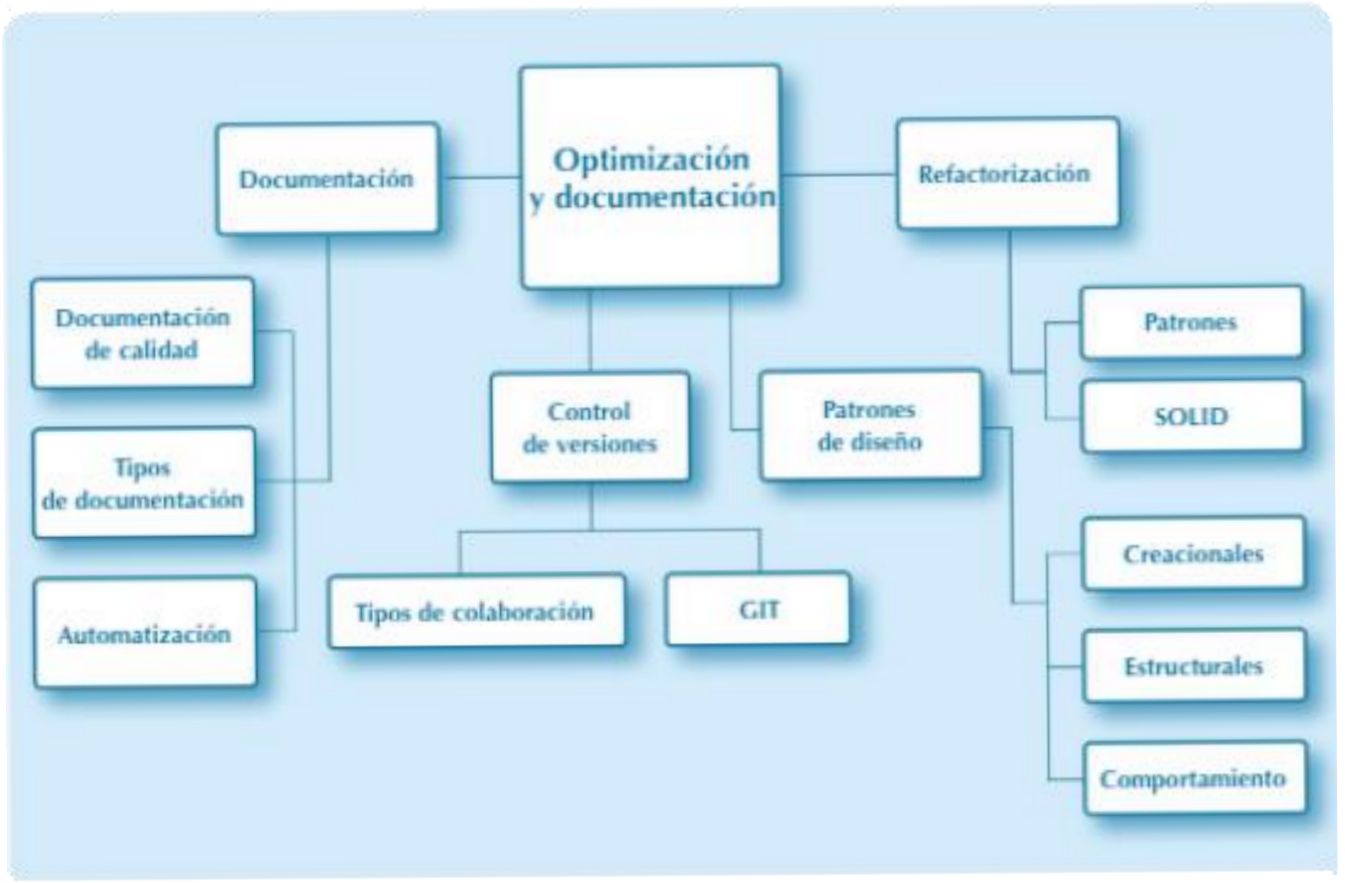
UNIDAD 5. OPTIMIZACIÓN Y DOCUMENTACIÓN

VICENT MARTÍ

OBJETIVOS

- Estudiar y conocer cómo es mejor desarrollar un software.
- Ver como utilizar la refactorización correctamente, como utilizar patrones de diseño y utilizar sistemas de control de versiones para realizar una documentación útil y eficaz.
  - Los puntos anteriores son en los que tienes mucho margen de mejora en tu práctica profesional.

MAPA CONCEPTUAL



CONTENIDO

1. Introducción
2. Refactorización
3. Patrones de refactorización más usuales
4. Control de versiones
5. Patrones de diseño
6. Documentación

1. INTRODUCCIÓN

Para convertirse en un buen programador, es preciso adquirir varias destrezas.
Una de ellas consiste en saber refactorizar código. Cuando el programador está comenzando, realiza los programas de la única forma que sabe, pero, cuando el desarrollador mejora, elige entre todas las posibles soluciones la más eficiente, efectiva, mantenible, etc.

En los siguientes apartados aprenderemos cuales de las posibles alternativas son las más correctas.
Además, actualmente, es raro participar en un proyecto sin utilizar un SCV o sistema de control de versiones.

En este capítulo, se abordarán sus principios básicos y se verá algo más en profundidad GIT, que es una de las herramientas de control de versiones más utilizada actualmente.

Por último, la documentación. Ningún desarrollador puede considerarse bueno si no le da la importancia que corresponde a la documentación. Recuérdese que un programa poco o mal documentado es un mal producto.

2. REFACTORIZACIÓN

El término refactorizar dentro del campo de la Ingeniería del Software hace referencia a la modificación del código sin cambiar su funcionamiento. Se emplea para crear un código más claro y sencillo, facilitando la posterior lectura o revisión de un programa

La refactorización nunca modificará el aspecto externo de una clase o programa (el comportamiento del software deberá siempre ser el mismo).

Para refactorizar un software habrá que realizar una serie de cambios internos, reestructurando componentes, siempre teniendo en cuenta que su comportamiento deberá permanecer inalterado.

2. REFACTORIZACIÓN II

Ventajas de refactorizar.

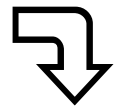
1. Ayudará al programador a encontrar errores de una forma más rápida y sencilla.
2. Ayudará a programar más rápido. Seguramente al encontrar menos errores se perderá menos tiempo en corregir y arreglar código.
3. Los programas se leerán y se interpretarán de una manera más fácil.
4. Los diseños serán más robustos.
5. Los programas tendrán más calidad.
6. Se evitará duplicar la lógica de los programas.

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Extract Method o reducción lógica

```
void printCuenta(String nombre, double cantidad){
    printLogo();
    System.out.println("nombre:"+nombre);
    System.out.println("cantidad:"+cantidad);
}
```

Código antes de refactorizar



```
void printCuenta2(String nombre, double cantidad){
    printLogo();
    printDetalles(nombre, cantidad);
}
void printDetalles(String nombre, double cantidad){
    System.out.println("nombre:"+nombre);
    System.out.println("cantidad:"+cantidad);
}
```

Código después de refactorizar

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Extract Method o reducción lógica

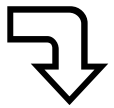
Este tipo de refactorización es de las más sencillas y típicas. Tiene varias ventajas:
- Realizan tareas específicas concretas (con lo cual pueden ser invocados por otros métodos de forma más sencilla)
- Permiten ir creando otros métodos con un nivel de complejidad mayor.

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Métodos inline / código embebido

```
int rondaGratis(){
    return (masde5cañas())?2:1;
}
boolean masde5cañas(){
    return numerodecañas > 5;
}
```

Código antes de refactorizar



```
int rondaGratis(){
    return (numerodecañas > 5)?2:1;
}
```

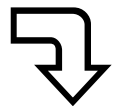
Código después de refactorizar

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Métodos inline / código embebido

```
double preciobase = pedido.preciobase();
    return (preciobase > 100);
```

Código embebido que se debería evitar



```
return (pedido.preciobase()>100);
```

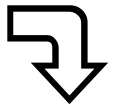
Sería mejor utilizar

En este caso se utiliza una variable temporal solo para realizar la comparación. Si la variable solo se utiliza para la comparación una forma más elegante y que evita problemas sería emplear la comparación directa a la hora de realizar el return.

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Variables autoexplicativas

```
if ((idioma.toUpperCase().index0f("RUS") > -1) &&
    (idioma.toUpperCase().index0f("ALE") > -1) &&
    (nivelingles > 0)) {
    // MENSAJES EN INGLES
}
```



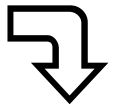
```
final boolean esRuso= idioma.toUpperCase().index0f( RUS") > -1;
final boolean esAleman = idioma.toUpperCase().index0f("ALE") > -1;
final boolean ingles = nivelingles > 0;
if (esRuso && esAleman && ingles) {
    // MENSAJES EN INGLES
}
```

Utilizando variables autoexplicativas correctamente definidas (tipo final), el código resultaría mucho más legible y sencillo. En la segunda imagen aparece el código original corregido.

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Mal uso de variables temporales

```
double tmp = pi radio radio;
System.out.println (tmp);
tmp = 2 pi radio;
System.out.println (tmp);
```



```
final double area = pi radio radio;
System.out.println (area);
final double perimetro = 2 pi radio;
System.out.println (perimetro);
```

¿Por qué se hace esto? Porque una variable temporal, generalmente se utiliza para ser instanciada o establecida solamente una vez por lo que, en caso, de que tenga que cambiarse su valor varias veces durante la ejecución del método, hay que plantearse crear una segunda variable temporal. En este caso se ve claro que habría que utilizar dos variable temporales en vez de una.

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Cambiar algoritmos

```
String BuscaAnimal(String[] animales){
    for (int i = 0; i c animales.length; i++) {
        if (animales[i].equals ("Perro")){
        return "Perro";
        }
        if (animales[i].equals ("Tortuga")){
        return "Tortuga";
        }
        if (animales[i].equals ("Loro")){
        return "Loro";
        }
    }
    return "No encontrado";
}
```

En el caso que se tenga que sustituir un algoritmo, trata de hacerlo de tal manera que funcione igual o mejor que antes. Las pruebas de regresión pueden ir bien para comparar resultados del antes y el después.

```
String BuscaAnimal2(String[] animales){
    for (String animale: animales) {
        if (animale.equals("Perro")) {
            return "Perro";
        }
        if (animale.equals("Tortuga")) {
            return "Tortuga";
        }
        if (animale.equals("Loro")) {
            return "Loro";
        }
    }
    return "No encontrado";
}
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Mover métodos entre clases I

Muchas veces se tienen clases complejas y otras con pocas tareas. En este caso hay que replantearse la carga de trabajo. Si algo no funciona, repartir trabajo es una buena estragia para acotar el problema.

Analicemos este ejemplo a continuación.

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Mover métodos entre clases II

Teniendo la clase Empleado con más carga de trabajo:

```
public class Empleado {
    private int horas;
    private int horasextra;
    private tipoEmpleado tipo;
    public double calculoHoras(){
    if (tipo.getTipo().equals("Supervisor")) {
        return horas + horasextra * 1.40;
    }
    if (tipo.getTipo().equals("Dependiente")) {
        return horas + horasextra * 1.75;
    }
    return horas + horasextra * 1.5;
    }
    public double getsueldo(){
        return tipo.getHorabase() calculoHoras();
    };
}
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Mover métodos entre clases II

Y tipoEmpleado con mucha menos carga:

```
public class tipoEmpleado {
    private String tipo;
    private double horabase;
    tipoEmpleado(String t,double h){
        this.tipo=t; this.horabase=h;
    }
    public String getTipo(){return tipo;
    }
    public double getHorabase(){return horabase;
    }
}
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Mover métodos entre clases III

Quedando tras la racionalización como sigue.
Clase Empleado:

```
public class Empleado {
    private int horas;
    private int horasextra;
    private tipoEmpleado tipo;
    public double getsueldo(){
        return tipo.getHorabase() tipo.calculoHoras(horas,horasextra);
    };
}
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Mover métodos entre clases IV

Quedando tras la racionalización como sigue.
Clase tipoEmpleado:

```
public class tipoEmpleado {
    private String tipo;
    private double horabase;
    tipoEmpleado(String t,double h){
        this.tipo=t; this.horabase=h;
    }
    public String getTipo(){
        return tipo;
    }
    public double getHorabase(){
        return horabase;
    }
    public double calculoHoras(int horas,int horasextra){
        if (tipo.equals("Supervisor")) {
        return horas + horasextra * 1.40;
        }
        if (tipo.equals("Dependiente")) {
        return horas + horasextra * 1.75;
        }
        return horas + horasextra * 1.5;
    }
}
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Mover variables miembro entre clases.

Mover métodos y campos es la base de la refactorización. En el ejemplo anterior, el precio de la hora base no depende del tipo de empleado y va a ser utilizado muchos métodos de la clase empleado. En vez de tenerlo en la clase tipoEmpleado, seguramente, sea mejor embeberla en la clase Empleado.
El resultado sería tener en la clase Empleado otra variable más y eliminar dicha variable de la clase tipoEmpleado:

```
public class Empleado {
    private int horas;
    private int horasextra;
    private double horabase;
    private tipoEmpleado tipo;
    …
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Autoencapsular campos (getters y setters)

Una buena práctica cuando una variable va a ser utilizada de forma asidua es autoencapsularla en la clase creando sus getters y setters.
En la clase tipoEmpleado, el resultado de tener setters y getters para la variable tipo sería el siguiente

```
private String tipo;
public String getTipo(){ return this.tipo;}
public void setTipo(String t){this.tipo = t;}
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Extraer clases I

¿Tiene en una clase código que hace más de lo que debería? Si tiene en una clase mucho código es que no se ha diseñado correctamente el esquema de clases. Por ejemplos véase la siguiente clase:

```
public class empleado2 {
    private int horas;
    private int horasextra;
    private final double horabase;
    private String tipo;
    public String getTipo(){return this.tipo;
}
public void setTipo(String t){this.tipo = t;
}
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Extraer clases II

Puede adivinarse que todavía habría que añadir más funcionalidad y su volumen ya es grande. Si continúan añadiéndose métodos, getters, setters, etc., la clase crecerá hasta el infinito.

La solución es extraer parte de la funcionalidad en otra clase. De esta forma, la clase será más fácil de mantener, ampliar y modular.

```
empleado2(String t,double h){
    this.tipo=t;
    this.horabase=h;
}
public double getHorabase(){return horabase;
}
public double calculoHoras(int horas,int horasextra){
    if (tipo.equals("Supervisor")) {
        return horas + horasextra * 1.40;
    }
    if (tipo.equals("Dependiente")) {
        return horas + horasextra * 1.75;
    }
    return horas + horasextra * 1.5;
}
    public double getsueldo(){
        return getHorabase() calculoHoras(horas,horasextra);
    }
}
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Extraer clases III

Dentro de la extracción de clases tendremos 2 subgrupos:

1. Clases delegadas
   Muchas veces, se tienen esquemas de clases en los que no interesa hacer accesible a cualquier clase parte de una clase concreta.

2. Foreign methods o métodos foráneos
   Muchas veces, se tiene una clase que hace todo lo que se pide y funciona a las mil maravillas, pero se necesita algo más. Si se modifica la clase, el desarrollador teme alterarla solamente para añadirle un método que, seguramente, no utilice de forma habitual, con lo cual se plantea otra alternativa.
   La alternativa es un foreign method o método foráneo.

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Reemplazar valores con objetos I

Muchas veces, al comienzo del diseño, se toman decisiones que pueden evitar un crecimiento o desarrollo futuro de la aplicación.

En ocasiones, se tienen datos que son tan simples que no tienen suficiente entidad para crear una clase. Pero, cuando se desarrolla más la aplicación, se observa que ese número de teléfono, por ejemplo, que se creía que era tan simple, necesitará formatearse de una manera determinada, extraer el prefijo, determinar si es móvil o fijo, etc.

Cuando se vea que esto puede ocurrir, hay que modificar la clase y extraer el campo creando una nueva clase.

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Reemplazar valores con objetos II

Imagínese que tiene la clase Jugador, que es parecida a la siguiente:

```
class jugador {
   private String telefono;
   …
```

Al final, se tenrá una clase Teléfono parecida a la siguiente:

```
public class telefono {
   private String _telefono;
   telefono(String s) {_telefono=s;}
   public String getTelefono() { return telefono; }
   public void setTelefono(String telefono) { this._telefono = telefono; }
   public String getPrefijo(){ return _telefono.substring(1,3); }
}
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Constantes

Muchas veces, cuando el programador cree que solo va a utilizar un valor una vez, utiliza una variable y se olvida de declarar una constante. Para ser un programador eficiente, es mejor utilizar constantes siempre que se pueda.
Sería preciso cambiar los métodos como este:

```
double descuento(double unidades, double preciounitario) {
    return unidades 0.95 preciounitario;
}
```

Por esta solución más acertada:

```
static final double DESCUENTO_DIRECTO = 0.95;
double descuento(double unidades, double preciounitario) {
    return unidades * DESCUENTO_DIRECTO * preciounitario;
}
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Encapsular arrays I

Los arrays son utilizados por muchos programadores, sobre todo por aquellos a los que no les gusta utilizar recursos más eficientes como las colecciones. Las colecciones, generalmente, ofrecen más versatilidad, pero, en caso de utilizar arrays, el encapsularlos puede dar mucho más juego.
Véase un ejemplo de setters y getters de un arrays:

```
String[] _asignaturas;
String[] getAsignaturas(){ return _asignaturas; }
void setAsignaturas(String[] var){ asignaturas = var; }
void setAsignatura(int donde,String titulo){ asignatu-
   ras[donde]=titulo; }
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Encapsular arrays II

Una vez establecidos los getters y setters, en ocasiones, es precios crear getters y setters específicos que utilicen copias de objetos para preservar los originales. Se utilizarán métodos parecidos a los siguientes:

```
// Setter a partir de una copia.
// Preservo el original.
void setAsignaturasCopia(String[] var){
   asignaturas = new String[var.length];
   for (int i=0; i < var.length; i++)
   setAsignatura(i,var[i]);
}
// En este getter creo una copia del array
// y to devuelvo con lo cual no se modifica
// el array original
String[] getAsignaturasCopia() {
   String [] var = new String[_asignaturas.length];
   System.arraycopy(_asignaturas, 0, var, 0, _asignaturas.length);
   return var;
}
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Encapsular arrays II

Una vez establecidos los getters y setters, en ocasiones, es precios crear getters y setters específicos que utilicen copias de objetos para preservar los originales. Se utilizarán métodos parecidos a los siguientes:

```
// Setter a partir de una copia.
// Preservo el original.
void setAsignaturasCopia(String[] var){
   asignaturas = new String[var.length];
   for (int i=0; i < var.length; i++)
   setAsignatura(i,var[i]);
}
// En este getter creo una copia del array
// y to devuelvo con lo cual no se modifica
// el array original
String[] getAsignaturasCopia() {
   String [] var = new String[_asignaturas.length];
   System.arraycopy(_asignaturas, 0, var, 0, _asignaturas.length);
   return var;
}
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Reemplazar tipos de objeto con subclases I

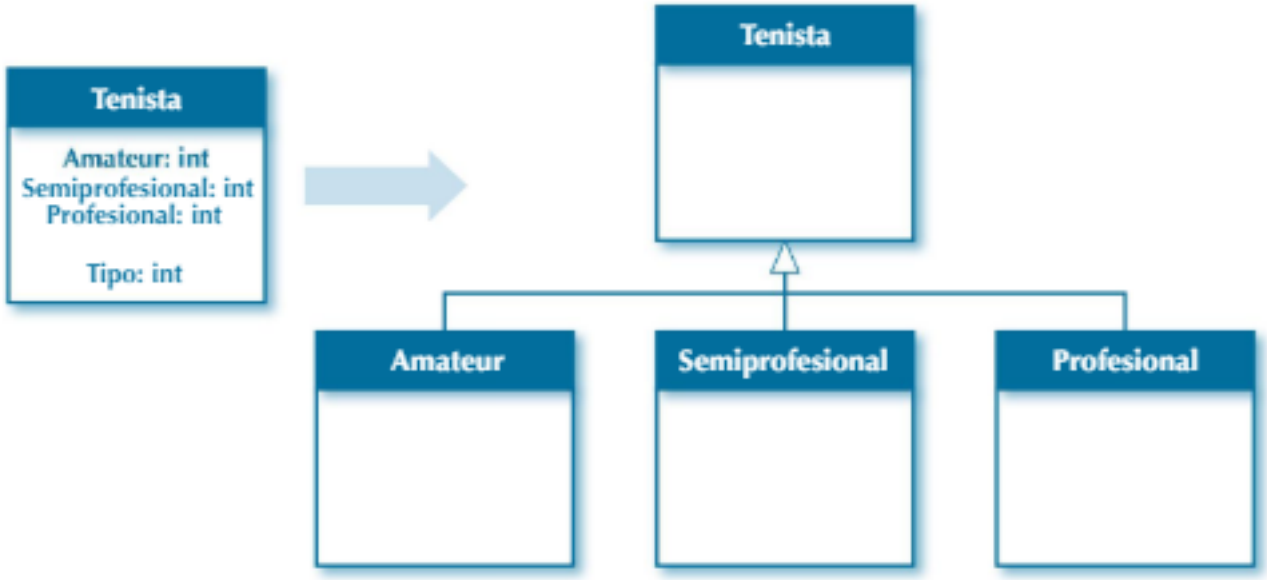
Muchas veces, se utilizan variables internas para clasificar objetos. Este tipo de programación implica que, dentro del código, existan muchas sentencias condicionales, ya sea de tipo if o de otro tipo. En este caso concreto, dependiendo del tipo de tenista, el código tendrá que ejecutar una cosa u otra.

Este tipo de programación no tiene sentido en una POO, luego, para refactorizar esta situación, lo mejor es cambiar toda la estructura condicional con polimorfismo. Se crearán clases derivadas y, mediante la herencia y el polimorfismo, se generá un código más limpio y fácil de mantener.

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Reemplazar tipos de objeto con subclases II

Ejemplo de subclases de la clase Tenista:



3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Reemplazar tipos de objeto con subclases III

Obsérvese el código antes de la refactorización.

```
public class tenista {
    private final int AMATEUR=1;
    private final int SEMIPROFESIONAL=2;
    private final int PROFESIONAL = 3;
    private final int _tipo;
    public tenista(int tipo){_tipo=tipo;}
    public int getTipo() { return _tipo; }
}
```

3. PATRONES DE REFACTORIZACIÓN MÁS USUALES

Reemplazar tipos de objeto con subclases IV

Obsérvese el código despuésde la refactorización.

```
public class tenista {
    private final int AMATEUR=1;
    private final int SEMIPROFESIONAL=2;
    private final int PROFESIONAL = 3;
    private int _tipo;
    public int getTipo() { return _tipo; }
    public tenista create(int tipo){
        switch (tipo) {
            case AMATEUR: return new amateur();
            case SEMIPROFESIONAL: return new semiprofesional();
            case PROFESIONAL: return new profesional();
            default: throw new IllegalArgumentException("tipo incorrecto");
        }
    }
}
```

4. PATRONES DE DISEÑO

Los patrones de diseño están íntimamente ligados a la programación orientada a objetos y quieren ir más allá a la hora de resolver problemas frecuentes en el desarrollo del software. Un patrón no es ni más ni menos que una solución a un problema típico de diseño.

Cabe destacar las siguientes características:

1. Reutilizabilidad.
2. Efectividad.
3. Utilización como vocabulario común entre ingenieros de software.
4. Estandarización.
5. Facilitan la comprensión de proyectos o estructuras más complejas.
6. No impiden que se utilice esta estrategia de diseño.

5. CONTROL DE VERSIONES

Un producto software se modifica infinitas veces durante su tiempo de vida y, por lo tanto, se necesita una herramienta que permita llevar un control de los distintos cambios que pueda sufrir a lo largo del desarrollo o vida útil.

GIT es una de las herramientas de control de versiones más utilizadas. Destacan las características:

1. Es una SCV gratuito y de código abierto.
2. Puede utilizarse para proyectos pequeños y grandes.
3. Es fácil de aprender y tiene un buen rendimiento.

En la mayoría de proyectos software no interviene una única persona en su desarrollo, por eso es imprescindible una buena gestión de versiones. No se debe hacer de manera manual ya que pueden cometerse muchos errores y el coste sería muy alto.

5. CONTROL DE VERSIONES

Almacenamiento de las distintas versiones

Los sistemas de control de versiones pueden clasificarse según cómo se almacena el código. Existen 2 tipos:

1. Sistemas centralizados

   Gestión más sencilla, repositorio único donde se almacena todo el código del software que ha de guardarse.

2. Sistemas distribuidos

   Cada programador tiene una copia del repositorio. Todos los usuarios tienen una copia del software. La ventaja es que en caso de fallo, habrá muchas copias de las cuales podrá recuperarse la última versión. En el servidor principal, residirá la copia principal.

5. CONTROL DE VERSIONES

Tipos de colaboración en un Sistema de Control de Versiones (SCV)

Existen diferentes formas de trabajar con un SCV y, por lo tanto, diferentes tipos de flujo de trabajo. A continuación, se comentarán los más frecuentes.

1. Flujo de trabajo centralizado o workflow centralizado
   Es un sistema muy propio de la vieja escuela. Suele utilizarse un servidor centralizado y los desarrolladores van publicando sus actualizaciones de código en él.
   Funcionan de forma exclusiva, lo que implica que, si alguien está modificando un elemento ningún otro desarrollador, puede realizar actualizaciones en este.
2. Flujo de trabajo con gestor integración
   En ocasiones, para un mayor control sobre las versiones del proyecto, se utiliza un getor de integración, que suele ser una persona del equipo de desarrollo que actualiza los trabajos en el repositorio común.
3. Flujo de trabajo con dictador y tenientes
   Este tipo de flujo de trabajo se utiliza solamente en casos de proyectos masicvos. Es una evolución del modelo anterior.

6. DOCUMENTACIÓN

Durante el proceso de elaboración de software, se genera mucha documentación. Los muchos documentos de innumerables programadores, analistas, jefes de proyecto, etc., dejan claro que es mejor no generar ningún tipo de documentación que generar documentación inservible o poco útil.

La documentación de un programa o producto va dirigido a muchas personas como programadores, usuarios, personas que van a testear la aplicación, etc.

Lo más importante de la documentación es que sea de calidad.

En ocasiones, el problema es que a los programadores no les gusta crear documentación y se hace de mala gana, luego eso se traduce en un pésimo resultado.

6. DOCUMENTACIÓN

Escritura de documentación de calidad

1. Hacer esquemas previos a la escritura.
2. Clasificar la información según la importancia.
3. Utilizar, si existen, estándares y herramientas de documentación (para Java, Javadoc).
4. Escribir con claridad.
5. Elegir bien la herramienta de documentación.
6. La guía de usuario y el manual de referencia.
7. Tener en cuenta el nivel de usuario que va a leer la documentación.

6. DOCUMENTACIÓN

Tipos de documentación

Cualquier tipo de software se puede ver desde dos puntos de vista diferentes: el técnico y el funcional. La documentación debe cubrir ambos aspectos.
Obsérvese la importancia de documentar en cada una de las fases.

1. Fase inicial
2. Análisis
3. Diseño
4. Codificación o implementación
5. Pruebas
6. Explotación
7. Mantenimiento

6. DOCUMENTACIÓN

1. Fase inicial

En esta fase, se planifica el proyecto, se hacen estimaciones, se conviene si el proyecto es o no rentable, etc.

De esta fase surgen muchos documentos, tanto de planificación como de estimaciones en los que se incluyen todos los datos económicos, posibles soluciones al problema y sus costes, etc. Son documentos que se realizan en un alto nivel y se acuerdan con la dirección de la empresa o las personas responsables del proyecto.

En estas fases iniciales, suelen tomarse decisiones que, a veces, afectan a todas las demás fases del proyecto, con lo cual tiene que estar todo bien documentado, detallado y soportado por datos concretos.

6. DOCUMENTACIÓN

2. Análisis

En esta dase, se analiza el problema. Consiste en recopilar, examinar y formular los requisitos del cliente y analizar cualquier restricción que pueda aplicarse.

Todas las entrevistas con el cliente, por lo tanto, tienen que estar registradas en documentos y, generalmente, esos documentos se consensuan con el cliente, y algunos tienen hasta carácter contractual.

El jefe de desarrollo, puede hacer firmar al cliente un documento de requisitos de la aplicación en la cual el equipo se compromete a realizar las especificaciones indicadas por el cliente y también el cliente se compromete a no variar sus necesidades hasta, por lo menos, terminar una primera release.

Como puede verse, es un documento que obliga a ambas partes a cumplir con lo acordado y, en muchas ocasiones, establece un marco de relación entre cliente y desarrollador.

6. DOCUMENTACIÓN

3. Diseño

Esta fase consiste en determinar los requisitos generales de la arquitectura de la aplicación y dar una definición precisa de cada subconjunto de la aplicación.

En esta fase, los documentos ya son más técnicos. Suelen crearse dos documentos de diseño: uno más genérico, en el que se tiene una visión de la aplicación más general, y otro detallado, en el que se profundizará en los detalles técnicos de cada módulo concreto del sistema.

Estos documentos los realizarán los analistas, junto con la supervisión del jefe de proyecto.

6. DOCUMENTACIÓN

4. Codificación o implementación

Esta fase consiste en la implementación del software en un lenguaje de programación para crear las funciones definidas durante la etapa de diseño.

Durante esta fase, se crea documentación muy detallada en la que se incluye código. Aunque mucho código suele comentarse en el mismo programa, también tienen que generarse documentos donde se indica, por ejemplo, para cada función, las entradas, salidas, parámetros, propósito, módulos o librerías donde se encuentra, quien ha realizado, cuándo, las revisiones que se han de realizar, etc.

Como puede observarse, el detalle es máximo teniendo en cuenta que ese código, en un futuro, va a tener que ser mantenido por la misma o, seguramente, otra persona y toda información que pueda recibir, a veces, es poca.

6. DOCUMENTACIÓN

5. Pruebas

En esta fase, se realizarán pruebas apra garantizar que la aplicación se programó de acuerdo con las especificaciones originales y los distintos programas de los que consta la aplicación están perfectamente integrados y preparados para la explotación.

Hay pruebas de todo tipo y podemos clasificar la documentación en dos bloques:

- Durante las pruebas funcionales se tienen que hacer todo tipo de anotaciones para luego plasmarlas en un documento, al que el cliente dará el visto bueno. En este documento, van detallándose tanto fallos como modificaciones que no cumplan con las especificaciones iniciales.

- En otro documento aparte, se detallarán los resultados de las pruebas técnicas realizadas.

6. DOCUMENTACIÓN

6. Explotación

En esta fase, se instala el software en el entorno real de uso y se trabaja con él de forma cotidiana.

Generalemnte, es la fase más larga y suelen surgir multitud de incidencias, nuevas necesidades, etc. Toda esta información suele detallarse en un documente en el que se registran los errores o fallos detectados intentando ser lo más explícito posible, puesto que luego los programadores y analistas deben revisar estos fallos o bugs y darles la mejor solución posible.

También surgirán otras necesidades que deben ir detallándose en un documento y que pasarán a realizarse en operaciones de mantenimiento.

6. DOCUMENTACIÓN

7. Mantenimiento

En esta fase, se realiza todo tipo de proedimientos correctivos y actualizaciones secundarias del software que consistirán en adaptar y evolucionar las aplicaciones.

Para realizar las labores de mantenimiento, hay que tener siempre delante la documentación técnica de la aplicación. Sin una buena documentación de la aplicación, las labores de mantenimiento son muy difíciles y su garantía en ese caso sería escasa.

Todas las operaciones de mantenimiento tienen que estar documentadas porque tiene que conocerse quien ha realizado la operación, qué ha hecho y cómo. También tienen que estar documentadas porque deberían probarlas otra persona distinta al programador.

6. DOCUMENTACIÓN

Ninguna aplicación debería entregarse ni desarrollarse si no cuenta con los siguientes documentos:

1- Manual de usuario. Documento de referencia que utilizará el usuario para desenvolverse con el programa.
2- Manual técnico. Con esta documentación, cualquier técnico que conozca el lenguaje con el que la aplicación ha sido creada debería poder conocerla casi tan bien como el personal que la creó.
3- Manual de instalación. En este manual, se explican, paso a paso, los requisitos y cómo se instala y pone en funcionamiento la aplicación.

6. DOCUMENTACIÓN

Generación automática de documentación

Es importante tener el código comentado y la mejor de las opciones es comentar el código dentro de él. Otras opciones implican mantener documentos que la mayoría de las veces se quedan obsoletos y desactualizados.

En Java, el código se documenta en el mismo fichero y, con una herramienta llamada Javadoc, pueden extraerse los textos y comentarios de dicho código fuente y transformarlos en páginas web con formato HTML.

Los comentarios se insertan con los caracteres / y /. Además hay algunas etiquetas más específicas como: @author, @version, @return o @since.

Tras ejecutar Javadoc, el cual puede ser ejecutado desde el propio IDE (Netbeans o Eclipse) o desde la línea de comandos, se obtiene una serie de páginas web con la documentación de la aplicación.