una wiki o breve doc.)
- Problema de escalabilidad (quizás funciona con pocos datos pero y con muchos?)
- **Confiabilidad del equipo y del sistema.**
  - Usuarios se quejan de programas lentos
  - Menos confianza de la gerencia en el producto encargado
- Horas/coste de refactorización

Métricas

- **Defectos:** Como desarrolladores debemos tener un estricto seguimiento de nuestros bugs. Tanto los resueltos como los que siguen ahí, los que siguen nos dan una idea de cuánto tiempo debemos invertir en solucionar un problema mientras que los resueltos nos dan una idea de que tan bien estamos mejorando nuestro sistema.

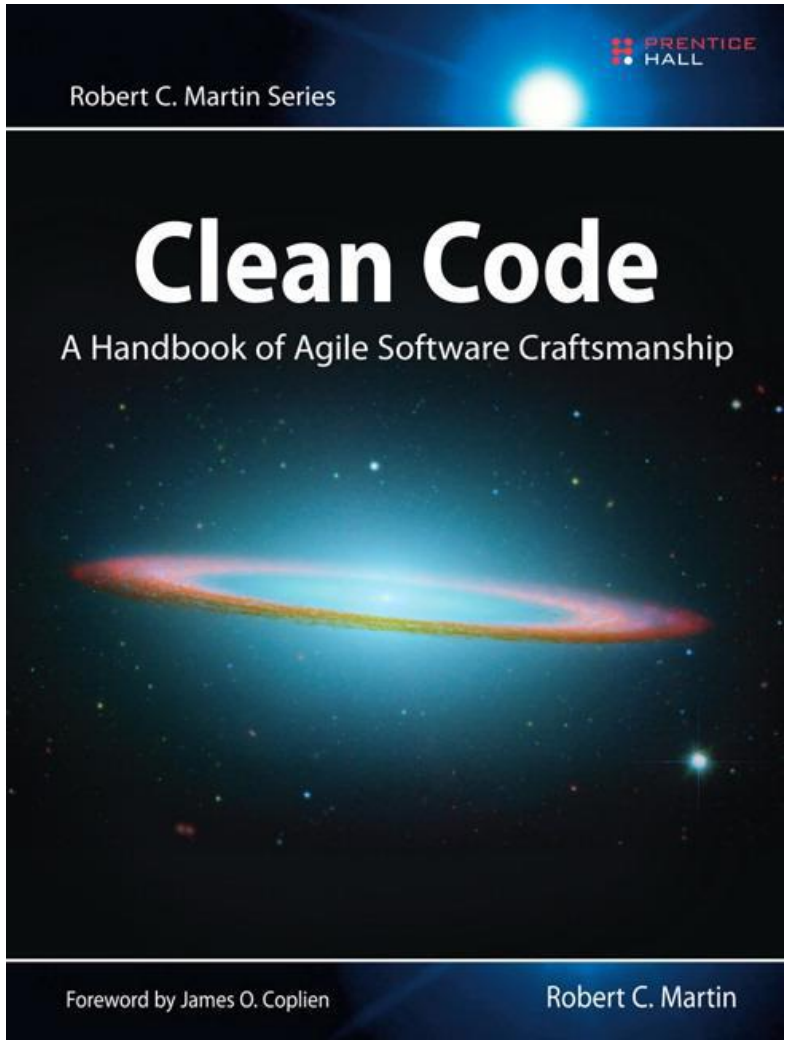
- **Calidad en el código:** A diferencia de los bugs, la calidad del código nos habla de lo que está por debajo de la superficie de nuestro sistema. Realizar análisis estático del código para ver su complejidad ciclomática, árboles de profundidad de herencia o que no se respetan los estándares y convenciones de código.

- **Adueñarse del código**: A medida que nuestros equipos crecen es más probable que aparezcan deudas técnicas no intencionales. Todo el equipo debe tener una actitud activa sobre el código. Los gerentes tienen que saber quién está trabajando en que sección del sistema en caso de un problema y el equipo de desarrollo se debe hacer responsable de dicho código, aun si el mismo es heredado.

- **Cohesión de código**: Buscando que nuestro sistema tenga una alta Cohesión y un bajo acoplamiento entre sus partes nos permite saber que el código está más auto contenido y por tanto es más fácil elaborar pruebas individuales para cada parte.

- **Puntos calientes**: Buscar lugares de nuestras soluciones que tiene mucho trabajo de reescritura o cambios. Debemos tener un cuidado especial en estos sectores ya que pueden estar evidenciando un área que puede necesitar una reingeniería o refactorización para mejorar.

Código Limpio



Robert C. Martin

El código limpio (clean code), no es algo recomendado o deseable, es algo **vital para las compañías y los programadores**

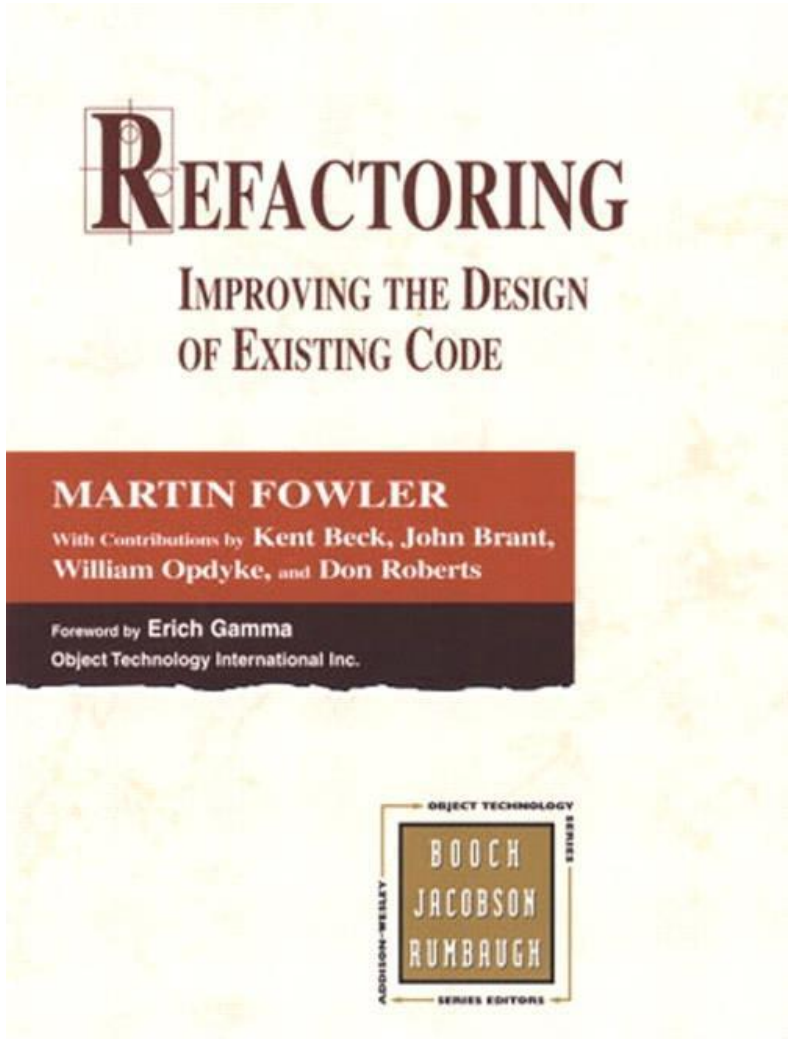
Código Limpio

- Mantenimiento del código más fácil y rápido
- Permite añadir nuevas funcionalidades de forma más sencilla
- Favorece una mayor reusabilidad y calidad del código, así como la encapsulación

Código Limpio

- Es simple y directo.
  - No debe esconder intenciones
  - Si hace falta incluir un comentario **probablemente** no deba considerarse limpio.
- Es específico y no especulativo.
- Contiene pruebas.
- Está redactado de forma legible.

Refactoring



Martin Fowler

[online catalog of refactorings]

Este libro nos ayuda a desarrollar el olfato y detectar “code smells”

Refactorizar es mejorar código no cambiar valor funcional

Refactorizar debe ser una tarea cotidiana

Principios SOLID

S: Single responsibility principle o Principio de responsabilidad única

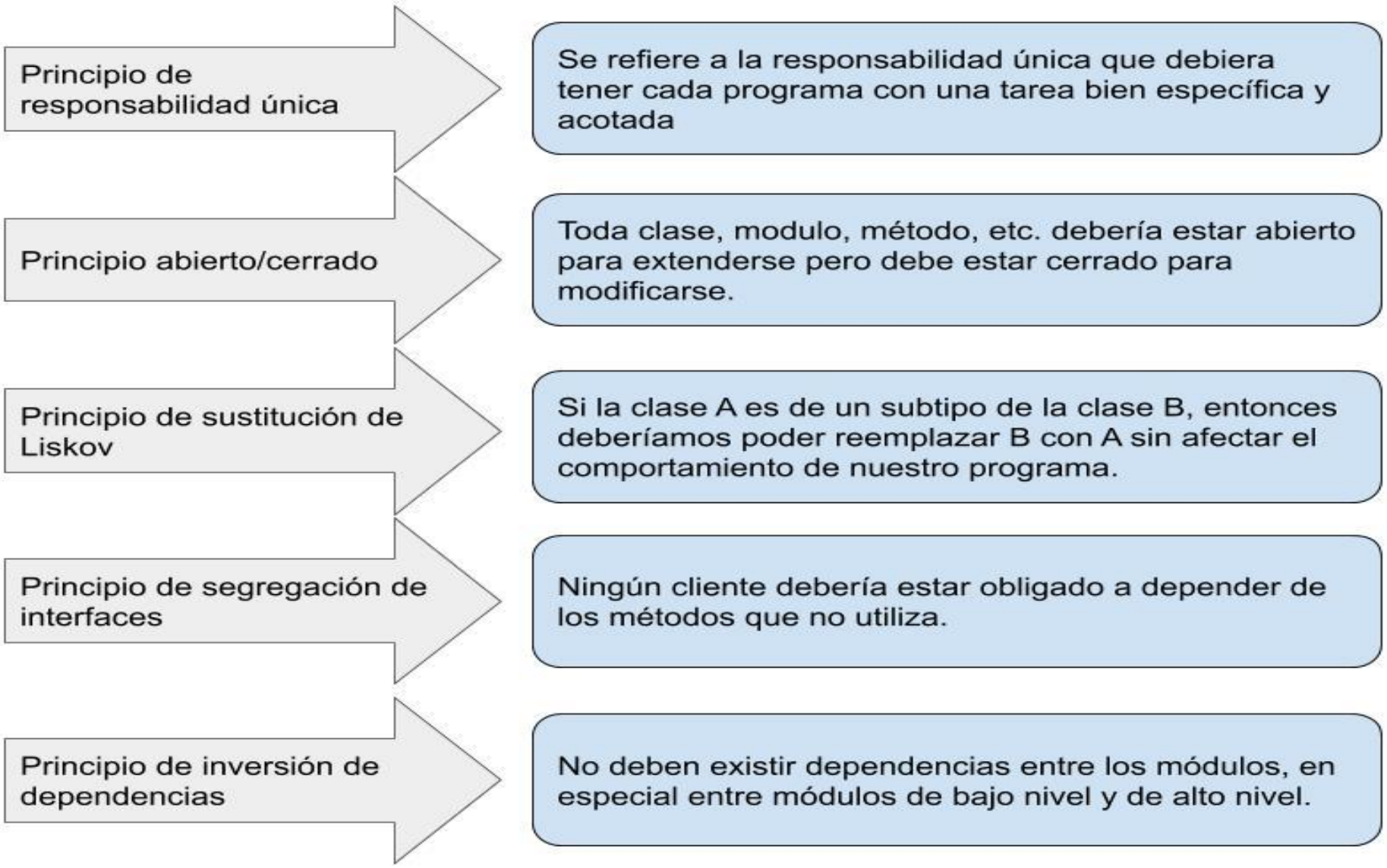
O: Open/closed principle o Principio de abierto/cerrado

L: Liskov substitution principle o Principio de sustitución de Liskov

I: Interface segregation principle o Principio de segregación de la interfaz

D: Dependency inversion principle o Principio de inversión de dependencia

Principios SOLID



Cualidad 1

El código limpio no hace demasiadas cosas, el código limpio es enfocado.

Cada clase, cada método o cualquier otro tipo de modulo debería cumplir con el **principio de responsabilidad única.**

Un módulo sólo debiera ser responsable de un único aspecto de los requisitos del sistema

Cualidad 1

El código limpio no hace demasiadas cosas, el código limpio es enfocado.

```
class UserLogin {

    private final DataBase db;

    UserLogin(DataBase db) {
        this.db = db;
    }

    void login(String userName, String password) {
        User user = db.findUserByUserName(userName);
        if (user == null) {
            // do something
        }
        // login process..
    }

    void sendEmail(User user, String msg) {
        // sendEmail email to user
    }

}
```

Cualidad 1

El código limpio no hace demasiadas cosas, el código limpio es enfocado.

```
class UserLogin {

    private final DataBase db;

    UserLogin(DataBase db) {
        this.db = db;
    }

    void login(String userName, String password) {
        User user = db.findUserByUserName(userName);
        if (user == null) {
            // do something
        }
        // login process..
    }
}
```

```
class EmailSender {

    void sendEmail(User user, String msg) {
        // send email to user
    }

}
```

Cualidad 1

El código limpio no hace demasiadas cosas, el código limpio es enfocado.

```
class Factura
{
    public string Codigo { get; set; }
    public DateTime FechaEmision { get; set; }
    public decimal ImporteFactura { get; set; }
    public decimal ImporteIVA { get; set; }
    public decimal ImporteDeduccion { get; set; }
    public decimal ImporteTotal { get; set; }
    public decimal PorcentajeDeduccion { get; set; }


    // Método que calcula el total de la factura
    public void CalcularTotal()
    {
        // Calculamos la deducción
        ImporteDeduccion = (ImporteFactura * PorcentajeDeduccion) / 100;
        // Calculamos el IVA
        ImporteIVA = ImporteFactura * 0.16m;
        // Calculamos el total
        ImporteTotal = (ImporteFactura - ImporteDeduccion) + ImporteIVA;
    }
}
```

Cualidad 1

El código limpio no hace demasiadas cosas, el código limpio es enfocado.

```
class IvaNormal
{
     private const decimal PORCENTAJE_IVA_NORMAL = 0.16m;
     public readonly decimal PorcentajeIvaNormal
    {
        get
        {
            return PORCENTAJE_IVA_NORMAL;
        }
    }

    public decimal CalcularIVA(decimal importe)
    {
        return importe * PORCENTAJE_IVA_NORMAL;
    }
}

class Deduccion
{
    private decimal m_PorcentajeDeduccion;

    public Deduccion(decimal porcentaje)
    {
        m_PorcentajeDeduccion = porcentaje;
    }

    public decimal CalcularDeduccion(decimal importe)
    {
        return (importe * m_PorcentajeDeduccion) / 100;
    }
}
```

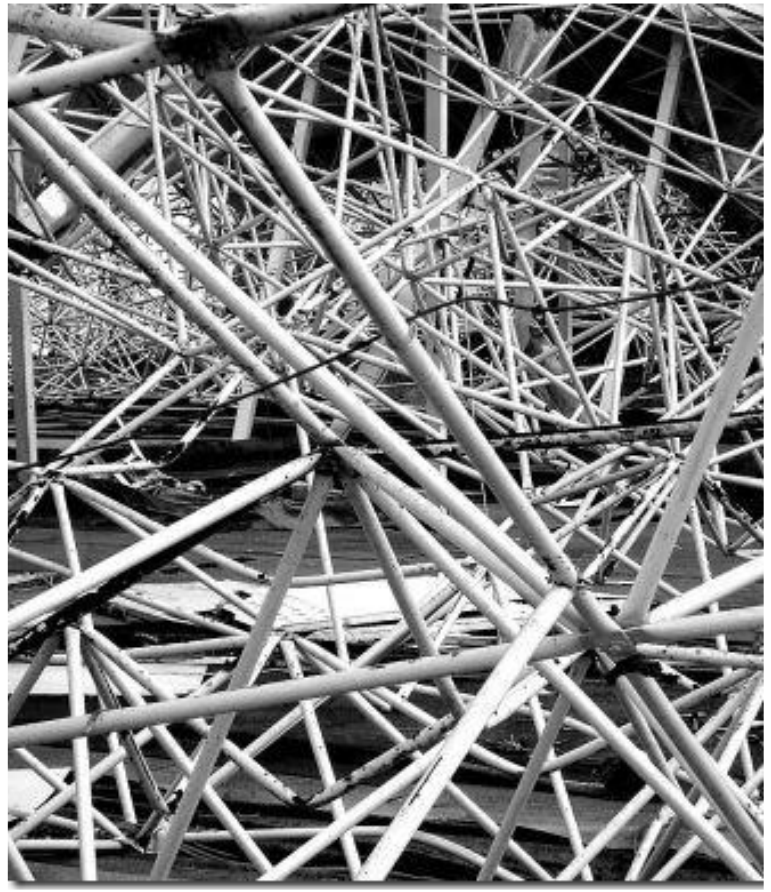
```
class FacturaFactorizada
{
    public string Codigo { get; set; }
    public DateTime FechaEmision { get; set; }
    public decimal ImporteFactura { get; set; }
    public decimal ImporteIVA { get; set; }
    public decimal ImporteDeduccion { get; set; }
    public decimal ImporteTotal { get; set; }
    public decimal PorcentajeDeduccion { get; set; }

    // Método que calcula el total de la factura
    public void CalcularTotal()
    {
        // Calculamos la deducción
        Deduccion deduccion = new Deduccion(PorcentajeDeduccion);
        ImporteDeduccion = deduccion.CalcularDeduccion(ImporteFactura);
        // Calculamos el IVA
        IvaNormal iva = new IvaNormal();
        ImporteIVA = iva.CalcularIVA(ImporteFactura);
        // Calculamos el total
        ImporteTotal = (ImporteFactura - ImporteDeduccion) + ImporteIVA;
    }
}
```

Cualidad 2

El código limpio no usa rodeos ni soluciones ofuscadas

La lógica debe ser directa, clara, para que a los fallos les sea difícil esconderse.



Cualidad 2

El código limpio no usa rodeos ni soluciones ofuscadas

```
public String weekday1(int day) {
    switch (day) {
        case 1:
            return "Monday";
        case 2:
            return "Tuesday";
        case 3:
            return "Wednesday";
        case 4:
            return "Thursday";
        case 5:
            return "Friday";
        case 6:
            return "Saturday";
        case 7:
            return "Sunday";
        default:
            throw new InvalidOperationException("day must be in range 1 to 7");
    }
}
```

Cualidad 2

El código limpio no usa rodeos ni soluciones ofuscadas

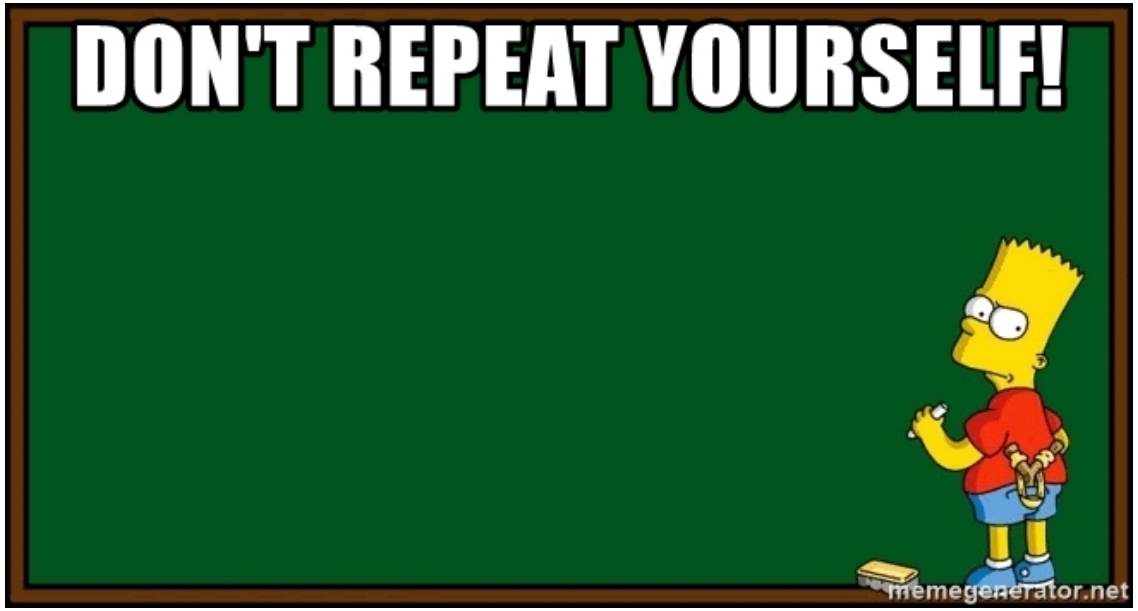
```
public String weekday2(int day) {
    if ((day < 1) || (day > 7)) throw new InvalidOperationException("day must be in range 1 to 7");

    string[] days = {
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"
    };

    return days[day - 1];
}
```

Cualidad 3

El código limpio no es redundante



Cumplir con la regla **DRY** (Don't Repeat Yourself).

Si se aplica correctamente DRY, los cambios de los requisitos solo debieran obligan a realizar cambios en un único lugar.

Cualidad 3

El código limpio no es redundante

```
public class Mechanic {
        public void serviceBus() {
                System.out.println("Servicing bus now");
                //Process washing
        }
        public void serviceCar() {
                System.out.println("Servicing car now");
                //Process washing
        }
}
```

Cualidad 3

El código limpio no es redundante

```
public class Mechanic {
        public void serviceBus() {
                System.out.println("Servicing bus now");
                washVehicle();
        }
        public void serviceCar() {
                System.out.println("Servicing car now");
                washVehicle();
        }
        public void washVehicle() {
                //Process washing
        }
}
```

Cualidad 3

El código limpio no es redundante

```
public class Main {
        static void showFirstMessage(){
                System.out.println(“Mens1”);
        }

        static void showSecondMessage(){
                System.out.println(“Mens2”);
        }

        static void showThirdMessage();{
                System.out.println(“Mens3”);
        }


        public static void main(String[] args) {
                showFirstMessage();
                showSecondMessage();
                showThirdMessage();
        }

}
```

Cualidad 3

El código limpio no es redundante

```
public class Main {
        static void showMessage(String message) {
                System.out.println(message);
        }

        public static void main(String[] args) {
                showMessage(“Mens1");
                showMessage(“Mens2");
                showMessage(“Mens3");
        }

}
```

Cualidad 3

El código limpio no es redundante

```
public class Dog  {
    public void eatFood() {
        System.out.println("Eat Food");
    }
    public void woof() {
        System.out.println("Dog Woof! ");
    }
}
```

```
public class Cat {
    public void eatFood() {
        System.out.println("Eat Food");
    }
    public void meow() {
        System.out.println("Cat Meow!");
    }
}
```

Cualidad 3

El código limpio no es redundante

```
public class Animal {
    public void eatFood() {
        System.out.println("Eat Food");
    }
}
```

```
public class Dog extends Animal {
    public void woof() {
        System.out.println("Dog Woof! ");
    }
}
public class Cat extends Animal {
    public void meow() {
        System.out.println("Cat Meow!");
    }
}
```

Cualidad 4

El código limpio es placentero de leer

Para lograr esto debemos cumplir con el principio KISS

Se debe evitar la complejidad innecesaria.

Esto es una práctica que nos alienta a enfocarnos exclusivamente en las cosas más simples que hagan funcionar al software.

Cualidad 4

```
public void processTransactions(List<Transaction> transactions) {
    if (transactions != null && transactions.size() > 0) {
        for (Transaction transaction : transactions) {
            if (transaction.getType().equals("PAYMENT")) {
                if (transaction.getStatus().equals("OPEN")) {
                    if (transaction.getMethod().equals("CREDIT_CARD")) {
                        processCreditCardPayment(transaction);
                    } else if (transaction.getMethod().equals("PAYPAL")) {
                        processPayPalPayment(transaction);
                    } else if (transaction.getMethod().equals("PLAN")) {
                        processPlanPayment(transaction);
                    }
                } else {
                    System.out.println("Invalid transaction type");
                }
            } else if (transaction.getType().equals("REFUND")) {
                if (transaction.getStatus().equals("OPEN")) {
                    if (transaction.getMethod().equals("CREDIT_CARD")) {
                        processCreditCardRefund(transaction);
                    } else if (transaction.getMethod().equals("PAYPAL")) {
                        processPayPalRefund(transaction);
                    } else if (transaction.getMethod().equals("PLAN")) {
                        processPlanRefund(transaction);
                    }
                } else {
                    System.out.println("Invalid transaction type");
                }
            }
        }
    }
}
```

Cualidad 5

El código limpio puede ser modificado fácilmente por cualquier otro desarrollador

No programamos para nosotros ni para el compilador

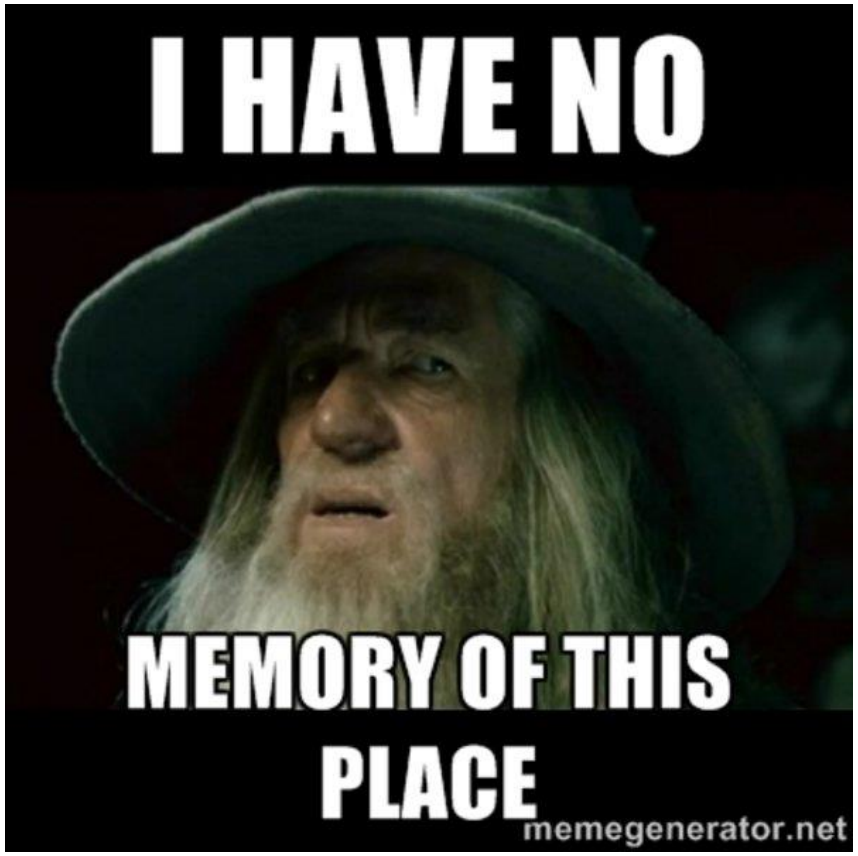
Trabajamos en equipo

Mantenemos el código en equipo

Escribimos código para otro desarrollador.

Cualidad 5

El código limpio puede ser modificado fácilmente por cualquier otro desarrollador



No programamos para nosotros ni para el compilador

Trabajamos en equipo

Mantenemos el código en equipo

Escribimos código para otro desarrollador.

Además, en pocos meses nosotros mismos vamos a ser "ese otro desarrollador".

Cualidad 5

El código limpio puede ser modificado fácilmente por cualquier otro desarrollador

```
public void processTransactions(List<Transaction> transactions) {
    if (transactions != null && transactions.size() > 0) {
        for (Transaction transaction : transactions) {
            if (transaction.getType().equals("PAYMENT")) {
                if (transaction.getStatus().equals("OPEN")) {
                    if (transaction.getMethod().equals("CREDIT_CARD")) {
                        processCreditCardPayment(transaction);
                    } else if (transaction.getMethod().equals("PAYPAL")) {
                        processPayPalPayment(transaction);
                    } else if (transaction.getMethod().equals("PLAN")) {
                        processPlanPayment(transaction);
                    }
                } else {
                    System.out.println("Invalid transaction type");
                }
            } else if (transaction.getType().equals("REFUND")) {
                if (transaction.getStatus().equals("OPEN")) {
                    if (transaction.getMethod().equals("CREDIT_CARD")) {
                        processCreditCardRefund(transaction);
                    } else if (transaction.getMethod().equals("PAYPAL")) {
                        processPayPalRefund(transaction);
                    } else if (transaction.getMethod().equals("PLAN")) {
                        processPlanRefund(transaction);
                    }
                } else {
                    System.out.println("Invalid transaction type");
                }
            }
        }
    }
}
```

```java
public void processTransactions(List<Transaction> transactions) {
    if (transactions != null && transactions.size() > 0) {
        for (Transaction transaction : transactions) {
            if (transaction.getType().equals("PAYMENT")) {
                if (transaction.getStatus().equals("OPEN")) {
                    if (transaction.getMethod().equals("CREDIT_CARD")) {
                        processCreditCardPayment(transaction);
                    } else if (transaction.getMethod().equals("PAYPAL")) {
                        processPayPalPayment(transaction);
                    } else if (transaction.getMethod().equals("PLAN")) {
                        processPlanPayment(transaction);
                    }
                } else {
                    System.out.println("Invalid transaction type");
                }
            } else if (transaction.getType().equals("REFUND")) {
                if (transaction.getStatus().equals("OPEN")) {
                    if (transaction.getMethod().equals("CREDIT_CARD")) {
                        processCreditCardRefund(transaction);
                    } else if (transaction.getMethod().equals("PAYPAL")) {
                        processPayPalRefund(transaction);
                    } else if (transaction.getMethod().equals("PLAN")) {
                        processPlanRefund(transaction);
                    }
                } else {
                    System.out.println("Invalid transaction type");
                }
            }
        }
    }
}
```

Refactoring

```java
public void processTranssactions(List<Transaction> transactions){

    if (transactions == null || transactions.size() <= 0){
        System.out.println("Problem");
        return;
    }

    for (Transaction transaction : transactions){

        if (!transaction.getStatus.equals("OPEN")){
            System.out.println("Invalid transaction type");
            continue;
        }

        if (transaction.getMethod.equals("PAYPAL")){
            if (transaction.getType().equals("PAYMENT")){
                processPayPalPayment(transaction);
            }
            else{
                processPayPalRefund(transaction);
            }
            continue;
        }

        if (transaction.getMethod.equals("PLAN")){
            if (transaction.getType().equals("PAYMENT")){
                processPlanPayment(transaction);
            }
            else{
                processPlanRefund(transaction);
            }
            continue;
        }

        if (transaction.getMethod.equals("CREDIT_CARD")){
            if (transaction.getType().equals("PAYMENT")){
                processCreditCardPayment(transaction);
            }
            else{
                processCreditCardRefund(transaction)
            }
            continue;
        }
    }
}
```

Cualidad 5

El código limpio puede ser modificado fácilmente por cualquier otro desarrollador

Cualidad 6

El código limpio debe tener dependencias mínimas

Mientras más dependencias tenga, más difícil va a ser de mantener y cambiar en el futuro.

Cualidad 6

El código limpio debe tener dependencias mínimas

```
void DisplayFriends(FacebookUser user, FacebookClient client) {
    Friend[] = (Friend[])client.GetConnection().ExecuteCommandForResult("getFriends:" + user.Name);
    ...
}
```

La clase de DisplayFriends está acoplada tanto a FacebookClient como a Connection

Si FacebookClient cambia repentinamente el tipo de conexión que utiliza, la aplicación ya no podrá obtener a los amigos de Facebook.

Cualidad 6

El código limpio debe tener dependencias mínimas

```
void DisplayFriends(FacebookUser user, FacebookClient client)
{
    Friend[] = client.GetFriends(user);
    ...
}
```

Podemos eliminar el acoplamiento entre la clase de DisplayFriends y Connection al pedirle a FacebookClient que nos brinde lo que necesitamos para nosotros.

Cualquier cambio llevado a su comportamiento interno no dañará nuestra propia clase.

Cualidad 6

El código limpio debe tener dependencias mínimas

```
int main (int argc, char *argv[]){

    const char *image1  = "C:\images\im1.png";
    const char *image2  = "C:\images\im2.png";

    float *I1, *I2;

    bool correct1 = read_image (image1, &I1);
    bool correct2 = read_image (image2, &I2);

    free (I1);
    free (I2);

    return 0;
}
```

Cualidad 6

El código limpio debe tener dependencias mínimas

```
int main (int argc, char *argv[]){

    const char *image1  = argv[1];
    const char *image2  = argv[2];

    float *I1, *I2;

    bool correct1 = read_image (image1, &I1);
    bool correct2 = read_image (image2, &I2);

    free (I1);
    free (I2);

    return 0;
}
```

Cualidad 7

El código limpio es pequeño

Tanto las clases como los métodos deberían ser cortos, preferentemente con pocas líneas de código.

Mientras más dividamos el código, más fácil se vuelve digerirlo.

Funciones reutilizables

El código limpio es pequeño

```
def add_blob(conn, file_path):
    try:
        if not os.path.isfile(file_path):
            return None

        with open(file_path, 'rb') as the_file:
            hash_file = hashlib.sha1(the_file.read()).hexdigest()

        extension = os.path.splitext(file_path)
        extension = extension.split('.')[1]
        extension.lower()

        mime = magic.Magic(mime=True)
        format_file = mime.from_file(file_path)
        format_file = format_file.split('/')[0]

        id_blob = add_file_in_the_database(conn, hash_file, extension, format_file)
        return id_blob
    except Exception as ex:
        message = "Failure in add_blob. Error = {}".format(ex)
        logger.exception(message)
        print(message)
        raise
```

El código limpio es pequeño

```python
@staticmethod
def get_hash_blob(path):
    with open(path, 'rb') as the_file:
        return hashlib.sha1(the_file.read()).hexdigest()

@staticmethod
def get_file_format(the_file):
    mime = magic.Magic(mime=True)
    fileformat = mime.from_file(the_file)
    fileformat = fileformat.split('/')[0]
    return fileformat

@staticmethod
def get_extension(file_path):
    extension = os.path.splitext(file_path)
    extension = extension.split('.')[1]
    extension.lower()
    return extension
```

```python
def add_blob(conn, file_path):
    try:
        if not os.path.isfile(file_path):
            return None


        hash_file   = get_hash_blob(file_path)
        extension   = get_extension(file_path)
        format_file = get_file_format(file_path)

        id_blob = add_file_in_the_database(conn, hash_file, extension, format_file)
        return id_blob
    except Exception as ex:
        message = "Failure in add_blob. Error = {}".format(ex)
        logger.exception(message)
        print(message)
        raise
```

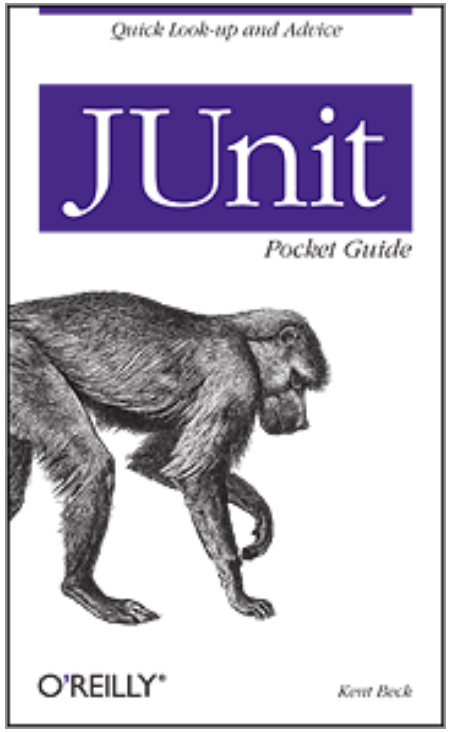
Cualidad 8

El código limpio tiene módulos de pruebas (testing)

¿Cómo podemos saber que nuestro código cumple con los requerimientos si no escribimos pruebas?

¿Cómo podemos mantener y extender el código sin miedo a romper algo?

El código sin pruebas no es código limpio.



Cualidad 9

El código limpio es expresivo

Usar nombres que revelen las intenciones

Elegir el nombre correcto lleva tiempo pero también ahorra trabajo.

El nombre de una variable, función o clase: ¿Por qué existe? ¿Qué hace? ¿Cómo se usa?

Cualidad 9

El código limpio es expresivo

Los nombres de los módulos deben ser significativos.

No tienen que resultar engañosos.

Tienen que ser distintivos.

El código se debe documentar a si mismo.

Cualidad 9

El código limpio es expresivo

```
int d; // Numero de días de vida
```

```
int diasDeVida;
```

Cualidad 9

El código limpio es expresivo

```
int dinero;
```

```
string usuario;
```

```
int cantidadDeDinero;
```

```
string infoUsuario;
```

Cualidad 9

El código limpio es expresivo

```
function copiar(a1,a2){
  a1 = a2;
}
```

```
function copiar(destino,datoACopiar){
  destino = datoACopiar;
}
```

Cualidad 9

El código limpio es expresivo

```
for(int i=0; i<5; i++){
   ...
}
```

```
const int NUMERO_DIAS_TRABAJADOS = 5;
for(int i=0; i<NUMERO_DIAS_TRABAJADOS; i++){
   ...
}
```

Cualidad 9

El código limpio es expresivo

```
// check to see if the employee is eligible for full benefit
if ((employee.flag == "1" ) && (employee.age > 65)){
    // do things
}
```

```
const int benefit_flag = "1";
const int minimum_age_for_full_benefits = "66"

if ((employee.flag == benefit_flag) && (employee.age >= minimum_age_for_full_benefits)){
    // do things

}
```

Cualidad 9

El código limpio es expresivo

```
// check to see if the employee is eligible for full benefit
if ((employee.flag == "1" ) && (employee.age > 65)){
    // do things
}
```

```
if (employee.isEligibleForFullBenefits()){
    // do things
}
```

Cualidad 9

Debemos usarlos porque no siempre sabemos como expresarnos sin ellos, pero su uso no es motivo de celebración.

El tiempo que inviertes en comentar, mejor invertirlo en expresar con código.

La energía empleada en escribir comentarios y mantenerlos actualizados, debería emplearse en crear un código más descriptivo, claro y expresivo.

Solo el código puede contar lo que hace. Es la única fuente de información precisa.

El código claro, expresivo y sin apenas comentarios, es muy superior al código enrevesado, complejo y lleno de comentarios.

Cualidad 9

Si puedes, no uses comentarios. Mejor usar funciones o variables

Muchas veces para explicar nuestras intenciones, basta con crear una función que diga lo mismo que el comentarios que pensaba escribir.

Comentarios al cerrar Estructuras

Los comentarios al cerrar las llaves para marcar que estructura esta cerrando, se suelen usar en funciones extensas.

Si tienes mucho que escribir… quizás tienes mucho que reducir

Fragmentos de código comentado

Mucho texto genera falso sentimiento de importancia
Los compañeros que vean el código no tendrán valor de borrarlo.
Pensaran que esta ahí por algo y que es demasiado importante para borrarlo.

Cualidad 9

Contexto del comentario

Si hay que comentar, mejor en el sitio necesario que de manera global.

Superfluo de información

Los comentarios no deben ser reflexiones

Debe explicar ¿qué hace?

Mucho cuidado si debes explicar motivos (por ej: ¿estamos justificando un bug?)

Cualidad 9

Posibles casos de utilización correcta de comentarios:

Explicar intenciones: Explicar el motivo de implementación en el código y no a otra.
Advertir las consecuencias: Advertir a otros programadores posibles consecuencias al modificar el código.
Amplificación: Para amplificar la importancia de al que, en caso contrario, parecería irrelevante.

Comentarios “TODO:”

TODO son tareas que el programador piensa que debería haber hecho pero no es así.

Un recordatorio para eliminar una función obsoleta.
Una solicitud para buscar un nombre más adecuado.
Para marcar un cambio que dependa de un evento planeado.

Cualidad 9

Contexto del comentario

Si hay que comentar, mejor en el sitio necesario que de manera global.

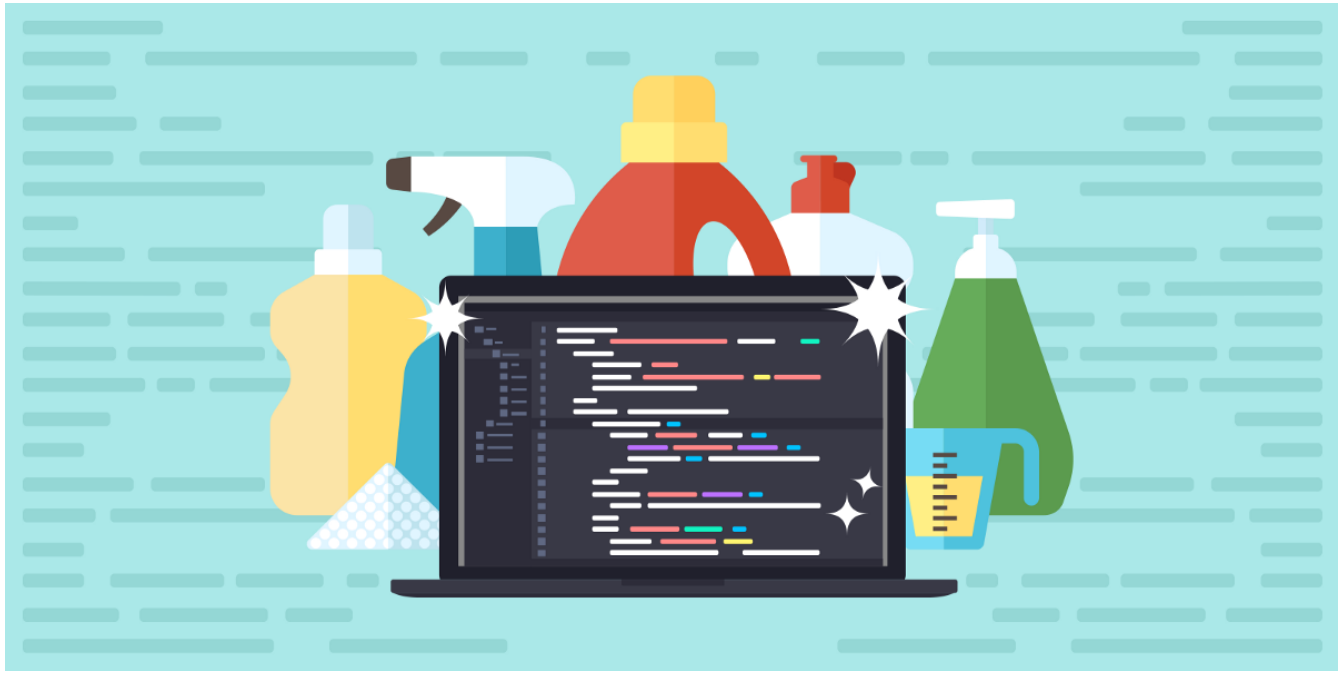
Superfluo de información

Los comentarios no deben ser reflexiones

Explicar el qué y cuidado con el porqué (por ej: ¿estamos justificando un bug?)

Cualidad 9

El código debe terminar leyéndose como un libro de instrucciones sin acertijos, que sea entendible por alguien que no ha visto el código anteriormente.



Producción del Software

Clean Code

Nelson Monzón López
nelson.monzon@ulpgc.es
Agustin Salgado de la Nuez
agustin.salgado@ulpgc.es
Daniel Santana Cedrés
daniel.santanacedres@ulpgc.es