UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Proyecto de mantenimiento para COPS, un entorno de programación en Java para personas con diversidad funcional

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Alison Beltrán Melgarejo

Tutor: Carlos David Martínez Hinarejos

y Natividad Prieto Sáez

2016 - 2017

Resumen

En este trabajo se abordan los problemas derivados de realizar un proyecto software sin seguir las pautas de calidad ni mantenibilidad desde la fase inicial. Este tipo problemas son muy comunes en entornos tanto laborales, como de investigación a pesar de que, generalmente, se conoce la importancia de seguir este tipo de procesos (especialmente en programas de gran envergadura y complejidad). El sistema software sobre el que se trabaja es COPS, un entorno de desarrollo software en Eclipse que facilita la programación en Java a usuarios con diversidad funcional.
Explicaremos los conceptos básicos para lograr software mantenible basándonos en este programa, ya que es un proyecto en que no se han seguido dichas pautas. Analizaremos su mantenibilidad y lo modificaremos con herramientas que optimizarán el proyecto para que cumpla las condiciones de mantenibilidad. Además, se establecerá un plan que se podrá seguir para que el proyecto cumpla las pautas de calidad de software.

Palabras clave: Ingeniería del software, entornos de programación, diversidad funcional, calidad de software, mantenimiento de software, programación en Java.

Abstract

This paper deals with the issues of carrying out a software project without following the quality nor the maintenance guidelines from the initial phase. These kinds of problems are very common in both work and research environments, although the importance of this type of process is generally known (especially in large scale and complex programs). The software system we will work on is COPS, a software development environment in Eclipse that facilitates users with functional diversity to programme Java.
In addition, we will explain the basic concepts to get maintainable software based on this program, since those guidelines have not been followed in the

project. We will also analyze its maintainability and we will modify it with tools that will optimize it, so that it can fulfil the maintainability conditions. Moreover, a plan will be established so that the projects meets the quality guidelines of software.

Keywords: Software engineering, programming environments, functional diversity, software quality, software maintenance, programming in Java.

Tabla de contenidos

1. Introducción

El proyecto COPS (Computer Programming using Speech) [1] es un *plugin* para Eclipse desarrollado en Java que permite escribir programas en Java al dictado. Para ellos se utiliza el sistema de reconocimiento de voz, iATROS.

En concreto, COPS extiende la funcionalidad de Eclipse permitiendo que personas con diversidad funcional puedan programar sin problemas en Java. Se trata de un *plugin* que nos permite revisar código ya escrito con síntesis de voz y dictar código.

La apariencia del programa es muy sencilla; se abre el programa de Eclipse y seleccionamos la vista COPS. Se abrirá un espacio en negro y ése será el espacio disponible para poder programar, como podemos ver en la Figura 1.

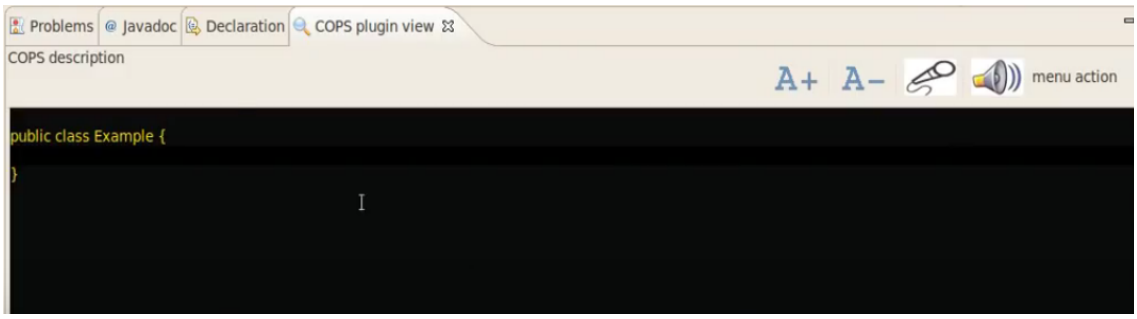

Figura 1. Vista de COPS en el entorno de programación de Eclipse.

A continuación, se muestran los botones de la vista y se describe su funcionalidad.

Cambiar el tamaño de la fuente para poder incrementar o decrementar el texto de nuestro programa.

Dictar una línea de código que se añadirá en la parte en la que esté el cursor. Cuando pulsamos el icono se vuelve de color rojo para indicar que estamos grabando. Se debe volver a pulsar para terminar la grabación.

La síntesis de voz de código nos permitirá escuchar con una voz sintetizada el texto que se haya seleccionado.

Desde aquí podemos ir al menú de preferencias del reconocedor, donde podemos cambiar los colores de la letra, el fondo y guardar nuestras preferencias en un perfil [1].

Para empezar a usar el *plugin* debemos ir a `Window`, `Show View`, `COPS View`. En la Figura 2 vemos un ejemplo de cómo se vería en nuestro Eclipse para seleccionar la vista de COPS.

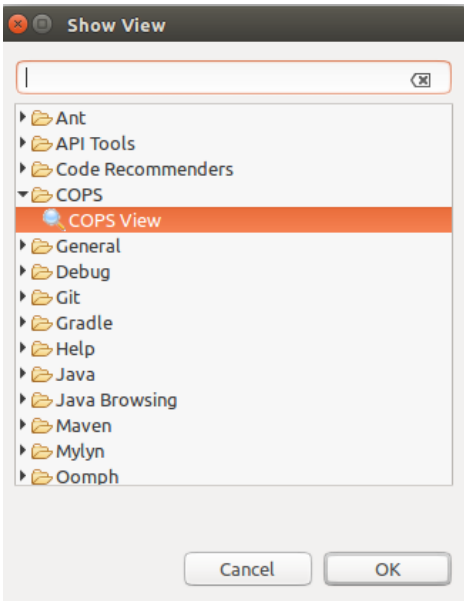

Figura 2. Seleccionar vista de COPS en Eclipse.

Después, creamos un nuevo proyecto de forma normal y una clase nueva. Una vez tenemos la clase creada ya podremos empezar a usar COPS. Situamos el cursor en el espacio en negro, le damos al botón del micrófono para grabar y cuando hayamos terminado de dictar el código pulsamos otra vez en el micrófono y el reconocedor escribirá lo que le hemos dictado. En la siguiente imagen vemos un ejemplo de cómo quedaría nuestro proyecto tras empezar a dictar. En la Figura 3, se muestra el resultado de la ejecución del reconocedor después de haber dictado el texto que se desea. Aunque puede haber errores por el reconocedor, en este caso no se producen.

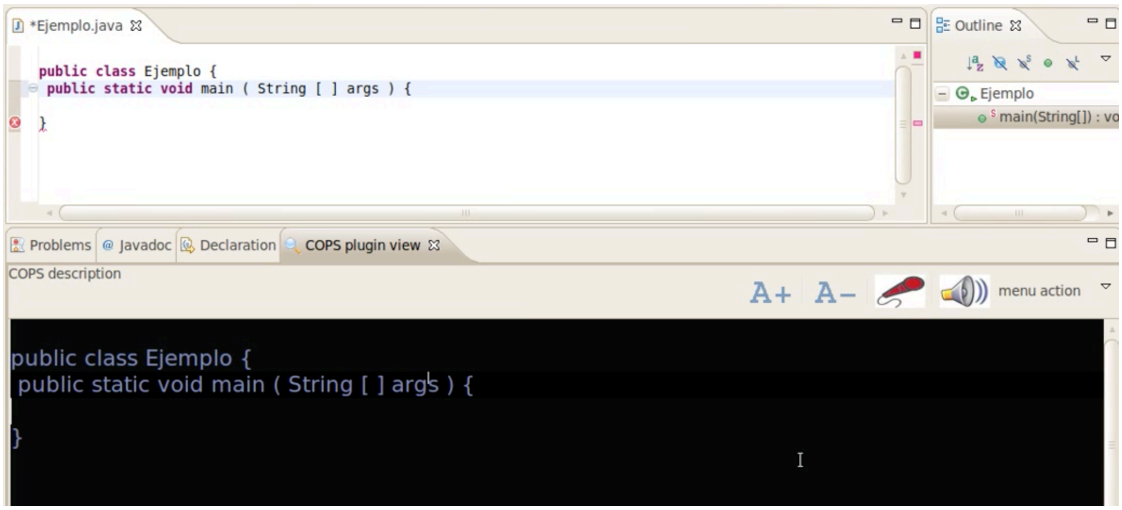

Figura 3. Resultado de la ejecución del reconocedor de voz en COPS.

El código queda replicado arriba, con la vista normal de Eclipse, y abajo con la vista que tiene COPS. Arriba podemos escribir como lo haríamos en cualquier clase de Eclipse y abajo haríamos uso de las herramientas que tiene disponible COPS.

En este trabajo vamos a analizar la importancia de llevar a cabo un mantenimiento del software tanto para proyectos académicos como empresariales, poniendo como ejemplo el proyecto COPS.

1.1 Motivación

No se puede mover el mundo moderno sin software. Las infraestructuras y los servicios públicos están controlados por sistemas informáticos, así como, trenes, aviones, coches, etc. La gran mayoría de electrodomésticos tienen software dentro. La ingeniería del software está formada por un proceso, un conjunto de métodos y herramientas que se deben seguir para poder elaborar software de calidad. Es relativamente sencillo si se conoce un lenguaje de programación crear software, pero, ¿qué es lo que define un software de calidad?

En este trabajo se va a intentar dar una respuesta clara a esta cuestión. Reformulando la primera frase: No se puede mover el mundo sin software que proporcione la funcionalidad, el rendimiento requerido para el usuario y además sea mantenible, confiable y usable [2].

1.2 Objetivos

Vamos a trabajar en un proyecto en el cual no se siguieron las directrices adecuadas para poder crear software de calidad, centrándonos en el mantenimiento. Vamos a explicar el concepto de mantenimiento, las métricas a seguir para poder decir que estamos creando software mantenible y las vamos a aplicar a nuestro proyecto, más concretamente, haremos uso del mantenimiento preventivo, de las técnicas de reingeniería para incrementar la mantenibilidad sin alterar sus especificaciones funcionales. También dejaremos el proyecto para que sea fácilmente configurable en cualquier equipo con un determinado sistema operativo.

Lo que se quiere demostrar con este trabajo es que es más fácil mejorar o resolver deficiencias de un programa si se siguen unas pautas de calidad mientras se está creando, intentar hacer que sea mantenible, en vez de una vez terminado el programa sin haber seguido ninguna norma.

1.3 Estructura del trabajo

Empezaremos explicando en el capítulo 2 las metodologías iniciales que tiene el proyecto y también las metodologías que aplicaremos que favorecen la mantenibilidad sin ser un requisito para que haya mantenibilidad. Después, en el capítulo 3, explicaremos el concepto de mantenimiento de software, los tipos de mantenimiento que tenemos, las dificultades que podemos encontrarnos si decidimos hacer un programa mantenible y unas soluciones que podemos aplicar en el proyecto. Además, definiremos que es un estándar y qué relación tiene con el mantenimiento, explicaremos el estándar más adecuado que podemos aplicar y las actividades que comprende. En el capítulo 4, analizaremos el concepto de calidad y lo relacionaremos con el concepto de mantenimiento para ver cómo se relaciona la calidad de software con la mantenibilidad. En al capítulo 5 definiremos nuestras acciones sobre el proyecto en las metodologías que explicamos en el capítulo 2, además del concepto de ingeniería inversa y reingeniería, que nos será muy útil para

nuestro proyecto y proyectos futuros. Y en el capítulo 6, las conclusiones que podemos sacar de todo este proceso.

2. Metodologías empleadas

2.1 Metodologías empleadas en COPS

En este apartado explicaremos las metodologías que se usaron inicialmente para el desarrollo del proyecto.

Java

El proyecto está escrito en el lenguaje Java. Java es un lenguaje orientado a objetos creado por Sun Microsystems en 1995. En la actualidad, es uno de los lenguajes de programación más popular. Su sintaxis deriva en gran parte de C y C++, pero éste tiene menos utilidades a bajo nivel. Las aplicaciones se pueden ejecutar en cualquier máquina virtual Java (JVM) sin importar el Sistema Operativo o el hardware del equipo [3].
En Java la organización de las clases es muy importante; se pueden dividir los proyectos en paquetes y cada paquete debe contener clases. Elegir bien un nombre para cada paquete es muy importante, ya que dentro de cada uno tienen que ir todas las clases que estén relacionadas con este paquete.
Dentro de cada paquete tenemos clases; las clases se pueden relacionar entre ellas para hacer uso de métodos o de interfaces.

Eclipse

En este caso el proyecto se ha implementado en el programa Eclipse. Eclipse es una plataforma de software compuesto por un conjunto de herramientas de programación de código abierto multiplataforma creado inicialmente por IBM, pero ahora gestionado por la Fundación Eclipse.
Eclipse es una comunidad para personas y organizaciones que desean colaborar desarrollando software comercial y de código abierto. Los proyectos que desarrollan se centran en la construcción de plataformas de código abierto compuesto por herramientas; implementación y administración de software a lo largo de su ciclo de vida. La fundación Eclipse es una corporación sin ánimo de lucro, apoyada por los miembros, que aloja los proyectos Eclipse y ayuda a

cultivar tanto una comunidad de código abierto como un ecosistema de productos y servicios complementarios. Además, es la encargada del desarrollo de Eclipse, el IDE y plataformas de código abierto para el desarrollo de aplicaciones Java [4].

En nuestro proyecto usaremos la versión Luna Service Release 2 (4.4.2), pero podríamos usar cualquier versión de Eclipse.

iAtros

iAtros es el acrónimo de “Improved Automatically Trainable Recognizer Of Speech”. Es una versión mejorada de su predecesor Atros que ha sido adaptado para ser utilizado como reconocedor automático tanto para habla como para texto manuscrito. Está compuesto de dos módulos de preprocesado y extracción de características (para la señal de voz y para las imágenes de texto manuscrito) y un módulo principal de reconocimiento. Los módulos de pre procesado y extracción de características proporcionan vectores de características al módulo de reconocimiento, que utiliza Modelos ocultos de Markov y modelos de lenguaje para realizar la búsqueda de las mejores hipótesis del reconocimiento [20].

2.2 Metodologías que vamos a emplear

Las herramientas descritas a continuación favorecen la mantenibilidad, pero no es obligatorio usarlas; podemos usar otras de índole parecida que resultarán igualmente eficaces si las aplicamos de forma apropiada.

Git

Git es un sistema de control de versiones distribuidas de código abierto y libre diseñado para manejar proyectos grandes y pequeños con eficiencia y velocidad [16]. Git se ha convertido en un excelente motor para el control de versiones con funcionalidad plena. Muchos proyectos de importancia usan Git, sobre todo los del grupo de programación del núcleo de Linux [17]. Es una herramienta muy interesante si estamos haciendo un proyecto entre varias personas para llevar un control sobre lo que modifica cada uno. Hay

herramientas parecidas, como Subversion. Hemos elegido GIT por ser más simple de usar y más conocida.

PMD

PMD es un analizador estático de código fuente. Encuentra fallos de programación comunes como variables no utilizadas, bloques de captura vacíos, creación de objetos innecesarios, etc. Soporta Java, JavaScript y SalesForce.com, entre otros muchos lenguajes. Además, incluye CPD (un detector de copiar y pegar). CPD encuentra código duplicado en Java, C, C ++, C #, Groovy, PHP, Ruby, Fortran o JavaScript entre otros [5]. Hay muchos analizadores estáticos de código; concretamente, para Java otro muy conocido es FindBugs, una herramienta muy parecida a PMD que identifica cientos de errores potenciales. Nosotros nos hemos decantado por PMD porque nos ha parecido la herramienta más completa y simple de utilizar.

Javadoc

Javadoc es una herramienta creada por Oracle que permite generar documentación a partir de comentarios en el código fuente [10]. Se generan páginas en HTML de documentación API de archivos de origen Java.
Los comentarios de tipo Javadoc se escriben entre /** … */. Está destinado principalmente a escribir comentarios en clases y métodos, aunque también tiene unos indicadores especiales que podemos usar prediciéndolos por un '@'. Por ejemplo, si una clase no la hemos escrito nosotros podemos poner junto con la descripción de la clase '@author' y el nombre del autor. Para escribir una buena documentación hay que tener en cuenta que los comentarios deben añadir claridad al código y ser simples. Además, hay que evitar la decoración que era tan común en los años 60 y 70 que no añade ningún valor al código [10].

Bugzilla

Bugzilla es un sistema de seguimiento y notificación de errores. Ha sido desarrollado por Mozilla.org y está escrito en Perl.

Es totalmente gratuito, por ser software libre, y cualquier software puede formar parte de él. Hay importantes proyectos en Bugzilla, como Eclipse, Linux Kernel, LibreOffice, OpenOffice o NetBeans, entre otras, en las que se pueden ver toda la lista de *bugs* [12].

¿Por qué es importante hacer uso de herramientas como Bugzilla? Porque así hacemos que los usuarios sean los propios *tester* de la aplicación. Ellos mismos serán los encargados de informar de errores, se reducen costes de servicio, se incrementa la productividad y se eleva la satisfacción del cliente. Además, los clientes pueden hacer un seguimiento de los *bugs* que indican y los *bugs* que describen otros usuarios. La empresa siempre está en contacto con los usuarios y pueden llevar un seguimiento fácil de los *bugs* pendientes [12].

3. Mantenimiento del Software

El software se actualiza. Lo vemos cada día en nuestro ordenador, tablet o teléfono móvil. Cada vez que sale un proyecto disponible a los usuarios casi inmediatamente ya se recibe una lista de errores, mejoras, peticiones de adaptación, etc. Al final las empresas gastan más dinero en mantener sus programas que en proyectos de creación de nuevos.

Hasta aquí, lo normal. Pero, ¿qué pasa si el equipo que creó ese programa ya no trabaja en la empresa? Una gran cantidad del software que se utiliza hoy en día fue creado hace años. Aunque se hayan seguido los modelos, las técnicas y la tecnología adecuada, estos programas han sufrido tantos cambios para adaptarse a migraciones, nuevas necesidades de los usuarios, cambios que se han hecho sin tener en cuenta la arquitectura general del sistema, etc. Si estos equipos de trabajo dejaron su proyecto mantenible, se podrá sacar a la luz. Pero, ¿y si no dejaron nada documentado? Es posible que el equipo que reciba ese código *heredado* no tenga conocimientos directos sobre el sistema. En estas situaciones los proyectos sufren retrasos, porque el tiempo que podrían estar empleando en modificar ese software lo tendrán que emplear en primero, entender el software, y segundo, modificarlo. Gerald Berns dijo: *"Mantenibilidad y comprensión de un programa son conceptos paralelos: mientras más difícil sea entender un programa, más difícil será darle mantenimiento"* [3].

Tradicionalmente se ha hablado del ciclo de vida del software como una secuencia estructurada de pasos que se deben seguir en el desarrollo de software.

Existen diversos modelos que plantean diferentes etapas:

- Análisis y Definición de requisitos.
- Especificación.
- Diseño.
- Programación.
- Prueba e instalación.
- Operación y mantenimiento

Podemos observar que, según este modelo clásico, el mantenimiento es la última etapa que se ha de realizar. Según la terminología ANSI-IEEE; cuando ya se ha presentado el proyecto al cliente y los usuarios ya lo están usando es cuando empieza la etapa del mantenimiento. Este modelo de creación de software lo que hace es incrementar los costes del mantenimiento, pues con el paso del tiempo existe una tendencia creciente a gastar más recursos en el mantenimiento del software [5].

En la Tabla 1 podemos observar como con el paso del tiempo se ha invertido más presupuesto en el mantenimiento.

Tabla 1. Inversión en mantenimiento del software a lo largo de los años [5].

Referencia	Fechas	% Mantenimiento
Pressman, 1993	1970 - 1979	35% - 40%
Lientz y Swanson, 1980	1976	60%
Pigoski, 1997	1980 - 1984	55 %
Pressman, 1993	1980 - 1989	60%
Rock-Evans y Hales, 1990	1987	67%
Schach, 1990	1987	67%
Pigoski, 1997	1985 - 1989	75%
Frazer, 1992	1990	80%
Pressman, 1993	1990	90%

El origen del incremento de costes del mantenimiento se debe a que es más costoso modificar algo en las últimas etapas del ciclo de vida que en las primeras. Una modificación en la especificación de requisitos en las fases finales de la programación puede suponer que se deba modificar gran parte del proyecto, mientras que, si se detecta en las primeras fases, cuando aún no se ha empezado a programar nada, el coste de solventar esa modificación es mucho menor.

3.1 Tipos de mantenimientos

Actualmente existen cuatro tipos de mantenimiento: correctivo, adaptativo, perfectivo, preventivo. Vamos a definir cada uno de ellos, a continuación [5].

Mantenimiento correctivo

Es aquel cuyo objetivo es localizar y corregir los defectos. Un defecto en el sistema incrementa las posibilidades de que éste falle. Un fallo en el sistema es cuando el programa tiene un comportamiento diferente al esperado. Aunque puede haber más, algunos fallos en el software pueden ser de:

- Procesamiento (por ejemplo, salidas incorrectas de un programa).
- Rendimiento (por ejemplo, se obtiene un tiempo de respuesta demasiado alto ante una acción).
- Programación (por ejemplo, inconsistencias a la hora de diseñar el programa).
- Documentación (por ejemplo, inconsistencias entre la documentación del programa y el manual del usuario) [5].

Mantenimiento adaptativo

Consiste en la modificación de un programa para adaptarse a cambios en el entorno en el que se ejecuta, tanto a nivel software como hardware.
Los cambios pueden ser de una gran envergadura, como, por ejemplo, reescribir parte del programa, o una simple modificación en un módulo. Estos cambios pueden deberse a cambios en el sistema operativo, cambios a nivel de arquitectura de redes o cambios en el desarrollo del programa, entre otros.

La necesidad de un cambio de tipo adaptativo solo se puede determinar monitorizando el entorno [6].

Los cambios a nivel software pueden ser de dos clases:

- En el entorno de datos (por ejemplo, cambio de tipo de bases de datos, o incluir una).
- En el entorno de los procesos (por ejemplo, migrando a una nueva plataforma de desarrollo con componentes distribuidos).

Gracias al mantenimiento adaptativo se consigue alargar la vida del software [5].

Mantenimiento perfectivo

Se basa en modificar el software por cambios en los requerimientos de los usuarios; pueden ser cambios en funciones ya implementadas o nuevas funcionalidades [6].

Podemos dividirlo en dos:

- De ampliación: orientado a ampliar el programa [5]. Esto se debe a que cuando el programa se vuelve un éxito los usuarios van experimentando la necesidad de incorporar nuevas funcionalidades, más allá de las que se pensaron cuando se creó el programa [6].
- De eficiencia: se busca la mejora de la ejecución [5]. Puede ser una mejora en la interfaz para hacerla más intuitiva, mejoras de rendimiento, evitar errores de ejecución, etc.

Mantenimiento preventivo

Consiste en incrementar la mantenibilidad del sistema sin alterar sus especificaciones funcionales [5]. Por ejemplo, podemos modificar o mejorar la documentación, incluir sentencias que comprueben la validez de datos de entrada, reestructurar el programa o incluir comentarios que ayuden a comprender el código. Este tipo de mantenimiento es el que más hace uso de las técnicas de reingeniería [5] y el que nosotros vamos a usar para mejorar nuestro programa.

3.2 Actividades y dificultades del mantenimiento

Desconocer las actividades que implica el mantenimiento del software puede inducir a pensar que no es importante. La idea clásica que se tiene del mantenimiento es que se debe dedicar más tiempo al mantenimiento correctivo, es decir, corregir errores de los programas. Sin embargo, los principales estudios indican lo contrario: se dedica más tiempo y dinero al mantenimiento perfectivo que al resto [5]. Como podemos observar en la Figura 4, en el gráfico se muestran los costes entre los diferentes tipos de mantenimiento.

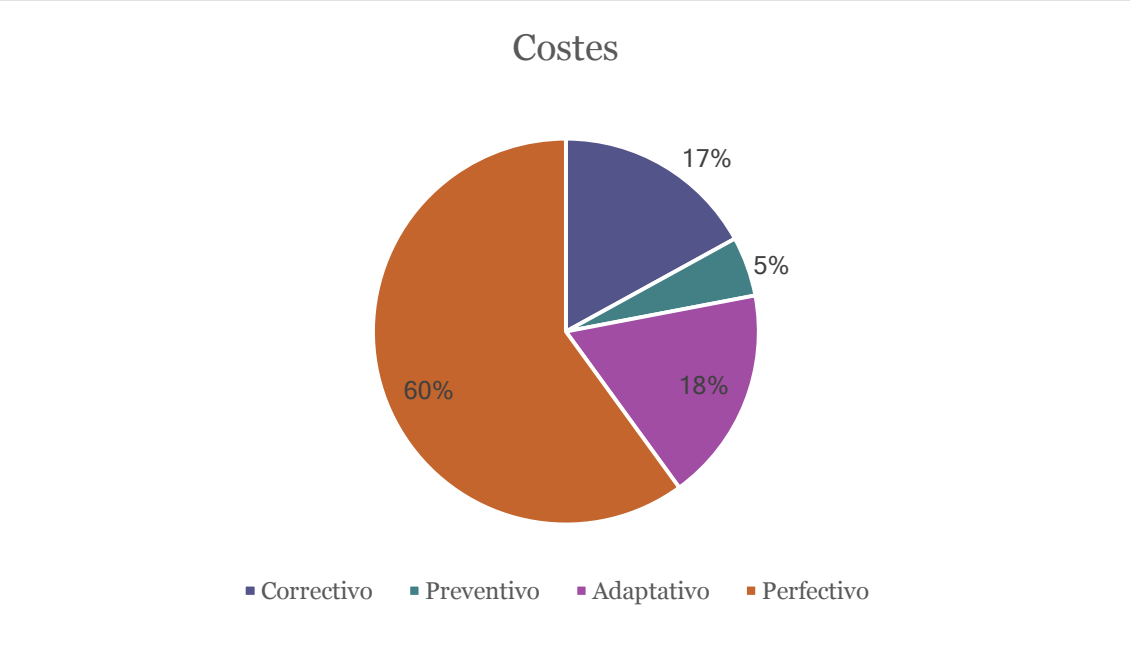

Figura 4. Costes relativos de cada tipo de mantenimiento [5]

Los principales problemas que se encuentran los programadores que deben dar mantenimiento a un proyecto son:

- *Comprensión del software y de los cambios a realizar:* antes de nada, hay que conocer bien el programa que se pretende mejorar. Se deben conocer bien las clases y los métodos para no introducir nuevos defectos. Cerca del 50% del tiempo se emplea en esta fase [5].
- *Modificación del software:* para incorporar los cambios será necesario modificar el software y se deben conocer las repercusiones que tiene. Cualquier cambio debe ser documentado y seguir estándares de calidad [5].

- *Realización de pruebas:* para validar todos los cambios se deben realizar pruebas selectivas que comprueben que los cambios no han afectado negativamente la ejecución. Las pruebas se deberían realizar al final de cada cambio [5].

3.2.1 Código heredado

Con el paso del tiempo se ha ido produciendo una gran cantidad de software; la gran mayoría de este software es antiguo, "heredado" [5].

Desafortunadamente, en muchas ocasiones este software fue implementado antes de que se establecieran unas normas de calidad o ha sido creado sin tenerlas en cuenta. El resultado de esto es que tenemos software de mala calidad. A veces hay sistemas que se han creado con código confuso, con un diseño complejo y una documentación inexistente o mala. Además, si añadimos que ha podido ser modificado varias veces y no haber registro de ello podemos tener serios problemas si nos encargan una migración o añadir nuevas funcionalidades.

Podríamos pensar que lo mejor es no modificar nada; ciertamente, y si satisface las necesidades de los usuarios, mejor es no modificar nada, pero esto no puede mantenerse así durante mucho tiempo. Es imposible que los programas tengan siempre la misma funcionalidad y aspecto. Por ejemplo, programas que se usaban en Windows XP como el paquete Office nunca cambiarían y seguirían iguales en Windows 10, o quizás no se pudieran haber migrado y necesitáramos una máquina virtual para ejecutarlos. El software avanza y aparecen nuevas tecnologías, nuevas maneras de hacerlo más sencillo y eficiente.

Podríamos decir que el software:

- Debe adaptarse para seguir cumpliendo las necesidades de los usuarios ante nuevos ambientes y tecnologías.
- Debe ser mejorado para implementar nuevos requerimientos del negocio.
- Debe ampliarse para que se pueda operar con otros sistemas o bases de datos modernas.

- La arquitectura debe rediseñarse para que pueda ser viable en otros ambientes y en otro tipo de redes [2].

La meta de la ingeniería de software moderna es "*Desarrollar metodologías que se basen en el concepto de evolución; es decir, el concepto de que los sistemas de software cambian continuamente, que los nuevos sistemas de software se desarrollan a partir de los antiguos y [...] que todo debe operar entre sí y cooperar con cada uno de los demás*" [2].

3.2.2 Problemas del mantenimiento

Además del código heredado, hay otros problemas que dificultan el mantenimiento:

- El mantenimiento es realizado por uno o varios programadores según su estilo. No en todas las ocasiones se sigue un estándar para realizar estas tareas.
- Al realizarse muchos cambios los programas tienden a ser menos estructurados, con código que no cumple los estándares, con una documentación desfasada, etc. Esto supone una pérdida de tiempo y un incremento de los costes, porque los programadores pierden tiempo comprendiendo este código, además de correr el riesgo de introducir efectos laterales no deseados.
- La falta de metodologías adecuadas hace que los usuarios participen poco en el proceso de la creación del software. Como consecuencia, el software desarrollado no cumple los requisitos de los usuarios y debe dedicarse más tiempo al mantenimiento. Si se hubieran seguido las metodologías necesarias para establecer de forma clara los requisitos de los usuarios no haría falta que se invirtiera tanto tiempo en el mantenimiento.
- También nos encontramos con problemas de gestión; los programadores tienden a dedicar poco tiempo a la documentación y a programar poniendo comentarios que haga el entendimiento del código más simple [5].

3.2.3 Efectos secundarios del mantenimiento

Las posibilidades de introducir nuevos errores y que esto provoque nuevas peticiones de mantenimiento son muy altas.

Los efectos secundarios de realizar tareas de mantenimiento son de tres tipos:

- Sobre el código: las modificaciones sobre el código fuente tienen una alta probabilidad de introducir nuevos errores. Las más probables son: cambios en el diseño que suponen cambios en el código, eliminación de subprogramas, modificación de funciones que dependen de otras y cambios para mejorar el rendimiento, entre otros [5].
- Sobre los datos: las estructuras de datos son la base de muchas aplicaciones de hoy en día y cualquier cambio en ellas puede provocar fallos irreparables. Los efectos secundarios pueden aparecer debido a: cambiar el tamaño de una matriz o estructuras similares, a modificación de variables locales o globales, redefinición de constantes, o los cambios en los argumentos de subprogramas [5]. Para evitar que ocurran estos efectos secundarios hay que dejar bien documentada la estructura de la base de datos y qué programas o subprogramas hacen uso de ella y con qué argumentos.
- Sobre la documentación: los efectos secundarios aquí se producen cuando se modifica cualquier elemento sobre el programa y no se documenta o no se documenta bien. Suele ocurrir que aparecen nuevos mensajes de error no documentados, texto no actualizado y tablas o índices no actualizados [5]. Es muy importante dejar la documentación bien escrita para reducir efectos secundarios de otro tipo.

3.2.4 Soluciones al problema del mantenimiento

Ante todas estas dificultades existen soluciones.

Podemos dividir las soluciones en dos apartados: soluciones de gestión (organizativas) y soluciones técnicas (metodologías y herramientas) [5]:

1. Soluciones de gestión: Para evitar que el mantenimiento haga un constante uso de recursos se necesita que los programadores más veteranos estén comprometidos a desarrollar software pensando siempre en el mantenimiento para evitar errores futuros. Se debe tener en cuenta también que debe aumentar la calidad del software que se desarrolla; los programadores deben seguir estándares de calidad, llevar una gestión estructurada del mantenimiento y una documentación correcta de todos los cambios [5].
2. Soluciones técnicas: Existen herramientas que han sido diseñas para facilitar la tarea del personal de mantenimiento. Los principales métodos empleados en el mantenimiento del software son la reingeniería, la ingeniería inversa y la reestructuración [5].

- La *reingeniería* consiste en examinar y modificar un sistema para reconstruirlo de otra forma.
- La *ingeniería inversa* es el proceso de analizar del sistema sus compontes, las relaciones existentes y crear representaciones del sistema de otro modo o en otro nivel de abstracción, como crear un diagrama de clases a partir de una base de datos.
- La reestructuración de datos consiste en la modificación del software para hacerlo más fácil de entender y cambiar [5].

Más adelante explicaremos estos términos con más profundidad para enfocarlos a su uso en nuestro proyecto.

3.3 El estándar ISO/IEC 14764:2006

Un estándar es un acuerdo documentado que contiene especificaciones técnicas, reglas, guías o definiciones que deben seguir los productos para asegurarse de que se ajustan a su propósito.
Hay muchos estándares que podemos definir, pero para nuestro proyecto hemos elegido el ISO/IEC 14764:2006, por ser el más útil para el ámbito del mantenimiento y el más reciente. Hay versiones parecidas (como el ISO/IEC 12207:1989 o el ISO/IEEE 1219:1993) que no vamos a definir.

Este estándar internacional propone una orientación, una guía de cómo llevar a cabo el proceso de mantenimiento. También enfatiza en la mantenibilidad de los productos software, la necesidad de modelos de servicio de mantenimiento y la necesidad de una estrategia y un plan de mantenimiento.
Describe con mayor detalle el proceso de mantenimiento que se define en el ISO/IEC 12207, sustituyéndolo.
Este estándar incluye definiciones para los distintos tipos de mantenimiento que hemos definido previamente, pero de otro modo. Además, proporciona orientación que se aplica a la planificación, ejecución, control, revisión y evaluación del cierre del proceso del mantenimiento.
Se ha descrito un proceso iterativo para administrar y ejecutar las actividades del mantenimiento del software. No tiene restricciones de tamaño, complejidad, criticidad o de tipo de aplicación software y se puede aplicar a cualquier proyecto. Se utilizan normas para discutir y representar cada fase del proceso software. Los criterios que se establecen se aplican tanto a la planificación del mantenimiento de software mientras se está desarrollando como a la planificación y ejecución del mantenimiento para proyectos ya terminados.
Proporciona el *framework* con el cual los planes genéricos y específicos del mantenimiento del software se pueden ejecutar, evaluar y adaptarlos al alcance del mantenimiento y a la magnitud de determinados productos software. Además, aporta el entorno conceptual, la terminología y los procesos para permitir que se pueda aplicar correctamente la tecnología (herramientas, técnicas y métodos) al mantenimiento de software [13].

Aunque lo mejor sería que la planificación del mantenimiento empezara en la etapa de planificación del desarrollo software.

Este estándar trabaja con una serie de conceptos que vamos a definir a continuación [13]:

- *Mantenimiento perfectivo*: modificación del producto software después de su entrega, para mejorar su rendimiento o mantenibilidad.
- *Mantenimiento preventivo*: modificación del producto software después de su entrega, para detectar y corregir defectos latentes antes de que produzca grandes fallos.
- *Mantenimiento adaptativo:* modificación del producto software después de su entrega, para que sea utilizable en un entorno nuevo.
- *Mantenimiento correctivo:* modificación reactiva de un producto software después de su entrega, para corregir defectos destacados [13].
- *Mejoras de mantenimiento:* modificación que se le hace a un producto para cumplir nuevos requisitos. Una mejora de este tipo puede ser adaptativa o perfectiva.
- *Peticiones de modificación*: es un término genérico que se usa para identificar las propuestas de modificación. Puede, posteriormente, clasificarse como cualquiera de los cuatro tipos de mantenimientos.
- *Informe de problemas:* es un término que se usa para identificar y describir problemas que se detectan en productos software.
- *Transición software*: es una secuencia controlada y coordinada de acciones en las que el desarrollo software pasa de la organización inicial que lo desarrolla a la organización que le dará el mantenimiento.
- *Mantenimiento del software:* el total de las acciones que se dan al software para que sea rentable. Las actividades se llevan a cabo durante la fase de pre-entrega, así como en las etapas posteriores.
- *Plan de mantenimiento:* documento que establece las prácticas, recursos y secuencias de actividades relevantes para mantener un producto software [13].

Como se muestra en la Figura 5, cuando se pide una modificación, puede ser de tipo corrección o de tipo mejora. Dependiendo del tipo que sea se corresponde con un tipo de mantenimiento.

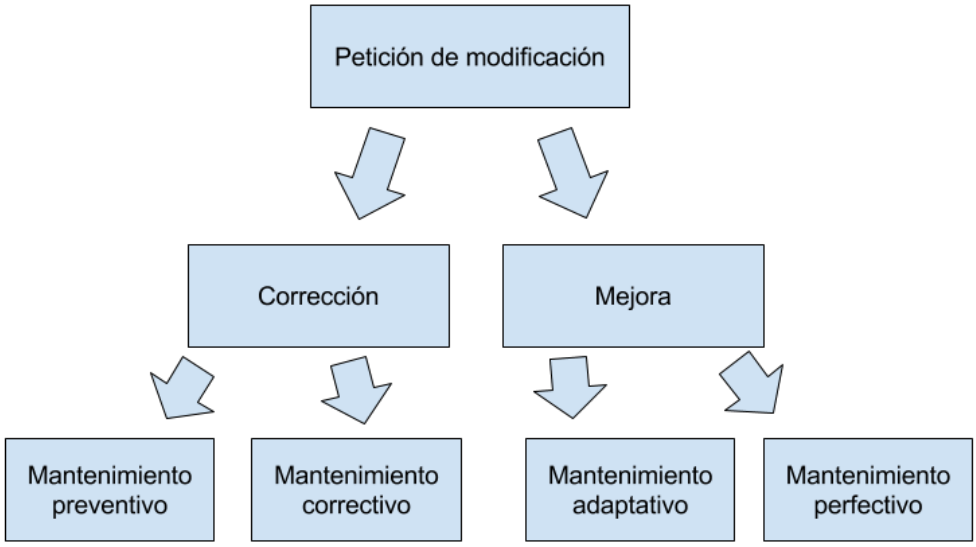

Figura 5. Esquema de la relación de conceptos [13].

El equipo o persona encargada del mantenimiento debe asegurarse de que el proceso de mantenimiento existe y es funcional antes de desarrollar cualquier producto software. El proceso debe activarse cuando exista un requisito de mantener un producto. A continuación, el estándar explica paso a paso y de forma ordenada cómo empezar el proceso del mantenimiento del software [13]. Las actividades que comprenden el mantenimiento de software en la ISO son las que se muestran en la Figura 6 y se detallarán a continuación:

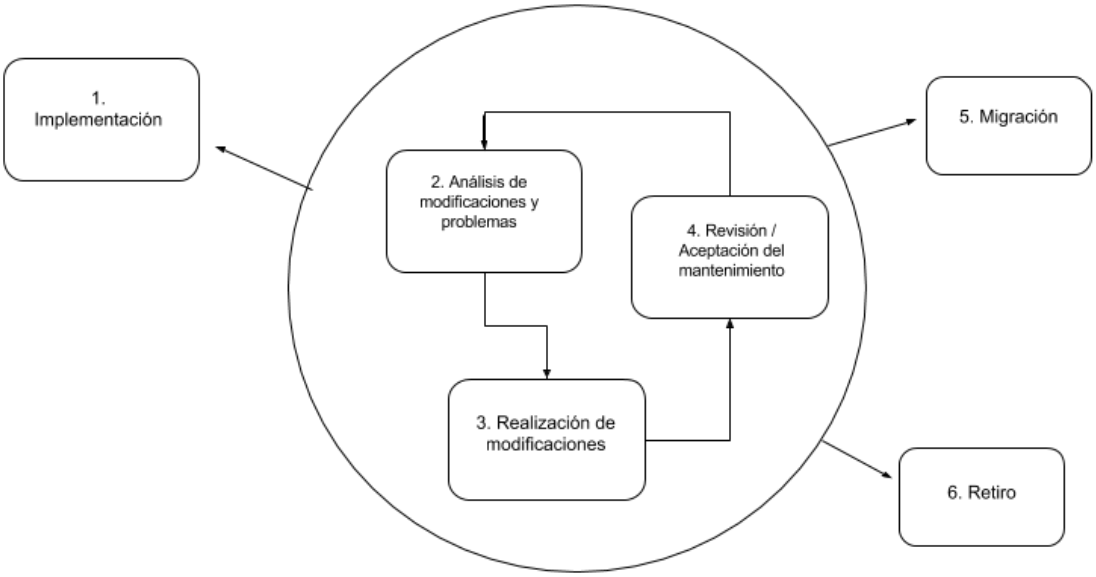

Figura 6. Actividades del mantenimiento de software en la ISO/IEC 14764:2006 [13].

1. Implementación.

Durante esta fase el mantenedor debe establecer los planes y procedimientos que se deben ejecutar durante el proceso de mantenimiento. El plan de mantenimiento debe desarrollarse en paralelo con el plan de desarrollo [13]. Algunos de los procedimientos que se definen sirven para recibir, registrar y seguir la pista a los informes de problemas y peticiones de modificación por parte de los usuarios. Además, se implementan o definen las interfaces organizacionales con el proceso de gestión de la configuración [14].

2. Análisis de modificaciones y problemas.

Esta y las actividades posteriores se activan después de la transición del software y se llaman iterativamente cuando surge la necesidad de alguna modificación [13].

El mantenedor debe:

- Analizar el informe del problema o requerimiento de modificación para determinar su impacto en la organización, en el sistema existente y en las interfaces.
- Comprobar que el problema existe o intentar reproducirlo.

- Definir varias opciones para implementar la modificación.
- Documentar el informe del problema o requerimiento de la modificación, los resultados e informes de implementación.
- Obtener la aprobación para realizar las modificaciones oportunas [14].

3. Realización de las modificaciones.

En esta fase el mantenedor desarrolla y prueba las modificaciones mediante testeos [13].

4. Revisión / Aceptación del mantenimiento.

Hay que asegurarse de que las modificaciones son correctas y de que se hayan realizado de acuerdo a las normas aprobadas utilizando la metodología correcta [13]. Para asegurarse de que son éstas las modificaciones necesarias, el mantenedor se pone en contacto con los usuarios o con el cliente para determinar los elementos que deben ser modificados [14].

5. Migración.

Esta fase no es obligatoria, pero durante el ciclo de vida del software es muy probable que tenga que ser modificado para funcionar en ambientes diferentes. Para migrar un sistema a un nuevo entorno de ejecución, el mantenedor debe determinar las acciones necesarias para llevar a cabo la migración y, después, desarrollar y documentar los pasos necesarios para realizarla [13]. Además, deberá:

- Notificar a los usuarios del proceso de migración, cuándo empezará y cuándo terminará.
- Evaluar el impacto que tendrá en el nuevo entorno.
- Formar a los usuarios en el nuevo entorno.
- Archivar el software antiguo [14].

6. Retirada.

Esta etapa tampoco es obligatoria, pero es posible que cuando el producto llega al fin de su vida útil se deba retirar. Antes de eso, se debe hacer un análisis para ayudar a tomar la decisión de retirar el producto. Normalmente se suelen basar en la economía y en la rentabilidad que da a la empresa. Una vez retirado se puede incluir en el Plan de Retiro [13].

Para cada una de las fases explicadas, el estándar tiene cinco apartados:

- Entradas: son documentos *input* necesarios para realizar las tareas.
- Pasos detallados a seguir.
- Los controles, unas guías para asegurarse de que se obtienen salidas correctas.
- Procesos de soporte que ayudan a realizar las tareas.
- Las salidas, que son documentos y objetos generados durante la tarea [13].

3.4 Organización inicial del Proyecto

Nuestro proyecto se divide en 5 paquetes y un total de 30 clases. La relación entre paquetes y clases inicialmente es:

- Codeshine; una clase.
- Preferences; siete clases.
- Speech; seis clases.
- Utils; doce clases.
- Views; cuatro clases.

La organización de las clases en los paquetes es muy importante, como hemos comentado en el capítulo 2. Dentro del paquete `codeshine.speech` encontramos las clases que nos permiten que el reconocedor de voz funcione y no debe estar ninguna clase que sirva para el funcionamiento del reconocedor fuera de ahí.

Todos los paquetes están relacionados, esto quiere decir que, las clases hacen uso de otras clases que están en distintos paquetes.

En la clase `Activator`, del paquete `codeshine`, tenemos estas dos líneas.

```
import codeshine.preferences.IPreferenceConstants;
import codeshine.speech.TtsClass;
```

Si ahora fuéramos a la clase *TtsClass* podríamos encontrar:

```
import codeshine.preferences.IPreferenceConstants;
import codeshine.utils.TokenList;
import codeshine.utils.Token;
import codeshine.utils.Trie;
```

Si están en distintos paquetes, la forma que tenemos de relacionarlas es con un *import*.

Algunas clases tienen comentarios en los métodos, pero la gran mayoría no. Además, había clases con líneas de código comentadas. Tampoco estaban las bibliotecas necesarias para que funcionara el proyecto, por lo que se han tenido que buscar una a una.

4. Calidad de Software

Hasta ahora, hemos estado definiendo conceptos que favorecen el mantenimiento del software y con ello la mantenibilidad de un producto, pero el propósito de que un producto sea mantenible, es que tenga calidad. ¿Cómo podemos determinar que nuestro producto cumple los requisitos para poder decir que tenemos software de calidad?

Durante años se ha pedido un modelo, un estándar que definiera características de calidad, subcaracterísticas y métricas asociadas, que se pueden utilizar no solo para evaluar un producto software, si no también para definir requerimientos de calidad y otros usos adicionales [5].

4.1 Calidad de software según el estándar ISO 9126

El estándar ISO 9126 define un modelo de calidad de software cuyos atributos se dividen en seis características, los cuales se dividen en subcaracterísticas, como vemos en la Figura 4:

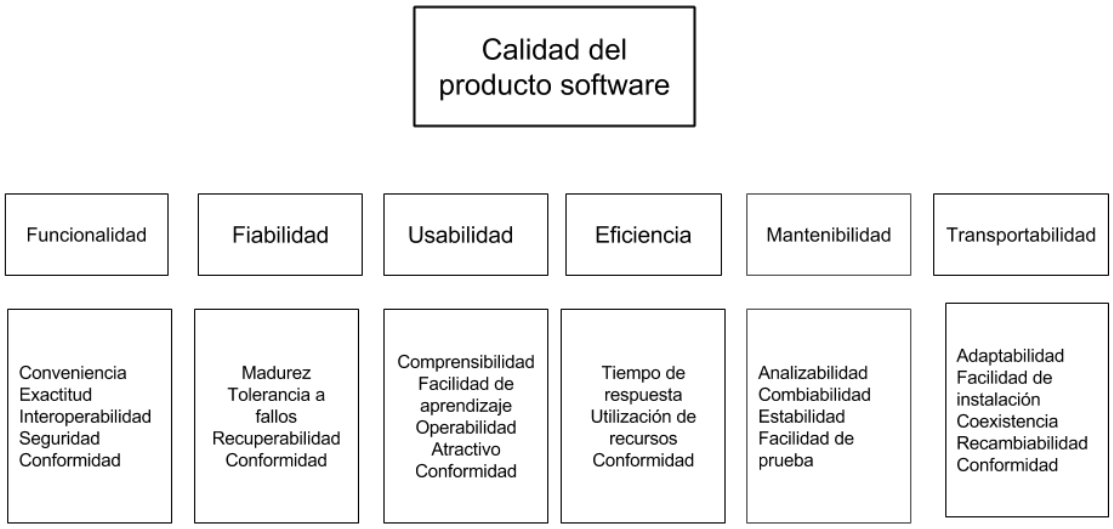

Figura 7. Características ISO 9126 [5].

Podemos observar que la ISO tiene la mantenibilidad como característica indispensable para obtener software de calidad.

Las subcaracterísticas que se deben cumplir son algunas de las que hemos estado mencionando [5].

- Analizabilidad: es la capacidad de un producto software de diagnosticar sus deficiencias, causas de fallos, o de identificar las partes que deben ser modificadas. Debe ser sencillo ver dónde falla un programa y no perder mucho tiempo buscando soluciones a un error.
- Cambiabilidad: es la capacidad que tiene un producto software de permitir implementar modificaciones especificadas previamente. La implementación incluye los cambios en el diseño, en el código y en la documentación. Si el software es modificado por el usuario final, entonces la cambiabilidad puede afectar a la operabilidad.
- Estabilidad: es la capacidad que tiene el software de minimizar los efectos inesperados sobre las modificaciones, esto quiere decir que, si modificamos una función, por ejemplo, no tengamos que modificar muchas clases. Este término está muy relacionado con la cambiabilidad: si en un programa es fácil incluir modificaciones; esto se refiere tanto como a modificar algo que ya estaba implementado como a añadir métodos o eliminarlos; si resulta sencillo hacer este tipo de acciones, podemos decir que también resultará sencillo que estos cambios no afecten a todas las clases.
- Facilidad de prueba: es la capacidad del software de poder evaluar las partes que han sido modificadas.
- Conformidad: es la capacidad del producto software de satisfacer los estándares relativos con la mantenibilidad.

Un término muy interesante que nos incluye la mantenibilidad en la ISO es que el software debe ser fácil probarlo; esto quiere decir que tiene que ser relativamente fácil desarrollar programas o utilizar herramientas que sirvan para el testeo de un programa. Resulta muy útil para comprobar que nuestro programa no tiene un comportamiento inesperado ante valores de entrada diferentes. Además, también se incluye el término conformidad, que nos indica que el software debe cumplir los estándares de calidad para que pueda ser presentado [5].

4.2 Medidas de mantenibilidad

Es importante saber no solo si un programa cumple las características de calidad y, más en concreto, las de mantenibilidad, si no también saber el grado de mantenibilidad de un programa. Si se tienen dos opciones de software para comprar, las dos son iguales, mismo lenguaje, mismo propósito, misma funcionalidad, pero, una es más mantenible que otra. ¿Se escogería la que es más mantenible porque otorga mayor facilidad a la hora de añadir o modificar el software o la opción en la que se tendría que esperar mucho más tiempo cuando se quisiera modificar? Es muy interesante conocer la forma en la que podemos medir la mantenibilidad del software [5].

4.2.1 Medidas externas de la mantenibilidad

Para medir la mantenibilidad de manera externa prestaremos atención a la velocidad en la que implementaremos un cambio una vez se establece la necesidad de realizar ese cambio. Por esta razón se establece una medida llamada *tiempo medio para reparación* (MTTR), que indica el tiempo medio en el que el departamento de mantenimiento implementa un cambio y pone de nuevo operativo el sistema software modificado. Para calcular este tiempo es necesario conocer [5]:

- Tiempo para identificar el problema.
- Tiempo de retraso administrativo.
- Tiempo para obtener las herramientas de mantenimiento.
- Tiempo para analizar el problema.
- Tiempo para hacer la especificación del cambio necesario.
- Tiempo para realizar el cambio (incluyendo pruebas y revisiones).

También se pueden utilizar otras medidas dependientes del entorno si las tenemos disponibles:

- Número de problemas sin resolver.
- Tiempo empleado en problemas no resueltos.
- Porcentaje de cambios que introducen nuevos defectos.
- Número de módulos modificados para implementar un cambio.

Con estas medidas se puede evaluar el grado de actividad de mantenimiento desarrollada y la efectividad del proceso de mantenimiento [5]. Si el equipo desarrollador ha seguido pautas del mantenimiento nos resultará sencillo continuarlas.

4.2.2 Medidas internas de la mantenibilidad

Algunas medidas estructurales pueden servir como medida, ya que existe una clara relación entre productos pobremente estructurados y documentados y la facilidad de mantenimiento de estos productos una vez se han implementado.
Para determinar las medidas (y sus atributos internos relacionados) que más afectan a la mantenibilidad, se debe hacer una combinación de medidas internas con medidas externas [5].
Por ejemplo; para medir internamente la Estabilidad, (frecuencia de fallos debidos a efectos laterales producidos después de una modificación), tendríamos que contar el número de fallos debidos a efectos laterales y comparar con el número total de fallos corregidos.
Así la estabilidad(x) se definiría por:

X=1-A/B

Donde:

- A= número de fallos debidos a efectos laterales detectados y corregidos.
- B = el número total de fallos corregidos.

0<=X<=1, Claramente cuanto más cerca de 1 mejor [21].

5. Soluciones técnicas

5.1 Reingeniería e Ingeniería inversa

Los tipos de mantenimiento que conocemos modifican parte o todo del software. Si para el desarrollo de productos utilizamos técnicas de Ingeniería del Software, ¿por qué no utilizar técnicas igualmente rigurosas para la reconstrucción de sistemas? [5].

El énfasis que se le ha dado a las técnicas de reingeniería y de ingeniería inversa ha sido por los problemas que tenían las empresas para dar mantenimiento a programas que llevaban mucho tiempo desarrollando y cambiando sin un mantenimiento adecuado [2].

En el gráfico mostrado en la Figura 8 podemos ver el proceso de reingeniería y las actividades que contiene, descritas a continuación:

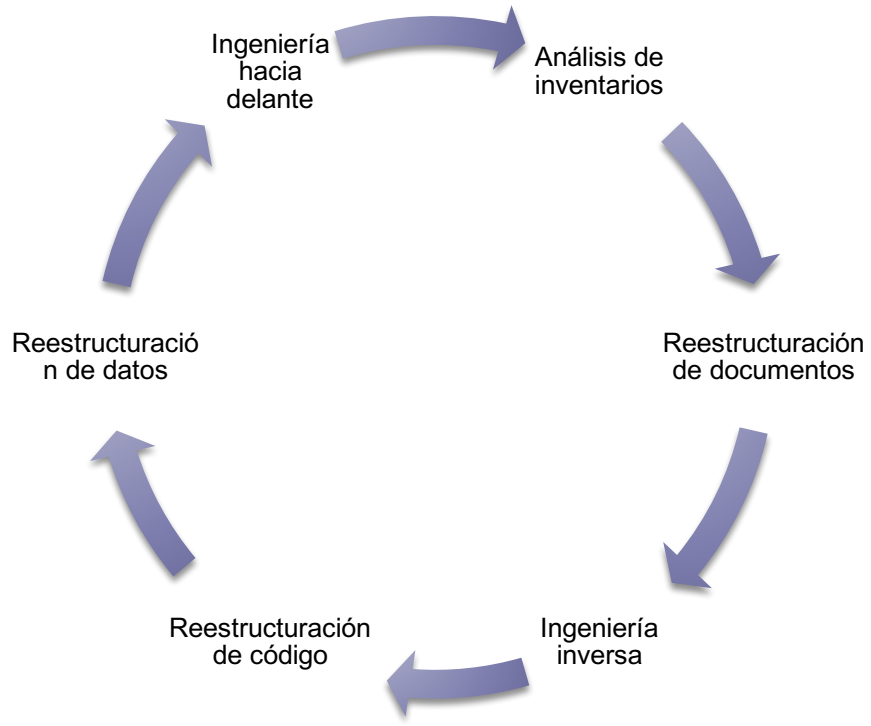

Figura 8. Proceso de Reingeniería [2].

- *Análisis de inventarios:* si estamos en una empresa, la empresa tiene que llevar un registro del software que produce. Examinando ese registro, si prestamos atención al tiempo que lleva el programa, el éxito que tiene, la mantenibilidad, etc., podemos determinar qué programas necesitan aplicar el proceso de reingeniería.
- *Reestructuración de documentos:* la mala documentación es un distintivo de muchos programas heredados; tenemos tres posibles casos que se pueden dar: el primero, si la creación de documentación consume demasiado tiempo, podemos elegir dejarlo así si el programa funciona; si no, tendremos que evaluar cuanta vida útil le queda a ese programa, pues crear documentación desde cero puede llegar a ser una tarea muy costosa. Si se da la situación de que debemos documentar porque el software tiene importancia empresarial y aún tardaremos años en retirarlo, tendremos el segundo caso; tendremos que documentarlo, pero habrá que recortar la documentación al mínimo posible. El último caso; cuando actualizamos programas, se da si tenemos documentación disponible; debemos actualizar la documentación y poco a poco iremos teniendo documentación de mejor calidad [2]. En nuestro proyecto nosotros hemos hecho uso de la reestructuración de documentos y hemos creado la documentación desde cero; al ser un proyecto pequeño lo hemos podido realizar, pero si hubiera sido uno de tamaño mayor se habría invertido demasiado tiempo en realizarlo.
- *Ingeniería Inversa:* es el proceso por el cual se analiza un programa con la intención de crear una representación a un nivel más alto de abstracción que el código fuente. Las herramientas que se usan en la ingeniería inversa extraen información del diseño de datos, del arquitectónico y del procedimental de un programa existente [2].
- Reestructuración de código: es la más común de todas las reingenierías. Se basa en analizar el código fuente con herramientas de reestructuración; se anotan los resultados y luego se reestructura o incluso se puede cambiar el lenguaje a uno más moderno, y se actualiza la documentación con cualquier cambio [2]. En nuestro caso, la reestructuración de código se ha basado en analizar código muerto y

eliminarlo; hay hasta una clase entera que está obsoleta y se ha marcado como *deprecated*.

- Reestructuración de datos: un programa con una mala estructura de datos será muy complicado de adaptar y mejorar. La reestructuración de datos es un proceso de reingeniería a gran escala. Se identifican objetos, atributos de datos y estructuras de datos.
- Ingeniería hacia delante: recupera información de diseño del sistema software existente y usa esa información para alterarlo y reconstruirlo.

Aplicar un proceso de reingeniería en un programa puede costar cantidades significativas de dinero, además de invertir mucho tiempo en ello. Este proceso puede costar meses e incluso años; por esto, todas las empresas necesitan tener un plan estratégico para poder acortar este periodo al mínimo posible, y con esto, conseguir un coste menor [2].

Antes de aplicar ningún proceso, primero debemos empezar comprendiendo el programa al que debemos aplicar el proceso de modificación. Esto es muy importante, ya que si no comprendemos bien el programa que queremos modificar podemos incluir nuevos errores y hacer una documentación errónea.

Durante el mantenimiento debemos tener un conocimiento sobre qué hace el sistema y cómo se comunica con su entorno, además de identificar dónde se requieren cambios en el sistema y conocer cuál es la mejor forma de corregir esto. Para poder corregir los errores tenemos que adquirir información acerca de aspectos del sistema, como el dominio del problema, qué resultados se obtienen de la ejecución, la relación causa-efecto y las funciones utilizadas para la toma de decisiones del programa [19].

No es necesario que todos los miembros del equipo de mantenimiento conozcan todos los detalles del sistema, puesto que cada uno tiene una parte del programa asignada. En el equipo podríamos tener, por ejemplo, gestores, analistas, diseñadores y programadores; cada uno tendría una parte del trabajo que dependería de él y no tiene por qué conocer al detalle la parte que realizarán sus compañeros, así pues:

- Los gestores tienen responsabilidad ante la toma de decisiones; deben tener conocimientos de soporte a las decisiones con el fin de tomar decisiones bien informadas. Dependiendo del tipo de decisiones que deben tomar tendrán que estar mejor informados. No tienen por qué conocer el diseño del sistema a nivel de arquitectura o detalles de implementación [19].
- Los analistas requieren tener conocimientos sobre el dominio del problema para realizar tareas como la determinación de requisitos funcionales y no funcionales. Además, deben establecer la relación entre el software y el entorno en el que se ejecuta para saber cómo afectan los cambios al sistema. Se debe tener una visión global del sistema. Se pueden emplear modelos físicos, como diagramas de contexto en los que se expliquen los principales componentes del sistema y cómo se relacionan entre ellos [19].
- Los diseñadores pueden trabajar a dos niveles: el diseño arquitectónico (que analiza componentes funcionales, estructuras de datos conceptuales y la interconexión de los componentes) y el diseño detallado (que trabaja más con representaciones de datos, estructuras de datos y con interfaces entre los distintos componentes). Los diseñadores extraen esta información y determinan cómo se podrían mejorar para ser encajadas entre la arquitectura, los flujos de datos, las estructuras de datos, etc. Examinan el código fuente para determinar una idea aproximada del tamaño del trabajo, las áreas del sistema que se verán afectadas y las habilidades necesarias que deberá tener el equipo de programación [19].
- Los programadores están obligados a conocer todos los efectos que pueda tener la ejecución del sistema en todos los niveles de abstracción. En un nivel más alto de abstracción el programador tiene que saber la función de los componentes individuales del sistema y su relación causal. A un nivel más bajo, debe entender para qué sirve cada método del programa, cuál es la secuencia de ejecución, la transformación de los objetos, etc. Conocido todo esto, al programador le será más fácil tomar decisiones sobre reestructuración de código, localización de errores y notificar sobre errores previsibles [19].

El proceso de conocer bien el código es un proceso complejo y ocupa gran parte del tiempo; además, se incrementa si no tenemos documentación, si es código muy complejo o si no tenemos suficiente conocimiento del dominio. El objetivo de la ingeniería inversa es ayudar a comprender la estructura interna de procesos complejos [5].

En el gráfico de la Figura 9, vemos las actividades que comprende el proceso de ingeniería inversa, así como la relación que tiene con la reingeniería.

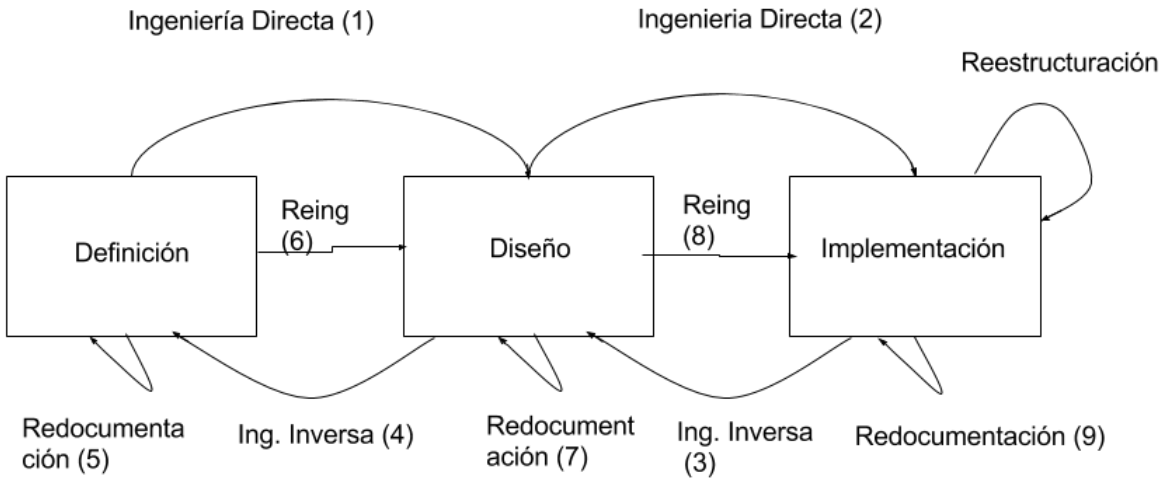

Figura 9. Proceso de Ingeniería Inversa [19].

Aplicar un proceso de ingeniería inversa se puede hacer en cualquier fase del proceso de desarrollar software, aunque lo aconsejable y lo más común es cuando tenemos el código fuente, ya que, es posible que con el paso del tiempo se hayan perdido los documentos con la especificación formal, el diseño, etc. [19].

Los términos que vemos en la figura 9 coinciden con los definidos en la reingeniería a excepción de la definición de Ingeniería Directa, que se refiere al enfoque tradicional de desarrollo de software, comenzando desde el análisis de requisitos, luego el diseño y después la implementación. Los números marcan el orden en el que debe aplicarse el proceso [19].

Una vez hemos comprendido el programa podemos proceder a aplicar el proceso de ingeniería inversa.

La ingeniería inversa es una técnica que nos ayudará a solventar estos problemas; no es una técnica que nos conduzca a la introducción de cambios en el programa, sino que facilita la implementación de éstos; los cambios se implementan utilizando técnicas como la ingeniería directa, reestructuración y reingeniería [19].

Éstas técnicas nos ayudarán a comprender las estructuras de datos, las funciones y los procesos:

- Las funciones se caracterizan a menudo por una relación entrada-salida; una función toma un valor como entrada y produce otro como salida. Durante este proceso, nosotros solo estaremos interesados en lo que hace la función y no en cómo funciona [19].
- Las estructuras de datos consisten en pasar el modelo físico al lógico y finalmente al modelo entidad relación o al diagrama de clases. Pasamos de tener el código de la base de datos a tener un gráfico con las tablas, la información más importante representada y sus relaciones [19].
- Los procesos los representamos también gráficamente, representando el orden exacto en el que se realizan las operaciones. Hay dos tipos de procesos que pueden ser representados: procesos concurrentes (que se comunican a través de compartir los datos que se almacenan en un espacio de memoria asignado) y los procesos distribuidos (que se comunican a través del paso de mensajes y no han compartido datos) [19].

5.2 Control de versiones

En nuestro proyecto hemos usado el *plugin* para Eclipse EGit. Git tiene una versión página web llamada GitHub donde está todo el repositorio de proyectos. Hay varios tipos de cuentas que podemos crear, según las necesidades de nuestro proyecto; si queremos que sea privado tendremos que pagar, pero si por el contrario no nos importa compartir nuestro proyecto podemos crear una cuenta gratuita. Nosotros tenemos una cuenta gratuita cuyo *nick* es *'albelmelg'*; ahí tenemos nuestro proyecto compartido con la comunidad de GitHub, con lo que cualquiera podría descargarlo y colaborar con nosotros.

Una vez creada la cuenta debemos sincronizarlo con el *plugin* que hemos instalado en Eclipse para poder subir al repositorio *online* copias del proyecto. Para ello hemos ido a Preferencias, Git, Configuración y en Añadir entrada, añadimos uno de user.email y otro de user.name para poder conectarnos al repositorio *online* de GitHub. Si queremos subir todo el proyecto a GitHub tenemos que hacer click derecho sobre el nombre del proyecto e ir a la opción Team y después Share Project; ahora seleccionamos Git y seguimos los pasos, nos pedirá la contraseña y, una vez terminado, tendremos todas las opciones disponibles de Git en Team.

Una vez está sincronizado lo primero que podemos hacer es Commit. Esta acción guardará los cambios hechos en local al repositorio. Cuando hacemos Commit no hay que actualizar todas las clases (podemos escoger si alguna no queremos sincronizarla) y siempre hay que escribir un comentario; el propósito de esto es que cada modificación en cada clase quede reflejada con un comentario. Como recomendación, lo mejor es que a cada cambio que se haga en una clase se haga Commit y que en el comentario quede reflejado qué método se ha cambiado, eliminado o añadido. El propósito de esto es, si trabajamos solos, identificar mejor dónde hemos cometido un error al poder descargar el proyecto en versiones antiguas; si trabajamos junto a otra persona, esa persona al actualizar su proyecto en local al del repositorio sabe en cada momento qué es lo que la otra persona ha modificado. A la parte izquierda del nombre de cada clase, si la modificamos nos aparecerá un símbolo “>”; esto nos indica que esa clase no está sincronizada con el repositorio.

EGit tiene muchas más opciones, pero comentaremos brevemente sólo las más utilizadas en este proyecto y en general.

La opción Synchronize nos comparará el *workspace* de Eclipse con el repositorio local o, si tenemos varias ramas en el proyecto, puede comparar la rama actual con otra. Con la opción Merge podemos combinar unas ramas con otras; con Fetch and Pull, al clonar repositorios remotos, Git crea copias de las ramas como ramas locales y como ramas remotas. Una operación Fetch actualizará sólo las ramas remotas. Para actualizar también las ramas locales tendrá que realizar una operación de Merge después de buscar. La operación

`Pull` combina `Fetch` y `Merge`. Por último, la opción `Push` hará un `Merge` entre la versión en local y la del repositorio. Todas las opciones mencionadas anteriormente pueden fallar; si fallan saldrá un aviso de que no se ha podido realizar y una pequeña descripción del error [18].

5.3 Análisis estático del código de COPS

En nuestro proyecto haremos uso de la versión 4.0.13, que es la versión más reciente adaptada para Eclipse.

Inicialmente ejecutaremos PMD para ver cuántos errores detecta en nuestro proyecto. Hemos decidido filtrar las reglas sobre las que trabaja PMD y quitar del análisis los *Warning*. El resultado de la ejecución de PMD se muestra en la Tabla 2:

Tabla 2. Resultado de la ejecución de PMD sobre el código inicial de COPS.

Paquete	Violations	Violations/Method
codeshine	17	3.40
codeshine.preferences	366	9.70
codeshine.speech	464	9.26
codeshine.utils	859	6.43
codeshine.views	535	7.34

Algunos errores no se podrán solucionar; por ejemplo, en los métodos `Start` y `Stop` tenemos un `thows Exception` porque el método de la clase padre, `AbstractUIPlugin` tiene un `throws Exception`. Lo correcto sería especificar qué excepción podría ejecutarse y no capturar cualquier excepción que pueda darse.

Algunos de los fallos que detecta PMD son:

- La falta de comentarios.
- Un método que no se usa.

- Una variable debería ser `final`.
- El nombre de una variable es demasiado corto o demasiado largo.
- Si empieza una variable o método por mayúscula y debería empezar por minúscula, etc.

Centrándonos en un análisis más generalizado de todos los paquetes, los errores más comunes que hemos encontrado en el proyecto son:

- *CommentRequired:* se han de comentar los métodos y variables; si este comentario no es del tipo JavaDoc, PMD nos avisa de que hay un error.
- *CommentSize*: los comentarios no pueden ser demasiado largos.
- *DoNotCallSystemExit:* en las aplicaciones de tipo J2EE no deben hacer llamadas a `System.exit()` porque esto pararía la ejecución del JVM. Es preferible que se use `Runtime.getRuntime().exit()` [7].
- *SystemPrintln:* En lugar del uso de `SystemPrintln` hemos optado por el uso de un *logger*. Importamos las clases de `org.apache.log4j.Logger`, creamos la variable estática *logger* (`static Logger logger = Logger.getLogger(NombreClase.class)`) al inicio de la clase y ya podemos usar el método `logger.info()` en lugar de `System.out.println`.
- *LocalVariableCoudlBeFinal:* si una variable se asigna solo una vez debe declararse como final [7].
- *VariableNamingConventions:* las variables deben empezar por minúscula a no ser que sean constantes [9].
- *Short/LongVariable*: modificamos el nombre de la variable para adaptarla a los estándares. Se considera variable *long* a las que tienen más de 16 caracteres y *short* a las que tienen menos de 7. Además, los nombres deben ir acorde a su funcionalidad, no a su tipo [9].
- *RedudantFieldInitializer*: este aviso sale cuando declaramos una variable, le damos un valor y luego en el código antes de usarla volvemos a darle un valor. Resulta redundante darle un valor al declararla y luego volver a asignarle un valor.
- UselessParenthesis: el uso abundante de paréntesis no es correcto, solo se deben usar los necesarios.
- *Raw type. References to generic types should be parameterized:* cuando creamos un *Array List* en una clase debemos ponerle un tipo. Por

ejemplo; `List <Trie> childrens = new ArrayList<Trie>();` en lugar de omitir el tipo de objeto `Trie`.

- *AvoidCatchingGenericExceptions:* capturar excepciones genéricas da la sensación de que el programador no sabe qué excepción puede salir al ejecutar ese código y por eso las captura todas. Debemos evitar este tipo de comportamiento.
- *Unnecesary use of fully qualified name:* este aviso nos quiere decir que, si ya hemos importado esa clase, no es necesario que la llamemos otra vez con su nombre completo para poder usarla. Por ejemplo; importamos `org.eclipse.jface.util.PropertyChangeEvent`; si queremos escribir un método que reciba como argumento un `PropertyChangeEvent` no es necesario volver a escribir todo el nombre para poder usar el objeto como parámetro; al estar importado basta con poner `PropertyChangeEvent` evento para poder hacer uso de él; resulta redundante importar la clase y luego usar su nombre completo otra vez para usarla.
- *UnusedPrivateMethods:* nos indica que hay algún método privado que no se usa dentro de la clase, por lo que se podría eliminar [8].

5.4 Documentación de COPS

Esto es en lo que más tiempo se ha invertido. Comentar código sin tener ningún comentario es una tarea complicada, ya que tenemos que entender el trabajo de otras personas para explicar su significado. A veces puede ser sencillo si el código es simple, si los nombres de los métodos, variables y clases son explicativos; pero si no tuviéramos como mínimo esto podría ser un quebradero de cabeza para los encargados de hacer una buena documentación.

Nosotros hemos preferido hacer una documentación corta y concisa, aunque en otras clases es un poco más larga. Documentar código no debe ser contar una historia, debe ser algo claro y corto.

Es muy importante escribir la documentación mientras se programa, porque es cuando mejor se sabe el propósito y qué hace un método o una clase o por qué se decide que una variable sea pública o privada. Además, debe documentarse

por qué se hace algo y no sólo lo que hace para que otros entiendan la importancia de ese método, clase, etc. [10].

Para documentar clases hemos explicado el propósito de la clase; podemos poner la versión y el autor, pero en nuestro caso no todas las clases tienen el autor y la versión. Para documentar métodos debemos poner primero de todo el propósito del método, con la etiqueta `@param` explicar los parámetros de entrada (diciendo qué son y para qué sirven) y con `@return` explicamos qué valor o valores puede devolver.

Los métodos también pueden contener las etiquetas:

- `@see` si queremos poner un método o una clase que está relacionada con ese método; debemos poner el nombre completo de esa clase.
- `@throws`, si el método puede lanzar alguna excepción, como `@throws java.lang.NullPointerException`.
- `@deprecated`, que indica si un método está obsoleto y ha sido sustituido por otro.

Para nuestro proyecto hemos usado Auriga Doglet, un *plugin* para Eclipse con el que poder generar la documentación en formato Javadoc en PDF. Lo descargamos de su web oficial y, una vez descargado, vamos a *Eclipse, Project, Generate Javadoc* y en el apartado *Use custom Doclet* ponemos `com.aurigalogic.doclet.core.Doclet` y en *Doclet Class Path* el `AurigaDoclet.jar` que hemos descargado. Le damos a *Next* y en *Extra Javadoc Options* indicamos el formato y en qué ubicación queremos que se guarde y con qué nombre:

`-format pdf -out /Users/aslimon/Documents/javadoc.pdf`

Se generará la documentación que hemos escrito en formato Javadoc, en PDF.

5.5 Publicación de COPS en Bugzilla

Los usuarios son los que abren incidencias, indicando que han encontrado un *bug* en la aplicación. Se debe indicar con máxima precisión dónde han encontrado el *bug* y en qué apartado de la aplicación (por ejemplo, en preferencias o en el reconocedor, etc.).

Cuando se envía un *bug* dependiendo de si hay varias personas en el equipo de mantenimiento y del apartado en el que se dé el *bug,* le llegaría un correo al

encargado de esa sección y el personal de mantenimiento tendría una lista ordenada de todos los *bugs* que hay pendientes, solucionados o a la espera de otra versión. Se mantiene una comunicación con el usuario final constante y el usuario puede ver si el encargado de solucionar ese *bug* ha puesto algún comentario o le pide más información.

Hemos publicado en una versión de Bugzilla que tiene el departamento DSIC de la Escuela Técnica de Ingeniería Informática, el proyecto COPS.

En Bugzilla, nuestro proyecto está dividido en 5 componentes, donde se deben reportar los errores según el apartado del *plugin* en que se encuentre el *bug:*

1. Instalación

Problemas en la instalación del *plugin*.

2. Reconocedor de voz

Problemas o peticiones de mejora del reconocedor de voz.

3. Vista del sistema

Problemas o peticiones de mejora con la vista de los iconos.

4. Ventana de preferencias

Problemas o peticiones de mejora en relación a la ventana de preferencias de COPS.

5. Otros

Cualquier problema o mejora que no se pueda clasificar entre los apartados anteriores.

5.6 Otras acciones sobre el proyecto

El proyecto hace uso de la tecnología iAtros para el funcionamiento del reconocedor. Junto al proyecto hay una carpeta llamada *iatros_cops* que contiene todos los ficheros de configuración y demás, para que el reconocedor pueda funcionar correctamente. Antes de poder ejecutar el proyecto se deben cambiar unos ficheros que hay en esta carpeta, concretamente, en la carpeta hay scripts que se ejecutan y necesitan la ruta concreta de donde se está ejecutando el proyecto. Por ejemplo, el fichero *wav2raw* es el encargado de transformar el fichero *wav* que se genera a *raw. El script* coge el valor que se ha generado con esta ruta que viene por defecto en

/home/demos/iatros_cops/cops.wav y lo transforma para dejarlo en la misma ruta (/home/demos/iatros_cops/cops.raw). Si nosotros no tenemos el fichero en la carpeta /home/demos, el script fallará, ya que no podrá encontrar esta ruta. Por eso, y para facilitar la correcta configuración del proyecto en otros equipos, cuando el usuario se descargue el proyecto en su ordenador, lo primero que debe ejecutar es el script que le preguntará en que carpeta tiene el *workspace* de Eclipse, leerá el *path* (la dirección) que ha introducido, moverá el codeshine.zip a esa carpeta, lo descomprimirá y automáticamente se modificarán 6 ficheros (scripts y ficheros de configuración), incluida la clase `AudioRecorder` dónde se modificará el valor de la variable `strPath`. Una vez modificados los ficheros, se recompilará iAtros y ya podremos ejecutar el *plugin* en Eclipse creando un nuevo proyecto en *File, Import, General, Existing Projects into Workspace.*

5.7 Estado final del proyecto

El proyecto sigue dividido en 5 paquetes, ya que son los indispensables para un correcto funcionamiento. Las clases se han reducido a las estrictamente necesarias para que funcione el proyecto. La división de clases y paquetes se ha quedado así:

- Codeshine: una clase.
- Preferences: seis clases.
- Speech: tres clases.
- Utils: once clases.
- Views: tres clases.

Lo que nos da un total de 24 clases, una reducción total de 6 clases que no eran necesarias.

Como vemos en la Tabla 3, la eliminación de clases innecesarias y la corrección de los avisos más frecuentes que daba PMD ha dado como resultado altos porcentajes de mejora de los paquetes.

Tabla 3. Errores corregidos con PMD.

Paquete	Inicialmente	Finalmente	Mejora
codeshine	17	6	64,7%
codeshine.preferences	366	173	52,73%
codeshine.speech	464	247	46,7%
codeshine.utils	859	398	53,66%
codeshine.views	535	230	57%

Además, se han reducido los *Warnings* de 169 a 4.

En cuanto a la documentación en Javadoc, nos ha generado un PDF de 68 páginas con todas las clases y métodos públicos comentados que anexaremos a la memoria como material complementario (Anexo A). Se podría haber hecho una documentación más exhaustiva, pero al partir desde cero, no podíamos dedicarle demasiados recursos.

6. Conclusiones

Se constata la dificultad para realizar tareas de mantenimiento sobre un software que ha sido desarrollado, sin incluir en dicho proceso las pautas requeridas de calidad y mantenibilidad.

En este proyecto, se ha observado que la complejidad del software y la dificultas de su mantenimiento son directamente proporcionales; es decir, cuanto más complejo es el software más difícil es darle mantenimiento.
La aplicación de las directrices de mantenibilidad, es indispensable para que los proyectos puedan tener una vida útil más larga y, como consecuencia, reducir costes a las empresas. Podemos inferir que la aplicación de estas directrices facilita estos trabajos, ya de por si complejos, reduciendo la inversión a realizar y optimizando resultados a medio y largo

Analizando los resultados que hemos obtenido aplicando herramientas como PMD, Javadoc, Git y Bugzilla, observamos que haber aplicado estas herramientas ha sido algo muy positivo para el proyecto, ya que se ha mejorado su mantenibilidad.

En un futuro convendría seguir con el trabajo de mantenimiento, ya que se ha dejado el proyecto en un buen estado para continuar con este proceso. En concreto, sería muy interesante aplicar los procesos de ingeniería inversa y seguir el modelo de la ISO 9126:2006, puesto que se ha dejado claro que la mantenibilidad es una propiedad indispensable para obtener software de calidad.

7. Bibliografía

[1] "COPS: Un entorno para facilitar la programación en Java a personas con diversidad funcional", Tecnologías E-Learning Accesibles, 2012 [En Línea]. Disponible en: https://goo.gl/gb4Mih [Accedido: 27 de abril de 2017]

[2] R.S Pressman, Ph.D. "Mantenimiento y reingeniería" en Ingeniería del software, un enfoque práctico, Ed. McGraw-Hill México D.F., 2010 pp. 655-673.

[3] "Java", Oracle [En Línea]. Disponible en: https://goo.gl/DvnslE [Accedido: 2 de mayo de 2017]

[4] "Eclipse: About us" Eclipse Fundation [En Línea]. Disponible en: https://goo.gl/dH9iLV [Accedido: 2 de mayo de 2017]

[5] M. Piatiini, J. Villalba, F. Ruiz, I. Fernandez, M. Polo, T. Bastanuchy, M.A. Martínez, "Mantenimiento del software: Conceptos, métodos, herramientas y outsourcing" Ed. Ra-ma 1998.

[6] K. Erdil, E.Finn K. Keating, J. Meattle, S.Park, D.Yoon. "Software Maintenance As Part of the Software Life Cycle" [En Línea] Disponible en: https://goo.gl/j3Ui3c [Accedido: 20 de mayo de 2017]

[7] "PMD", PMD Github 2017 [En Línea]. Disponible en: https://goo.gl/zmwbe4 [Accedido: 22 de mayo de 2017]

[8] "PMD Documentation", Sourceforce [En línea]. Disponible en: https://goo.gl/b3CMw3 [Accedido: 23 de mayo de 2017]

[9] A. Hristov "Manual de estilo de programación; Rev 9.01" Ed: Planetalia 2007. Capitulo 9.2 https://goo.gl/lnErfl [Accedido: 5 de junio de 2017]

[10] Asignatura Mantenimiento del Software. Tema 8 "Guias de estilo y Javadoc" Curso 2016 – 2017. Universidad Politécnica de Valencia.

[11] Asignatura Mantenimiento del Software. Tema 4 "Bugzilla" Curso 2016 – 2017. Universidad Politécnica de Valencia.

[12] "Bugzilla", [En Línea] Disponible en: https://goo.gl/m6gpNb [Accedido: 14 de junio 2017]

[13] ISO/IEC 14764 "Software Engineering — Software Life Cycle Processes — Maintenance" [En Línea] Disponible en: https://goo.gl/lBgCvv

[14] Asignatura Mantenimiento del Software. Tema 6 “El estándar ISO/IEC 14764” Curso 2016 – 2017. Universidad Politécnica de Valencia.

[15] Aenor, “UNE 189802” Julio 2009

[16] “Git” [En Línea]. Disponible en: https://goo.gl/wjqjya [Accedido: 17 de junio 2017]

[17] “Git”. Wikipedia [En línea]. Disponible en: https://goo.gl/2Uj3KK [Accedido: 17 de junio 2017]

[18] “Tutorial EGIT” Eclipse Source [En Línea]. Disponible en: https://goo.gl/iHqKRx [Accedido: 19 de junio 2017]

[19] Asignatura Mantenimiento del software. Tema 5 “Ingeniería Inversa y otros” Curso 2016 - 2017 Universidad Politécnica de Valencia.

[20] M. Lujan-Mares, V. Tamarit, V. Alabau, C. Martínez-Hinarejos, M. Pastor, A. Sanchis, and A. Toselli (2008). iATROS: A speech and handwritting recognition system. V Jornadas en Tecnologías del Habla (VJTH’2008), pp 75–78.

[21] Asignatura Mantenimiento del software. Tema 9 “Mantenibilidad de software” Curso 2016 - 2017 Universidad Politécnica de Valencia.

Anexo A: Javadoc de COPS

Debido a la longitud y al formato en que se genera el Javadoc se incluirá en un fichero aparte.

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València