Unidad 4

Optimización y documentación

Entornos de desarrollo

4

Índice

Introducción

Un buen programador no debe contentarse con escribir código, sino que siempre debe buscar la mejora de este, siempre empleando los diversos elementos y técnicas a las que puede acceder para llevar a cabo su trabajo.

En este tema se mostrarán diversos elementos y técnicas que nos permitirán crear un código de mayor calidad y documentar correctamente el proceso.

La primera parte contará con diversas técnicas empleadas para la mejora de un código ya escrito, lo cual nos permitirá, sin cambios en el programa, mantener nuestro código limpio y robusto.

La segunda parte nos dará el conocimiento de los SCV, sistemas de control de versiones, los cuales nos permitirán ir registrando y guardando nuestro trabajo, lo cual nos permitirá retirar cualquier cambio sin complicaciones. Este punto es especialmente relevante para la creación de código en colaborativo.

Por último, estudiaremos los temas relevantes de la documentación dentro de nuestra labor, tanto la manera de crear una documentación de calidad como el conocimiento de la documentación que necesitamos en cada etapa del proceso.

Al finalizar esta unidad

- Conoceremos los mecanismos que podemos usar para mejorar nuestro código de programación, simplificándolo y evitando errores.
- Estudiaremos los patrones de diseño y como pueden ayudarnos.
- Describiremos qué es un sistema de control de versiones y como emplearlo.
- Entenderemos cómo elaborar una documentación de calidad.
- Descubriremos cuáles son los documentos que debemos realizar en cada una de las etapas.

4.1. Refactorización

La refactorización es el proceso de reestructuración interna del código, generalmente para mejorar su eficiencia, sin cambiar su estructura externa, es decir, es una reescritura para optimizar el comportamiento del código sin modificarlo. Este proceso es comúnmente llamado "limpiar del código".

A pesar de que el programa manejado por el código no sufre ninguna modificación por este cambio, se producen numerosas ventajas con este proceso, entre las que se encuentran:

- Permitirá descubrir errores ocultos en el código.
- Con menos errores un código el programa se ejecutará más rápido.
- Permitirá la simplificación del código.
- La simplificación llevará a un código más robusto y por lo tanto de mayor calidad.
- Permitirá evitar redundancias y duplicados en la lógica.

4.1.1. Patrones de refactorización más usuales

Algunos de los modelos de refactorización más empleados son:

Extract method o reducción lógica

Se basa en la creación de códigos cortos con instrucciones muy precisas, lo cual supone una serie de ventajas como son:

- Realizar tareas concretas específicas.
- Permite el empleo de otros métodos de complejidad mayor.

```
public void construirCuerpo() {
    construirCabeza();
    construirBrazo();
    construirPierna();
    construirPie();
}

private void construirPie() {
    // Código para construir un pie
}

private void construirPierna() {
    // Código para construir una pierna
}

private void construirBrazo() {
    // Código para construir un brazo
}

private void construirCabeza() {
    // Código para construir una cabeza
}
```

Imagen 1. Ejemplo de Extract method o reducción

Métodos inline o código embebido

En busca de mostrar claramente la función del código podemos llegar a ralentizar el programa al factorizar el código, resulta mejor la creación de un código más simple, lo cual está especialmente recomendado para programas de java.

Antes

```
public boolean esNumeroMayor(int num1, int num2) {

    boolean resultado;

    if (num1 > num2)
        resultado = true;
    else
        resultado = false;

    return resultado;

}
```

Refactorizado

```
public boolean esNumeroMayor(int num1, int num2) {
    return num1 > num2 ? true : false;
}
```

Imagen 2. Ejemplo de inline o código embebido

Podemos encontrar este problema también en el empleo de variables temporales cuya única función es la comparación.

Encontramos también problemas en el empleo de variables temporales para el almacenamiento de información como intermedio en un proceso. En estos casos puede ser mejor eliminar los elementos variables y sustituirlos por otro método de mayor utilidad.

Antes

```
public float obtenerAreaSegunLimite (float limite) {

    float area = base * altura;

    if(area <= limite) {
        return area / 2;
    } else {
        return area * 2;
    }

}
```

Refactorizado

```
public float obtenerAreaSegunLimite (float limite) {

    if(getArea() <= limite) {
        return getArea() / 2;
    } else {
        return getArea() * 2;
    }

}

public float getArea() {
    return base * altura;
}
```

Imagen 3. Ejemplo con variables temporales

Si se desean incluir variables sin factorizarlas se recomienda la "final", ya que evita la doble instanciación, la cual puede ser problemática.

```
public float obtenerAreaSegunLimite (float limite, float base, float altura) {
    final float area = base * altura;

    if(area <= limite) {
        return area / 2;
    } else {
        return area * 2;
    }

}
```

Imagen 4. Ejemplo de variable con final

Variables autoexplicativas

El empleo de variables autoexplicativas permite la creación de un código sencillo y legible lo cual permite evitar el empleo de códigos con líneas excesivamente complejas.

Antes

```
public static void contarDinero(int dinero) {

    int monedero = dinero;

    if (monedero / 500 > 0) {
        System.out.println("Billetes de 500: " + dinero / 500);
        monedero -= ((monedero / 500) * 500);
    }

    if (dinero / 200 > 0) {
        System.out.println("Billetes de 200: " + dinero / 200);
        monedero -= ((monedero / 200) * 200);
    }
}
```

Refactorizado

```
public static void contarDinero(int dinero) {

    int monedero = dinero;
    int cantidadBilletes500;
    int cantidadBilletes200;

    cantidadBilletes500 = monedero/500;
    monedero-= cantidadBilletes500 * 500;

    cantidadBilletes200 = monedero/200;
    monedero-= cantidadBilletes200 * 200;

    System.out.println("Billetes de 500: " +cantidadBilletes500);
    System.out.println("Billetes de 200: " +cantidadBilletes200);

}
```

Imagen 5. Ejemplo de variables autoexplicativas

Mal uso de variables temporales

El empleo de variables temporales puede provocar problemas ya que se crean con la intención de definir su valor una única vez, por lo que el cambio de su valor implicará la necesidad de la creación de una segunda variable durante el proceso, las cuales, además, para evitar la doble instanciación ambas variables deben declararse como "final".

Debemos ser conscientes de que los parámetros deberían modificarse solo de forma excepcional, por lo que se debe configurar el código para evitar las circunstancias que puedan llevar a ello.

Antes

```
public static void sumarRestarNumeroXVeces(int numero, int cantidad) {

    int j = 0;
    int a = cantidad;

    while(a>0) {
        j += numero;
        a--;
    }
    System.out.println("Suma: " + j);

    j = 0;
    a = cantidad;

    while(a>0) {
        j -= numero;
        a--;
    }

    System.out.println("Resta: " + j);
}
```

Refactorizado

```
public static void sumarRestarNumeroXVeces(int numero, int cantidad) {

    int resultadoSuma = 0;
    int resultadoResta = 0;
    int numeroVecesSuma = cantidad;
    int numeroVecesResta = cantidad;

    while(numeroVecesSuma>0) {
        resultadoSuma += numero;
        numeroVecesSuma--;
    }

    while(numeroVecesResta>0) {
        resultadoResta -= numero;
        numeroVecesResta--;
    }

    System.out.println("Suma: " + resultadoSuma);
    System.out.println("Resta: " + resultadoResta);

}
```

Imagen 6. Ejemplo de Mal uso de variables

Cambiar algoritmos

Siempre que sea posible se debe reformular el código para simplificarlo, por supuesto sin modificar en ningún momento la función de este. Con el fin de evitar problemas con las modificaciones del código, es recomendable el empleo de pruebas de regresión antes de la implementación, las cuales evitarán la mayoría de los problemas.

Este proceso de simplificación permite mejorar la eficiencia del código y su legibilidad.

Antes

```
public static int productoSPrimerosNumeros() {
  int i = 0; int resultado = 1; while (i < 5) { if (i == 0) resultado =
  resultado * 1; else resultado=resultado*(i+1); i++; } return resultado;
}
```

Refactorizado

```
public static int productoSPrimerosNumeros() {

    int resultado = 1;

    for (int i = 1; i <= 5; i++) {
        resultado *= i;
    }

    return resultado;

}
```

Imagen 7. Ejemplo de cambios de algoritmos

Mover métodos entre clases

Es importante repartir las tareas entre las diferentes clases con el fin de evitar una sobreexplotación de algunas de las clases mientras que otras permanecen con una carga de trabajo inferior.

Este tipo de actos prevendrá la ralentización del programa al evitar que uno de los elementos, con un exceso de labores, arrastre al resto.

Mover variables miembro entre clases

En muchas ocasiones la refactorización no depende de un cambio en las líneas de código, sino en el cambio de lugar de estas, moviendo los métodos y campos a las clases donde se van a emplear con asiduidad y eliminándolos de las clases donde no serán empleados.

Autoencapsular campos (getters y setters)

En los casos donde en una clase una de sus variables es empleado con asiduidad es posible autoencapsularla mediante el empleo de getters y setters para mejorar su rendimiento.

Antes

```
public class Coche {

    public String tipo;
    public String modelo;

}
```

```
ain(String[] args) {
    Coche();
    coche.modelo = "modelo de coche";
}
```

Refactorizado

```
public class Coche {

    private String tipo;
    private String modelo;

    public String getTipo() {
        return tipo;
    }

    public void setTipo(String tipo) {
        this.tipo = tipo;
    }

    public String getModelo() {
        return modelo;
    }

    public void setModelo(String modelo) {
        this.modelo = modelo;
    }

}
```

```
public static void main(String[] args) {
    Coche coche = new Coche();
    coche.setModelo("modelo de coche");
}
```

Imagen 8. Ejemplos de autoencapsulado de campos

Extraer clases

Cuando una clase se sobrecarga con código y no es posible distribuirlo entre diferentes clases podemos decir que la creación del esquema de clases no se diseñó correctamente, ya que este tipo de casos deberían evitarse mediante la previsión de las necesidades desde el inicio de la creación de esquemas.

Si nos encontramos con este problema existen diversas formas de solucionarlo.

Clases delegadas

La resolución de este problema se puede realizar mediante la creación y distribución de los elementos en las llamadas clases delegadas, permitiendo que cierta información, en lugar de encontrarse en una clase principal, se mueva a dichas clases inferiores donde si es relevante, liberando así parte de la carga en la principal donde dicha información no era necesaria.

Imagen 9. Ejemplos de clases delegadas

Foreign methods o métodos foráneos

Con el fin de evitar estropear clases con mucha tarea que funcionan correctamente, si se necesita añadir algo más a ella, se opta por usar el llamado método foráneo, donde los nuevos elementos se incluyen en una clase cliente con un objeto de la clase utilidad como argumento.

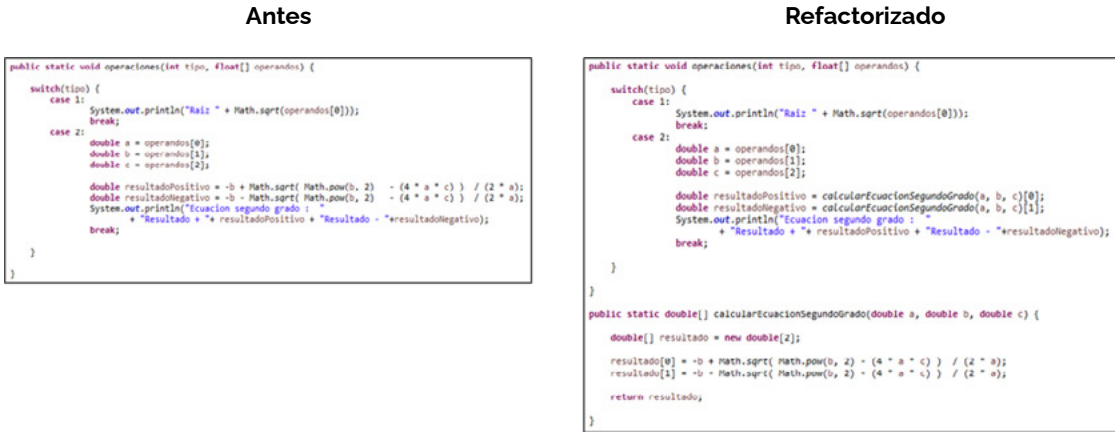

Imagen 10. Ejemplo de Foreign methods

Reemplazar valores con objetos

Es posible que los elementos de una clase, con el tiempo, requieran de un aumento de importancia, pasando de poder permanecer como un simple elemento dentro de una clase hasta requerir la formación de su propia clase, este tipo de suceso es común en los programas que se expanden.

Este tipo de problema se soluciona de manera sencilla con la creación de la nueva clase y la mudanza de los elementos requeridos a ella, permitiendo que la clase inicial se aligere y que la nueva clase cuente con espacio para crecer.

Antes

```
public class Persona {

    private String nombre;
    private String direccion;
    private String ciudad;
    private String codigopostal;
    private String pais;

}
```

Refactorizado

```
public class Persona {

    private String nombre;
    private Direccion direccion;

}
```

```
public class Direccion {

    private String tipoVia;
    private String nombre;
    private String ciudad;
    private String codigopostal;
    private String pais;


}
```

Imagen 11. Ejemplo de Reemplazo de valores

Empleo de constantes

El empleo de constantes es necesario para mejorar la eficiencia, ya que evita los problemas de emplear un valor en una sola línea y emplear continuas variables. No debemos olvidar que es mejor el empleo de constantes de manera periódica.

```
public class Celda {

    public static final String COLOR_NEGRO = "NEGRO";
    public static final String COLOR_BLANCO = "BLANCO";

    //..

}
```

Imagen 12. Ejemplo de empleo de constantes

Encapsular arrays

Los arrays no son recomendados, ya que existen alternativas mejores, pero su uso es amplio. Con el fin de mejorar la eficiencia de estos es necesario siempre que se pueda encapsularlos mediante el establecimiento de getters y setters, en ocasiones de copias de objetos en lugar del objeto mismo.

```
public class EncapsularArrays {

    private int[] billetes = {5,10,20,50,100,200,500};

    public int[] getBilletes() {
        return billetes;
    }

    public int[] getBilletesCopia() {
        int [] var = new int[billetes.length];
        System.arraycopy(billetes, 0, var, 0, billetes.length);
        return var;
    }

    public void setBilletes(int[] billetes) {
        this.billetes = billetes;
    }

    public void setBillete(int posicion, int billete) {
        billetes[posicion] = billete;
    }

}
```

Imagen 13. Ejemplo de encapsulación de arrays

Reemplazar tipos de objeto con subclases

La clasificación interna de un objeto no es recomendable ya que esto desemboca en el empleo de numerosas condicionales, que complicarán en el texto y pueden ralentizarlo.

En lugar de realizar dicha acción se recomienda el empleo del polimorfismo y la herencia para la creación de subclases, o clases derivadas, que permitan un código más limpio y la liberación de la superclase original.

Antes

```
public class Animal {

    private final String tigre = "Tigre";
    private final String tortuga = "Tortuga";

    public String getTipo(String animal) {
        if(animal.equalsIgnoreCase(tigre)) {
            return "Soy de tipo tigre";
        } else if (animal.equalsIgnoreCase(tortuga)) {
            return "Soy de tipo tortuga";
        }

        return "No soy de ningun tipo";
    }

}
```

Refactorizado

```
public class Animal {

    public Animal getTipo(String animal) {
        if(animal.equalsIgnoreCase("Tigre")) {
            return new Tigre();
        } else if (animal.equalsIgnoreCase("Tortuga")) {
            return new Tortuga();
        }

        return null;
    }
}
```

```
public class Tortuga extends Animal {
    // Código tortuga
}
```

```
public class Tigre extends Animal {
    // Código tigre
}
```

Imagen 14. Ejemplo de reemplazo de objetos con subclases

4.2. Patrones de diseño

Los patrones de diseños, como patrones que son, intentan resolver problemas comunes con soluciones predefinidas, por lo que se relacionan en gran medida a la búsqueda de soluciones para las problemáticas que se encuentran en los programas orientados a objetos.

Poseen múltiples características por las que se destacan:

- **Reutilización** de los mismos patrones tantas veces como sea necesario.
- **Efectividad** probada.
- **Vocabulario común** entre ingenieros de software.
- **Estandarización** del código.
- **Facilitación de la comprensión** de estructuras complejas.
- **Permite el empleo otras estrategias** simultáneamente.

Existen tres tipos de patrones de diseño:

- **Patrones creacionales:** Permiten la creación con facilidad de nuevos objetos. Permite la reutilización del código y otorga flexibilidad de creación.
  - Abstract Factory.
  - Builder Patterns.
  - Factory Method.
  - Prototype.
  - Singleton.

- **Patrones estructurales:** Trabaja en los problemas de relaciones entre objetos, generalmente a partir del concepto de herencia para composición de interfaces o nuevas funcionalidades.
  - Adapter.
  - Bridge.
  - Composite.
  - Decorator.
  - Facade.
  - Flyweight.
  - Proxy.

- **Patrones de comportamiento:** Permite descubrir comunicaciones entre objetos de clase y modificarlas.
  - Chain of responsability.
  - Command.
  - Interpreter.
  - Iterator.
  - Mediator.
  - Memento.
  - Observer.
  - State.
  - Strategy.
  - Template method.
  - Visitor.

4.3. Control de versiones

Es normal, y muy probable, que cada programa creado sufra continuas modificaciones a lo largo de su vida, por lo que en ocasiones poco queda del programa original, pero tanto cambio puede llevar a errores imprevistos. La mejor manera de evitar estos problemas es mantener un registro de todas las versiones implementadas, de modo que, ante un imprevisto, sea posible volver a una versión anterior.

Entre las herramientas de este tipo podemos encontrar Git, el cual, entre otras cosas destaca por:

- Ser gratuito y de código abierto.
- Servir tanto para grandes como para pequeños proyectos.
- Buen rendimiento.
- Muy sencillo e intuitivo.

El empleo de este tipo de sistemas es especialmente necesario cuando se trabaja en un trabajo colaborativo, ya que una modificación de un colaborador, realizada junto con otras del resto de colaboradores, puede crear un problema sin que el resto sepa cuál es el elemento que produce el error. En casos como estos la posibilidad de simplemente volver a una versión anterior e ir probando hasta encontrar el error es invaluable.

4.3.1. Almacenamiento de las distintas versiones

El modo de almacenamiento de las versiones anteriores será el elemento empleado para la división de los tipos de SCV, sistemas de control de versiones, quedando en dos grupos:

Sistemas centralizados

Se almacena todo el código de las versiones anteriores en un único repositorio común.

Es el sistema con mayor control y de gestión más sencilla, ya que este tipo de almacenamiento permite que las versiones anteriores se guarden con una única numeración, aunque en ocasiones puede ser demasiado rígido.

Sistemas distribuidos

Se crea un repositorio para cada usuario, por lo que otorga una gran seguridad por la existencia de diversas copias que servirán de respaldo.

Al permitir que cada usuario trabaje desde su hogar permitirá que el servidor no sufra tanto, por lo que no se requerirá que este sea tan potente.

Además, otorgará una gran flexibilidad, ya que pondrá en manos de los desarrolladores todas las copias de las versiones del programa, pudiendo "jugar" con estas versiones para encontrar mejores soluciones.

4.3.2. Tipos de colaboración en un SCV

Existen diversos modos de colaboración con los SCV, estos modos de trabajo son llamados workflow o flujo de trabajo y determinarán el modelo de trabajo. Los principales modelos son:

Flujo de trabajo centralizado o workflow centralizado

Se emplea para trabajar con un servidor centralizado y a costa de una mayor rigidez permite una gran seguridad a la hora de cargar cambios, ya que la localización del fallo y su recuperación será sencilla.

Este tipo de método de trabajo emplea el sistema de trabajo en exclusiva el cual permite el trabajo en segmentos y con un único desarrollador modificándolo simultáneamente, es decir, una vez que un desarrollador a escogido un segmento para su modificación este se bloqueará para el resto de usuarios hasta que se haya cargado el trabajo modificado, en ese momento se desbloqueará a la espera de una futura modificación.

Este modelo de trabajo es bastante rígido y puede ralentizar el proyecto, pero, como ya se ha dicho, otorga una gran seguridad a la hora de conocer la procedencia de las actualizaciones y evita conflictos entre modificaciones simultaneas.

Flujo de trabajo con gestor de integración

Otro tipo de modelo de trabajo es el trabajo colaborativo, donde todos pueden trabajar en una copia local de la versión para cargarla posteriormente, este modelo da una gran libertad y agilidad, ya que permite el trabajo simultaneo de una misma versión por lo que nadie debe esperar turno. A pesar de las ventajas de este sistema, cuenta con grandes desventajas, como problemas para encontrar modificaciones que den problemas o problemas surgidos por el conflicto entre dos elementos sobre los que se trabaja simultáneamente y que dan problemas al cargarlos juntos.

Este tipo de problemas se soluciona trabajando con un gestor de integración, una persona que recibirá todas las versiones modificarlas y tendrá la responsabilidad de subirlas él al repositorio común. Esto evita que los desarrolladores puedan modificar directamente el repositorio común, aunque si podrán verlo para poder trabajar con él, al tiempo que crea una línea de filtrado con la que se pueden detectar algunos errores obvios.

Flujo de trabajo con dictador y tenientes

Es una evolución del método anterior empleado para proyectos masivos, los cuales, por su tamaño requieren medidas especiales de control, como ocurre con los kernel de Linux. En este sistema de trabajo colaborativo se crean un total de tres niveles:

- **Desarrolladores:** Los encargados de modificar el código.
- **Tenientes:** Encargados de recoger por equipos el trabajo creado por los desarrolladores y revisarlo.
- **Dictador:** El único con acceso a repositorio, recibe los cambios ya recopilados y revisados de manos de los tenientes y los carga en el repositorio común.

4.4. Documentación

El proceso de documentación de un proyecto es importante, tanto en el proceso de creación como en las partes previas y posteriores. Este tipo de elementos que no influyen directamente en el proceso suelen descuidarse por parte de los desarrolladores.

El proceso de documentación debe hacerse con cuidado para evitar que lo que debería ser una ayuda se convierta en un trabajo extra o generar elementos de baja calidad que llegan a ser inservibles, como suele pasar con elementos como los manuales de instrucciones.

4.4.1. Escritura de documentación de calidad

Para llevar a cabo la elaboración de elementos de calidad es conveniente seguir ciertas pausas que garanticen su calidad, como son:

- **Esquemas:** La elaboración de esquemas nos permite estructurar las ideas al tiempo que evita el olvido de algún elemento. Estos mismos esquemas pueden incorporarse a los documentos si pueden ser de utilidad.
- **Clasificar:** No toda la información es relevante o importante, por lo que su inclusión puede entorpecer nuestra labor, para evitar esto es mejor clasificar la información destacando la que sea relevante, de modo que se incluya esa en el documento, mientras que la menos importante puede incluirse en anexos.
- **Síntesis:** Preparar resúmenes, índices y esquemas de los documentos permite un control mayor de ellos y facilita su acceso a nuevos usuarios.
- **Empleo de estándares:** Muchos lenguajes de códigos o herramientas poseen elementos estandarizados, en la medida de lo posible es mejor emplear estos.
- **Anticipación:** Es necesario, en la medida de lo posible resolver las posibles dudas que puedan llegar a surgir a los usuarios anticipándose a ellas con elementos como los apartados de "Preguntas frecuentes".
- **Claridad:** La información debe ser entendida, en ocasiones por personas no expertas, por lo que es necesario que se mantenga un lenguaje claro y preciso y, por lo tanto, fácilmente entendible sin ambigüedades.
- **Empleo de las herramientas adecuadas:** El empleo de procesadores de texto está ampliamente extendido por su manejo simple y flexibilidad, pero no siempre son la herramienta idónea, por lo que es posible que sea necesario buscar alternativas que se adapten mejor a nuestras necesidades.
- **Empleo de los documentos apropiados:** Las guías de usuario y los manuales de referencia no son lo mismo, cada uno, a pesar de su semejanza tienen unas características propias y, por lo tanto, unos usos idóneos. Emplear el documento adecuado en el momento adecuado es fundamental.
- **El nivel del usuario:** Como expertos podemos usar un lenguaje especializado, pero debemos tener en cuenta que es posible que el usuario no posea los conocimientos necesarios para descifrarlo, por lo que debemos adaptar nuestro nivel de lenguaje al suyo, evitando, en la medida de lo posible, el empleo de jerga técnica.

4.4.2. Tipos de documentación

Existen dos tipos de documentos fundamentales en la creación de software en función de su naturaleza:

- **Documentos técnicos:** Son aquellos que contienen información técnica sobre el producto, ya sea sobre sus componentes, características, bases de datos, etc.
- **Documentos funcionales:** Son aquellos que contienen información sobre la funcionalidad del producto, como puede ser el funcionamiento del programa, el modo de empleo, etc.

Dependiendo de la fase de vida en la que se encuentre el producto requerirá con mayor urgencia un tipo de documento u otro.

Fase inicial

Es la fase de planificación del proyecto, por lo que no existe aún nada del proyecto, es posible que los desarrolladores aún no estén contratados, por lo que se centrará más en los temas económicos del costo y la rentabilidad del proyecto.

Se generan numerosos documentos de esta parte del proyecto con las decisiones tomadas en esta época del proceso, las estimaciones que de él se han supuesto y los objetivos que pretende lograr.

En esta etapa la información registrada es más hipotética que objetiva.

Análisis

Es el primer acercamiento al proyecto como tal, se centra en el estudio de los objetivos del futuro proyecto y los problemas que surgirán de ellos.

Se reúnen los problemas que se pueden predecir durante la elaboración del proyecto, así como los recursos y requisitos necesarios para llevarlos a cabo.

Los documentos de este proceso registrarán los requisitos del proyecto, así como los posibles inconvenientes, con el fin de que sean visualizados por el cliente. En muchos casos estos documentos pueden llegar a ser contractuales.

Diseño

Es el inicio de la creación del proyecto propiamente dicho, ya que implica la búsqueda de las necesidades reales y precisas del proyecto, así como la creación de un esquema arquitectónico de este y de las aplicaciones que lo acompañarán.

En esta etapa ya se cuenta con los técnicos por lo que los documentos de esta parte serán de carácter técnico, aunque se elaborará una segunda copia de carácter divulgativo con el fin de que el cliente y el resto de elementos de la empresa necesarios, pero sin conocimiento técnico lo puedan entender.

Codificación o implementación

Mediante un lenguaje de programación se lleva a la práctica los planes que se elaboraron en la fase de diseño, siguiendo estos o modificándolos en los casos de necesidad.

En esta etapa los documentos serán muy técnicos y relacionados a la elaboración del código. Generalmente será de carácter interno y registro para posibles errores, por lo que será imposible de comprender sin los conocimientos necesarios.

Pruebas

Tras la elaboración del proyecto el periodo de pruebas será una mezcla entre personal cualificado y no cualificado, con la creación de dos tipos de documentos.

Un primer documento contendrá las expectativas del programa junto a todos los elementos que deberá contener. Con este documento se realizarán las pruebas, siempre en presencia del cliente, apuntando en él cualquier tipo de anotación necesaria, así como los resultados de las pruebas llevadas a cabo, sus éxitos y especialmente fracasos o elementos que no cumplen el estándar.

Un segundo documento se realizará con las pruebas llevadas a cabo por los técnicos, ya sin presencia del cliente, por lo que todos los documentos de este proceso serán de carácter técnico.

Explotación

En este periodo se hace uso del sistema por los usuarios, por lo que suele ser el proceso más largo. En él se registran los errores e incidencias con el fin de registrarla en un documento.

Estos fallos o carencias, al ser registrados por los usuarios, no serán registrados con un lenguaje técnico, pero sí siendo lo más específico posible, con el fin de que los programadores puedan entenderlos y solucionar los fallos.

Mantenimiento

En esta etapa se modifica el código para solventar fallos y carencias, con el fin de realizar esto es necesario emplear los documentos técnicos del programa y los documentos recogidos durante la explotación, con los cuales encontrar los fallos y realizar las modificaciones necesarias.

Esta etapa registrará los cambios realizados, el desarrollador que los realizo y siempre que sea posible la identidad del probador que realiza la prueba de esa modificación.

Por último, es necesario tener en cuenta que, además de la documentación generada en las distintas etapas, no es correcto entregar una aplicación sin los siguientes documentos:

- **Manual de usuario**: Debe ser fácilmente comprensible para los usuarios sin el conocimiento técnico, debe permitir comprender la finalidad y los modos de empleo del producto sin ambigüedades.
- **Manual técnico**: Dirigido a los técnicos este documento contendrá conocimientos con un lenguaje especifico y no apto para usuarios normales, con el fin de que sus destinatarios puedan conocer toda la información necesaria de un producto.
- **Manual de instalación**: Al igual que el manual de usuario este debe ser comprensible para personas sin conocimiento técnico. Su función será la de explicar clara y detalladamente los procesos de instalación de un programa.

4.4.3. Generación automática de documentación

El código cambia constantemente por la que la creación de documentos de este para anotaciones no es efectiva, ya que quedan desactualizados con rapidez. Una mejor solución a este problema es el empleo de comentarios en el propio código, los cuales se pueden extraer posteriormente como HTML con programas como Javadoc.

En el código de Java es posible incluir comentarios en el mismo fichero, estos deben insertarse siempre precedidos de dos barras inclinadas "//" y durarán hasta el final de la línea, los comentarios de varias líneas se intercalarán entre "/*" y "*/".

```
//Comentarios de una línea.

/*Comentarios
de varias
líneas*/
```

Si deseamos ser más específicos podemos, iniciando con "/**" y, acabando con "*/", con "*" en cada línea, incluir las siguientes etiquetas:

- **@author:** Indica la identidad del autor del código.
- **@versión:** Indica la versión del código.
- **@return:** Indica un valor de retorno que no sea void.
- **@since:** Indica la fecha de creación.
- **@deprecated:** Indica elementos obsoletos.
- **@param <nombre_par>:** describe el parámetro indicado.
- **@see:** Indica una redirección para el lector.
- **@link <URL>:** Indica una redirección a la URL especificada.

```
/**
* @author Nombre del autor
* @version Número de la versión
* @since Fecha de modificación
*/
```

www.universae.com