Mantenimiento y Evolución de Sistemas de Información

Ricardo Pérez del Castillo • Francisco Ruiz González
Ignacio García Rodríguez de Guzmán • Macario Polo Usaola
Mario G. Piattini Velthuis

Ra-Ma®

Mantenimiento y Evolución de Sistemas de Información

Mantenimiento y Evolución de Sistemas de Información

Ricardo Pérez del Castillo

Ignacio García Rodríguez de Guzmán

Francisco Ruiz González

Macario Polo Usaola

Mario G. Piattini Velthuis

Ra-Ma®

La ley prohíbe
fotocopiar este libro

Mantenimiento y Evolución de Sistemas de Información

Editado por:
RA-MA Editorial
Calle Jarama, 3A, Polígono Industrial Igarsa
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: *editorial@ra-ma.com*
Internet: *www.ra-ma.es* y *www.ra-ma.com*
ISBN: 978-84-9964-759-3
Depósito legal: M-28954-2018
Maquetación: Antonio García Tomé
Diseño de portada: Antonio García Tomé
Filmación e impresión: Safekat
Impreso en España en septiembre de 2018

A esposa e hijos.
A Nacho y Mario quienes me introdujeron
en el mundo del mantenimiento software

Ricardo Pérez del Castillo

A Juan Manuel Jiménez,
de quien tanto aprendí en tan poco tiempo.
A Ricardo Pérez, por tu calidad como persona.

Ignacio García Rodríguez de Guzmán

A mis cuatro chicas:
Prado, Irene, Violeta y Gala

Francisco Ruiz

A Cristina, Marta, Carmen y Nuria

Macario Polo Usaola

A Félix Navarro, por su capacidad de
evolucionar tecnológica y personalmente,
con admiración

Mario G. Piattini Velthuis

ÍNDICE

AUTORES

RICARDO PÉREZ DEL CASTILLO

Doctor por la Universidad de Castilla-La Mancha, en la que también obtuvo los títulos de Ingeniero en Informática e Ingeniero Técnico en Informática de Gestión. Es miembro del grupo de investigación Alarcos especializado en sistemas de información, bases de datos e ingeniería del software. Sus temas de investigación incluyen la arqueología de procesos de negocio y la refactorización de modelos de procesos de negocio, así como otras técnicas de ingeniería inversa aplicadas al mantenimiento de software. Además, ha colaborado en proyectos con universidades y centros de investigación extranjeros como las Universidades de Innsbruck (Austria) y Bari (Italia) y nacionales como la Complutense de Madrid y la Universidad de La Laguna. Por otra parte, su labor investigadora y docente la ha compatibilizado con trabajos profesionales en la industria de la consultoría trabajando para Itestra GmbH (Alemania) y Deloitte (España), donde ha podido realizar trasferencia tecnológica y trabajar en grandes proyectos de desarrollo y mantenimiento.

IGNACIO GARCÍA RODRÍGUEZ DE GUZMÁN

Doctor por la Universidad de Castilla-La Mancha, en la que también realizó sus estudios de Ingeniero en Informática e Ingeniero Técnico en Informática de Sistemas. Ha sido Profesor en la Universidad Rey Juan Carlos de Madrid. Actualmente es Profesor Titular de Universidad en la Escuela Superior de Informática de Ciudad Real de la misma Universidad y Director del Instituto de Tecnologías y Sistemas de Información (ITSI) de la UCLM. Es miembro del grupo de investigación Alarcos

especializado en sistemas de información, bases de datos e ingeniería del software. Sus temas de interés giran en torno a la reingeniería del software, modernización del software, arquitectura dirigida por modelos y los procesos de negocio. En relación a estos temas, ha escrito varios artículos en revistas y conferencias nacionales e internacionales.

FRANCISCO RUIZ GONZÁLEZ

Doctor Ingeniero en Informática y Licenciado en Ciencias Químicas. Desde 1984 ha desarrollado actividad profesional como analista-programador, gestor de proyectos, director TI y consultor. Director de los Servicios Informáticos de la Universidad de Castilla-La Mancha en su periodo fundacional (1985-1989). Desde 1990 es profesor universitario, incluidos más de 10 años de decano/director. Sus temas de trabajo incluyen: dirección y gobierno de TI, arquitecturas empresariales, tecnología BPM, ingeniería del software, y sistemas de información. Catedrático de Lenguajes y Sistemas Informáticos en la Escuela Superior de Informática de la UCLM en Ciudad Real, fue fundador del grupo Alarcos. Certificado PMP y miembro, entre otras asociaciones, de ACM, IEEE-CS, ATI, '*CIO Index*', '*Association of Enterprise Architects*' y SISTEDES).

MACARIO POLO USAOLA

Doctor por la Universidad de Castilla-La Mancha y licenciado en Informática por la de Sevilla. Es profesor titular de Lenguajes y Sistemas Informáticas en la Escuela Superior de Informática de la UCLM. Acreditado como catedrático de universidad. Interesado en la automatización de los procesos de la Ingeniería del Software, especialmente el testing, ha desarrollado herramientas y publicado artículos en esta línea. Es también autor de diferentes novelas, como *Fuera de ningún sitio*, *El pecador mudo* o *Si yo soy yo*.

MARIO GERARDO PIATTINI VELTHUIS

Doctor y Licenciado en Informática por la Universidad Politécnica de Madrid. Licenciado en Psicología por la Universidad Nacional de Educación a Distancia. Máster en Auditoría Informática (CENEI), Máster en Dirección de RR.HH. (IMAFE) y Master´s Certificate en Dirección de Proyectos (George Washington University). Especialista en la Aplicación de Tecnologías de la Información en la Gestión Empresarial (CEPADE-UPM). CISA (Certified Information System

Auditor), CISM (Certified Information System Manager), CRISC (Certified in Risk and Information System Control) y CGEIT (Certified in the Governance of Enterprise IT) por la ISACA. PMP (Project Management Professional) por el PMI. Diplomado en Calidad por la Asociación Española para la Calidad. Auditor Jefe ISO 15504/33000 por AENOR.

Ha trabajado como consultor para numerosos organismos y empresas, entre los que destacan: Ministerio de Industria y Energía, Ministerio de Administraciones Públicas, Siemens-Nixdorf, Unisys, Hewlett-Packard, Oracle, ICM, Atos-Ods, Avanzit, Sistemas Técnicos de Loterías, Indra/Soluziona, Alhambra/Eidos, Mundo Reader (BQ), etc. Socio fundador de las empresas Cronos Ibérica S.A. (actualmente Alten), Kybele Consulting S.L. (actualmente Intelligent Environments), Lucentia Lab, S.L., DQTeam, S.L. y AQCLab, primer laboratorio acreditado por ENAC para la evaluación de la calidad de producto software y de los datos. Ha sido profesor asociado en la Universidad Complutense y en la Universidad Carlos III de Madrid. Ha sido Director del Centro Mixto de Investigación y Desarrollo de Software UCLM-Indra, Coordinador del Área de Ciencias de la Computación y Tecnología Informática de la Agencia Nacional de Evaluación y Prospectiva (ANEP), y Director del Instituto de Tecnologías y Sistemas de Información (ITSI) de la UCLM.

Catedrático de Universidad de Lenguajes y Sistemas Informáticos en la Escuela Superior de Informática (ESI) de la Universidad de Castilla-La Mancha (UCLM), donde dirige el grupo de investigación Alarcos, especializado en Calidad de Sistemas de Información.

Entre los 15 “Top scholars in the field of systems and software engineering (2004-2008)”, Premio Nacional a la Trayectoria Profesional del Ingeniero Informático de la Federación de Asociaciones de Ingenieros Informáticos de España, y Premio Aritmel por la Sociedad Científica Informática de España (SCIE).

PREFACIO

Desde hace décadas que el mantenimiento y la evolución del software son una de las principales fuentes de preocupación de los responsables de sistemas de información, que ven cómo el presupuesto dedicado a estas tareas, en algunos casos, supone hasta el 80% del total de los recursos, imposibilitando de esta manera el desarrollo de nuevos sistemas; y cómo les resulta casi imposible adaptar el software existente a nuevos requisitos o cambios tecnológicos.

Mantenimiento proviene del latín *manu tenere* (tener en la mano), en definitiva, controlar, y evolución viene del latín *evolutio*, que implica una serie de transformaciones que tiene que experimentar el software. Sin embargo, parece que no hay nada más difícil de gestionar que la evolución del software y las aplicaciones informáticas que se encuentran actualmente en las empresas.

Actualmente, la transformación digital ha enfatizado la importancia de la gestión y evolución de los sistemas heredados (legacy). Como señala Ramírez (2017): "*El dilema es cómo evolucionar la plataforma actual e incorporar las nuevas tecnologías al ritmo adecuado desde el punto de vista de la eficiencia y experiencia del cliente. Todo un reto para los CIOs porque las expectativas en las compañías son muy elevadas*".

La presente obra recoge trabajos en este campo, resultado de numerosos proyectos realizados por el grupo Alarcos de la UCLM desde finales de los años noventa hasta la actualidad en colaboración con varias empresas: MANTEMA (Iniciativa ATYCA, Ministerio de Industria y Energía), MANTICA (Definición de un Conjunto de Métricas para la Mantenibilidad de Bases de Datos Objeto-Relacionales) (CICYT-FEDER, 1FD97-0168 TIC), MPM (Iniciativa ATYCA, Ministerio de Industria y Energía), MANTIS (Entorno para el Mantenimiento Integral del Software) (CICYT/Unión Europea, 1FD97-1608 TIC), TAMANSI (Técnicas Avanzadas

para el Mantenimiento de Sistemas de Información) (JCCM, PCB-02-001), MÁS (Mantenimiento Ágil del Software)(Ministerio de Ciencia y Tecnología, TIC 2003-02737-C02-02), ENIGMAS (ENtorno Inteligente para la Gestión del Mantenimiento Avanzado del Software) (FEDER, JCCM), COMPETISOFT (Mejora de Procesos para Fomentar la Competitividad de la Pequeña y Mediana Industria del Software de Iberoamérica)(CYTED), ESFINGE (Evolución de Software Factories mediante Ingeniería del Software Empírica)(Ministerio de Educación y Ciencia, TIN 2006-15175-C05-05), MAESTRO (Modernización de Sistemas Heredados hacia Procesos de Negocio)(Consejería de Educación y Ciencia y Fondos JCCM, HITO-2010-122), GINSENG (Green in Software Systems and Software Engineering) (Proyectos de I+D+I, del Programa Estatal de Investigación, Desarrollo e Innovación Orientada a los Retos de la Sociedad, TIN2015-70259-C2-1-R), SEQUOIA (SEcurity and QUality in PrOcesses with BIg Data and Analytics) (Proyectos de I+D+I, del Programa Estatal de Investigación, Desarrollo e Innovación Orientada a los Retos de la Sociedad, TIN2015-63502-C3-1-R), MOTERO (MOdernización de sisTEmas heredados hacia líneas de pROducto) (JCCM, PEII-2014-044-P), PISCIS (CDTI, EXP 00063762 / ITC-20131007), "An Approach Supported by Tests for Architectural Modernization of Adaptive Systems" (FAPESP, 2016/03104-0), TESTIMO ("Mejora del Proceso de Testing del Software en base a sus Tareas Manuales", JCCM, SBPLY_17_180501_000503), y SOS ("Sostenibilidad Software", JCCM).

En este libro se persiguen los siguientes objetivos:

- Presentar de forma clara y resumida los conceptos fundamentales relacionados con el mantenimiento y la evolución del software.
- Ofrecer un tratamiento sistemático de los estándares internacionales relacionados con el proceso de mantenimiento y la mantenibilidad del software.
- Analizar las técnicas y herramientas que pueden facilitar la evolución del software.
- Ofrecer metodologías para abordar el mantenimiento y la evolución del software.

A lo largo de esta obra se ha combinado el rigor académico con la experiencia práctica, proporcionando a sus lectores una panorámica actual y completa sobre la problemática asociada al mantenimiento.

CONTENIDO

La obra consta de dos partes claramente diferenciada: la primera de fundamentos en las que se presentan los conceptos, procesos, metodologías y herramientas para el mantenimiento y la evolución del software; y la segunda de temas avanzados, en las que se abordan aspectos como la modernización, arqueología de procesos de negocio, o el mantenimiento *green*.

La primera parte "FUNDAMENTOS" consta de siete capítulos. El primer capítulo introduce los conceptos básicos de mantenimiento y evolución de software, analizando sus costes, y distinguiendo entre diversos tipos de mantenimiento (correctivo, adaptativo, perfectivo y preventivo). Se analizan también las dificultades que plantea el mantenimiento y algunas posibles soluciones.

El capítulo 2 presenta una ontología sobre los conceptos que giran entorno al mantenimiento del software y la relación que guardan entre sí.

El capítulo 3 resume el estándar ISO 12207 sobre procesos de ciclo de vida software, profundizando en las actividades y tareas que se proponen para el proceso de mantenimiento; proceso que se considera como uno de los principales del ciclo de vida, junto con la adquisición, suministro, desarrollo y explotación.

El capítulo 4 propone metodologías para el mantenimiento software (MANTEMA y Ágil MANTEMA), y resume otros estudios que profundizan en aspectos metodológicos del mantenimiento software.

El capítulo 5 se dedica por completo a discutir los aspectos relacionados con la mantenibilidad del software: parámetros que influyen, atributos, propiedades, medidas, etc.

El capítulo 6 explora las métricas que se han propuesto tanto de producto como de proceso, así como para medir la calidad de un departamento de mantenimiento. También se presentan algunas métricas específicas para COBOL, lenguajes orientados a objetos y modelos de datos relacionales.

El capítulo 7 analiza el soporte que ofrecen las herramientas al proceso de mantenimiento, tanto herramientas generales como otras desarrolladas ex profeso para mantenimiento.

La segunda parte de "TEMAS AVANZADOS" consta de tres temas. El capítulo 8 introduce las técnicas de ingeniería inversa y reingeniería, analizando sus costes y beneficios. Este capítulo también presenta la modernización de software dirigida por la arquitectura, donde al proceso de reingeniería tradicional se le han sumado aspectos de la ingeniería dirigida por modelos.

El proceso de mantenimiento *green* que incorpora aspectos de sostenibilidad al proceso de mantenimiento tradicional es el objeto del capítulo 9 del libro.

El capítulo 10, a modo de solución técnica, analiza la arqueología de procesos de negocio como un método que articula la evolución de los sistemas de información considerando el conocimiento de negocio embebido en estos.

El libro finaliza con una amplia bibliografía, parte de la cual se recomienda en los diversos capítulos, así como con un apéndice de acrónimos.

ORIENTACIÓN A LOS LECTORES

Aunque un conocimiento en profundidad del problema del mantenimiento y la evolución de los sistemas de información puede estar reservado a expertos en la materia, esta obra puede dirigirse a una audiencia más amplia que comprende:

- Profesionales informáticos que estén trabajando en el área de mantenimiento de sistemas de información.
- Directivos que tengan entre sus responsabilidades la evolución del software.
- Usuarios avanzados que tengan interés en adquirir unos conocimientos sobre los conceptos y técnicas del mantenimiento y la evolución del software.

- Participantes en seminarios o cursos monográficos sobre Ingeniería del Software.
- Alumnos de Escuelas y Facultades de Informática.
- Analistas o consultores que quieran abordar esta materia de manera más sistemática.

Concretamente, y en base al perfil del lector o sus necesidades, proponemos a continuación una organización lógica del libro que permitirá al lector abordar los contenidos de una forma determinada. Así, y tal como se observa en la Figura 0.1, proponemos los siguientes perfiles de lectura:

1. Lector novel en el mantenimiento que desea obtener un buen conocimiento sobre el tema; podría ser el caso del profesional que va a comenzar a desempeñar su labor en el área de mantenimiento, o del alumno de alguna carrera relacionada con la Informática.
2. Lector con perfil profesional y amplia experiencia, que ya está familiarizado con los aspectos y la práctica del mantenimiento, o investigador en este tema. Estos capítulos presentan las aplicaciones más recientes de las técnicas derivadas del mantenimiento, como puede ser la identificación y refactorización de procesos de negocio, mantenimiento aplicado a la mejora de la sostenibilidad del software, o técnicas avanzadas de reingeniería y modernización del software.
3. Este tercer perfil está pensado para profesionales de la industria del software que, sin necesitar conocimientos demasiado prácticos del mantenimiento, sí necesitan en cierto modo supervisar, auditar o evaluar procesos de mantenimiento en su organización.

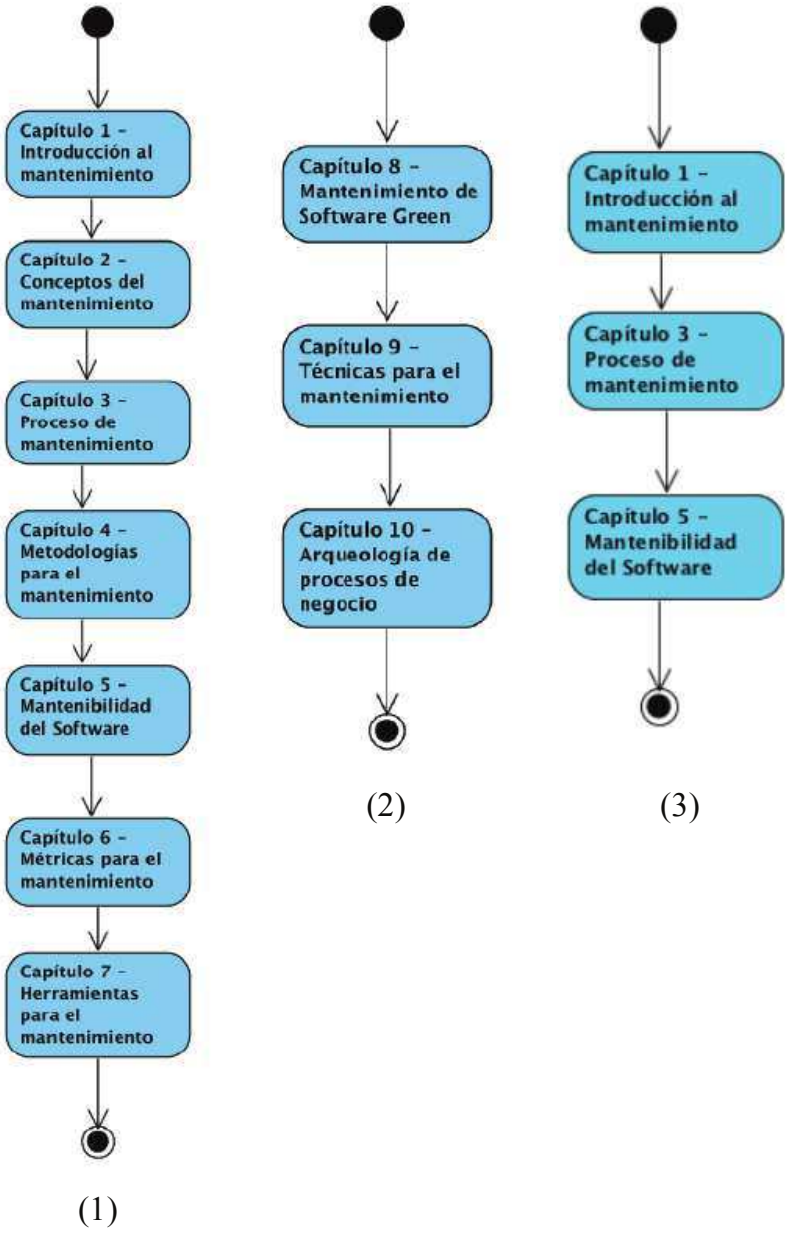


Figura 01. Propuestas de lectura del libro según el perfil del lector

OTRAS OBRAS RELACIONADAS

Queremos destacar que existen algunos libros que complementan la visión de la presente obra:

- *Calidad de Sistemas de Información 4º edición.* Piattini, M., García, F., García, I. y Pino, F.J., 2018, Madrid, Ra-Ma, que recoge las principales técnicas para la mejora de la calidad de los procesos, productos y servicios relacionados con los sistemas de información.
- *Gobierno y Gestión de las Tecnologías y los Sistemas de Información*. Piattini, M. y Ruiz, F., 2018, Madrid, Ra-Ma, en el que se tratan estándares y metodologías par llevar a cabo el gobierno y gestión de los sistemas de información, optimizando los beneficios y minimizando los riesgos.
- *Calidad del Software*. Rodríguez, M. y Piattini, M., 2018, Madrid, Ra-Ma, en el que se profundiza en los modelos de calidad y métricas para el producto software.

AGRADECIMIENTOS

Querríamos expresar nuestro agradecimiento, en primer lugar, a los alumnos de las asignaturas *Ingeniería del Software I y II, Procesos de Ingeniería del Software, Ingeniería de Requisitos y Calidad de Sistemas de Información* de la Escuela Superior de Informática de Ciudad Real, así como a los asistentes a los diferentes seminarios y conferencias que hemos organizado sobre diferentes aspectos del mantenimiento y evolución de los sistemas informáticos en los más de veinticinco años que llevamos trabajando sobre el tema. También a los compañeros del Grupo Alarcos que han participado con nosotros en numerosos proyectos sobre este tema.

El trabajo ha sido adicionalmente soportado por la ayuda que el Dr. Ricardo Pérez-Castillo disfruta de una ayuda de la JCCM, con número de expediente SBPLY/16/180501/000397 dentro de las iniciativas de retención y retorno de talento en línea con los objetivos RIS3.

A todos los profesionales de las empresas y organizaciones con las que hemos trabajado en proyectos de investigación y de aplicación práctica en este tema, queríamos agradecerles las experiencias que nos han transmitido, y su interés por las soluciones que hemos venido proponiendo y aplicando en todos estos años.

También deseamos dar las gracias a Sandra Ramírez por sus valiosas sugerencias que, como en otras muchas ocasiones, han contribuido a mejorar considerablemente este libro, y a la editorial Ra-Ma, especialmente a Jesús Ramírez Martín, Jesús Ramírez Galán y Julio Santoro Sánchez, por su apoyo y confianza.

Los autores
Ciudad Real, agosto 2018.

PARTE I.

FUNDAMENTOS

1

INTRODUCCIÓN AL MANTENIMIENTO

Frente a la considerable velocidad con que se ha desarrollado el hardware, el desarrollo del software ha sufrido un retraso histórico en cuanto a la elaboración y disposición de un cuerpo de doctrina tecnológico (metodologías y herramientas) y científico (modelos o teorías en los que basar lo anterior).

Una de las principales causas de esta situación ha sido la poca importancia que se le ha dado al mantenimiento y a la evolución del software desde todos los colectivos afectados (gestores de empresas, responsables de centros de proceso de datos, informáticos y usuarios). En este tema realizaremos una introducción a los conceptos básicos de esta área, sus características, importancia (especialmente en términos económicos), problemática y soluciones.

1.1 CONCEPTOS GENERALES

En esta sección se introduce una definición de mantenimiento y los cuatro tipos de mantenimiento que se puede identificar: correctivo, adaptativo, perfectivo y preventivo.

1.1.1 Definición de mantenimiento

El estándar IEEE 1219 [IEEE, 1998] define el Mantenimiento del Software como «la modificación de un producto software después de haber sido entregado (a los usuarios o clientes) con el fin de corregir defectos, mejorar el rendimiento u otros atributos, o adaptarlo a un cambio en el entorno».

En el estándar ISO 12207, de Procesos del Ciclo de Vida del Software [ISO/IEC, 2017] se establece que el propósito del proceso de mantenimiento es conservar la capacidad del sistema de proporcionar un servicio

En otras fuentes bibliográficas clásicas aparecen definiciones similares a las anteriores. Por ejemplo, Pressman [2014] dice que «la fase de mantenimiento se centra en el cambio que va asociado a la corrección de errores, a las adaptaciones requeridas a medida que evoluciona el entorno del software, y a cambios debidos a las mejoras producidas por los requisitos cambiantes del cliente».

En las anteriores definiciones de mantenimiento aparecen indicados, directa o indirectamente, cuatro tipos de mantenimiento: correctivo, adaptativo, perfectivo y preventivo, que suelen ser los referenciados en la bibliografía [Ruiz *et al.*, 1999]. Un resumen del papel que representa cada tipo de mantenimiento aparece en la Figura 1.1. Puede observarse que, mientras que el cambio tecnológico afecta indirectamente a los sistemas software, el entorno de trabajo y los usuarios lo hacen directamente [Hybertson *et al.*, 1997], generando peticiones de mantenimiento adaptativo y perfectivo respectivamente.

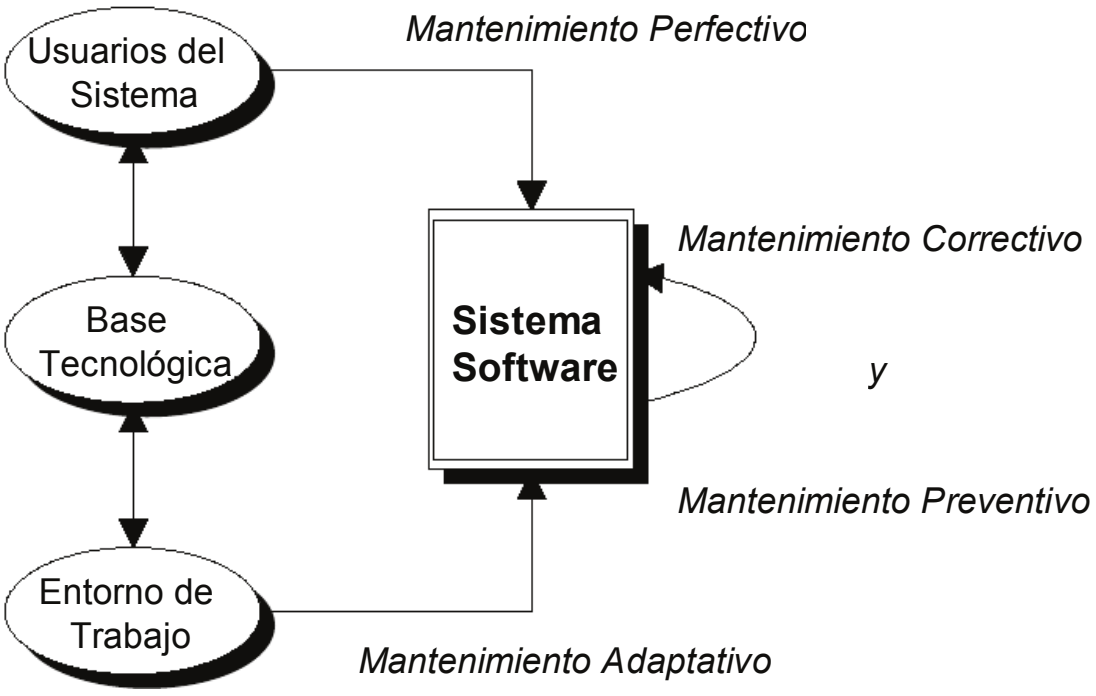


Figura 1.1. Las fuentes del mantenimiento del software

1.1.2 Mantenimiento correctivo

A pesar de las pruebas y verificaciones que aparecen en etapas anteriores del ciclo de vida del software, los programas pueden tener defectos. El mantenimiento correctivo tiene por objetivo localizar y eliminar los posibles defectos de los programas. Un defecto es una característica del sistema con el potencial de causar un fallo. Un fallo ocurre cuando el comportamiento de un sistema es diferente del establecido en la especificación. Entre otros, los fallos en el software pueden ser de:

- Procesamiento, por ejemplo, salidas incorrectas de un programa.
- Rendimiento, por ejemplo, tiempo de respuesta demasiado alto en una búsqueda de información.
- Programación, por ejemplo, inconsistencias en el diseño de un programa.
- Documentación, por ejemplo, inconsistencias entre la funcionalidad de un programa y el manual de usuario.

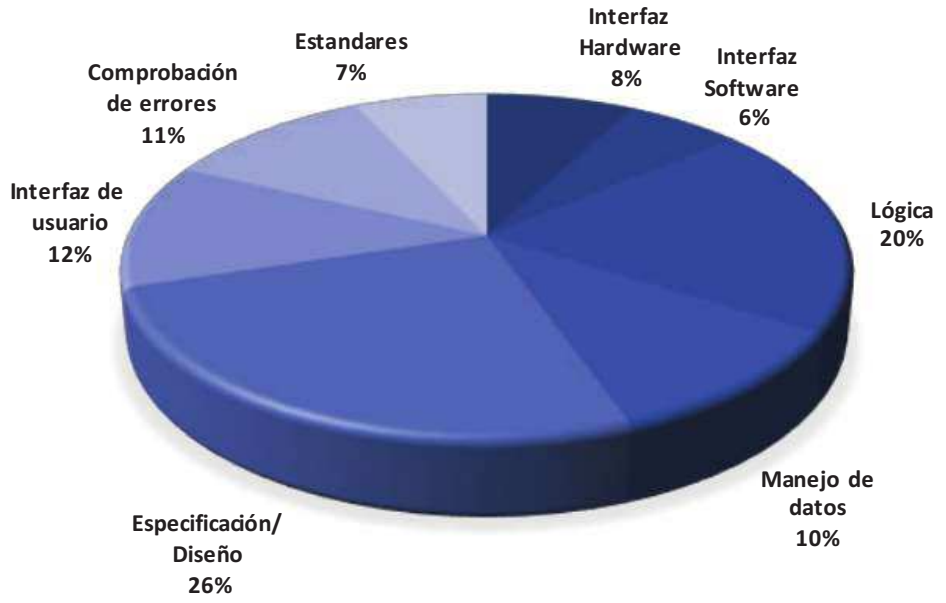


Figura 1.2. Origen de los defectos del software

En la Figura 1.2 se muestra una distribución de las causas de los defectos en un estudio clásico realizado por Grady [1994], según el cual el 37,4% de los defectos se origina en la fase de especificación de requisitos, el 25,5% en la fase de diseño y el 36,3% en la fase de codificación.

1.1.3 Mantenimiento adaptativo

Este tipo de mantenimiento consiste en la modificación de un programa debido a cambios en el entorno (hardware o software) en el cual se ejecuta. Estos cambios pueden afectar al sistema operativo (cambio a uno más moderno), a la arquitectura física del sistema informático (paso de una arquitectura de red de área local a Internet/Intranet o a la nube) o al entorno de desarrollo del software (incorporación de nuevos elementos o herramientas). La envergadura del cambio necesario puede ser muy diferente: desde un pequeño retoque en la estructura de un módulo hasta tener que reescribir prácticamente todo el programa para su ejecución en un ambiente distribuido en una red.

Los cambios en el entorno software pueden ser de dos clases:

- En el entorno de los datos, por ejemplo, al dejar de trabajar con un sistema de gestión de bases de datos relacionales y sustituirlo por una base de datos NoSQL.
- En el entorno de los procesos, por ejemplo, migrando a una nueva plataforma de desarrollo con componentes distribuidos, Java, ActiveX, etc.

El mantenimiento adaptativo es cada vez más usual debido principalmente al cambio, cada vez más rápido, en los diversos aspectos de la informática: nuevas generaciones de hardware, nuevos sistemas operativos —o versiones de los antiguos, y mejoras en los periféricos o en otros elementos del sistema. Frente a esto, la vida útil de un sistema software puede superar fácilmente los veinte años.

1.1.4 Mantenimiento perfectivo

Cambios en la especificación, normalmente debidos a cambios en los requisitos de un producto software, implican un nuevo tipo de mantenimiento llamado perfectivo. La casuística es muy variada. Desde algo tan simple como cambiar el formato de impresión de un informe, hasta la incorporación de un nuevo módulo aplicativo. Podemos definir el mantenimiento perfectivo como el conjunto de actividades para mejorar o añadir nuevas funcionalidades requeridas por el usuario.

Algunos autores dividen este tipo de mantenimiento en dos:

- Mantenimiento de Ampliación: orientado a la incorporación de nuevas funcionalidades o soportar mayor carga de trabajo (por ejemplo, por un aumento de usuarios).

- Mantenimiento de Eficiencia: que busca la mejora de la eficiencia de ejecución.

Este tipo de mantenimiento aumenta cuando un producto software tiene éxito comercial y es utilizado por muchos usuarios, ya que cuanto más se utiliza un software, más peticiones de los usuarios se reciben demandando nuevas funcionalidades o mejoras en las existentes.

1.1.5 Mantenimiento preventivo

Este último tipo de mantenimiento consiste en la modificación del software para mejorar sus propiedades (por ejemplo, aumentando su calidad y/o su *mantenibilidad*) sin alterar sus especificaciones funcionales. Por ejemplo, se puede reestructurar los programas para mejorar su legibilidad, o bien incluir nuevos comentarios que faciliten la posterior comprensión del programa. Este tipo de mantenimiento es el que más partido saca de las técnicas de *ingeniería inversa* y *reingeniería* (véase capítulo 9).

En algunos casos se ha planteado el *Mantenimiento para la Reutilización*, consistente en modificar el software (buscando y modificando componentes para incluirlos en bibliotecas) para que sea más fácilmente reutilizable. En realidad, este tipo de mantenimiento es preventivo, especializado en mejorar la propiedad de *reusabilidad* del software.

1.2 ACTIVIDADES DE MANTENIMIENTO

El desconocimiento de las actividades que implica el mantenimiento del software puede inducir a minusvalorar su importancia, y se tiende a asociar el mantenimiento del software con la corrección de errores en los programas. Por esta causa, la impresión más generalizada entre los gestores, usuarios, e incluso entre los propios ingenieros informáticos, es que la mayor parte del mantenimiento que se realiza en el mundo es de tipo correctivo. Sin embargo, varios autores ([McKee, 1984], [Frazer, 1992], [Basili *et al.*, 1996]) indican que esta impresión es equivocada, mostrando cómo los mayores porcentajes de esfuerzo se dedican a mantenimiento perfectivo (véase Figura 1.3, tomada de Frazer [1992]).

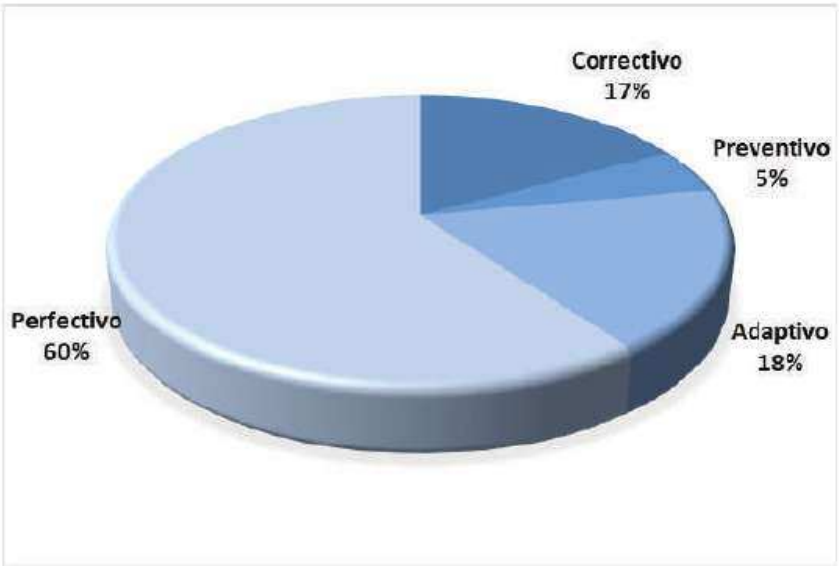


Figura 1.3. Costes relativos de cada tipo de mantenimiento

El establecimiento de analogías entre el mantenimiento del software y el mantenimiento del hardware puede conducir a confusión, ya que el software, a diferencia del hardware, no se desgasta y, por tanto, la principal actividad asociada con el mantenimiento del hardware —reemplazar o reparar las piezas estropeadas o defectuosas— no es aplicable al software [Bardou, 1997].

Algunos autores ([McClure, 1992], [Bennet *et al.*, 1991], [Harjani y Queille, 1992], [Briand *et al.*, 1998]) han identificado diferentes tipos de actividades que se realizan en cada modificación del software. Basili *et al.* [1996] identifican las siguientes once actividades, que se realizan con cada modificación del software:

- Análisis de impacto y de costes/beneficios: se dedica esta actividad a analizar diferentes alternativas de implementación y/o a comprobar su impacto en la planificación, coste y facilidad de operación.
- Comprensión del cambio: puede consistir en localizar el error y determinar su causa, o en comprender los requisitos de una mejora solicitada.
- Diseño del cambio: se refiere al diseño propuesto para el cambio, pudiéndose incluir un rediseño del sistema.
- Codificación y pruebas unitarias: se codifica y prueba el funcionamiento de cada componente modificado.

- Inspección, certificación y consultoría: esta actividad se dedica a inspeccionar el cambio, comprobar otros diseños, reuniones de inspección, etc.
- Pruebas de integración: se refiere a comprobar la integración de los componentes modificados con el resto del sistema.
- Pruebas de aceptación: en esta actividad, el usuario comprueba, junto al personal encargado del mantenimiento, la adecuación del cambio a sus necesidades.
- Pruebas de regresión: en esta actividad se somete el software modificado a casos de pruebas previamente almacenados y por los que ya pasó.
- Documentación del sistema: se revisa y reescribe, en caso necesario, la documentación del sistema para que se ajuste al producto software ya modificado.
- Otra documentación (del usuario, por ejemplo): se revisa y reescribe, en caso necesario, los diferentes manuales de usuario y otra documentación, excepto la documentación del sistema.
- Otras actividades, como las dedicadas a la gestión del proyecto de mantenimiento.

Estos autores controlaron el esfuerzo dedicado a cada una de estas actividades en cinco proyectos diferentes de un sistema de control de satélites de la NASA. Con el fin de mostrar más claramente la distribución del esfuerzo durante las modificaciones, clasifican las actividades anteriores en cinco grupos. En la Tabla 1.1 reproducimos la distribución media de esfuerzo de los cinco grupos.

Otros resultados interesantes de este mismo estudio se refieren a la distribución del esfuerzo en cada grupo de actividades según se trate de mantenimiento correctivo o perfectivo, resultando que:

- La proporción de esfuerzo dedicado a comprensión es mucho mayor en el caso de mantenimiento correctivo que en el de perfectivo.
- La proporción de esfuerzo empleado en inspección, certificación y consultoría es mucho mayor en el caso de mantenimiento perfectivo que en el de correctivo.
- La proporción de esfuerzo dedicado a diseño, codificación y pruebas es muy similar en ambos tipos de mantenimiento.

Grupo	Actividades	Porcentaje de esfuerzo
Análisis y comprensión	Análisis de impacto y de costes/beneficios Comprensión del cambio	13%
Diseño	Diseño del cambio 50% de inspección, certificación y consultoría	16%
Implementación	Codificación y pruebas unitarias 50% de inspección, certificación y consultoría	29%
Pruebas	Pruebas de integración Pruebas de aceptación Pruebas de regresión	24%
Otras	Documentación del sistema Otra documentación Otras actividades	18%

Tabla 1.1. Distribución del esfuerzo en las actividades de mantenimiento

1.3 COSTES DEL MANTENIMIENTO

Múltiples estudios señalan que el mantenimiento es la parte más costosa del ciclo de vida del software. Está comprobado que el coste de mantenimiento de un producto software a lo largo de toda su vida útil supone más del doble que los costes de su desarrollo [Schach, 1992].

Según Singer [1998], los programadores pasan el 61% de su vida profesional realizando trabajos de mantenimiento, y sólo un 39% nuevos desarrollos. Algunos autores [Frazer, 1992] estiman que la situación puede llegar a ser casi insostenible, ya que existen empresas que se acercan a porcentajes del 95% de los recursos dedicados al mantenimiento, con lo cual se hace imposible el desarrollo de nuevos productos software. Esta situación se conoce como *Barrera de Mantenimiento*. En general, el porcentaje de recursos necesarios para mantenimiento se incrementa a medida que se produce más software [Hanna, 1993] y la producción de éste ha tenido, desde sus inicios, una tendencia siempre creciente.

Por otra parte, el famoso «*Efecto 2000*» confirmó todavía más la importancia económica y social del mantenimiento del software. Son ampliamente conocidas las grandes inversiones y esfuerzos que prácticamente todas las empresas y administraciones públicas realizaron en este tema. En el caso de la Unión Europea, se unió también otra segunda situación de necesidad de mantenimiento de software a gran escala: la adaptación a la implantación del *Euro* como moneda única. Ambos

casos son situaciones caracterizadas del porque debió realizarse el mantenimiento de miles de aplicaciones software, con decenas de millones de líneas de código.

Son varias las causas de que en la mayoría de las organizaciones se requiera mucho trabajo de mantenimiento [Osborne y Chikofsky, 1990]. En primer lugar, una gran cantidad del software que existe actualmente ha sido desarrollado hace más de 20 o 30 años. Aunque estos programas fuesen creados utilizando las mejores técnicas de diseño y codificación existentes en su momento (y la mayoría no lo fueron), se construyeron con restricciones de tamaño y espacio de almacenamiento y se desarrollaron con herramientas tecnológicamente limitadas. En segundo lugar, estos programas han sufrido una o varias migraciones a nuevas plataformas o sistemas operativos. Y, por último, han experimentado múltiples modificaciones para mejorarlos y adaptarlos a las nuevas necesidades de los usuarios. Todos estos cambios se realizaron sin tener en cuenta la arquitectura general del sistema (no se aplicaron técnicas de ingeniería inversa o reingeniería). El resultado de todo ello es la existencia de sistemas software que tienen que seguir funcionando en la actualidad con una baja calidad (diseño pobre de las estructuras de datos, mala codificación, lógica defectuosa y documentación escasa).

Una causa directa de los grandes costes del mantenimiento es que el coste relativo aproximado de reparar un defecto aumenta considerablemente en las últimas etapas del ciclo de vida del software [Boehm, 1981] de forma que la relación entre el coste de detectar y reparar un defecto en la fase de análisis de requisitos y en la fase de mantenimiento es de 1 a 100 respectivamente (véase Figura 1.4).

Algunas de las razones por las que es menos costoso detectar y corregir un error durante las etapas iniciales del ciclo de vida que durante las etapas últimas son [Schach, 1992]:

- Es más fácil cambiar la documentación (por ejemplo, los documentos de especificación o de diseño) que modificar el código.
- Un cambio durante una fase tardía puede requerir que sea modificada la documentación de todas las fases anteriores.
- Es más fácil encontrar un defecto durante la fase en la cual se ha introducido el defecto que tratar de detectar y corregir los efectos provocados por el defecto en una fase posterior.
- La causa de un defecto puede esconderse en la inexistencia o falta de actualización de los documentos de especificación o diseño.

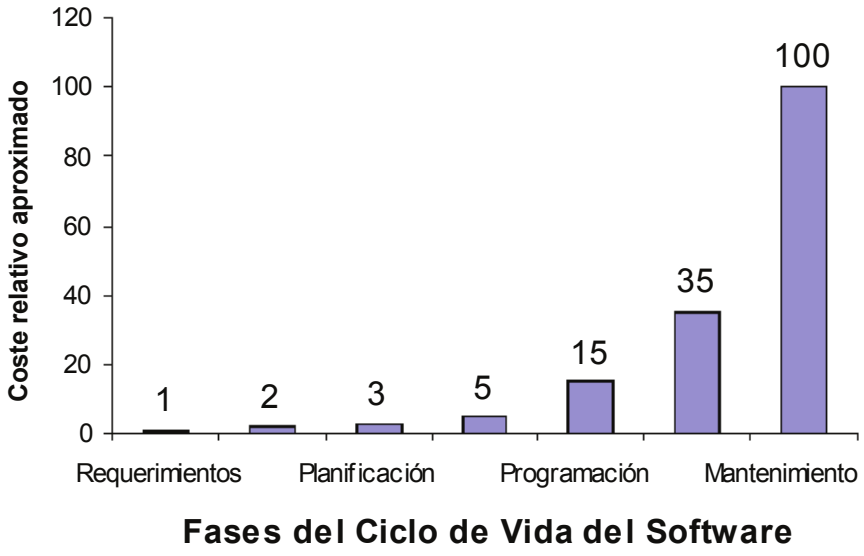


Figura 1.4. Coste relativo aproximado de detectar y corregir defectos

Cuando se planifican los costes de mantenimiento, los analistas-programadores experimentados tienen la impresión de que el mantenimiento es algo descontrolado y que nunca se sabe qué va a pasar (es algo así como predecir el futuro). Parece como si el mantenimiento del software fuese un *iceberg* del cual sólo se percibe una pequeña parte, pero bajo cuya superficie se esconde una gran cantidad de problemas potenciales y de costes encubiertos [Canning, 1972].

En la parte sumergida de este iceberg se ocultan otros costes, menos tangibles que los monetarios, pero que pueden ser causa de muchas preocupaciones. Según McCracken [1980] un coste intangible del mantenimiento del software se encuentra en las oportunidades de desarrollo que se han de posponer o que se pierden, debido a que los recursos disponibles están dedicados a las tareas de mantenimiento. Otros *costes intangibles* son los siguientes:

- Insatisfacción del cliente cuando no se puede atender en un tiempo aceptable una petición de reparación o modificación que parece razonable.
- Los errores ocultos introducidos al cambiar el software durante el mantenimiento reducen la calidad global del producto.
- Perjuicio en otros proyectos de desarrollo cuando la plantilla tiene que dejarlos, total o parcialmente, para atender peticiones de mantenimiento.

En suma, un coste final del mantenimiento del software es la reducción que se produce en la productividad de los ingenieros informáticos al iniciar el mantenimiento de aplicaciones antiguas. Algunos autores [Boehm, 1979] han calculado reducciones de la productividad —medida en LDC por persona y mes— de 40 a 1, es decir, el coste de mantener una línea de código puede llegar a ser 40 veces más alto que en el proceso de desarrollo.

1.4 DIFICULTADES DEL MANTENIMIENTO Y LA EVOLUCIÓN

La problemática del mantenimiento se resume en realizar el mantenimiento del software de forma tan rigurosa que la calidad no se deteriore como resultado de este proceso, y así facilitar la evolución del software. De esta forma [Pang y Hindle, 2016] definen el termino de *Mantenimiento Continuo* como el mantenimiento de repositorios y artefactos de forma apropiada y consistente a través de la automatización, resumen, compactación, archivación y eliminación. De esta forma se mantiene la salud no sólo del sistema bajo mantenimiento sino también de los entornos de desarrollo y los procesos relevantes.

Aunque el Mantenimiento Continuo puede ayudar a las dificultades del mantenimiento y la evolución del software, la pregunta a formular es la siguiente: ¿cómo debe mantenerse el software para preservar su fiabilidad y capacidad de evolución? A continuación, veremos las circunstancias que hacen que la respuesta a esta pregunta no sea fácil y esté ciertamente condicionada.

1.4.1 Código heredado

Con el paso de los años se ha ido produciendo un volumen muy grande de software. En la actualidad, la mayor parte de este software está formado por código antiguo «heredado» (del inglés *legacy code*); es decir, código de aplicaciones desarrolladas hace algún tiempo, con técnicas y herramientas en desuso y probablemente por personas que ya no pertenecen al colectivo responsable en este momento del mantenimiento del software concreto. En muchas ocasiones la situación se complica porque el código heredado fue objeto de múltiples actividades de mantenimiento. La opción de desechar este software y reescribirlo para adaptarlo a las nuevas necesidades tecnológicas o a los cambios en la especificación es muchas veces inadecuada por la gran carga financiera que supuso el desarrollo del software original y la necesidad económica de su amortización.

Los problemas específicos del mantenimiento de código heredado han sido caracterizados en las llamadas *Leyes del Mantenimiento del Software* [Lehman, 1980]:

- **Continuidad del Cambio** – Un programa utilizado en un entorno del mundo real debe cambiar, ya que si no cada vez será menos usado en dicho entorno. En otras palabras, esto significa que, tan pronto como un programa ha sido escrito, está ya desfasado. Las razones que conducen a esta afirmación son varias: a los usuarios se les ocurren nuevas funcionalidades cuando comienzan a utilizar el software; nuevas características en el hardware pueden permitir mejoras en dicho software; se encuentran defectos en el software que deben ser corregidos; el software debe instalarse en otro sistema operativo o máquina; o el software necesita ser más eficiente

- **Incremento de la Complejidad** – A la par que los cambios transforman los programas, su estructura se hará progresivamente más compleja salvo que se haga un esfuerzo activo para evitar este fenómeno. Esto significa que al realizar cambios en un programa (excluyendo el mantenimiento preventivo), la estructura de dicho programa se hace más compleja cuando los programadores no pueden o no quieren usar técnicas de ingeniería del software

- **Evolución del Programa** – La evolución de un programa es un proceso autorregulado. Las medidas de determinadas propiedades (tamaño, tiempo entre versiones y número de errores) revelan estadísticamente determinadas tendencias e invariantes.

- **Conservación de la Estabilidad Organizacional** – A lo largo del tiempo de vida de un programa, la carga que supone el desarrollo de dicho programa es aproximadamente constante e independiente de los recursos dedicados.

- **Conservación de la Familiaridad** – Durante todo el tiempo de vida de un sistema, el incremento en el número de cambios incluidos con cada versión (release) es aproximadamente constante.

Casi veinte años después, el mismo autor vuelve a confirmar las leyes anteriores [Lehman *et al.*, 1998]: la afirmación «los grandes programas no llegan nunca a completarse y están en constante evolución» se ve confirmada por el hecho de que, como ya hemos mencionado, algo más del 60% de las modificaciones que se realizan en el software se refieren a mantenimiento perfectivo.

1.4.2 Problemas del mantenimiento

Además de las dificultades de mantenimiento que señalan las leyes anteriores, existen otros problemas clásicos que complican el mantenimiento [Schneidewind, 1987]:

- A menudo, el mantenimiento es realizado de una manera *ad hoc* en un estilo libre establecido por el propio programador (esta opinión también es compartida por Pressman [1993], quien afirma que «raramente existen organizaciones formales, de modo que el mantenimiento se lleva a cabo *como se pueda*»). No en todas las ocasiones esta situación es debida a la falta de tiempo para producir una modificación diseñada cuidadosamente. Prácticamente todas las metodologías se han centrado en el desarrollo de nuevos sistemas y no han tenido en cuenta la importancia del mantenimiento. Por esta razón, no existen o son poco conocidos los métodos, técnicas y herramientas que proporcionan una solución global al problema del mantenimiento.

- Cambio tras cambio, los programas tienden a ser menos estructurados. Esto se manifiesta en una documentación desfasada (como afirman Baxter y Pigdeon [1997] al indicar que «la documentación completa o inexistente del sistema es uno de los cuatro problemas más importantes del mantenimiento de software»), código que no cumple los estándares, incremento en el tiempo que los programadores necesitan para entender y comprender los programas o en el incremento en los efectos secundarios producidos por los cambios. Todas estas situaciones implican casi siempre unos costes de mantenimiento del software muy altos.

- Es muy habitual que los sistemas que están siendo sometidos a mantenimiento sean cada vez más difíciles de cambiar (lo cual, como confirman Griswold y Notkin [1993], provoca que el mantenimiento sea cada vez más costoso). Esto se debe al hecho de que los cambios en un programa por actividades de mantenimiento dificultan la posterior comprensión de la funcionalidad del programa. Sommerville [1992] también apunta que «cualquier cambio conlleva la corrupción de la estructura del software y, a mayor corrupción, la estructura del programa se torna menos comprensible y más difícil de modificar». Por ejemplo, el programa original puede basarse en decisiones de programación no documentadas a las que no puede acceder el personal de mantenimiento. En estas situaciones, es normal que el software no pueda ser cambiado sin correr el riesgo de introducir efectos laterales no deseados debidos a

interdependencias entre variables y procedimientos que el personal de mantenimiento no ha detectado.

- La falta de una metodología adecuada suele conducir a que los usuarios participen poco durante el desarrollo del sistema software. Esto tiene como consecuencia que, cuando el producto se entrega a los usuarios, no satisface sus necesidades y se tienen que producir esfuerzos de mantenimiento mayores en el futuro.

- Además de los problemas de carácter técnico anteriores, también pueden existir problemas de gestión. Muchos programadores consideran el trabajo de mantenimiento como una actividad inferior —menos creativa— que les distrae del trabajo —mucho más interesante— del desarrollo de software. Esta visión puede verse reforzada por las condiciones laborales y salariales y crea una baja moral entre las personas dedicadas al mantenimiento. Como resultado de lo anterior, cuando se hace necesario realizar mantenimiento, en vez de emplear una estrategia sistemática, las correcciones tienden a ser realizadas con precipitación, sin pensarse de forma suficiente, no documentadas adecuadamente y pobremente integradas con el código existente. No es extraño, pues, que el propio mantenimiento conduzca a la introducción de nuevos errores e ineficiencias que conducen a nuevos esfuerzos de mantenimiento con posterioridad.

Todos estos problemas se pueden atribuir —parcialmente— al gran número de programas existentes que han sido desarrollados sin tener en cuenta la ingeniería del software. Esta área de la informática no es una panacea, pero, por lo menos, aporta soluciones parciales a los diversos problemas planteados con el mantenimiento del software.

1.4.3 Efectos secundarios del mantenimiento

La posibilidad de error al cambiar un procedimiento lógico tan complejo como el que constituye la mayor parte de los programas actuales es muy grande. Por esta razón, una de las principales dificultades del mantenimiento del software es el riesgo del llamado efecto bola de nieve, de manera que los cambios producidos por una petición de mantenimiento introducen *efectos secundarios* que implicarán nuevas peticiones de mantenimiento en el futuro. Estos efectos secundarios suponen nuevos defectos que aparecen como consecuencia de las modificaciones realizadas.

Según las consecuencias que se derivan, los efectos secundarios del mantenimiento del software son de tres clases [Freedman y Weinberg, 1990]:

1.4.3.1 EFECTOS SECUNDARIOS SOBRE EL CÓDIGO

Todos los desarrolladores de software han «sufrido» en algún momento de su vida profesional los problemas originados por olvidar añadir un «;» o por confundir por un simple error de mecanografía un signo de puntuación con otro. Las consecuencias de estos «despistes» pueden ser muy importantes y sirven para corroborar que los efectos secundarios por cambios en el código son difíciles de prever. Las modificaciones en el código fuente que tienen una mayor probabilidad de inducir a nuevos errores son:

- Cambios en el diseño que suponen muchos cambios en el código.
- Eliminación o modificación de un subprograma.
- Eliminación o modificación de una etiqueta.
- Eliminación o modificación de un identificador.
- Cambios para mejorar el rendimiento.
- Modificación de la apertura/cierre de ficheros.
- Modificación de operaciones lógicas.

1.4.3.2 EFECTOS SECUNDARIOS SOBRE LOS DATOS

Las estructuras de datos constituyen una parte fundamental y básica en cualquier producto software, por lo que cualquier cambio que se produzca en ellas puede conducir a fallos importantes del sistema. Los efectos secundarios de este tipo pueden aparecer debido a los siguientes cambios:

- Redefinición de constantes locales o globales.
- Modificación de los formatos de registros o archivos.
- Cambio en el tamaño de una matriz u otras estructuras similares.
- Modificación de la definición de variables globales.
- Reinicialización de indicadores de control o punteros.
- Cambios en los argumentos de los subprogramas.

Para reducir esta clase de efectos secundarios es importante realizar una correcta documentación de todos los datos, incluyendo tablas de referencias cruzadas que los asocien con los subprogramas que los utilizan.

1.4.3.3 EFECTOS SECUNDARIOS SOBRE LA DOCUMENTACIÓN

Los efectos secundarios de esta clase se producen cuando los cambios sobre el código de una aplicación no se reflejan en la documentación de diseño y/o en la documentación de usuario. Si la documentación técnica no se corresponde con el estado actual del software, se producirán efectos secundarios debidos a una incorrecta caracterización de las propiedades de dicho software. Por otro lado, la estima que los usuarios tendrán del producto software se reducirá considerablemente si comprueban que la documentación no se adapta a los ejecutables. Los cambios que con mayor probabilidad pueden producir efectos secundarios sobre la documentación son:

- Modificar el formato de las entradas interactivas.
- Nuevos mensajes de error no documentados.
- Tablas o índices no actualizados.
- Texto no actualizado correctamente.

Es muy recomendable revisar la configuración entera del software, incluyendo la documentación, para evitar estos efectos secundarios. De hecho, existen peticiones de mantenimiento que se pueden satisfacer sólo con corregir, ampliar o clarificar la documentación sin necesidad de producir cambios en los programas.

1.5 SOLUCIONES AL PROBLEMA DEL MANTENIMIENTO

Las diversas propuestas para resolver este problema pueden dividirse en dos categorías: las que proponen soluciones de gestión (organizativas) y las que proponen soluciones técnicas (metodologías y herramientas).

1.5.1 Soluciones de gestión

En términos financieros, el mantenimiento del software puede ser visto como un continuo consumidor de recursos, mientras que los beneficios no están claros ni cuantificados. Para ayudar a evitar esta situación se necesita un mayor apoyo por parte de la dirección de las organizaciones para las actividades de mantenimiento del software. Para ello es necesario que los gestores veteranos (*seniors*) de las organizaciones sean conscientes de:

- La importancia de las tecnologías de la información para la organización.
- El software es un activo corporativo que puede suponer una ventaja competitiva.

Los gestores que estén descontentos con la situación y que quieran cambiarla, tendrán que adquirir un compromiso personal y visible con las soluciones organizativas propuestas.

Recursos dedicados al mantenimiento

El recurso fundamental y clave para el mantenimiento del software es el humano. Por tanto, una manera de mejorar el mantenimiento podría ser constituir un grupo separado de programadores dedicados a mantener código antiguo [1]. Sin embargo, debido al carácter poco atractivo de este trabajo, es habitual que el personal nuevo recién incorporado sea asignado a esta actividad. Estos programadores inexpertos deben intentar comprender la lógica de diseño del sistema, a pesar de que no pueden comprender el modelo conceptual del software debido a que carecen de experiencia de uso de las técnicas de ingeniería del software y de conocimiento del dominio de lo que el programa realiza. Así, raramente saben cómo encontrar y corregir defectos o realizar modificaciones.

Gestión de la calidad

El aumento de los recursos humanos y económicos dedicados al mantenimiento del software puede suponer una solución a corto plazo, pero para resolver el problema a largo plazo se hace necesario adoptar una aproximación que permita mejorar la calidad del proceso en su conjunto. Los métodos para aumentar la calidad, tanto de un producto software como del proceso de su producción, se parecen cada vez más a los empleados en la industria en general. Entre las mejores técnicas de gestión de la calidad del software se incluyen [Daly, 1979]:

- Uso de técnicas estándares para la descomposición del software en entidades funcionales;
- Empleo estricto de estándares de documentación del software;
- Diseño paso a paso en cada nivel de descomposición del software;
- Uso de código estructurado; y
- Definición de todas las interfaces y estructuras de datos importantes antes de comenzar el diseño detallado.

---

1 Copiando a otras disciplinas de ingeniería también se ha propuesto que el mismo equipo realice ambas tareas: el desarrollo y el mantenimiento [Sneed, 1991].

Adicionalmente, pueden utilizarse métricas de productos y de procesos, y así como utilizar mejores herramientas de desarrollo de software (por ejemplo, un entorno único que integre editor, compilador y depurador).

Gestión estructurada del mantenimiento

Tal como se señala en el apartado anterior, es importante emplear una gestión estructurada y organizada del proceso de mantenimiento del software. Este *Mantenimiento Estructurado* [Pressman, 1993] aparece como resultado de la anterior aplicación de una metodología de ingeniería del software. La existencia de una adecuada *Configuración del Software* (documentación e información sobre los requerimientos, especificación, diseño y pruebas) reduce la cantidad de esfuerzo requerido en el mantenimiento y mejora la calidad general de los cambios.

Cuando el mantenimiento no es estructurado, se sufren las consecuencias de la falta de metodología: «dolorosa» evaluación del código (muchas veces poco legible), complicada comprensión del sistema por la pobre documentación interna (desconocimiento de la estructura del programa, las estructuras de datos globales, las interfaces y otros requisitos de diseño y/o rendimiento), dificultad para descubrir las consecuencias de los cambios en el código y, por último, imposibilidad de realizar pruebas de regresión (repetición de pruebas anteriores) al no existir ningún registro de pruebas.

En los casos en que no hay más remedio que mantener código heredado, las dificultades pueden atenuarse siguiendo algunas sugerencias propuestas por Yourdon [1975]:

- Prevenir antes que curar, es decir, obtener la mayor información posible sobre el programa antes de que surjan las emergencias de mantenimiento.
- Conocer y entender el flujo de control general del programa. En caso de que no exista, modelar los diagramas de estructura y de flujo de alto nivel.
- Evaluar la documentación.
- Añadir comentarios al código para facilitar su entendimiento posterior.
- Utilizar las ayudas que, habitualmente, proporcionan los compiladores: listados de referencias cruzadas, tablas de símbolos, etc.
- Al realizar cambios, respetar en la medida de lo posible el estilo y formato previos.

- Utilizar variables propias para evitar los posibles efectos secundarios que pueden surgir al utilizar las variables existentes previamente.
- Llevar un registro completo de todas las actividades de mantenimiento.
- Añadir comprobación de errores.

Antes de optar por deshacerse de un programa y reescribirlo de nuevo, es necesario hacer un estudio detallado para evaluar las ventajas e inconvenientes de una y otra opción. En cualquier caso, hay que ser conscientes de que todavía existen aplicaciones en funcionamiento cuyo código está formado por programas tipo «espagueti», mal estructurados y nada documentados, que son prácticamente imposibles de mantener.

Organización del equipo humano

Puesto que las tareas relacionadas con el mantenimiento comienzan mucho antes de que se realice la primera petición de mantenimiento, es muy aconsejable que se establezca una organización del equipo de mantenimiento, estableciendo claramente las personas que participarán en cada actividad para tratar de evitar que el mantenimiento se realice «como se pueda». Esta organización puede ser creada formalmente o simplemente constituirse de hecho, pero, en cualquier caso, se deberán establecer claramente los procedimientos de evaluación, control, supervisión e información de cada petición de mantenimiento.

Existen muchas alternativas sobre cómo organizar el equipo de mantenimiento, aunque es esencial, incluso en pequeños equipos, establecer una delegación de responsabilidades. En Polo *et al.* [1999] se distinguen las tres siguientes *organizaciones lógicas* involucradas en el proceso de mantenimiento, identificándose una serie de perfiles para cada una de ellas:

1. Cliente: es la organización propietaria del software y, por tanto, la que recibe el servicio de mantenimiento. Para esta organización se distinguen los siguientes perfiles:
   - *Solicitante*: es quien presenta las solicitudes de modificación. Establece los requerimientos necesarios para su implementación y los entrega a la organización de mantenimiento.
   - *Organización del Sistema*: es el departamento que conoce el sistema que será mantenido.
   - *Atención a Usuarios*: es el departamento que presta asistencia a los usuarios.

2. Organización de mantenimiento: es la organización que realiza el servicio de mantenimiento. En esta organización se identifican cuatro perfiles:
   - *Gestor de peticiones*: acepta o rechaza las peticiones modificación y decide el tipo de mantenimiento que debe aplicarse.
   - *Planificador*: planifica la cola de peticiones de modificación aceptadas.
   - *Equipo de Mantenimiento*: es el grupo de personas que implementa la solicitud de modificación.
   - *Responsable de Mantenimiento*: prepara la etapa de mantenimiento y establece las normas y procedimientos necesarios para llevar a cabo la metodología de mantenimiento usada.
3. Usuario: es la organización que utiliza el software objeto del mantenimiento. En esta organización sólo se identifica el perfil *Usuario*.

Dependiendo de la situación, estas tres organizaciones lógicas pueden estar constituidas por tres organizaciones distintas, o bien pueden coincidir dos o incluso las tres organizaciones reales en una sola, dependiendo del papel que desempeñe cada una en el proyecto de mantenimiento: por ejemplo, las organizaciones lógicas *Cliente* y *Usuario* podrían estar formadas por la misma empresa si ésta poseyera y utilizara el software, que es mantenido por una organización ajena. Del mismo modo, diferentes perfiles pueden coincidir en una sola persona o grupo de trabajo: el *Gestor de peticiones* puede encargarse también de planificar la cola de peticiones, con lo que realizaría también las funciones de *Planificador*.

Documentación de los cambios

En la organización del mantenimiento es muy importante realizar una correcta documentación de los cambios. Por esta razón, es conveniente que las peticiones de mantenimiento se realicen utilizando un formulario estandarizado. Así mismo, el equipo encargado del mantenimiento deberá elaborar un *informe de cambios* para cada petición de mantenimiento que deberá incluir un estudio del esfuerzo requerido para satisfacer la petición, la naturaleza de las modificaciones necesarias, y la prioridad (urgencia) del cambio.

En general, la mayoría de los autores coinciden al detallar en el conjunto de datos que deben recogerse para cada cambio ([Swanson, 1976], [Jorgensen, 1995], [Basili *et al.*, 1996], [Briand *et al.*, 1998]), aunque puedan diferir en algunos de ellas. Briand *et al.* [1998] proponen recoger la siguiente información con cada modificación, con el fin de permitir la evaluación y mejora del proceso de mantenimiento:

1. Descripción del cambio.
   - Localización
     - Subsistemas afectados.
     - Módulos afectados.
     - Entradas/salidas afectadas.
   - Tamaño
     - Líneas de código añadidas, modificadas y eliminadas.
     - Módulos examinados, añadidos, modificados y eliminados.
     - Tipo del cambio
       - Correctivo
       - Perfectivo
       - Preventivo
       - Adaptativo
2. Descripción del proceso de cambio.
   - Esfuerzo dedicado.
   - Experiencia del personal de mantenimiento.
     - Tiempo que ha pasado el personal de mantenimiento trabajando en el sistema.
     - Tiempo que ha pasado el personal de mantenimiento trabajando en el dominio de la aplicación.
   - Si el cambio generó documentación, relacionarla.
3. Descripción del problema.
   - Descripción del error.
     - Causa y origen del error (véase la referencia).
     - Momento del proceso en que se produjo el error.
   - Dificultad.
     - Causas que dificultaron la modificación.
     - Actividad más difícil relacionada con la modificación.
   - Cantidad de esfuerzo desperdiciado.
   - Decisiones que se podrían haber tomado para disminuir la dificultad de los errores.

1.5.2 Soluciones técnicas

Las soluciones técnicas al problema del mantenimiento del software son de dos clases: herramientas y métodos. Las primeras sirven para soportar de forma más efectiva y cómoda los segundos. Estas herramientas han sido diseñadas para ayudar al personal de mantenimiento a comprender el software y a probar sus modificaciones para asegurar que no han sido introducidos errores. Muchas de estas herramientas son iguales o similares a las utilizadas para la prueba (*testing*) del software: formateador, analizador estático, estructurador, documentador, depurador interactivo, generador de datos de prueba y comparador.

Los principales métodos empleados en el mantenimiento del software son la reingeniería, la ingeniería inversa, la reestructuración y la modernización:

- **Reingeniería** consiste en el examen y modificación de un sistema para reconstruirlo en una nueva forma [Bennett *et al.*, 1990]. La reingeniería como tal consta de tres fases: ingeniería inversa, reestructuración, e ingeniería directa [Arnold, 1993].

- **Ingeniería Inversa** es el proceso de analizar un sistema para identificar sus componentes y las interrelaciones que existen entre ellos, así como para crear representaciones del sistema en otra forma o en un nivel de abstracción más elevado [Chikofsky y Cross, 1990].

- **Reestructuración** del software consiste en la modificación del software para hacerlo más fácil de entender, cambiar, o menos susceptible de incluir errores en cambios posteriores [Arnold, 1986]. Un aspecto muy importante de la reestructuración es que su objetivo es mejorar la calidad (en alguna de sus dimensiones), pero nunca modificar la funcionalidad del software que se está reestructurando. La reestructuración también se conoce como *software refactoring*. El esquema básico de la reestructuración o *refactorización* se compone de dos etapas: identificación de los defectos a resolver (para lo que se pueden aplicar técnicas de identificación tan sofisticadas como sea necesario [Ghannem et al., 2015]), y aplicación de la regla o patrón de que resuelva el problema de calidad. Un caso común de defectos que se resuelve durante la reestruturación del software son los *Design Code Smells* o malos olores del software [Fowler y Beck, 1999].

- **Modernización** del software *(Architecture-Driven Modernization*[2]), se presenta como una nueva visión de la reingeniería, donde se utilizan

2 *http://adm.omg.org/*

un conjunto de modelos y estándares para llevar a cabo la reingeniería y evolución de los sistemas heredados, utilizando el paradigma MDE (*Model-Driven Engineering*), y concretamente, su instanciación más conocida, MDA (*Model-Driven* Architecture) [OMG, 2014]. Siguiendo la filosofía de la modernización del software, tanto la ingeniería inversa como la reestructuración y la ingeniería directa se llevan a cabo empleando modelos estandarizados, y concretamente, tomando como base el metamodelo KDM (*Knowledge-Discovery Metamodel*) [OMG, 2016], que es un estándar ISO/IEC [Pérez-Castillo et al., 2011d]. El metamodelo KDM presenta un conjunto de submetamodelos interrelacionados que permiten representar un sistema heredado desde distintas vistas que abarcan visiones muy cercanas al código, hasta vistas próximas al negocio.

Estas cuatro técnicas tienen relación entre ellas y a menudo se utilizan varias conjuntamente.

1.6 LECTURAS RECOMENDADAS

✓ Mens, T., Serebrenik, A. y Cleve, A. (eds.). (2014). Evolving Software Systems. Springer Heidelberg.

En este libro se ofrece una visión de los avances más recientes en la evolución del software, así como los desafíos a los que se enfrenta esta área.

1.7 SITIOS WEB RECOMENDADOS

✓ *https://alarcos.esi.uclm.es/*

En este portal se recoge información sobre los artículos, libros y proyectos realizados por el grupo Alarcos en más de veinte años sobre el tema de mantenimiento y evolución del software.

1.8 EJERCICIOS

Ejercicio 1

Responda a las siguientes cuestiones respecto del mantenimiento del software:

- ¿Cuándo se debe comenzar a organizar?
- ¿Por qué se tiende a pensar que las necesidades de mantenimiento son menores que las reales?
- ¿Quién debe realizarlo?
- ¿Cuáles son los tipos de mantenimiento?
- ¿Cuál es el tipo de mantenimiento más habitual?

Ejercicio 2

Enumere cinco razones que hacen que el mantenimiento sea necesario.

Ejercicio 3

Dé una definición lo más completa posible de mantenimiento del software.

Ejercicio 4

Indique tres causas de que el mantenimiento del software sea tan costoso económicamente.

Ejercicio 5

Enumere cinco actividades de mantenimiento. ¿Cuál de ellas es la que más tiempo consume según los estudios estadísticos realizados?

Ejercicio 6

Indique cómo describiría un error de manera que se facilite la tarea de mantenimiento.

Ejercicio 7

Comente con cuál de las leyes de Lehman está más de acuerdo y con cuál más en desacuerdo.

Ejercicio 8

Señale tres efectos secundarios del mantenimiento sobre el código y otros tres sobre los datos.

Ejercicio 9

Enumere la información que se deberá registrar para cada cambio.

Ejercicio 10

¿Cuáles son los cuatro métodos de la ingeniería del software que aportan soluciones técnicas al problema del mantenimiento?

2

ONTOLOGÍA DEL MANTENIMIENTO

El mantenimiento, como cualquier proceso del ciclo de vida del software, maneja y propone su propia terminología, que debe ser conocida, entendida y utilizada de la forma adecuada.

Dado el gran número de conceptos que se maneja en el ámbito del mantenimiento, este capítulo propone al lector una organización de los conceptos más importantes del contexto del mantenimiento, relacionados mediante una *ontología*.

2.1 VISIÓN GENERAL

Las ontologías están llamadas a jugar un papel cada vez más importante en la mejora de los procesos de negocio, permitiendo la convergencia de dos puntos de vista hasta ahora divergentes: la ingeniería del software y la gestión de empresas (en particular para el análisis de negocios). Las tecnologías orientadas al modelado suponen una ayuda importante en esta línea [Bertrand y Bezivin, 2000].

Las ontologías sirven para aclarar los conceptos y las relaciones entre ellos, por eso incluiremos en este capítulo la ontología de la gestión de proyectos de mantenimiento de software que definimos en [Ruiz et al., 2004]. Esta ontología consta de cuatro subontologías (por razones de claridad y comprensibilidad). En la Figura 2.1 se muestra un esquema de los factores de dominio considerados y los principales conceptos que abarcan. En consecuencia, las subontologías de los productos, de las actividades, de la organización del proceso y de los agentes se corresponden (aunque no al cien por cien) con los factores de dominio que afectan al Proceso de Mantenimiento Software (PMS).

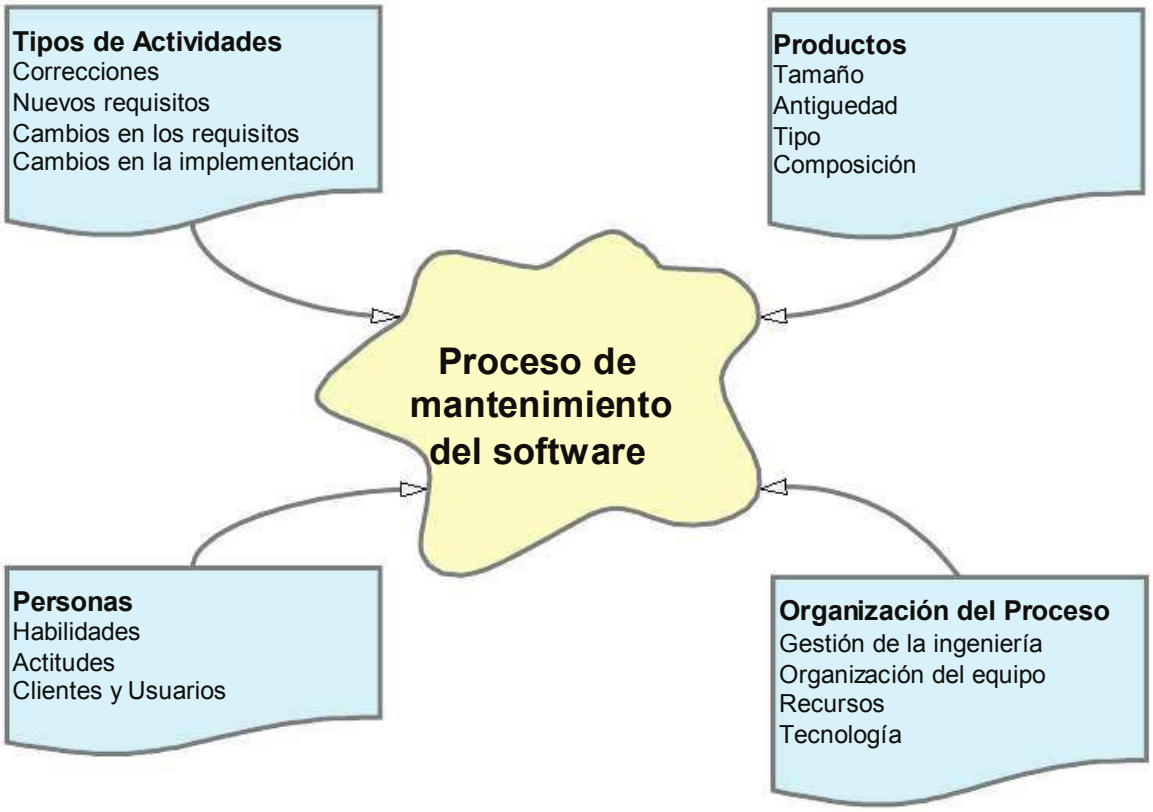


Figura 2.1. Resumen de los factores de dominio que afectan al PMS.

La integración de estas cuatro subontologías se realiza por medio del concepto de proyecto tal como lo define la "*Guide to the Project Management Body of Knowledge (PMBOK Guide)*" del *Project Management Institute* [PMI, 2000]. Así, un proyecto es "un esfuerzo temporal emprendido para crear un producto o servicio único". En esta definición, "temporal" significa que un proyecto siempre tiene un inicio y un final en el tiempo y "único" significa que el producto o servicio es diferente de alguna manera de los demás productos y servicios. Por ejemplo, trata de servicios de mantenimiento que tienen circunstancias específicas en cada proyecto: el software mantenido, las condiciones de servicio, el cliente y usuarios, etc. Un proyecto se caracteriza, principalmente, porque es:

- Realizado por personas [3];
- Está limitado por restricciones, especialmente en cuanto a recursos.
- Es planificado, ejecutado y controlado (es decir, es gestionado).

3 Con la matización de que se pueden utilizar herramientas que amplían y mejoran las capacidades de dichas personas.

Los atributos incluidos en las representaciones ontológicas que siguen son únicamente los que tienen significación especial para la gestión del PMS. Además, sólo incluimos una tabla única con todas las clases de interrelaciones (mostradas como estereotipos en los diagramas UML) para la ontología del mantenimiento.

En la Figura 2.2 se muestra el diagrama de clases UML que representa estos conceptos (con el concepto central de proyecto resaltado), junto con las siguientes consideraciones:

- Las restricciones pueden ser precondiciones, que se deben cumplir para poder empezar a ejecutar el proyecto, u objetivos, es decir, postcondiciones que se deben alcanzar para considerar que se ha completado con éxito.
- Un proyecto puede estar formado por varios sub-proyectos.
- En la gestión de un proyecto o sub-proyecto se manejan colecciones de elementos de diversos tipos: artefactos (o productos), actividades, recursos y agentes (humanos o automáticos).

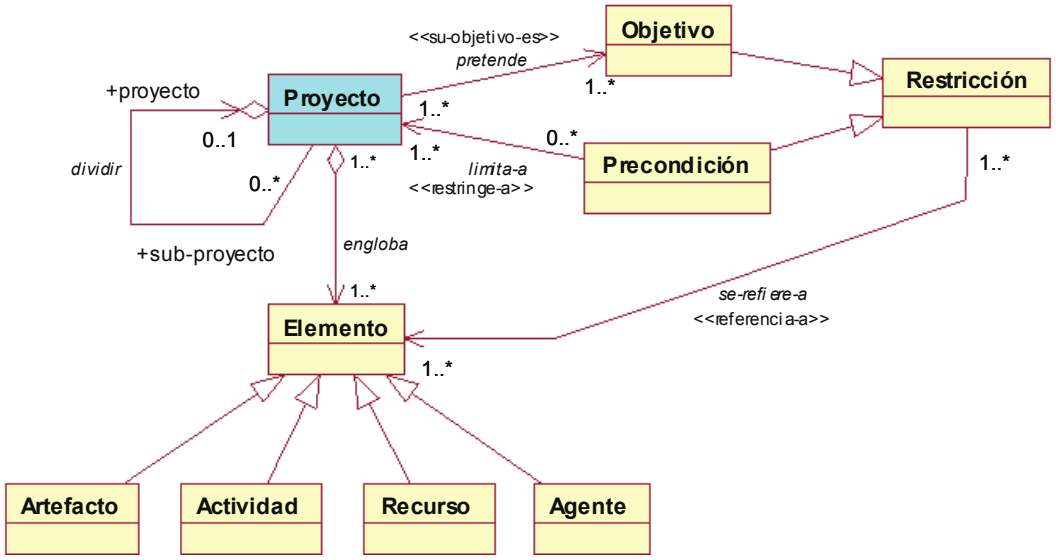


Figura 2.2. Esquema básico de la ontología del mantenimiento.

El glosario de conceptos y la tabla de interrelaciones (sin mostrar la tabla de atributos porque no hay ninguno significativo a este nivel) se muestra en la Tabla 2.1 y Tabla 2.2.

Concepto	Super-Concepto	Descripción	Propósito
Actividad	-	Ver subontología de las actividades.	-
Agente	-	Ver subontología de los agentes.	
Artefacto	-	Ver subontología de los productos.	-
Elemento	Concepto	Cada uno de los diferentes conceptos manejados en la gestión de proyectos de mantenimiento.	Manejar los diferentes tipos de elementos de un proyecto.
Objetivo	Restricción	Postcondición que debe cumplirse para considerar que un proyecto se ha completado con éxito. Sinónimo: postcondición. Ejemplo: tener construido un producto software concreto.	Gestión de los proyectos.
Precondición	Restricción	Restricción que debe cumplirse antes de que un proyecto puede comenzar a ejecutarse. Ejemplo: Tener designado un jefe de proyecto.	Gestión de los proyectos.
Proyecto	Concepto	Esfuerzo temporal emprendido para crear un producto o servicio único [PMI, 2000].	Representar los proyectos de servicios de mantenimiento que una organización lleva a cabo.
Recurso	-	Ver subontología de las actividades.	-
Restricción	Concepto	Condición de entorno que limita las opciones posibles para planificar, ejecutar y controlar un proyecto.	Gestión de los proyectos.

Tabla 2.1. Esquema general de la ontología del mantenimiento: glosario de conceptos.

Nombre	Descripción
Dividir	Un proyecto puede dividirse en varios sub-proyectos, que son realizados y gestionados de forma autónoma, aunque coordinada.
Engloba	La gestión de un proyecto de mantenimiento engloba una colección de elementos de diferentes tipos. Un elemento puede aparecer en varios proyectos (una persona participa en más de un proyecto, un recurso se utiliza en más de un proyecto).
Pretende	Un proyecto se lleva a cabo buscando satisfacer unos determinados objetivos.
Limita-a	Un proyecto no puede empezar si no se cumplen unas determinadas precondiciones.
Se-refiere-a	Una restricción se define en referencia a unos determinados elementos de un proyecto. Ejemplo: El personal disponible para el proyecto son el ingeniero “Luis ...” y el programador “Antonio ...”.

Tabla 2.2. Esquema general de la ontología del mantenimiento: tabla de interrelaciones.

2.2 SUBONTOLOGÍA DE LOS PRODUCTOS

En esta subontología se definen los productos software que son mantenidos, su estructura interna y composición y las versiones que existen de ellos. A continuación, se muestran el diagrama UML y las tablas de atributos y de interrelaciones. Aunque el concepto central es el de producto, el concepto de artefacto también es fundamental ya que es uno de los tipos de elementos que se manejan durante la gestión de proyectos de mantenimiento (Figura 2.3).

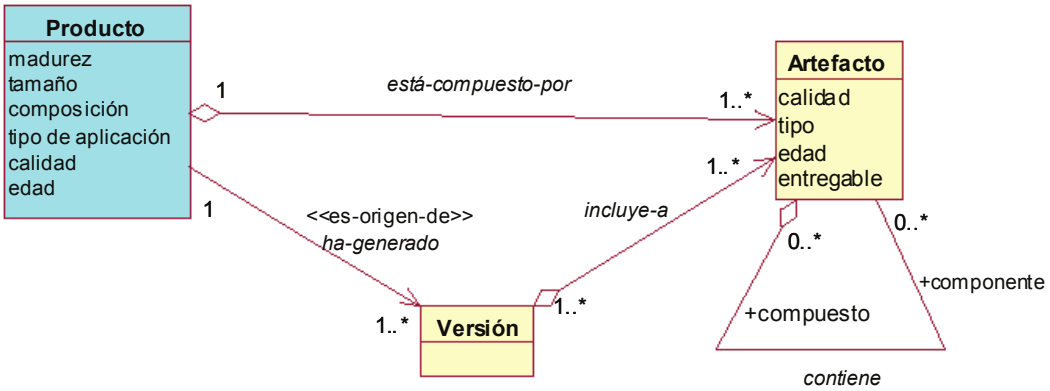


Figura 2.3. Diagrama de la subontología de los productos.

Concepto	Super-Concepto	Descripción	Propósito
Artefacto	Elemento	Parte diferenciada de un producto software que es creada o modificada por las actividades. Puede ser un documento (textual o gráfico), un componente COTS, o un módulo de código. Ejemplos: documento de especificación de requisitos, plan de calidad, módulo de clase, rutina, informe de pruebas, manual de usuario. Sinónimos: elemento software, producto de trabajo, ítem de producto.	Definir la estructura interna y composición del software.
Producto	Concepto	Aplicación software que está siendo mantenida. Es un conglomerado de diversos artefactos. Sinónimo: Software.	Mantenerlo.
Versión	Concepto	Un cambio en la línea base de un producto. Puede ser una nueva versión completa, *upgrade*, *release* o actualización, o un simple parche en el código.	Implantar el proceso de gestión de configuración.

Tabla 2.3. Subontología de los productos: glosario de conceptos.

Concepto	Atributo	Descripción	Cardinalidad
Artefacto	calidad	Medida cualitativa de la calidad, especialmente de la documentación existente sobre el artefacto.	1
	tipo	Naturaleza del artefacto. Ejemplos: documento, módulo, componente, archivo.	1..*
	edad	Número de años desde que se obtuvo la primera versión.	1
	entregable	Indica si el artefacto debe ser entregado al cliente para considerar que el proyecto ha sido completado.	1
Producto	madurez	Etapa en el ciclo de vida del producto: inicial, evolución, servicio, retirada, o cierre. La frecuencia e importancia de cada tipo de mantenimiento es diferente en cada una de estas etapas.	1
	tamaño	Medida cualitativa del tamaño. Existe una relación clara entre este atributo y la organización y tamaño del equipo de mantenimiento.	1
	composición	Nivel de abstracción de los artefactos que lo forman: componentes tipo caja negra o componentes abiertos con documentos de diseño.	1
Producto	tipo de aplicación	La productividad y tipos de mantenimiento están muy influidas por ella. Ejemplos: de gestión, científica, específica, de control, empotrada, etc.	1
	calidad	Medida cualitativa de la calidad, especialmente de la documentación.	1
	versión	Número de años desde que se obtuvo la primera versión. Afecta al mantenimiento en función de la edad del propio producto y también de la edad de la tecnología usada.	1

Tabla 2.4. Subontología de los productos: tabla de atributos.

Nombre	Descripción
Está-compuesto-por	Un producto software está compuesto por artefactos de diferentes clases.
Ha-generado	Un producto software es el origen de una o varias versiones de él a lo largo de su ciclo de vida.
Incluye-a	Una versión incluye un subconjunto de todos los artefactos que forman parte de un producto software.
Contiene	Un artefacto puede estar compuesto por otros más simples y viceversa.

Tabla 2.5. Subontología de los productos: tabla de interrelaciones.

La única interrelación de las cuatro anteriores que no es de una clase predefinida es "ha-generado". Como se observa en el diagrama UML, su estereotipo indica que es de la clase "es-origen-de".

Por tanto, en la tabla de clases de interrelaciones mostrada únicamente es necesario incluir esta clase.

2.3 SUBONTOLOGÍA DE LAS ACTIVIDADES

Esta subontología engloba dos de los cuatro tipos de elementos básicos para gestionar un proyecto de mantenimiento: actividades y recursos. Más en concreto, en ella se definen los tres aspectos siguientes:

- *Taxonomía de tipos de actividades*. Una actividad es una abstracción de "*cómo se hace el trabajo*". Mediante la jerarquía de tipos de actividades se modela el conocimiento de que existen diferentes "tipos de trabajo" a llevar a cabo.

- *Taxonomía de tipos de recursos*. Los más importantes son el hardware y el software, aunque también existen otros como locales, consumibles, etc. Se considera a las personas no como un recurso humano (a pesar de ser ésta una denominación muy común) sino agentes que participan activamente en los proyectos. Es importante distinguir entre el concepto de recurso y su materialización. Por ejemplo, un diagrama de clases puede estar materializado en forma de dibujo en papel, en un archivo PDF o en un archivo SVG.

- *Relación entre artefactos, actividades y recursos*. Esta relación está basada en el patrón "*Basic Process Structure*" (parte superior en Figura 2.4), de forma que las actividades consumen (tienen como entrada) y producen o modifican (tienen como salida) artefactos usando unos determinados recursos.

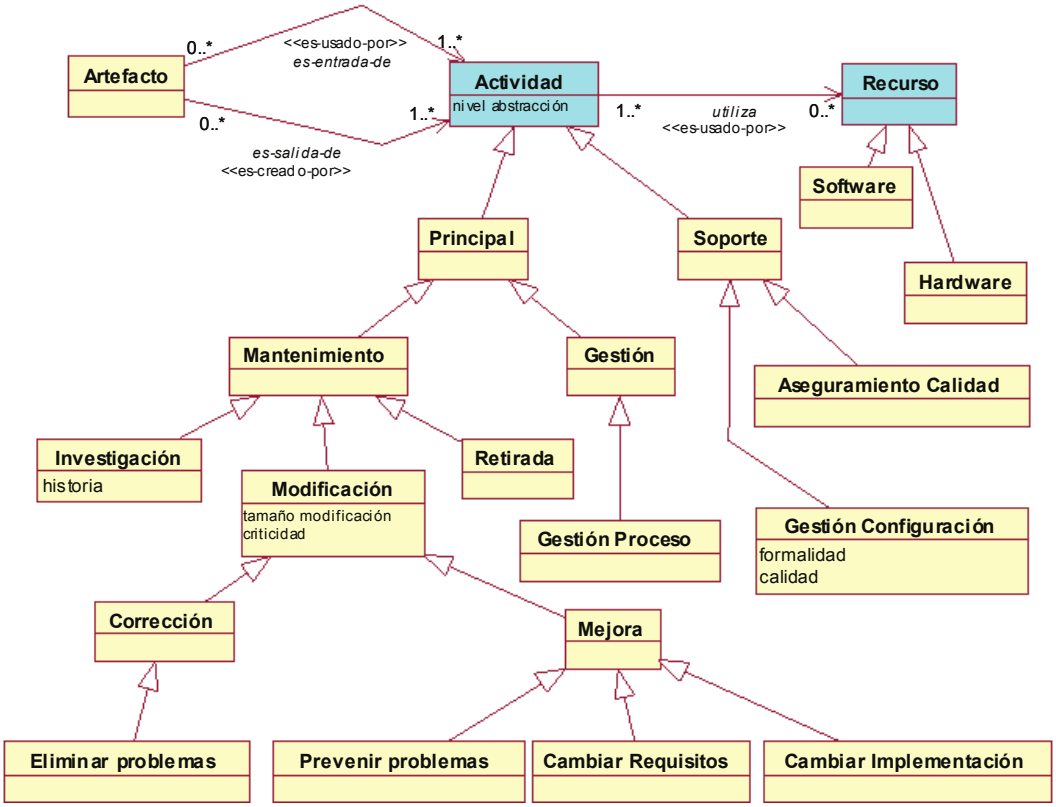


Figura 2.4. Diagrama de la subontología de las actividades.

Es importante tener en cuenta que, al referirnos a *Actividades*, la cantidad de trabajo y complejidad interna puede ser tan grande o pequeña como se desee: una categoría de procesos, un proceso, un grupo de actividades, una actividad, o incluso una sub-actividad (tarea).

En esta subontología no se han incluido todos los tipos de actividades que desempeñan un papel central en la gestión de proyectos de mantenimiento. Así, las actividades se dividen en principales y de soporte. Entre las segundas, las más significativas son las pertenecientes a los procesos de "Aseguramiento de Calidad" y de "Gestión de Configuración". Las principales se subdividen en actividades propias del proceso de mantenimiento y actividades de gestión de dicho proceso. En estas últimas destacan las actividades de gestión del proceso, que se concretan y detallan en la subontología de organización del proceso en el apartado siguiente.

Las actividades de mantenimiento se clasifican, en función del modelo de proceso del estándar ISO 14764 [ISO/IEC, 2006], en actividades de investigación, de modificación y de retirada. Las actividades de modificación pueden consistir en corregir los artefactos software o en mejorarlos. Las primeras, que se corresponden con el tipo de mantenimiento correctivo, suelen consistir en la eliminación de problemas detectados en el software. Las segundas pueden ser de tres tipos: prevenir problemas mejorando ciertas características del software (mantenimiento

preventivo); implementar cambios en los requisitos o en algunas propiedades de calidad del software (mantenimiento perfectivo); o cambiar sólo aspectos de implementación, es decir, el entorno operativo del software, sin tocar los requisitos (mantenimiento adaptativo).

El glosario de conceptos y las tablas de atributos e interrelaciones de esta subontología son los siguientes:

Concepto	Super-Concepto	Descripción	Propósito
Actividad	Elemento	Una acción que debe realizarse para lograr los objetivos del proyecto. Sinónimos: Tarea, paso. Ejemplos: análisis de petición de mantenimiento, implementación de la modificación en el código.	Describir el trabajo a realizar.
Artefacto	-	Ver subontología de los productos.	-
Aseguramiento Calidad	Soporte	Actividades de soporte realizadas para asegurar que los productos y los procesos son conformes a los requisitos y a los planes establecidos.	Garantizar la calidad del producto.
Cambiar implementación	Mejora	Actividad de mejora realiza para adaptar un producto software a cambios en su entorno de implementación sin afectar a los requisitos. Sinónimo: Mantenimiento adaptativo. Ejemplo: adaptarlo a un nuevo sistema operativo.	Dar servicio de mantenimiento.
Cambiar requisitos	Mejora	Actividad de mejora realizada para adaptar el funcionamiento de un software a cambios en los requisitos o a la inclusión de nuevos requisitos. Sinónimo: Mantenimiento perfectivo.	Dar servicio de mantenimiento.
Corrección	Modificación	Actividad de modificación que consiste en la eliminación de defectos en un producto software para que su funcionamiento se adapte a los requisitos.	Dar servicio de mantenimiento.
Eliminar problemas	Corrección	Actividad de corrección realizada para eliminar problemas detectados como defectos en el funcionamiento de un software. Sinónimo: Mantenimiento correctivo.	Dar servicio de mantenimiento.
Gestión	Principal	Actividad para gestionar el PMS incluida en el subsistema organizacional definido en el sistema de procesos.	Gestionar el PMS.
Gestión Configuración	Soporte	Actividades de soporte cuyo objetivo es establecer y mantener la integridad de los artefactos y hacerlos disponibles, mediante diferentes versiones, a los agentes del proyecto.	Garantizar la integridad de las versiones.

Gestión del Proceso	Gestión	Actividad de gestión realizada para organizar, supervisar y controlar la iniciación y realización del PMS buscando alcanzar sus objetivos.	Gestionar el PMS.
Hardware	Recurso	Recurso formado por un sistema informático, una computadora y un dispositivo periférico.	Utilizar dispositivos informáticos.
Investigación	Mantenimiento	Actividad de mantenimiento que evalúa las diversas maneras de satisfacer una petición de mantenimiento y su impacto en un producto software.	Dar servicio de mantenimiento.
Mantenimiento	Principal	Actividad incluida en el PMS según la definición de ISO/IEC.	Realizar el PMS.
Mejora	Modificación	Actividad de modificación que implementa cambios en un producto software que modifican su comportamiento o implementación o que mejora alguna de sus características de calidad.	Dar servicio de mantenimiento.
Modificación	Mantenimiento	Actividad de mantenimiento que crea o modifica uno o varios artefactos, cambiando el comportamiento o implementación de un producto.	Dar servicio de mantenimiento.
Prevenir problemas	Mejora	Actividad de mejora realizada para eliminar problemas futuros, aunque todavía no se han manifestado como defectos. Sinónimo: Mantenimiento preventivo.	Dar servicio de mantenimiento.
Principal	Actividad	Actividad perteneciente al subsistema principal o al subsistema organizacional definidos en el sistema de procesos.	Realizar y gestionar el PMS.
Recurso	Elemento	Algo de naturaleza no humana que es necesario para realizar una actividad pero que no forma parte del producto software.	Gestionar recursos.
Retirada	Mantenimiento	Actividad de mantenimiento que se realiza para concluir la vida útil de un producto software.	Concluir un servicio de mantenimiento.
Software	Recurso	Herramienta software utilizada para la automatización total o parcial de alguna(s) actividad(es).	Utilizar herramientas software.
Soporte	Actividad	Actividad cuyo objetivo es facilitar la realización de las actividades principales. Pertenece al subsistema de soporte definido en el sistema de procesos.	Dar soporte a las actividades principales.

Tabla 2.6. Subontología de las actividades: glosario de conceptos.

Concepto	Atributo	Descripción	Cardinalidad
Actividad	Nivel de abstracción	Indica el grado de abstracción de una actividad (declaración de trabajo) dentro de la jerarquía del sistema de procesos. Ejemplos: Ciclo de vida, Proceso, Subproceso, Grupo de actividades, Actividad, etc.	1
Gestión de Configuración	Calidad	Grado de cumplimiento de los objetivos del proceso de gestión de configuración. Es un factor determinante para la calidad y eficiencia de un servicio de mantenimiento.	1
	Formalidad	Indicador del nivel de definición del proceso de gestión de configuración. Ayuda a preservar la integridad y consistencia de un producto software y sus artefactos.	1
Investigación	Historia	Información sobre el estado actual y la evolución previa de la actividad de investigación. Su disponibilidad determina en gran parte la eficiencia y calidad de los mantenedores.	1
Modificación	Criticidad	Indica la rapidez con que los usuarios necesitan que la corrección o mejora sea realizada.	1
	Tamaño de la modificación	Estimación del esfuerzo necesario para realizar la modificación, en función de la cantidad de artefactos afectados y su tamaño.	1

Tabla 2.7. Subontología de las actividades: tabla de atributos.

Nombre	Descripción
Es-entrada-de	Las actividades necesitan ciertos artefactos como entradas para poder ser llevadas a cabo.
Es-salida-de	Las actividades producen (crean o modifican) artefactos.
Utiliza	Las actividades necesitan utilizar ciertos recursos para poder ser realizadas.

Tabla 2.8. Subontología de las actividades: tabla de interrelaciones.

2.4 SUBONTOLOGÍA DE ORGANIZACIÓN DEL PROCESO

Esta subontología engloba los conceptos que permiten definir la forma de llevar a cabo las actividades y la forma en que está organizado el proceso de mantenimiento propiamente dicho, incluyendo la definición de las actividades centrales que se realizan durante la planificación y ejecución de un proyecto de mantenimiento.

Una de las principales diferencias entre desarrollo y mantenimiento de software es que el primero está dirigido por los requisitos mientras que el segundo está dirigido por eventos. Esto significa que los estímulos (es decir, las entradas) que inician las actividades de mantenimiento son eventos aleatorios no planificados: la recepción de peticiones de mantenimiento (PM). Por esta razón, es fundamental que una organización de mantenimiento (mantenedor) gestione adecuadamente la cola de peticiones que le llegan. En esta subontología hemos generalizado las características comunes a las principales metodologías de mantenimiento conocidas, utilizando como eje común el modelo general del PMS propuesto por [ISO/IEC 2006].

Debido a la complejidad del diagrama global de esta subontología, hemos optado por dividir su explicación en tres subapartados, centrados en los conceptos siguientes:

- *Procedimientos* disponibles para realizar las actividades;
- Actividades de mantenimiento y de gestión que permiten *gestionar las peticiones de mantenimiento* (PM); e
- Identificación de los *problemas* y sus tipos.

2.4.1 Procedimientos

Cada actividad puede ser realizada utilizando uno o varios procedimientos, aunque no siempre se utilizan todos ellos porque suelen existir varios procedimientos alternativos para realizar el mismo trabajo (Figura 2.5). Para cada procedimiento se definen su tipo (método, técnica o guía) y las restricciones de utilización que tiene en función de los factores de desarrollo del producto software original: paradigma y tecnología utilizados (Falbo et al, 1998). También se identifican las herramientas software útiles en la automatización de cada procedimiento y los artefactos modificados (o creados) por el procedimiento. Estos últimos son todos o algunos de los definidos como salidas de las actividades asociadas. La tecnología de desarrollo es un factor decisivo a considerar en el caso de productos software con un ciclo

de vida previsto muy largo ya que habrá que elegir aquella que tenga garantías de disponibilidad al cabo de muchos años.

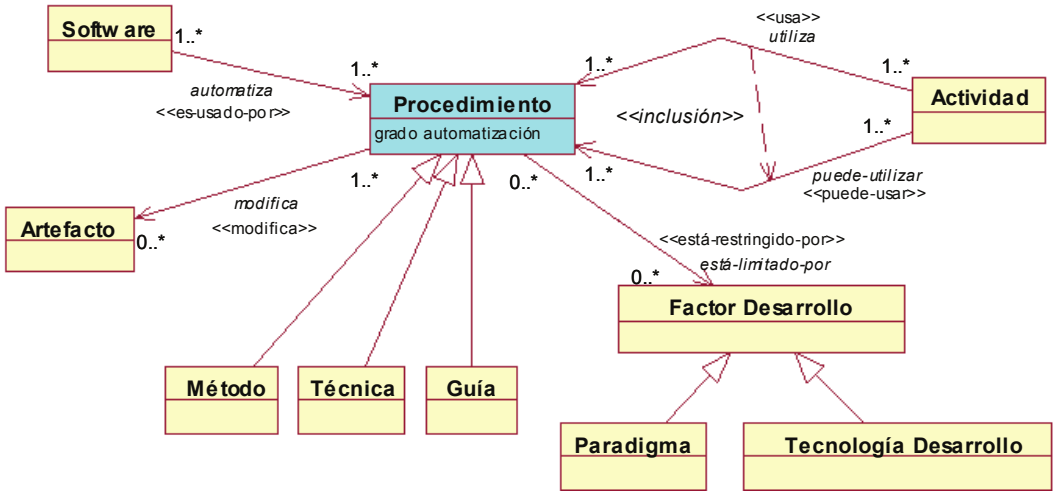


Figura 2.5. Subontología de organización del proceso: los procedimientos.

2.4.2 Gestión de Peticiones

En la Figura 2.6 se muestra la parte de la subontología que representa las actividades y artefactos centrales en la gestión del proceso de mantenimiento. La gestión del proceso consiste en una colección de actividades de gestión de peticiones de mantenimiento y de control de cambios (aunque existen otras actividades de gestión del proceso, estas son las fundamentales) que originan actividades de mantenimiento (de investigación y de modificación). Estas actividades utilizan dos clases principales de artefactos: peticiones de mantenimiento (PM) e informes de investigación. El evento que dispara estas actividades es la recepción de una PM. Según su contenido, una PM puede ser de dos tipos diferentes: la descripción de un problema detectado en el software (*informe de problema*) o una petición de un cambio justificada (*solicitud de cambio*). La actividad de "Gestión de PM" recibe la PM y la analiza en función del "Acuerdo de Nivel de Servicio" (ANS) que exista acordado con la organización cliente. En caso de que dicha PM se englobe dentro de las condiciones de servicio del ANS, una actividad de investigación produce un "informe de investigación". Este informe es recibido por la actividad de "control de cambio" que es la encargada de decidir las actividades de modificación que se aprueban (en caso de que haya alguna). La gestión del proceso también determina la estructura organizativa utilizada por el mantenedor que soporta cada producto software.

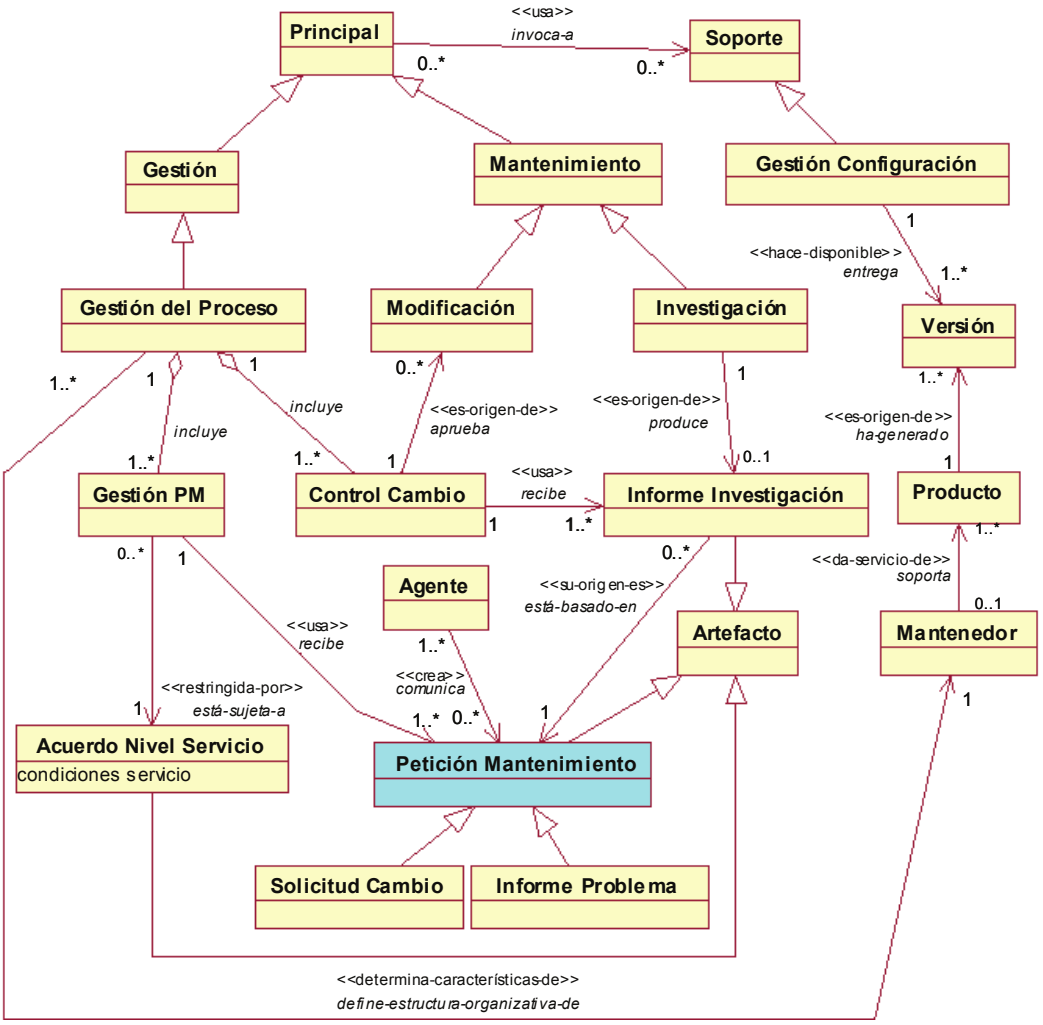


Figura 2.6. Subontología de organización del proceso: gestión de peticiones de mantenimiento.

Un aspecto complementario en esta parte son las actividades de soporte invocadas por las principales de mantenimiento y de gestión del proceso. En especial, las actividades de "gestión de configuración" tienen una repercusión muy fuerte sobre la calidad y rendimiento del servicio de mantenimiento, ya que son las encargadas de la entrega a los usuarios de nuevas versiones de un producto software. Además, son básicas para garantizar la integridad y conocer el estado de un producto software y sus artefactos.

2.4.3 Problemas

Tal como se ha comentado, las actividades de investigación producen informes de investigación asociados a las peticiones de mantenimiento. Estos informes de investigación permiten conocer la información interesante de los diversos problemas encontrados en el software. Un informe de investigación es un artefacto de tipo documento que identifica las causas de un problema (comunicado inicialmente mediante una petición de mantenimiento), los efectos que produce en el funcionamiento del software, y los artefactos donde se han localizado dichas causas (ver Figura 2.7). Un informe de investigación puede referirse a un problema que ya ha sido analizado en uno o varios informes previos, a los cuales complementa y amplia. Las causas de un problema pueden estar en el propio software (defectos software) o ser de otra naturaleza. Los defectos software más frecuentes son los que producen fallos, es decir, malos funcionamientos durante la ejecución [Kajko-Mattsson, 1999].

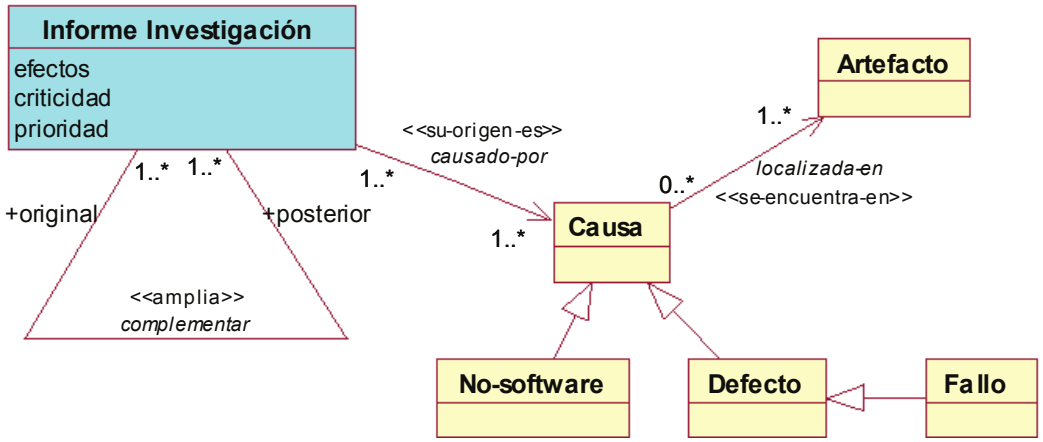


Figura 2.7. Subontología de organización del proceso: informes de investigación.

2.4.4 Integración de los tres aspectos

En a Figura 2.8 se muestra el diagrama conjunto integrando los tres aspectos explicados dentro de la subontología de organización del proceso.

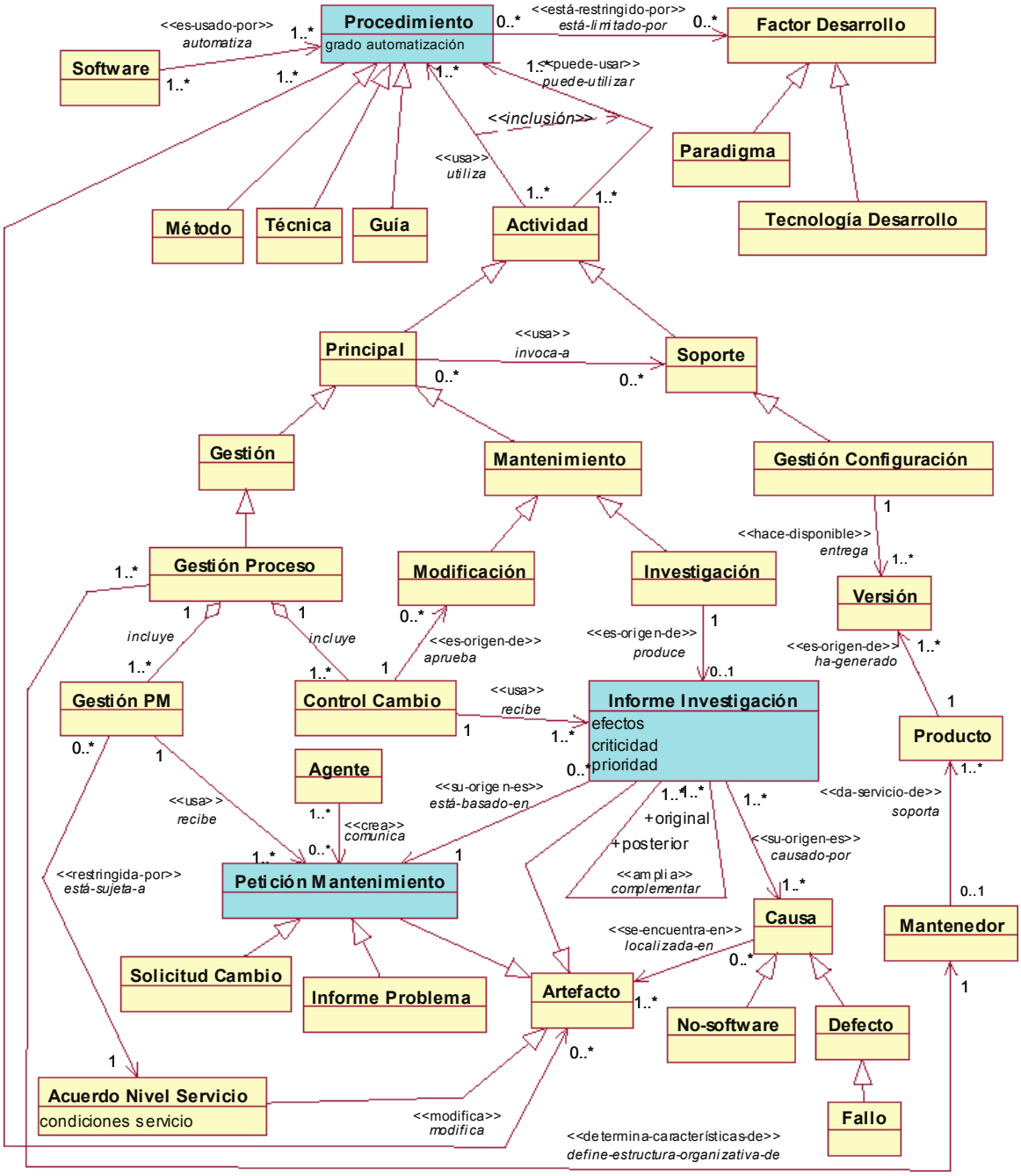


Figura 2.8. Diagrama global de la subontología de organización del proceso.

El glosario de conceptos y las tablas de atributos e interrelaciones de esta subontología se muestran en las siguientes tablas.

Concepto	Super-Concepto	Descripción	Propósito
Actividad	-	Ver subontología de las actividades	-
Acuerdo Nivel Servicio	Artefacto	Documento con los objetivos y condiciones en que el mantenedor prestará el servicio de mantenimiento al cliente.	Incorporar perspectiva de servicio al PMS.
Agente	-	Ver subontología de los agentes	-
Artefacto	-	Ver subontología de los productos	-
Causa	Concepto	Causa de un problema en un producto software.	Analizar los problemas detectados.
Control Cambio	Gestión Proceso	Actividad de gestión que permite determinar las actividades de modificación adecuadas para atender un informe de investigación, y aprobar las modificaciones realizadas.	Autorizar o rechazar modificaciones.
Defecto	Causa	Anomalía de funcionamiento de un producto software respecto a lo establecido en los requisitos.	Clasificar las causas de problemas.
Factor Desarrollo	Concepto	Un factor relativo a la manera en que se desarrolló el producto original que supone una restricción para determinar los procedimientos válidos y útiles.	Delimitar el catálogo de procedimientos.
Fallo	Defecto	Defecto encontrado en el código ejecutable.	Clasificar las causas de problemas.
Gestión	-	Ver subontología de las actividades	-
Gestión Configuración	-	Ver subontología de las actividades	-
Gestión PM	Gestión Proceso	Actividad de gestión que consiste en gestionar la cola de PMs, comprobar si se adecuan a las condiciones de servicio establecidas en el ANS y, en su caso, invocar las actividades de mantenimiento adecuadas.	Conocer problemas en el software.
Gestión Proceso	-	Ver subontología de las actividades	-
Guía	Procedimiento	Guía para construir o corregir un tipo específico de documento. Sinónimos: Plantilla, *Script*.	Definir tipos de documentos a producir.

Concepto	Super-Concepto	Descripción	Propósito
Informe Investigación	Artefacto	Documento que incluye un análisis del impacto de la resolución de un problema o de un cambio solicitado, las diferentes soluciones para implementarlo y la alternativa elegida y su justificación.	Documentar análisis de impacto.
Informe Problema	Petición Mantenimiento	Petición de mantenimiento que describe los síntomas de un problema encontrado en el producto.	Documentar necesidades de mantenimiento.
Investigación	-	Ver subontología de las actividades	-
Mantenedor	-	Ver subontología de los agentes	-
Mantenimiento	-	Ver subontología de las actividades	-
Método	Procedimiento	Procedimiento sistemático, incluyendo sus pasos y heurísticas, para permitir la realización de una o varias actividades. Ejemplo: estimación de esfuerzo mediante COCOMO II.	Definir actividades.
Modificación	-	Ver subontología de las actividades	-
No-software	Causa	Una causa que no es un defecto en el software. Ejemplos: equivocación de los usuarios en la entrada de datos, mal funcionamiento del hardware.	Analizar los problemas detectados.
Paradigma	Factor Desarrollo	Filosofía de desarrollo de software utilizada durante la construcción original del producto. Los procedimientos útiles están limitados por el paradigma. Ejemplo: orientación a objetos.	Identificar procedimientos útiles.
Petición Mantenimiento	Artefacto	Documento describiendo un problema detectado en el producto mantenido (informe de problema) o explicando y justificando un cambio del producto (solicitud de cambio).	Documentar necesidades de mantenimiento.
Principal	-	Ver subontología de las actividades	-
Procedimiento	Concepto	Conducta bien definida y precisa para realizar una actividad. Sinónimo: directiva.	Explicar la manera de realizar las actividades.
Producto	-	Ver subontología de los productos	-
Software	-	Ver subontología de las actividades	-

Concepto	Super-Concepto	Descripción	Propósito
Solicitud Cambio	Petición Mantenimiento	Petición de mantenimiento que explica y justifica la necesidad de un cambio en el producto.	Documentar necesidades de mantenimiento.
Soporte	-	Ver subontología de las actividades	-
Técnica	Procedimiento	Procedimiento usado para realizar una actividad definido de forma menos rigurosa que un método. Técnica: estimación funcional de tamaño.	Definir actividades.
Tecnología Desarrollo	Factor Desarrollo	La tecnología utilizada cuando el producto y sus artefactos fueron originalmente desarrollados. En función de cual esta tecnología los procedimientos de mantenimiento podrán ser unos u otros. Ejemplos: almacén de datos, SGBD relacional.	Decidir procedimientos válidos.
Versión	-	Ver subontología de los productos	-

Tabla 2.9. Subontología de organización del proceso: glosario de conceptos.

Concepto	Atributo	Descripción	Cardi-nalidad
Acuerdo Nivel Servicio	Condiciones servicio	Descripción del alcance del servicio de mantenimiento, es decir, de los compromisos que adquiere el mantenedor frente al cliente.	1..*
Informe Investigación	Efectos	Descripción de los efectos producidos por un problema concreto en el funcionamiento del software.	1..*
	Criticidad	Evaluación cualitativa del grado de impacto que la resolución del problema tiene en el funcionamiento ordinario del producto.	1
	Prioridad	Nivel de importancia asignado a la resolución del problema. Está determinado en parte por la criticidad.	1
Procedimiento	Grado automatización	Indicador del nivel de automatización actual de un procedimiento: ninguno, semi-automático, automático.	1

Tabla 2.10. Subontología de organización del proceso: tabla de atributos.

Nombre	Descripción
Aprueba	El resultado de las actividades de control de cambios es la aprobación de modificaciones.
Automatiza	Cada recurso software automatiza unos determinados procedimientos.
Causado-por	El problema analizado en un informe de investigación está originado por una o varias causas.
Complementar	Un informe de investigación puede complementar a otros anteriores.
Comunica	Los agentes que participan en el proyecto envían las PMs al mantenedor.
Define-estructura-organizativa-de	En las actividades de gestión del proceso se define la estructura organizativa utilizada por el mantenedor para llevar a cabo el proyecto.
Entrega	Las actividades de gestión de configuración entregan nuevas versiones y mejoras.
Está-basado-en	Cada informe de investigación está basado en una petición de mantenimiento.
Está-limitado-por	La utilidad de un procedimiento está limitada por los factores de desarrollo del producto software.
Está-sujeta-a	Las decisiones en las actividades de gestión de PM están sujetas a lo estipulado en el ANS.
Ha-generado	Ver subontología de los productos.
Incluye	La gestión del proceso incluye, entre otras, un conjunto de actividades de gestión de PM y de control de cambio.
Invoca-a	Las actividades principales invocan a actividades de soporte.
Localizada-en	Cada causa de un problema se encuentra localizada en uno o varios artefactos.
Modifica	Cada procedimiento puede modificar (o crear) unos determinados artefactos.
Produce	Las actividades de investigación producen informes de investigación.
Puede-utilizar	Para realizar cada actividad se pueden utilizar unos determinados procedimientos.
Recibe	Las actividades de gestión de PM se activan cuando se recibe una petición de mantenimiento.
Soporta	Ver subontología de los agentes.
Utiliza	En cada actividad se utilizan algunos de los procedimientos que son utilizables para su realización.

Tabla 2.11. Subontología de organización del proceso: tabla de interrelaciones.

2.5 SUBONTOLOGÍA DE LOS AGENTES

Esta subontología se refiere a la *jerarquía de tipos de agentes* que existen durante la gestión de proyectos de mantenimiento. Esta jerarquía tiene las características siguientes:

1. Los agentes pueden ser humanos o automáticos (herramientas software).

2. Los agentes humanos pueden ser personas individuales u organizaciones. Estas últimas están formadas por personas, de manera que cada persona desempeña un determinado puesto en cada organización que la emplea.

3. Cada organización tiene un modelo organizacional, que se representa mediante una jerarquía de agregación de organizaciones subordinadas completado con las relaciones de subordinación entre las personas que forman parte de ellas (en realidad, entre los puestos que ocupan).

4. Existen tres tipos de organizaciones virtuales: mantenedor, cliente y usuario. Se identifican como virtuales porque pueden corresponder a una organización real diferente cada una de ellas o las tres ser la misma organización real. También es común el caso en que existen dos organizaciones reales, el mantenedor y el cliente, que a la vez es usuario. También puede ocurrir que las tres organizaciones virtuales sean organizaciones reales subordinadas de una misma organización real, por ejemplo, cuando un departamento de informática da servicio de mantenimiento a otros departamentos de la misma empresa.

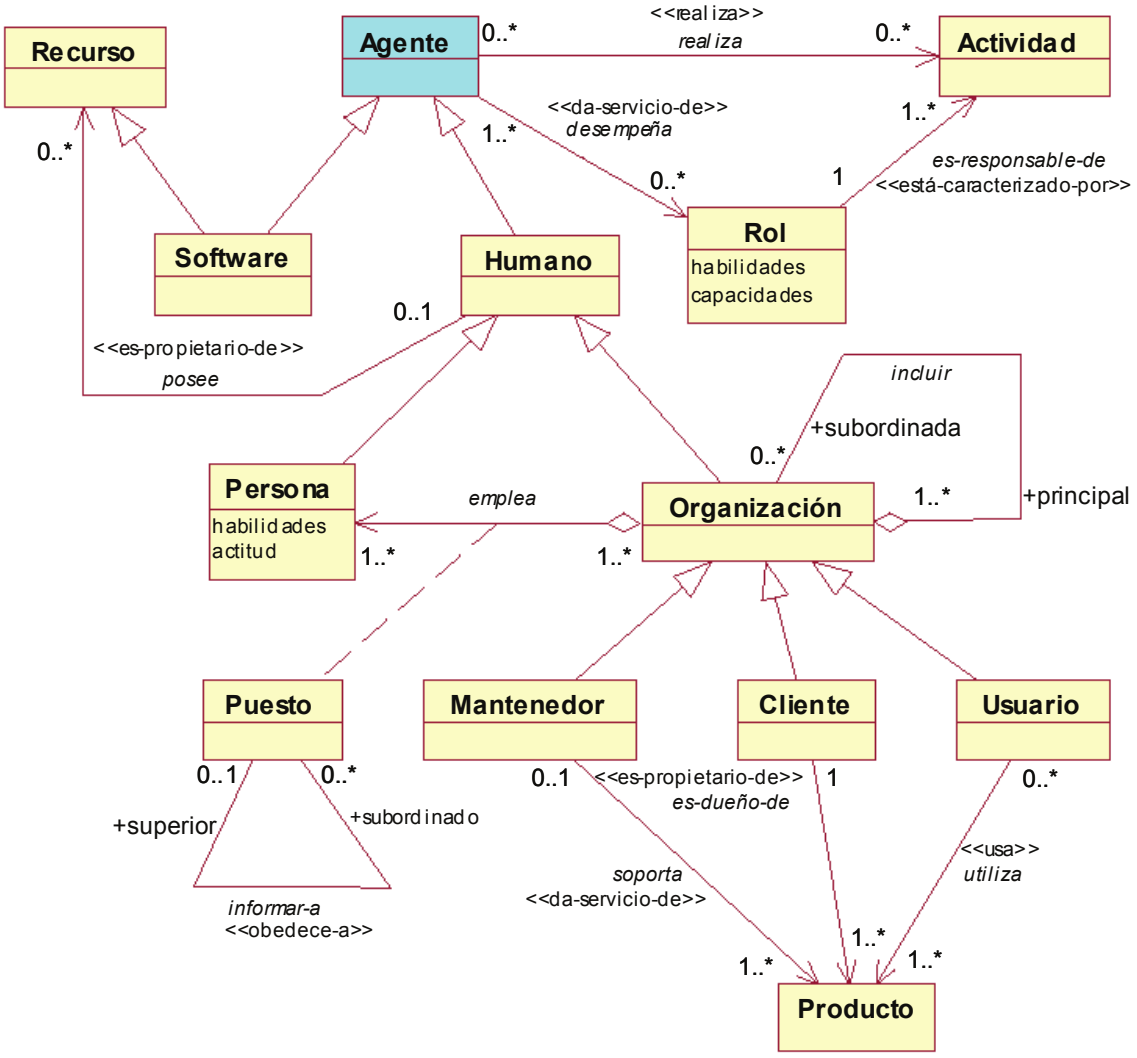


Figura 2.9. Diagrama de la subontología de los agentes.

Como muestra la Figura 2.9, el modelo de agentes también está basado en el concepto de rol, lo que permite mayor generalidad y flexibilidad para representar cualquier posible situación que se produzca en un proyecto real [Polo *et al.*, 1999]. De esta manera, durante la definición de los procesos, se establecen los roles existentes y las actividades que son responsabilidad de cada rol. En cambio, durante la planificación del proyecto se establecen los agentes participantes y los roles que desempeñarán. Por último, durante la reificación de los procesos, es decir, durante la ejecución del proyecto, se identifican las realizaciones concretas de actividades de cada agente.

El glosario de conceptos y las tablas de atributos e interrelaciones de esta subontología son los siguientes:

Concepto	Super-Concepto	Descripción	Propósito
Actividad	-	Ver subontología de las actividades	-
Agente	Elemento	Una persona, organización o aplicación software que desempeña un papel activo en la realización de un proyecto de mantenimiento, es decir, realiza alguna(s) actividad(es). Sinónimo: actor.	Identificar los participantes en el proyecto.
Cliente	Organización	Organización propietaria del producto software mantenido.	Definir las organizaciones participantes.
Humano	Agente	Agente que es una persona individual o un grupo de personas (organización).	Definir estructural organizacional.
Mantenedor	Organización	Organización que realiza el servicio de mantenimiento del producto software.	Definir las organizaciones participantes.
Organización	Humano	Un agente humano formado por un grupo de personas u organizaciones más simples (sub-organizaciones) que actúan con una identidad común en el proyecto. Ejemplos: empresa cliente, equipo de mantenimiento, departamento de informática.	Definir estructural organizacional.
Persona	Humano	Agente humano individual.	Identificar personas participantes.
Producto	-	Ver subontología de los productos	-
Puesto	Concepto	Un puesto de trabajo dentro de una organización. Es ocupado por una persona. Ejemplos: Jefe de proyecto, Director de TI.	Definir estructural organizacional.
Recurso	-	Ver subontología de las actividades	-
Rol	Concepto	Abstracción de un conjunto de habilidades o capacidades necesarias para realizar una o varias actividades. Sinónimos: papel, función. Ejemplos: responsable de pruebas, ingeniero de mantenimiento.	Establecer responsabilidades.
Software	-	Ver subontología de las actividades	-
Usuario	Organización	Organización que utiliza el producto software mantenido.	Definir las organizaciones participantes.

Tabla 2.12. Subontología de organización del proceso: glosario de conceptos.

Concepto	Atributo	Descripción	Cardinalidad
Persona	Actitud	La actitud o motivación de una persona para realizar su trabajo. Produce un impacto importante en la calidad de los procesos.	1
	Habilidades	Habilidades que posee una persona (relacionadas con proyectos software). Similares a las de los roles.	1..*
Rol	Capacidades	Capacidades (conocimientos y dominios) necesarias para poder desempeñar un rol. Ejemplo: análisis y diseño con UML.	1..*
	Habilidades	Habilidades personales que ayudan a poder desempeñar mejor un rol. Ejemplo: liderazgo.	1..*

Tabla 2.13. Subontología de organización del proceso: tabla de atributos.

Nombre	Descripción
Desempeña	Un agente puede desempeñar varios roles en un proyecto. Igualmente, un rol puede ser desempeñado por varios agentes diferentes.
Emplea	Cada persona en cada organización es empleada en un determinado puesto.
Es-dueño	Un recurso puede tener un propietario, que debe ser un agente humano.
Es-responsable-de	Cada actividad tiene un rol que es responsable de su correcta realización.
Incluir	Las organizaciones pueden estar formadas por otras organizaciones más simples (sub-organizaciones) y pueden ser parte de otras organizaciones más complejas.
Informar-a	Los puestos de trabajo de una organización tienen una organización jerárquica de forma que cada puesto informa y obedece a su inmediato superior.
Posee	Los agentes humanos pueden ser propietarios de los recursos utilizados en el proyecto.
Realiza	Durante la fase de ejecución del proyecto, agentes concretos realizan las actividades.
Soporta	La organización de mantenimiento (mantenedor) da servicio de mantenimiento de unos determinados productos software.
Utiliza	Una organización o usuario utiliza unos determinados productos software.

Tabla 2.14. Subontología de las actividades: tabla de interrelaciones.

La integración de estas cuatro subontologías se ha realizado teniendo en cuenta los principios de ingeniería ontológica Los diagramas y tablas anteriores ya están libres de inconsistencias semánticas y sintácticas. La única excepción es en el caso de repeticiones de nombres de interrelaciones ya que, en el caso de ontologías tan complejas como las presentes, es prácticamente imposible evitar esta duplicidad si se desea utilizar nombres auto-explicativos. Esto no es problema siempre y cuando se cumpla que las dos interrelaciones etiquetadas con igual nombre tengan la misma clase de interrelación (es decir, su semántica sea la misma) y que, en caso de duda, se precedan o acaben con el nombre de un concepto participante que permita distinguirlas; por ejemplo, [Agente].posee o incluye.[Gestión PM].

La tabla de clases de interrelaciones de la ontología del mantenimiento (englobando las cuatro subontologías anteriores), es la siguiente:

Clase	Nombre inverso	Propósito
Amplía	Es-ampliado-por	Las instancias de un concepto permiten conocer en más detalle las características de las instancias de otro concepto.
Caracteriza-a	Está-caracterizado-por	Una instancia de un concepto identifica a las instancias de otro concepto.
Crea	Es-creado-por	Una instancia de un concepto provoca la creación de instancias de otro concepto.
Da-servicio-de	Es-servido-por	Las instancias de un concepto llevan a cabo un trabajo para atender una necesidad de las instancias de otro concepto.
Determina-características-de	Sus-características-están-determinadas-por	La estructura interna de las instancias de un concepto es determinada por las instancias de otro concepto.
Es-objetivo-de	Su-objetivo-es	La existencia de una instancia de un concepto es la razón de la existencia de instancias de otro concepto.
Es-origen-de	Su-origen-es	Una instancia de un concepto se origina como consecuencia y después de la existencia de otra instancia de otro concepto.
Es-propietario-de	Su-propietario-es	Las instancias de un concepto que refiere a personas u organizaciones humanas, son propietarias de instancias de otro concepto.
Hace-disponible	Está-disponible-por	La existencia de una instancia de un concepto hace posible que las instancias de otro concepto pueden ser utilizadas por instancias de otros conceptos diferentes.
Modifica	Es-modificado-por	Una instancia de un concepto provoca la modificación de instancias de otro concepto.

Obedece-a	Es-obedecido-por	Las acciones de las instancias de un concepto están determinadas por instancias de otro concepto.
Puede-usar	Es-usable-por	Una instancia de un concepto puede utilizar – opcionalmente- una instancia de otro concepto.
Realiza	Es-realizado-por	Las instancias de un concepto que refiere a un trabajo o acción, son realizadas por instancias de otro concepto.
Referencia-a	Referenciado-por	La definición de una instancia de un concepto se hace en referencia a instancias de otro concepto.
Restringe-a	Restringido-por	Los estados posibles de una instancia de un concepto están limitados por instancias de otro concepto.
Se-encuentra-en	Es-encontrada-en	Las instancias de un concepto están localizadas dentro de la estructura interna de instancias de otro concepto.
Usa	Es-usado-por	Una instancia de un concepto utiliza instancias de otro concepto.

Tabla 2.15. Ontología del mantenimiento: tabla de clases de interrelaciones.

2.6 LECTURAS RECOMENDADAS

✓ Calero, C., Ruiz, F: y Piattini, M. (eds.)(2008). Ontologies for Software Engineering and Software Technology, Springer.

En este libro se ofrece una panorámica bastante completa sobre la utilización de las ontologías en la ingeniería del software.

2.7 SITIOS WEB RECOMENDADOS

✓ *www.oeg-upm.net*

Para profundizar más en los aspectos de la ingeniería ontológica, invitamos al lector a visitar la web del "Ontology Engineering Group" , dirigido por la profesora Asunción Gómez-Pérez.

2.8 EJERCICIOS

Ejercicio 1

¿Qué diferencias existen entre una ontología y un metamodelo?

Ejercicio 2

Intente formalizar las ontologías en alguna plataforma adecuada.

Ejercicio 3

¿Cómo integraría la ontología del mantenimiento con una ontología de la medición? Recomendamos al lector consultar: García et al. (2006). Towards a consistent terminology for software measurement. Information & Software Technology 48(8), 631-644.

Ejercicio 4

Intente ampliar la ontología incluyendo conceptos de refactoring, *smells,* patrones, etc. Recomendamos al lector consultar: Garzás, J. y Piattini, M. (2005). An Ontology for Microarchitectural Design Knowledge. IEEE Software 22(2), 28-33.

Ejercicio 4

Si el mantenimiento se lleva a cabo de manera global, es decir mediante equipos en diferentes partes del mundo, ¿cómo se podría extender la ontología?. Recomendamos al lector consultar: Vizcaíno, A., García, F., Piattini, M., y Beecham, S. (2016). A validated ontology for global software development. Computer Standards & Interfaces 46, 66-78

3

EL PROCESO DE MANTENIMIENTO EN EL CICLO DE VIDA SOFTWARE

Resulta fundamental entender el mantenimiento como uno de los procesos principales dentro del contexto del ciclo de vida software, para lo que conviene examinar los principales estándares internacionales relativos a este tema. En estos estándares se especifican las principales actividades y tareas de todos los procesos del ciclo de vida, y, por supuesto, las de mantenimiento, que son objeto del presente capítulo.

3.1 PROCESOS DEL CICLO DE VIDA DEL SOFTWARE

Uno de los problemas más importantes en cualquier departamento de sistemas de información es definir un marco de referencia común que pueda ser empleado por todas las personas que participan en el desarrollo de los sistemas, y en el que se definan los procesos, las actividades y las tareas a llevar a cabo.

Precisamente, las principales organizaciones profesionales y organismos internacionales se han venido ocupando del ciclo de vida de los sistemas y del software, así, por ejemplo, tanto IEEE como ISO/IEC han publicado a lo largo del tiempo diversos estándares titulados, respectivamente, "*IEEE Standard for Developing Software Life Cycle Processes*" e "*Information technology – Software life-cycle processes*". En la actualidad proponen una norma conjunta ISO/IEC 12207 también conocida por IEEE Std 12207. Esta norma entiende por modelo de ciclo de vida "*un marco de referencia de procesos y actividades que conciernen el ciclo de vida, que puede organizarse en etapas, y que sirve como referencia común para la comunicación y el entendimiento*". El **ciclo de vida** abarca toda la "vida" del sistema,

producto, servicio, o proyecto, comenzando con su concepción y finalizando cuando ya no se utiliza.

A continuación, se resumen los procesos del ciclo de vida del software según la norma ISO/IEC 12207 [ISO/IEC, 2017] que agrupa las actividades que se pueden realizar durante el ciclo de vida del software se agrupan en cuatro grandes categorías (véase Figura 3.1).

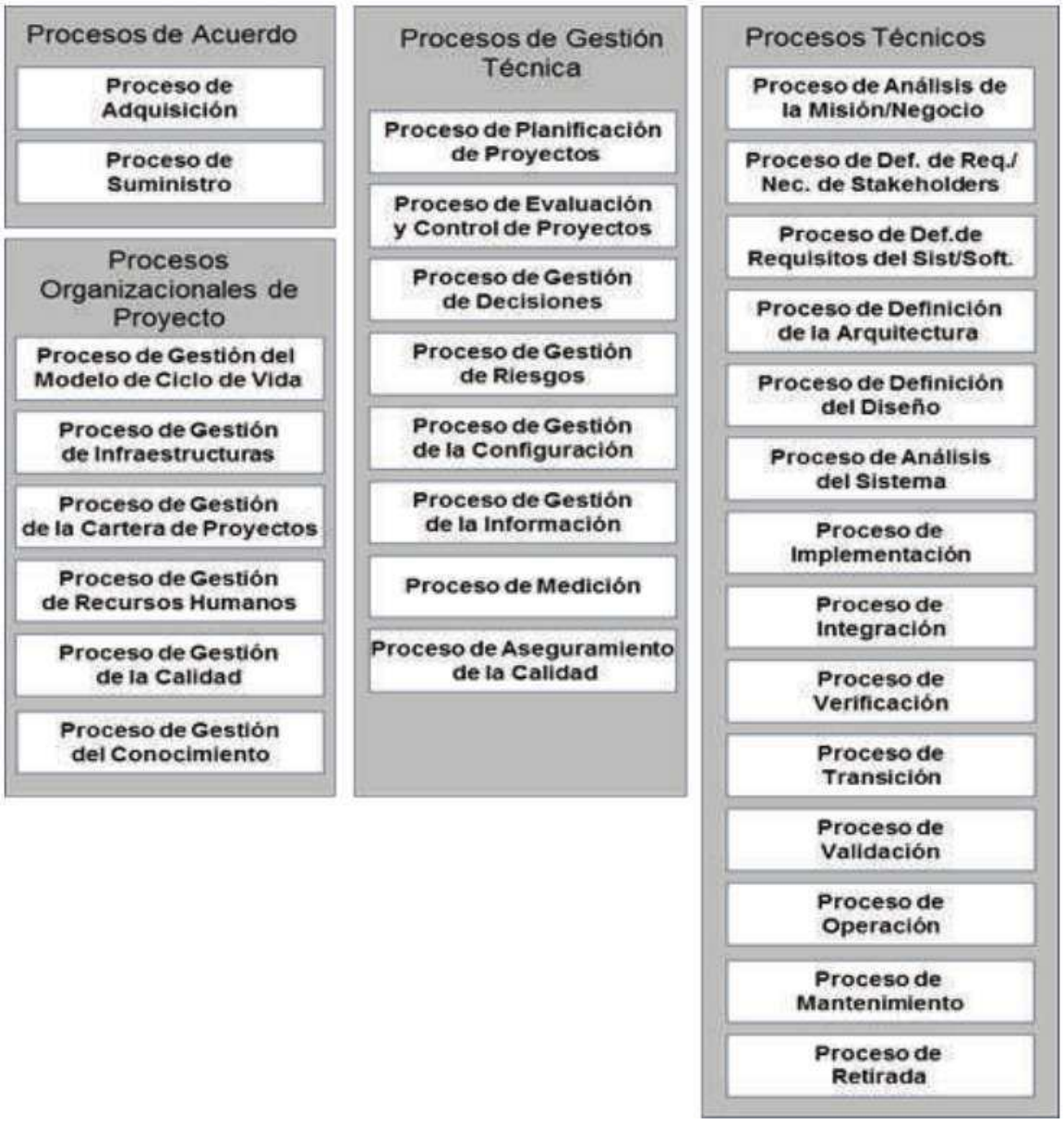


Figura 3.1. Procesos del ciclo de vida según la norma ISO 12207

Los procesos del ciclo de vida se agrupan en cuatro categorías.

3.1.1 Procesos de Acuerdo

Definen las actividades necesarias para establecer un acuerdo entre dos organizaciones (la compradora y la proveedora de los productos o servicios). Se encuadran en esta categoría:

- El **Proceso de Adquisición**, cuyo propósito es obtener el producto o servicio de acuerdo con los requisitos del comprador. Se compone de las siguientes actividades:
  - Prepararse para la adquisición
  - Publicitar la adquisición y seleccionar el proveedor
  - Establecer y mantener un acuerdo
  - Monitorizar el acuerdo
  - Aceptar el producto o servicio
- El **Proceso de Suministro**, cuyo propósito es proveer un producto o servicio al comprador, que cumpla los requisitos acordados. Se compone de las siguientes actividades:
  - Prepararse para el suministro
  - Responder a una licitación
  - Establecer y mantener un acuerdo
  - Ejecutar el acuerdo
  - Entregar y dar soporte al producto o servicio

3.1.2 Procesos Organizacionales que Posibilitan los Proyectos

Estos procesos se encargan de asegurar la disponibilidad de los recursos necesarios para que el proyecto satisfaga las necesidades y expectativas de las stakeholders de la organización. Son los siguientes:

- **Proceso de Gestión del Modelo de Ciclo de Vida**, que se encarga de definir, mantener y asegurar la disponibilidad de políticas, procesos, modelos y procedimientos del ciclo de vida. Consta de tres actividades básicas como son: establecer, evaluar y mejorar el proceso.

- **Proceso de Gestión de Infraestructuras**, cuyo propósito es proveer la infraestructura (sus principales actividades son establecer y mantener la infraestructura) y los servicios a los proyectos para dar soporte a los objetivos del proyecto y de la organización durante el ciclo de vida.

- **Proceso de Gestión de la Cartera de Proyectos,** encargado de iniciar y mantener los proyectos necesarios, suficientes y adecuados para cumplir los objetivos estratégicos de la organización. Sus principales actividades son: definir y autorizar proyectos, evaluar la cartera de proyectos y finalizar los proyectos.

- **Proceso de Gestión de Recursos Humanos**, que asegura que se proveen los recursos humanos necesarios y mantiene sus competencias, en consonancia con las necesidades de negocio. Para ello se llevan a cabo las siguientes actividades: identificar, desarrollar, adquirir y proporcionar habilidades.

- **Proceso de Gestión de la Calidad**, que pretende asegurar que los productos, servicios e implementaciones del proceso de gestión de la calidad cumplen los objetivos de calidad del proyecto y de la organización y logran la satisfacción del cliente. Según la norma, como resultado de este proceso se tiene que conseguir que:

  - Se definan e implementen objetivos, políticas y procedimientos de gestión de la calidad de la organización.
  - Se establezcan criterios y métodos de evaluación de la calidad.
  - Se proporcionen a los proyectos los recursos y la información para dar soporte a la operación y monitorización de las actividades de aseguramiento de la calidad del proyecto.
  - Se obtienen y analizan los resultados de evaluación del aseguramiento de la calidad.
  - Se mejoran las políticas y procedimientos de gestión de la calidad basándose en resultados de proyecto y organizacionales.

  Este proceso contempla las siguientes actividades:

  - Planificar la gestión de calidad.
  - Evaluar la gestión de la calidad.
  - Llevar a cabo acciones preventivas y correctivas de gestión de la calidad.

- **Proceso de Gestión del Conocimiento**, cuyo propósito es crear la capacidad y los activos que permiten a la organización explotar las oportunidades de volver a aplicar conocimiento existente. Comprende las actividades de planificar la gestión del conocimiento, compartir conocimientos y habilidades en la organización, compartir activos de conocimiento, y gestionar conocimiento, habilidades y activos de conocimiento.

3.1.3 Procesos de Gestión Técnica

Estos procesos se ocupan de la gestión de los recursos y activos asignados por la dirección de la organización y de aplicarlos para cumplir los acuerdos a los que se compromete la organización:

- **Procesos de Planificación de Proyectos**, que produce y coordina los planes realistas y efectivos, para lo cual se descompone en las actividades siguientes: definir el proyecto, planificar la gestión técnica y la del proyecto, y activar el proyecto.

- **Proceso de Control y Evaluación de Proyectos**, cuyo propósito es asegurar que los planes están alineados y son factibles, determinar el estado del proyecto, y del desempeño técnico y del proceso, y dirigir la ejecución para asegurar que el desempeño está de acuerdo con lo planificado, con los presupuestos previstos y satisfacer los objetivos técnicos. Comprende las actividades relativas a: planificar el control y la evaluación del proyecto, evaluar el proyecto y controlar el proyecto.

- **Proceso de Gestión de Decisiones,** proporciona un marco analítico y estructurado para identificar, caracterizar y evaluar, de manera objetiva, un conjunto de alternativas para una decisión en cualquier punto del ciclo de vida y seleccionar la opción más beneficiosa. Para ello, engloba las siguientes actividades: prepararse para las decisiones, analizar la información para la decisión, tomar y gestionar las decisiones.

- **Proceso de Gestión de Riesgos,** cuyo propósito es identificar, analizar, tratar y monitorizar de manera continua los riesgos, para lo que se llevan a cabo las siguientes actividades: planificar la gestión de riesgos, gestionar el perfil de riesgo, analizar los riesgos, tratar los riesgos y monitorizar los riesgos.

- **Proceso de Gestión de la Configuración**, encargado de gestionar y controlar los elementos y las configuraciones del sistema a lo largo del ciclo

de vida. Comprende las actividades relativas a: planificación de la gestión de configuración, realización de la identificación de la configuración, de la gestión del cambio de la configuración, de la determinación del estado de la configuración, de la evaluación de la configuración y del control de liberaciones.

- **Proceso de Gestión de la Información**, cuyo propósito es generar, obtener, confirmar, transformar, conservar, recuperar, diseminar y destruir la información a los stakeholders designados. Se compone de dos actividades: prepararse para la gestión de información y llevar a cabo la gestión de la información.

- **Proceso de Medición**, encargado de recoger, analizar y reportar datos e información objetivos para dar soporte a una gestión efectiva y demostrar la calidad de productos, servicios y procesos. Se descompone en: preparar la medición y llevarla a cabo.

- **Proceso de Aseguramiento de la Calidad**, que sirve para ayuda a asegurar la aplicación efectiva del proceso de gestión de calidad de la organización al proyecto.

3.1.4 Procesos Técnicos

Estos procesos son los que se utilizan para definir los requisitos de un sistema intensivo en software, transformar los requisitos en un producto efectivo, permitir la reproducción consistente del producto cuando sea necesario, usar el producto para proporcionar los servicios requeridos, mantener la provisión de estos servicios y eliminar el producto cuando se retire del servicio. La norma incluye en este apartado los siguientes procesos:

- **Proceso de Análisis de la Misión o Negocio**, para definir el negocio o la misión o la oportunidad, caracterizar el espacio de la solución y determinar las soluciones potenciales que pueden abordar el problema o sacar ventaja de una oportunidad.

- **Proceso de Definición de Requisitos y Necesidades de los Stakeholders**, para que un sistema o producto software pueda proveer las capacidades necesarias a los usuarios y otros stakeholders en un entorno definido.

- **Proceso de Definición de Requisitos del Sistema/Software,** cuyo propósito es transformar la visión orientada al usuario o stakeholder de

las capacidades deseadas a una vista técnica de una solución que cumple las necesidades operacionales del usuario.

- **Proceso de Definición de la Arquitectura**, para generar alternativas de la arquitectura del sistema, seleccionar una o más alternativas que atiendan las preocupaciones de los stakeholders y cumplan los requisitos del sistema y expresar la arquitectura en un conjunto de vistas consistentes.
- **Proceso de Definición del Diseño**, para proporcionar suficientes datos e información detallada acerca del sistema y sus elementos que permitan la implementación consistente con las entidades arquitectónicas definidas en los modelos y vistas de la arquitectura del sistema.
- **Proceso de Análisis del Sistema**, que proporciona una base rigurosa de datos e información para la comprensión técnica que sirve para ayudar a la toma de decisiones a lo largo del ciclo de vida.
- **Proceso de Implementación**, cuyo propósito es realizar un elemento de software o de sistema específico.
- **Proceso de Integración**, que sintetiza un conjunto de elementos software o de sistema en un software o sistema realizado (producto o servicio) que satisface los requisitos, arquitectura y diseño de software y de sistema.
- **Proceso de Verificación**, cuyo propósito es proporcionar evidencia objetiva de que un sistema o elemento software o de sistema cumple sus requisitos o características especificadas.
- **Proceso de Transición**, para dotar a un producto software o sistema de la capacidad de proporcionar servicios especificados por los requisitos de los stakeholders en el entorno operacional.
- **Proceso de Validación**, cuyo propósito es proporcionar evidencia objetiva de que el sistema, cuando se usa, satisface los requisitos de los stakeholders y los objetivos organizacionales logrando su uso previsto en el entorno operacional previsto.
- **Proceso de Operación**, por el que se usa el producto software o sistema para proporcionar sus servicios.
- **Proceso de Mantenimiento**, cuyo propósito es conservar la capacidad del sistema de proporcionar un servicio. Este es el proceso clave y en el cual se dedicará el siguiente punto del capítulo.

- **Proceso de Retirada**, que finaliza la existencia de un elemento software o un sistema para un uso determinado, manipulando los elementos retirados o reemplazados de forma apropiada y atendiendo las necesidades críticas de retirada que se han identificado (por los acuerdos, políticas, aspectos legales, medioambientales, de seguridad, etc.).

3.1.5 Proceso de Adaptación

El anexo A de la norma muestra cómo adaptarla a casos particulares con el fin de obtener procesos del ciclo de vida nuevos o modificados teniendo en cuenta circunstancias tan variadas como: estabilidad o variedad de entornos operacionales; riesgos (comerciales o de desempeño) de las diferentes stakeholders; novedad, tamaño y complejidad; fecha de inicio o duración de la utilización; cuestiones de integridad como seguridad, seguridad en el funcionamiento (safety), privacidad, usabilidad o disponibilidad; oportunidades tecnológicas emergentes; perfil del presupuesto y de los recursos organizacionales disponibles; disponibilidad de los servicios de los sistemas de soporte; roles, responsabilidades y autorizaciones en el ciclo de vida del sistema; necesidad de cumplir otros estándares, etc.

3.2 ACTIVIDADES Y TAREAS DEL PROCESO DE MANTENIMIENTO

Como hemos visto en el apartado anterior, la norma ISO/IEC 14764:2006 [ISO/IEC, 2006] considera el mantenimiento como uno de los procesos principales del ciclo de vida del software, ya que «define las actividades de la organización (*mantenedora*) que proporciona el servicio de mantener el producto software; es decir, gestionar las modificaciones al producto software con el fin de mantenerlo actualizado y adecuado a su uso». Mientras que la ontología presentada en el capítulo 2 nos permite tener una visión general de los conceptos manejados, ahora profundizamos en las actividades concretas. En esta norma se pueden distinguir las siguientes actividades del proceso de mantenimiento:

3.2.1 Preparación para el mantenimiento

En esta actividad se define y se planifica el proceso de mantenimiento software. Esta actividad consiste en las siguientes tareas:

- Definir una estrategia de mantenimiento, que incluya la consideración de lo siguiente:

- Establecimiento de prioridades, horarios típicos y procedimientos para realizar, verificar, distribuir e instalar cambios de mantenimiento de software de acuerdo con los requisitos de disponibilidad operacional.
- Establecer técnicas y métodos para tomar conciencia de la necesidad de soluciones correctivas, adaptativas y mantenimiento perfectivo.
- Evaluación periódica de las características de diseño en caso de evolución del sistema de software y de su arquitectura.
- Pronosticar la obsolescencia potencial de componentes y tecnologías utilizando información técnica de los cambios en los sistemas relacionados.
- Establecer prioridades y recursos para obtener acceso a las versiones correctas del producto e información del producto necesaria para realizar el mantenimiento (por ejemplo, instalación programada o por fases, parches de mantenimiento o actualizaciones de software).
- Medidas de mantenimiento que proporcionarán información sobre los niveles de rendimiento, efectividad y eficiencia, incluido el acceso a fallos y fallos continuados en el tiempo.
- Derechos acordados sobre los datos y el impacto en los datos del sistema durante la resolución del problema y la actividad de mantenimiento.
- Enfoque para garantizar que los elementos del sistema falsos o no autorizados no se introduzcan en el sistema.
- Impacto del cambio de mantenimiento en otros elementos de sistemas de software frente al riesgo de dejar constancia de una anomalía en el software.
- Los niveles de habilidad técnica del personal necesarios para efectuar reparaciones o reemplazos de sistemas o software, soluciones, parches, actualizaciones, teniendo en cuenta los requisitos legales y reglamentarios relacionados con la salud y seguridad, protección y medio ambiente.

◤ Para los elementos que no son de software, definir una estrategia de logística durante todo el ciclo de vida, incluida la adquisición y consideraciones operacionales: la cantidad y el tipo de elementos de reemplazo que se almacenarán, las ubicaciones y condiciones de su almacenamiento, su tasa de reemplazo anticipada, y su vida de almacenamiento y frecuencia de renovación.

- Identificar las restricciones del mantenimiento que se incorporarán en los requisitos del sistema/software, arquitectura o diseño. A menudo, esto se debe a la necesidad de reutilizar los sistemas de mantenimiento y verificación existentes, reutilizar existencias de elementos reemplazables del sistema y acomodar las limitaciones de reabastecimiento o realizar mantenimiento en ubicaciones o entornos específicos.
- Identificar los intercambios de modo que el sistema y las acciones de mantenimiento y logística asociadas den como resultado una solución que sea asequible, operable, compatible y sostenible.
- Identificar y planificar los sistemas o servicios necesarios que habilitan el mantenimiento.
- Obtener o adquirir acceso a los sistemas o servicios de habilitación que se utilizarán.

3.2.2 Ejecución del Mantenimiento

En esta actividad se realiza el esfuerzo de mantenimiento definido en la actividad anterior. Esta actividad consta de las siguientes tareas:

- Revisar los requisitos de los stakeholders, quejas, eventos, incidentes e informes de problemas para identificar necesidades de mantenimiento correctivo, adaptativo, perfectivo y preventivo. Hay que tener en cuenta que, para los sistemas de software con ciclos de vida iterativos, los requisitos cambiantes se pueden considerar como la fuente de actividades de mantenimiento adaptativo y perfectivo. Para el mantenimiento del software, este proceso realiza correcciones, cambios y mejoras en el software implementado, así como parches y actualizaciones para mantener la seguridad del sistema.
- Analizar el impacto de los cambios de mantenimiento en las estructuras de datos, datos y funciones de software, usuarios, documentación e interfaces relacionados. Las revisiones y análisis a menudo incluyen factores tales como la categoría de acción de mantenimiento, tamaño de la modificación, costo involucrado, tiempo para modificar, e impacta en el rendimiento, la seguridad o la protección.
- Al encontrar fallos inesperados que causan un fallo del sistema de software, definir las acciones para restaurar el sistema a un estado operativo.

- Implementar los procedimientos para la corrección de defectos y errores, o para el reemplazo o actualización de elementos del sistema. La corrección de errores y fallos utiliza la resolución de problemas y puede gestionarse a través de los procesos de aseguramiento de calidad y control y evaluación de proyectos. Adicionalmente, se realizan las pruebas de regresión para verificar que el cambio de mantenimiento no ha introducido nuevos problemas.

- Realizar mantenimiento preventivo mediante la sustitución, el parcheo, o la actualización del sistema software para mejorar el rendimiento de un sistema de software que se prevé que llegue a niveles inaceptables. Por ejemplo, falta de capacidad debido a aumentos en la demanda u otras condiciones inaceptables de operación como la ejecución de un software de seguridad obsoleto.

- Identificar cuando se requiere mantenimiento adaptativo o perfectivo. Las acciones de mantenimiento adaptativas y perfectivas generalmente implican cambios en los requisitos del sistema software y/o la arquitectura y diseño, teniendo en cuenta que se puede iniciar un nuevo proyecto para modificar el sistema de software existente.

3.2.3 Soporte Logístico

Las acciones de logística permiten que el sistema software mantenga la disponibilidad operativa. Las acciones incluyen disposiciones para la dotación de personal, soporte de suministros, equipos de soporte, necesidades de datos técnicos (documentación de usuario) y derechos sobre los datos acordados, soporte de capacitación, comunicaciones, soporte de recursos informáticos/equipos e instalaciones. Esta actividad considera las siguientes tareas:

- Obtener recursos para soportar el sistema de software a lo largo de su ciclo de vida o la vida del proyecto (adquisición logística).

- Monitorizar la calidad y disponibilidad de los elementos de reemplazo y los sistemas de habilitación, mecanismos de entrega y su integridad continua durante el almacenamiento.

- Implementar mecanismos para la distribución de elementos o sistemas de software, incluidos el embalaje, manejo, almacenamiento y comunicaciones o transporte necesarios para durante el ciclo de vida.

- Confirmar que las acciones logísticas cumplan con los requisitos del sistema de software o de soporte de elementos o logren que la preparación operacional está planificada e implementada.

3.2.4 Gestión de resultados del mantenimiento y su logística

Esta actividad se descompone en las siguientes tareas para la gestión de los resultados del proceso de mantenimiento software y la logística asociada:

- Registrar incidentes y problemas, incluidas sus resoluciones, y los resultados significativos de mantenimiento y logística. Esto incluye las anomalías debidas a la estrategia de mantenimiento, los sistemas de habilitación de mantenimiento, la ejecución del mantenimiento y logística, o la definición incorrecta del sistema. Esta actividad puede incluir cambios en la logística o procedimientos de distribución de software. Los cambios en los requisitos del sistema de software, arquitectura o diseño son realizados dentro de otros procesos técnicos.

- Identificar y registrar tendencias de incidentes, problemas y acciones de mantenimiento y logística. Los informes con datos de tendencias y resolución de problemas se utilizan para informar al personal de operaciones y mantenimiento, clientes, y otras stakeholders de proyectos que están creando o utilizando entidades de sistema similares.

- Mantener la trazabilidad de los elementos del sistema que se están manteniendo. Se mantiene la trazabilidad bidireccional entre las acciones de mantenimiento registradas y los artefactos del sistema de software durante del ciclo de vida.

- Proporcionar artefactos clave y elementos de información que se han seleccionado para las líneas de base. En este sentido, se utiliza el proceso de gestión de configuración para establecer y mantener elementos de configuración.

- Monitorizar y medir la satisfacción del cliente con el sistema y acerca del soporte de mantenimiento. El estándar ISO 10004 contiene pautas para monitorear y medir la satisfacción del cliente, cuya información se puede utilizar posteriormente en el proceso de gestión de la calidad.

3.3 EL MANTENIMIENTO EN LA NORMA ISO/IEC 14764

La norma ISO/IEC 14764 [ISO/IEC, 2006] propone para el proceso de mantenimiento la estructura que se muestra en la Figura 3.2.

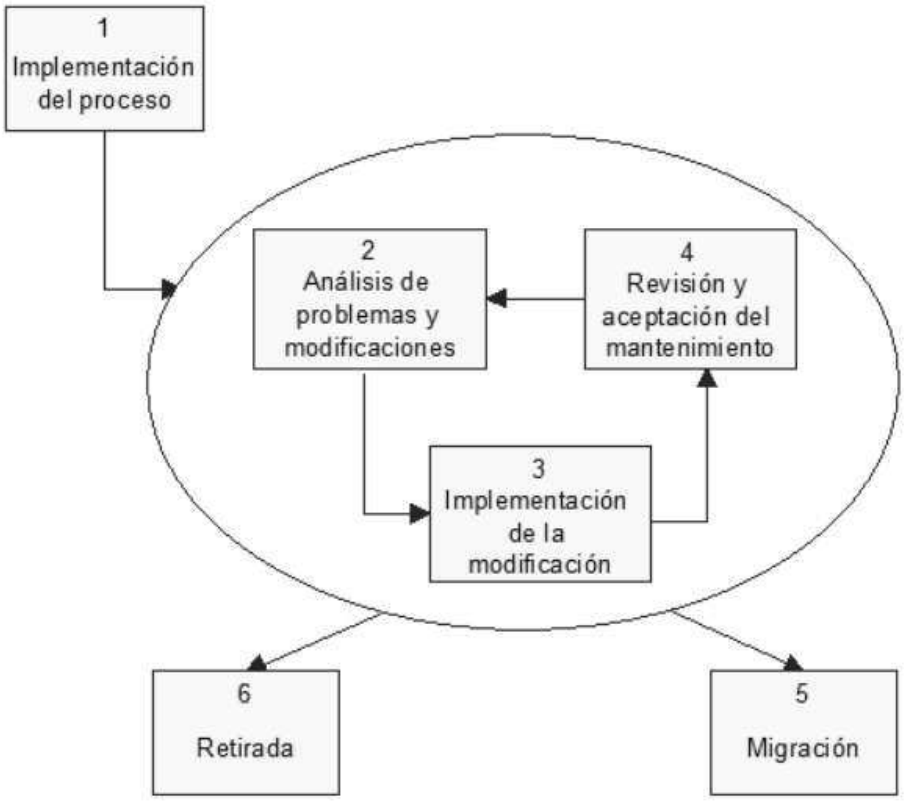


Figura 3.2. Proceso de mantenimiento de ISO/IEC 14764

3.3.1 Implementación del proceso

La organización de mantenimiento establece los planes y procedimientos que serán ejecutados durante el proceso de mantenimiento. Consta de tres tareas:

1. Desarrollar, documentar y ejecutar planes y procedimientos para dirigir las actividades y tareas del Proceso de Mantenimiento.

2. Definir procedimientos para recibir, almacenar y controlar los informes de problemas y solicitudes de modificación de los usuarios, y proporcionar a éstos retroalimentación.

3. Implementar o establecer una interfaz con el Proceso de Gestión de Configuración, para gestionar las modificaciones del sistema existente.

Como salida de la implementación del proceso se obtiene:

- Plan de mantenimiento.
- Plan de entrenamiento.
- Procedimientos de mantenimiento.
- Plan de gestión de proyectos.
- Procedimientos de resolución de problemas.
- Plan de medición.
- Manual de mantenimiento.
- Planes para comentarios del usuario.
- Plan de transición.
- Evaluación de la sustentabilidad.
- Plan de gestión de la configuración.

3.3.2 Análisis de problemas y modificaciones

En esta actividad, la organización de mantenimiento analiza las peticiones de modificación e informes de problemas, replica o verifica el problema, desarrolla alternativas para implementar la modificación, documenta los resultados y opciones de implementación, y obtiene la aprobación para ejecutar la alternativa seleccionada.

En la tarea de análisis de la petición de modificación o informe del problema, la organización de mantenimiento debe indicar el tipo de mantenimiento de la futura intervención; así como la magnitud del cambio (por ejemplo: tamaño de la modificación, coste, tiempo necesario).

3.3.3 Implementación de la modificación

La organización de mantenimiento desarrolla y prueba las modificaciones realizadas sobre el producto software. Las tareas que componen esta actividad son las siguientes:

- Dirigir y determinar análisis para determinar qué documentación, unidades de software y versiones necesitan ser modificadas.
- Entrar en el Proceso de Desarrollo para implementar las modificaciones.
- Realizar una revisión conjunta de la modificación realizada.
- Documentar y asegurar la calidad de la modificación.

3.3.4 Revisión y aceptación del mantenimiento

En esta actividad se asegura que las modificaciones realizadas sobre el sistema son correctas. Para su ejecución, deben realizarse revisiones con la organización que autorizó la modificación y obtener la aprobación correspondiente.

3.3.5 Migración

Esta actividad se realiza cuando el sistema tiene que ser modificado para ser ejecutado en un nuevo entorno. El encargado de mantenimiento ha de definir las tareas necesarias para conseguir la migración y por lo tanto desarrollar y documentar los pasos requeridos para una migración efectiva.

3.3.6 Retirada

Se realiza cuando el producto software ha alcanzado el final de su vida útil. Se puede llevar a cabo un análisis para tomar la decisión o no de la retirada del sistema software. El análisis debería determinar desde un punto de vista de la eficacia económica respecto a:

- Preservar tecnologías obsoletas
- Cambiar a tecnologías modernas mediante el desarrollo de un nuevo producto o servicio software
- Desarrollar un nuevo software para lograr algunos de los siguientes puntos:
  - Modularidad
  - Estandarización
  - Facilitar el mantenimiento
  - Independencia de proveedor

3.4 LECTURAS RECOMENDADAS

- ISO (2017). Information technology – *Software life cycle processes*. International Standard ISO/IEC 12207. Ginebra, International Organization for Standardization.

  Recomendamos la lectura completa del estándar ISO 12207, ya que aporta una visión integradora de los procesos del ciclo de vida software.

3.5 EJERCICIOS

Ejercicio 1

¿En qué lugar del ciclo de vida incluiría el proceso de mantenimiento?

Ejercicio 2

Elabore un diagrama en el que se muestren con detalle las interacciones del proceso de mantenimiento con el resto de procesos principales, así como con los procesos de soporte y organizacionales según la norma ISO 12207.

Ejercicio 3

Defina un procedimiento para la gestión de peticiones de modificación.

Ejercicio 4

¿En función de qué parámetros adaptaría el proceso de mantenimiento de la norma ISO/IEC 12207?

Ejercicio 5

Proponga un entorno para la gestión del soporte logístico para el proceso de mantenimiento.

Ejercicio 6

Compare la propuesta de la norma ISO/IEC 12207 para la gestión de incidencias y problemas en el proceso de mantenimiento, con la que propugna la familia de normas ISO/IEC 20000 para los sistemas de gestión de servicios.

Ejercicio 7

Elabore una lista de control (*checklist*) que permita caracterizar la madurez para el proceso de mantenimiento de software definido en la norma ISO/IEC 12207.

Ejercicio 8

¿Cuáles son las actividades del proceso de mantenimiento de acuerdo a la norma ISO/IEC 14764?

Ejercicio 9

Estudie la correspondencia entre las actividades de la norma ISO/IEC 14764 y el proceso de mantenimiento de la norma ISO/IEC 12207.

Ejercicio 10

Analice cómo se trata el proceso de mantenimiento en otros modelos como CMMI-DEV.

4

METODOLOGÍAS PARA EL MANTENIMIENTO

La mayor parte de metodologías de Ingeniería del Software en la actualidad consideran, en mayor o menor medida, aspectos del mantenimiento software, pero no han sido pensadas ni diseñadas específicamente para el proceso de mantenimiento. En este capítulo se ofrece una visión de las metodologías que proponemos para llevar a cabo el proceso de mantenimiento software.

4.1 MANTEMA: UNA METODOLOGÍA PARA EL MANTENIMIENTO DE SOFTWARE

MANTEMA es una metodología para mantenimiento de software que integra todas las actividades relacionadas con este proceso, cuyo objetivo es convertir el mantenimiento de software en un proceso controlable y mensurable mediante la identificación y definición clara de todos los elementos (software, documentos, personas, tareas...) que intervienen en el mantenimiento.

MANTEMA aborda por entero el problema del mantenimiento, intentando disminuir los costes de todas sus actividades, para lo cual es necesario la definición clara del conjunto de actividades que se deben ejecutar a lo largo del proceso de mantenimiento. Por ello, en MANTEMA se definieron inicialmente cinco tipos diferentes de mantenimiento, cada uno con un conjunto diferentes de tareas:

1. Correctivo urgente, que tiene lugar cuando existe un error en el software que bloquea el funcionamiento normal de la organización o de la aplicación, siendo crítico el tiempo de solución.

2. Correctivo no urgente, que ocurre cuando existe un error en el software que no es crítico, pero que tal vez impida el funcionamiento de la aplicación o el normal funcionamiento de la empresa en un periodo de tiempo relativamente corto.

3. Perfectivo, que tiene lugar cuando se van a añadir nuevas características o funcionalidades al software en explotación.

4. Adaptativo, que se aplica cuando el software en explotación va a cambiarse para que continúe funcionado correctamente en un entorno cambiante.

5. Preventivo, que es aplicado cuando se desean mejorar las características internas de un producto para hacerlo, por ejemplo, más fácilmente mantenible.

Sin embargo, durante la aplicación de la metodología se observó que, a pesar de que las intervenciones de correctivo no urgente, perfectivo, preventivo y adaptativo poseen características propias que las diferencian unas de otras, sus líneas de diseño y ejecución son bastante similares, por lo que se decidió agrupar estos cuatro tipos de mantenimiento bajo una única denominación, «mantenimiento planificable», pasando a denominar «no planificable» al correctivo urgente.

Además de detallar la secuencia de operaciones que se debe ejecutar para las intervenciones de cada tipo de mantenimiento, MANTEMA hace una especial consideración a la puesta en marcha del proceso de mantenimiento, identificando y definiendo dos conjuntos adicionales de actividades:

- En el primero, «Actividades y tareas iniciales comunes», se engloban todas las actividades que deben ser ejecutadas al comenzar el proceso de mantenimiento y que serán anteriores a la ejecución de cualquier intervención.

- El segundo, «Actividades y tareas finales comunes», agrupa las actividades que serán realizadas tanto al finalizar el proceso de mantenimiento como aquellas que serán ejecutadas con posterioridad a las intervenciones, independientemente de su tipo.

Con este planteamiento, el proceso de mantenimiento definido por MANTEMA puede entenderse como el grafo polietápico que mostramos en la Figura 4.1.

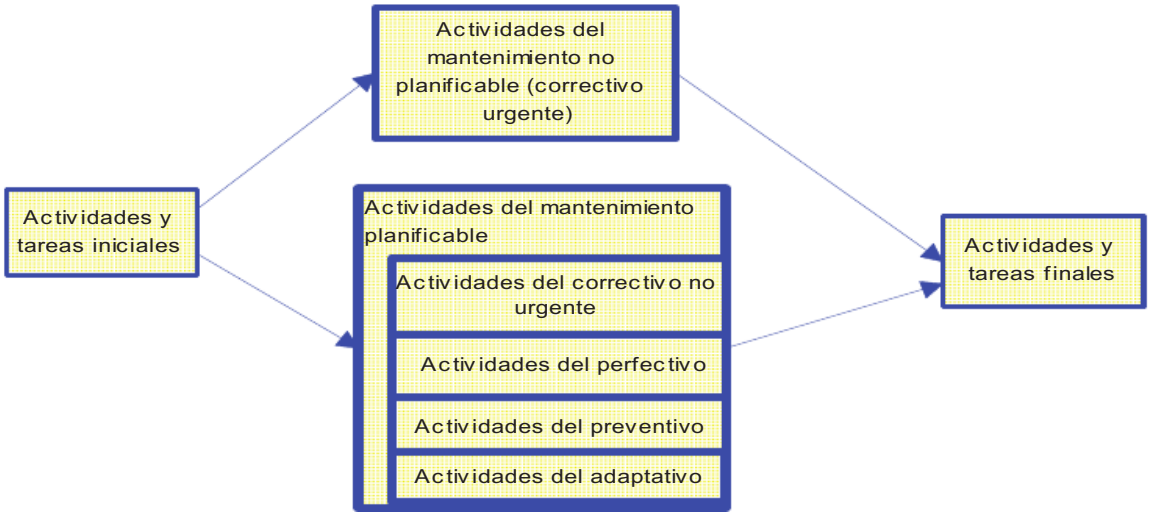


Figura 4.1. Vista general de la metodología MANTEMA

4.1.1 Descripción de las tareas

En MANTEMA, cada tarea se detalla especificando sus:

- Entradas, que son los elementos necesarios para la correcta ejecución de la tarea. Estos elementos podrán ser programas, documentos, etc., y ser tomados bien de tareas anteriores, bien del entorno del proceso.
- Salidas, que son los elementos que se generan tras la ejecución de la tarea. Las salidas podrán ir dirigidas a otras tareas posteriores o bien al entorno.
- Técnicas, que son las técnicas que pueden utilizarse para ejecutar la tarea.
- Métricas, que deben recogerse para mantener el proceso y el producto bajo control.
- Responsables, representados por los roles encargados de la ejecución de la tarea, y que serán algunos de los enumerados en [Polo *et al.*, 1999].
- Interfaces con otros procesos, que se establecerán durante la ejecución de la tarea con el resto de procesos definidos por la organización para el ciclo de vida software (por ejemplo, con el proceso de «Gestión de la configuración»).

Al hilo de este último punto, en el proceso de mantenimiento definido en la metodología se integran las actividades necesarias para el mantenimiento de algunos de los procesos del ciclo de vida definidos en ISO/IEC 12207. De este modo, la relación del proceso de mantenimiento descrito en MANTEMA con el resto de procesos del ciclo de vida software se puede representar con la Figura 4.2, que muestra, como procesos satélites al mantenimiento, aquellos que no se han integrado y que son con los que se establece interfaz en algún momento.

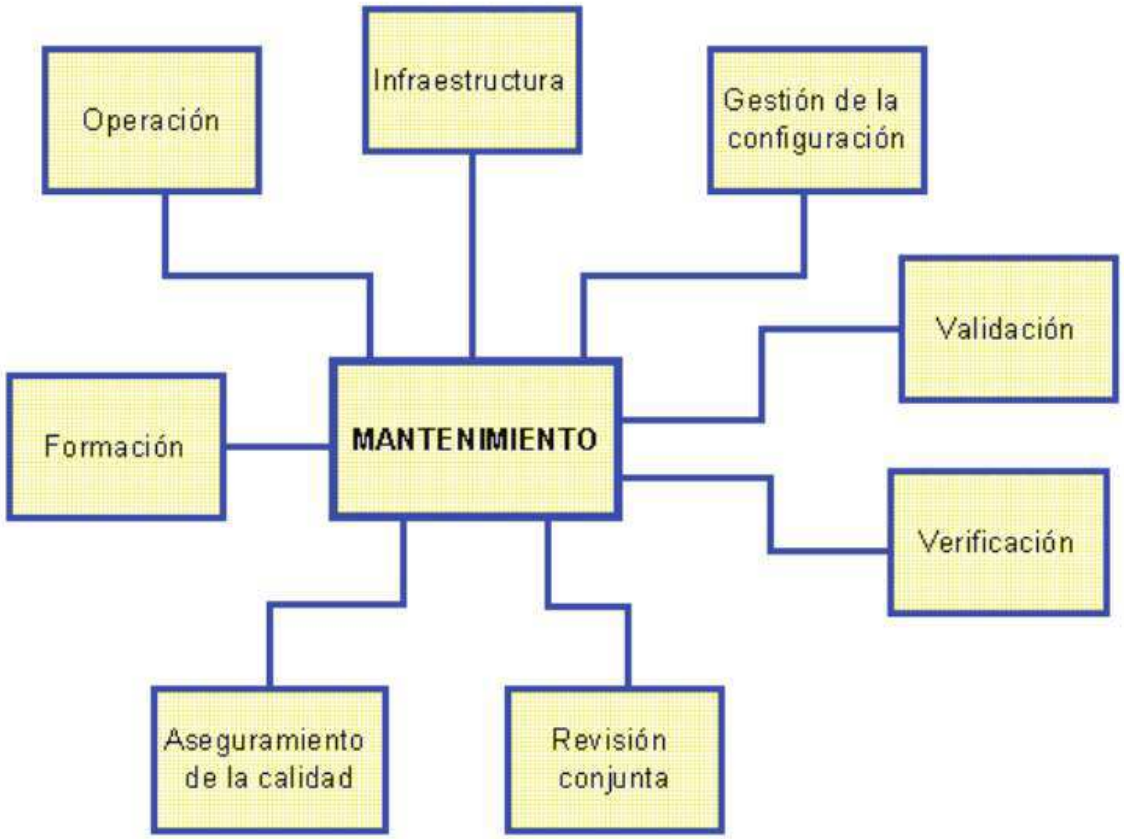


Figura 4.2. Procesos con los que se establece interfaz.

4.1.2 Estructura detallada de MANTEMA

En esta sección describimos más detalladamente los nodos que componen la metodología MANTEMA (veáse la Figura 4.1).

4.1.2.1 ACTIVIDADES Y TAREAS INICIALES COMUNES

Este conjunto de actividades está representado por el nodo inicial del grafo mostrado en la Figura 3.1. Consta de tres actividades. En la ejecución de las dos primeras (Estudio inicial y Planificación del proceso) se prepara el proceso de mantenimiento, mientras que la tercera está dedicada a la recepción y clasificación de las peticiones de modificación.

Durante la primera actividad (Estudio inicial), la organización de mantenimiento recoge información acerca del software que se va a mantener, con el objeto de realizar la planificación del proceso de mantenimiento y, en su caso, de presentar una propuesta de mantenimiento a la organización Cliente.

Una vez que la información anterior está recopilada, las dos siguientes tareas están dedicadas a la preparación de la propuesta de mantenimiento y a la redacción y firma del contrato de prestación del servicio, ambas muy importantes en el caso de que exista una relación de externalización entre la organización de Mantenimiento y el Cliente.

Concluida la primera actividad, durante la segunda de este nodo inicial se realiza la Planificación del proceso de mantenimiento. Durante esta actividad, la organización de mantenimiento adquiere conocimiento de la aplicación. En los casos en que existe externalización, la organización de mantenimiento actúa durante esta tarea simplemente observando cómo ejecuta sus tareas el actual equipo de mantenimiento. Tras este periodo «mudo», la organización de mantenimiento debe entregar al cliente documentación completa del software que incluya informes de auditoría, posibles mejoras, etc., además de ir construyendo documentación de consumo interno que incluya valores de métricas, tablas de referencias cruzadas, diccionarios de datos, etc.

Dentro todavía de la segunda actividad, se definen los procedimientos que deberán seguirse para presentar las peticiones de modificación, se implementa el proceso de gestión de configuración (en caso de que se carezca de uno) y, deseablemente, se preparan los entornos en que se realizarán las pruebas.

A partir del momento en que queda terminada la segunda actividad, la organización de mantenimiento está preparada para ejecutar las acciones necesarias para servir las peticiones de modificación. Entraríamos, por tanto, en un conjunto de actividades y tareas cíclico, en el sentido de que serán ejecutadas para cada Petición de Modificación que se reciba.

Precisamente la tercera y última actividad de este conjunto inicial de actividades y tareas está dedicada a la recepción de la Petición de Modificación y a su clasificación según uno de los dos tipos de mantenimiento que distinguimos en la sección 1 (aunque, para las peticiones de mantenimiento planificable, será necesario seguir precisando si se tratan de correctivo no urgente, perfectivo, preventivo o adaptativo, ya que la forma de actuar difiere ligeramente en cada caso).

4.1.2.2 ACTIVIDADES Y TAREAS DEL MANTENIMIENTO NO PLANIFICABLE (CORRECTIVO URGENTE)

Detallamos en la Tabla 4.1 y la Tabla 4.2 la serie de actividades y tareas que deben seguir las intervenciones de mantenimiento que, en la tercera actividad del nodo inicial del proceso, se hayan clasificado como correspondientes a mantenimiento «no planificable» (correctivo urgente).

Actividad	Análisis del error	Intervención correctiva urgente (continúa)	
Tarea	NP1.1 Investigar y analizar causas	NP2.1 Realizar acciones correctivas	NP2.2 Cumplimentar documentación
Entradas	Producto software en explotación con error bloqueante o crítico. Petición de modificación	Conjunto de elementos software a corregir	Elementos software antiguos (con errores visibles). Elementos software corregidos
Salidas	Conjunto de elementos software a corregir	Conjunto de elementos software corregidos	Documentación de las acciones correctivas realizadas
Técnicas		Codificación	
Responsable	Equipo de mantenimiento Usuario	Equipo de mantenimiento	Equipo de mantenimiento
Interfaces con otros procesos		Aseguramiento de la calidad. Gestión de la configuración	

Tabla 4.1. Estructura del mantenimiento no planificable

Actividad	Intervención correctiva urgente (continuación)	Cierre intervención
Tarea	NP2.3 Ejecutar pruebas unitarias	NP3.1 Pasar a producción
Entradas	Elementos software corregidos Casos de prueba	Elementos software corregidos y probados
Salidas	Elementos software corregidos y probados Documentación con las pruebas unitarias realizadas	Producto software en explotación corregido
Técnicas	Técnicas de prueba del software	
Responsable	Equipo de mantenimiento	Equipo de mantenimiento Usuario
Interfaces con otros procesos	Aseguramiento de la calidad	Gestión de la configuración

Tabla 4.2. Estructura del mantenimiento no planificable (continuación)

4.1.2.3 ACTIVIDADES Y TAREAS DEL MANTENIMIENTO PLANIFICABLE

En este tipo de mantenimiento se incluye la definición de las actividades y tareas para los mantenimientos correctivo no urgente, perfectivo, preventivo y adaptativo. Cada nodo de la Figura 4.3 representa una tarea de este tipo de mantenimiento. Como se puede observar, la secuencia de tareas seguida por las intervenciones correctivas no urgentes y perfectivas es exactamente la misma, mientras que existen algunas diferencias entre éstas con las preventivas y adaptativas.

Los nodos de la Figura 4.3 se pueden detallar utilizando tablas del estilo de las usadas para el no planificable, de manera que también para estas tareas se especifican entradas, salidas, técnicas, etc.

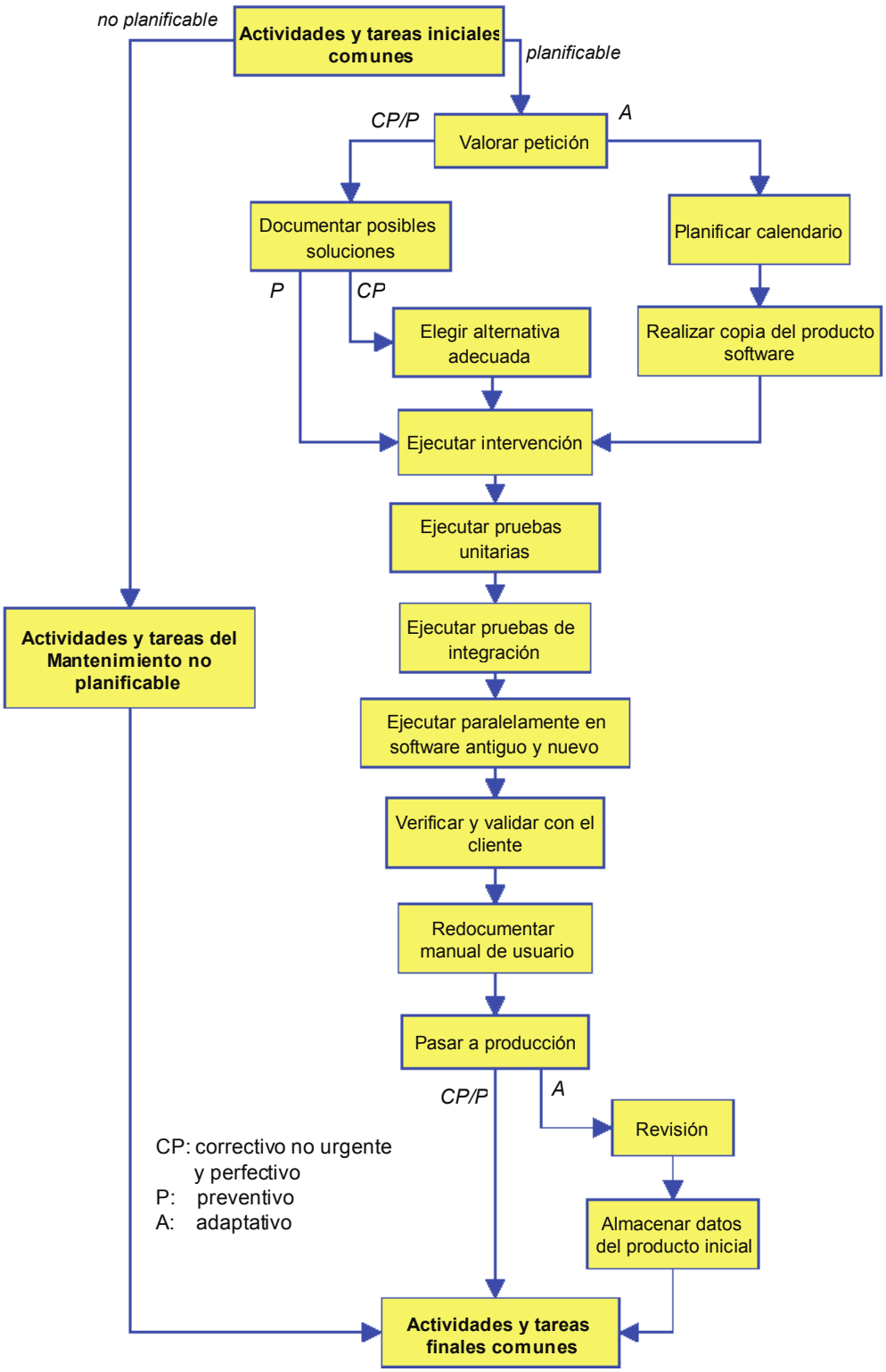


Figura 4.3. Estructura del mantenimiento planificable

4.1.2.4 ACTIVIDADES Y TAREAS FINALES COMUNES

Este nodo consta de las siguientes cuatro actividades:

- Registro de la intervención, tras la cual la intervención (incluyendo toda su documentación asociada) queda registrada según los procedimientos que se establecieran en la actividad «Planificación del proceso» del conjunto de actividades y tareas iniciales comunes.
- Actualización de la base de datos histórica, que consiste en almacenar (si no se ha hecho ya) los valores de las diferentes métricas que deben recogerse en cada tarea.
- Retirada, que será realizada conforme al proceso homónimo de la norma ISO/IEC 12207.
- Fin de la externalización, que ocurrirá si ha existido relación de *outsourcing* entre la Organización de Mantenimiento y el Cliente.

4.1.2.5 DOCUMENTACIÓN

Gran parte de los elementos que las tareas utilizan como entradas o salidas son documentos que deben ser generados durante el proceso. En la metodología se definen plantillas con los contenidos de todos o casi todos los documentos generables durante el proceso. En la Tabla 4.3 se recogen algunos de ellos.

4.1.2.6 MÉTRICAS

En MANTEMA definimos una larga serie de métricas que deben ser recogidas con cada intervención de mantenimiento y que se incorporan, para posteriores análisis, a una base de datos histórica en la segunda tarea de la última actividad.

Es interesante recoger métricas tanto de producto como de proceso. Entre las primeras, aparte de las clásicas, más conocidas y más útiles en el proceso de mantenimiento y Piattini et al. (2018).

1. Cuestionario inicial	10. Alternativas de implementación
2. Propuesta de mantenimiento	11. Acciones perfectivas realizadas
3. Contrato de mantenimiento	12. Lista de elementos software y propiedades mejorables
4. Tabla de factores de riesgo	13. Acciones preventivas realizadas
5. Resumen técnico	14. Plan de migración
6. Petición de modificación	15. Notificación de futura migración
7. Acciones correctivas realizadas	16. Medidas del producto
8. Pruebas unitarias realizadas	17. Plan de mantenimiento del periodo
9. Diagnóstico y posibles soluciones	

Tabla 4.3. Algunos de los documentos de la metodología MANTEMA

4.2 ÁGIL MANTEMA

Ágil_MANTEMA está creado a partir de la agilización de MANTEMA a través de la aplicación de Scrum. La propuesta metodológica Ágil_MANTEMA está enfocada sobre todo a medianas y pequeñas organizaciones, y pretende definir un proceso de mantenimiento (descripción, desglose de trabajo, roles, paquetes de trabajo, etc.), detallando qué debe realizarse, cuándo, cómo y por quién, es decir, busca guiar paso a paso el proceso de mantenimiento del software en este tipo de organizaciones. En [Pino et al., 2012] se ofrece una visión clara y concisa de esta metodología.

4.2.1 Estructura General de Ágil MANTEMA

La estructura general de Ágil MANTEMA está ideada pensando en ayudar a las pymes que desean disponer de una guía metodológica para llevar a cabo el proceso de mantenimiento de software. Dicha estructura, presentada en la Figura 4.4, se basa en la estructura de MANTEMA y en las indicaciones de la norma ISO/IEC 12207.

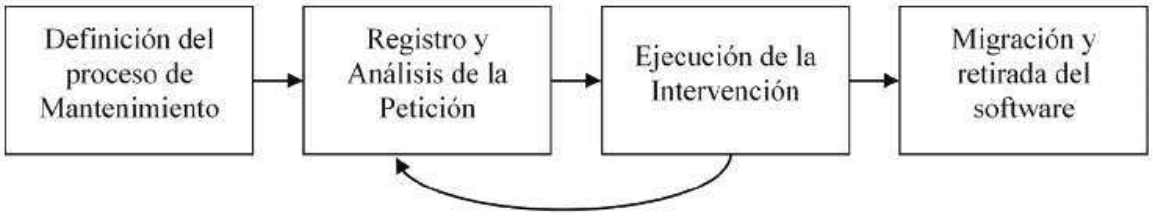


Figura 4.4. Estructura general de Ágil_MANTEMA

La estructura general, o modelo de proceso, presentada en la figura anterior se complementa con los siguientes elementos, que también se incluyen en Ágil_MANTEMA:

- Niveles de servicio extraída de Métrica V3 y adaptada a esta metodología.
- Nivel de capacidad del proceso basado en la norma ISO/IEC 15504-2 [ISO/IEC, 2003], y adaptada también a esta metodología.

Estos dos conceptos se introducen y relacionan en Ágil_MANTEMA mediante una representación bidimensional, de forma que en una dimensión se encuentran los niveles de servicio y en la otra dimensión se encuentran los niveles de capacidad del proceso de mantenimiento (ver Tabla 4.4).

		Niveles de Servicio		
		Básico	**Intermedio**	**Avanzado**
Niveles de Capacidad	**Uno**			
	Dos			
	Tres			

Tabla 4.4. Representación bidimensional de Ágil_MANTEMA

El propósito de ofrecer esta representación bidimensional es que cualquier pyme pueda manejar mejor la complejidad inherente al proceso de mantenimiento de software, mediante una adaptación en función de sus características organizacionales y sus objetivos de negocio. La intención es que pueda elegir qué nivel de servicio y con qué nivel de capacidad desea implementar su propio proceso de mantenimiento, de acuerdo a sus necesidades e infraestructura.

Como se observa de la Tabla 4.4, Ágil_MANTEMA plantea que el nivel de servicio básico solo tiene un nivel de capacidad, el nivel de servicio intermedio puede realizarse con dos niveles de capacidad diferentes, y el nivel de servicio avanzado puede llevarse a cabo con tres niveles de capacidad.

4.2.1.1 NIVELES DE SERVICIO DE ÁGIL_MANTEMA

Ágil_MANTEMA define tres niveles de servicio (ver Tabla 4.5): el nivel básico abarca el mantenimiento correctivo urgente; el nivel intermedio añade al anterior el correctivo no urgente y el perfectivo, y el nivel avanzado abarca todos los tipos de mantenimiento, incorporando a los del nivel anterior los mantenimientos adaptativo y preventivo. Los tipos de mantenimiento se explican en el apartado 2.4.

	Nivel Básico	**Nivel Intermedio**	**Nivel Avanzado**
Tipos de Mantenimiento	- Correctivo Urgente	- Correctivo Urgente - Correctivo No Urgente - Perfectivo	- Correctivo Urgente - Correctivo No Urgente - Perfectivo - Adaptativo - Preventivo
Interfaz fundamental	- Soporte al cliente - Gestión de resolución de problemas	- Soporte al cliente - Gestión de resolución de problemas - Gestión de la Configuración - Aseguramiento de la Calidad	- Soporte al cliente - Gestión de resolución de problemas - Gestión de la Configuración - Aseguramiento de la Calidad - Gestión de cambio de requisitos - Gestión de proyectos

Tabla 4.5. Niveles de servicios definidos en Ágil_MANTEMA

La Tabla 4.5 también muestra las interfaces que tiene cada nivel de servicio de mantenimiento con procesos que brindan actividades y productos de trabajo de soporte y gestión (llamados procesos auxiliares), a través de los cuales se pretende incrementar la capacidad del proceso de mantenimiento. Cada una de las interfaces define una serie de actividades y productos de trabajo de tipo organizativo o de soporte al proceso de mantenimiento (que depende del tipo de mantenimiento a realizar). Esto permite a la organización realizar otras actividades complementarias al proceso de mantenimiento.

4.2.1.2 NIVELES DE CAPACIDAD DE ÁGIL_MANTEMA

La implementación de las actividades descritas por las interfaces permite llevar a cabo prácticas base y de gestión que incrementan el nivel de capacidad del proceso de mantenimiento. Es por ello que, siguiendo esta estrategia, en Ágil_MANTEMA se establecen tres niveles de capacidad para el proceso de mantenimiento, en función de las interfaces implementadas (ver Tabla 4.6).

Nivel de Capacidad	Proceso auxiliar (Interfaces)	Grado esperado de cumplimiento
Nivel Uno	Soporte al cliente	AI o CI
	Gestión de resolución de problemas	AI o CI
Nivel Dos	Soporte al cliente	CI
	Gestión de resolución de problemas	CI
	Gestión de la configuración	AI o CI
	Aseguramiento de calidad	AI o CI
Nivel Tres	Soporte al cliente	CI
	Gestión de resolución de problemas	CI
	Gestión de la configuración	CI
	Aseguramiento de calidad	CI
	Gestión de cambio de requisitos	AI o CI
	Gestión de proyectos	AI o CI

Tabla 4.6. Niveles de capacidad del proceso de mantenimiento Ágil_MANTEMA

Cada uno de los procesos auxiliares (interfaces) que se relacionan en la Tabla 4.6 tienen una escala específica para su medición, las prácticas base de estos procesos se valoran con una escala discreta compuesta por los elementos:

- CI: completamente implementado. La práctica base se cumple entre un 86% y 100 %.
- AI: ampliamente implementado. La práctica base se cumple entre un 51% y 85%.
- PI: parcialmente implementado. La práctica base se cumple entre un 16% y 50%.
- NI: no implementado. La práctica base se cumple entre un 0% y 15%.

El grado esperado de cumplimiento del proceso auxiliar se obtiene de promediar el valor de las prácticas base descritas por cada proceso. Por ejemplo, se puede tener un proceso de mantenimiento con nivel de servicio básico y nivel de capacidad 1, lo cual indica que el tipo de mantenimiento que se realiza es el correctivo urgente y además se implementan ampliamente las interfaces Soporte al cliente y Gestión de resolución de problemas. Cada organización puede escoger el nivel de servicio de mantenimiento que quiere implementar y el nivel de capacidad, teniendo en cuenta sus necesidades y características propias.

4.2.1.3 TIPOS DE MANTENIMIENTO

Los tipos de mantenimiento en Ágil_MANTEMA son los identificados en MANTEMA, ya que éste no es un factor que se vea afectado por la búsqueda de mayor agilidad. Dependiendo de las características de cada organización, así como de las circunstancias particulares de cada proyecto o servicio de mantenimiento concreto, se determinarán los tipos de mantenimiento que serán soportados, respetando siempre la estructura establecida por los niveles de servicio. Así, en Ágil_MANTEMA no se recomienda soportar el mantenimiento correctivo urgente y el correctivo no urgente sin soportar también el perfectivo (ver nivel de servicio intermedio).

A continuación, se describen los tipos de mantenimiento en Ágil_MANTEMA organizados en las categorías de planificable y no planificable. Se pretende con esta división lograr una mejor gestión y optimización del Registro de Peticiones[4], ofreciendo un criterio al Gestor de Peticiones (rol que se encarga de ordenar las peticiones de modificación) para clasificar y priorizar las peticiones de mantenimiento.

4 Registro de Peticiones es un grupo clasificado, ordenado y priorizado de peticiones de mantenimiento.

1. Mantenimiento No Planificable.
   - Correctivo urgente (Nivel Básico): Es aquel que se da en situaciones en que existe un error en el producto software que bloquea la aplicación o el proceso de funcionamiento de la empresa, que debe ser resuelto con brevedad (por ejemplo, el día veintiocho falla la aplicación de cálculo de nóminas). Estas peticiones deben ser rápidamente atendidas.
2. Mantenimiento Planificable.
   - Correctivo no urgente (Nivel Intermedio): Se produce cuando existe un error en el producto software que no es crítico, pero que tal vez impida el funcionamiento de la aplicación o el normal funcionamiento de la empresa en un periodo de tiempo relativamente corto (por ejemplo, el fallo en la aplicación de nóminas se produce el día diez).
   - Perfectivo (Nivel Intermedio): Se ocupa de añadir al software en explotación nuevas características o funcionalidades, habitualmente solicitadas por el cliente.
   - Adaptativo (Nivel Avanzado): Se aplica cuando el software en explotación va a cambiarse para que continúe funcionado correctamente en un entorno cambiante. Un caso típico es la adaptación a un nuevo sistema operativo, o el cambio del sistema de gestión de la base de datos.
   - Preventivo (Nivel Avanzado): Es aplicado cuando se desea mejorar las características internas de un producto software buscando que en un futuro el esfuerzo de mantenimiento sea menor. Un ejemplo típico es la aplicación al código de técnicas de refactorización o la revisión y mejora de los comentarios.

4.2.2 Descripción del Proceso de Mantenimiento

A continuación, se presenta el proceso de mantenimiento propuesto por Ágil_MANTEMA, en el cual primero se muestran los participantes de este proceso y luego se ofrece una visión general del proceso. Finalmente se presenta la estructura detalla de la metodología.

4.2.2.1 PARTICIPANTES EN EL PROCESO DE MANTENIMIENTO

Ágil_MANTEMA define a sus participantes en un modelo basado en roles, esto quiere decir, que cada organización define a unos integrantes con un

conjunto de tareas y responsabilidades para que se desempeñe dentro del proceso de mantenimiento. Estas tareas y responsabilidades se presentan a continuación.

1. Cliente: Es la organización propietaria, por lo tanto, es quien recibe el servicio de mantenimiento.
   - Propietario del Producto: Representa a todos los interesados en el producto final. Sus áreas de responsabilidad son: La financiación del proyecto, retorno de la inversión del proyecto y el lanzamiento del proyecto. El propietario del producto por lo general formula peticiones de modificación del tipo perfectivo o adaptativo.
2. Usuario: Es quien utiliza el software. Propone las peticiones de modificación correctivas (urgentes o no urgentes) y perfectivas.
3. Mantenedor: Es quien realiza la modificación del software.
   - Gestor de Peticiones: Es quien acepta o rechaza las peticiones de modificación y decide el tipo de mantenimiento que corresponde. En caso de ser perfectivo pone al tanto al propietario del producto (cliente) para ver la viabilidad del mantenimiento. En caso de ser cualquiera de los otros tipos, sitúa la petición en la Lista de Espera de Peticiones, asignándole una prioridad.
   - Responsable de Mantenimiento: Es quien prepara el proceso y establece las normas y procedimientos necesarios para aplicar la metodología. Interactúa con el cliente y el equipo de mantenimiento, y es el responsable de que se lleven a cabo las prácticas, valores y reglas de Scrum. Además, es miembro del equipo de mantenimiento y trabajar a la par con el resto de miembros, coordina los encuentros permanentes del equipo, y se encarga de eliminar eventuales obstáculos.
   - Equipo de Mantenimiento: Es el grupo de personas que implementa las peticiones de mantenimiento. Tiene autoridad para reorganizarse y definir las acciones necesarias o sugerir remoción de impedimentos. También puede proponer peticiones de mantenimiento preventivo.

4.2.2.2 VISIÓN GENERAL DEL PROCESO DE MANTENIMIENTO

El proceso de mantenimiento comienza cuando el mantenedor y el cliente inician a trabajar en conjunto, es decir, asignan responsables, criterios y explicación de cómo se va a trabajar. Estas tareas se agrupan en la actividad llamada Planificación del proceso (véase Figura 4.5).

Al concluir la actividad descrita anteriormente se inicia el ciclo que cada petición de mantenimiento tendrá que seguir. La primera actividad a realizar en el ciclo es la Atención de la petición mediante la cual se formula o recibe la petición de mantenimiento, que luego pasa a manos del Gestor de Peticiones que se encargará de asignar el tipo y prioridad de la petición. El grupo ordenado de peticiones se llama Registro de Peticiones. Desde este registro se selecciona un primer grupo de peticiones llamado Lista de Espera de Peticiones, este grupo puede entrar a dos diferentes tipos de Sprint de Mantenimiento (SprintM) [5], uno corto para las peticiones del tipo no planificable y otro más largo para las del tipo planificable. Dentro del SprintM se realizarán una serie de reuniones con el fin de obtener su estado de avance y posibles problemas que puedan ocurrir dentro de su ejecución. Cuando una petición ha sido resuelta se finaliza el ciclo con la actividad Finalizar intervención. La finalidad de esta actividad es la validación y verificación del producto por parte del cliente, el paso del producto a producción, registro de documentos y reuniones de retrospección.

Para finalizar el proceso de mantenimiento, cuando ya no se van a recibir más peticiones porque ya se acabó el tiempo del proyecto/servicio, se cuenta con una actividad final llamada Finalización del servicio, que sirve para una cesión de actividades por parte del mantenedor de forma que no repercuta negativamente en la organización cliente. En algunas ocasiones, antes de llevar a cabo la finalización es necesario realizar la actividad de Retirada con el fin de aplicar el plan de retirada del software.

Se puede observar que las actividades de Planificación del proceso, Retirada y Finalización del servicio, se desarrollarán tan solo una vez y no entorpecerán la agilidad del proceso. Por otro lado, están los dos tipos de SprintM que serán un conjunto repetitivo de actividades. Esto no quiere decir que los cambios no estén permitidos, por lo contrario, son bienvenidos y existe un mecanismo para incorporarlos gracias a que existe una retroalimentación rápida con el cliente, junto con una entrega rápida y periódica de atención a las peticiones de mantenimiento.

Es frecuente en las empresas que hacen mantenimiento distinguir entre "informe de problema" (o incidencia) y "solicitud de cambio", según se trata respectivamente de comunicar un fallo (mantenimiento correctivo urgente o no urgente) o de demandar un cambio cuyo origen no es un fallo (mantenimientos perfectivo, adaptativo y preventivo). Por sencillez, en Ágil_MANTEMA se ha optado por hablar sólo de "peticiones de modificación", englobando en ellas tanto a los informes de problemas como a las solicitudes de cambio.

---

5 SprintM: Ciclo de mantenimiento básico de duración recomendada dependiendo del tipo de mantenimiento (para correctivo urgente de entre uno y siete días, para los otros de entre ocho y quince días) en el que atiende y resuelve una petición de mantenimiento.

Teniendo en cuenta estas observaciones, podemos precisar un poco más el concepto de "proceso de mantenimiento" indicando que está formado por:

1. Un conjunto de actividades destinadas a preparar y planificar las actividades de mantenimiento.
2. El conjunto de intervenciones que producen modificaciones sobre el software.
3. Reuniones para medir el estado de avance y resolver problemas dentro de las intervenciones.
4. Un conjunto de actividades que deben realizarse con posterioridad a cada modificación sobre el software.
5. Opcionalmente:
   - Tareas que incorporen interfaces con los procesos auxiliares establecidos.
   - Una actividad que guíe en la finalización de la prestación del servicio de mantenimiento.

Con estas descripciones se genera el proceso de mantenimiento en Ágil_MANTEMA el cual podremos observar en la Figura 4.5 y que describiremos en el siguiente punto.

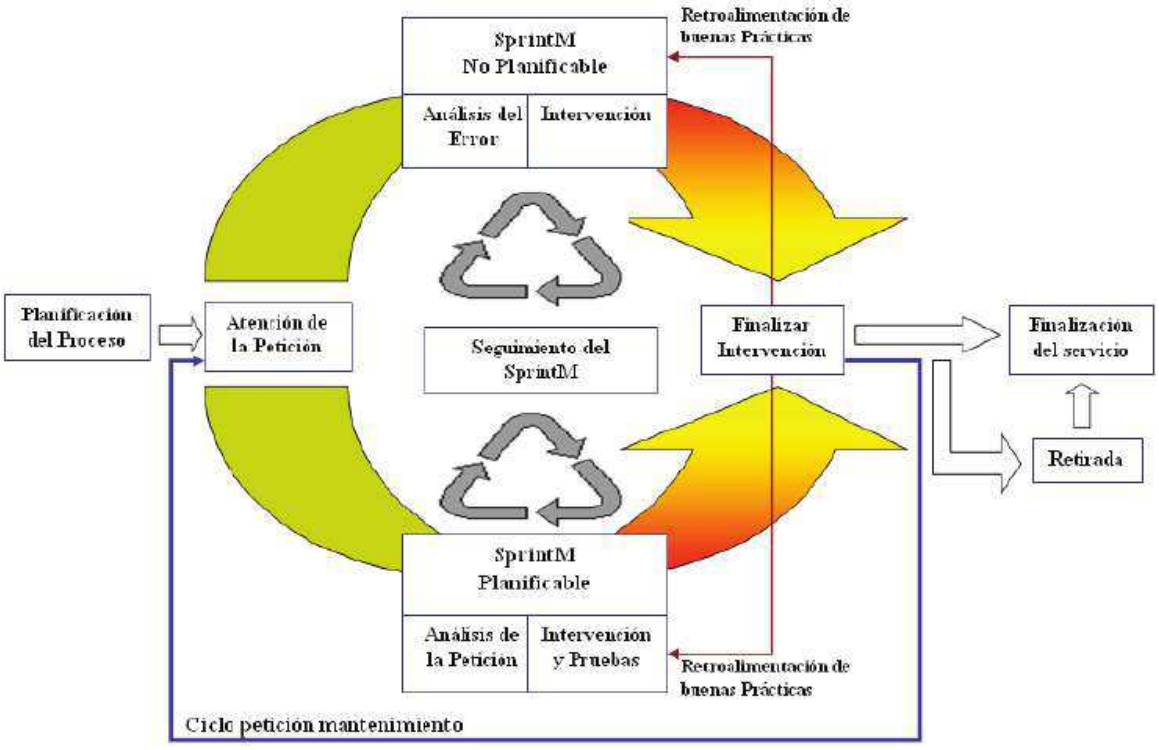


Figura 4.5. Proceso de Ágil_MANTEMA

4.2.2.3 ESTRUCTURA DETALLADA DE LA METODOLOGÍA

En esta sección dedicamos un epígrafe para cada actividad y tarea del ciclo de vida de Ágil_MANTEMA, entre las que podemos encontrar la planificación del proceso de mantenimiento, el análisis de la petición de modificación, el Sprint no Planificable, el Sprint Planificable (dependiendo del tipo de mantenimiento), el seguimiento del Sprint de Mantenimiento, y para concluir el ciclo con las actividades y tareas finales, en caso de que fuera necesario se incorporará la finalización del servicio. Del mismo modo, detallamos las entradas, salidas, personal responsable y técnicas de cada tarea, de manera que determinamos los "qué, cómo, dónde, cuándo y por qué" del proceso.

Muchos de los artefactos que las tareas toman como entradas y que producen como salida son documentos, que en el texto aparecerán etiquetados con el prefijo DOC seguido de un número.

4.2.2.3.1 Planificación del proceso

El propósito de esta actividad es llevar a cabo la planificación del proceso de mantenimiento. La actividad inicial planificación del proceso sólo se ejecuta cuando el cliente contacta con el mantenedor para que realice el proceso de mantenimiento. La descripción de esta actividad y las tareas correspondientes se muestran a continuación.

- **Tarea I0.1. Asignar responsables.** Se asignan los roles del proceso de mantenimiento a las personas involucradas en el mismo. El Propietario del Producto, el Gestor de Peticiones, el Responsable de Mantenimiento, y el Equipo se Mantenimiento están claramente establecidos e identificados en la organización.

- **Tarea I0.2. Adquirir conocimiento de la aplicación.** El Equipo de mantenimiento obtiene la información del software que se va a mantener. Luego el Equipo de mantenimiento estudia la documentación, código fuente, referencias cruzadas, se entrevista con los usuarios, observar cómo trabaja éste, con el fin de obtener el conocimiento necesario y suficiente del producto software a mantener. Durante el tiempo que dure esta tarea no se realiza ningún tipo de mantenimiento. Al finalizar esta tarea, el Equipo de mantenimiento entrega al Cliente un informe acerca del estado de su software, de manera que el cliente verifique que el Equipo de Mantenimiento ha adquirido un conocimiento adecuado del software.

- **Tarea I0.3. Preparar entornos de pruebas.** En esta tarea el Equipo de mantenimiento prepara el entorno de pruebas. Realiza copias del software, preparar la base de datos y archivos (el entorno), que sean semejantes a la realidad y que cubran la totalidad de las funcionalidades del sistema. El objetivo es que el software pueda funcionar en un ambiente aislado, para que no afecte la operación normal de los sistemas en uso.
- **Tarea I0.4. Definir procedimientos de petición de modificación.** El Equipo de mantenimiento genera el documento que el usuario presentara para solicitar mantenimiento. Dicho documento será llamado Petición de Modificación (DOC1). En esta tarea también se establecerá, junto con el cliente, los procedimientos de la Petición de Modificación, a los usuarios se les indicará quien será el receptor de dicha Petición llamado Gestor de Peticiones.

En la Figura 4.6 se representa el diagrama de actividades de la planificación del proceso.

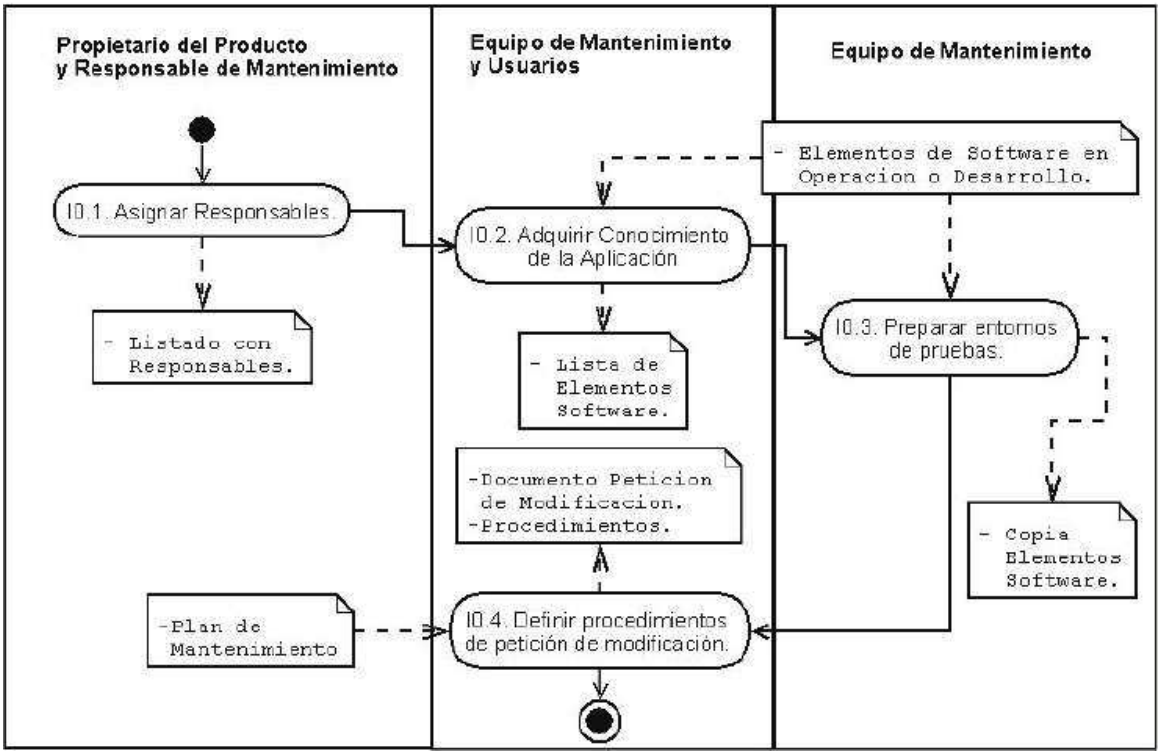


Figura 4.6. Diagrama de actividad de Planificación del proceso de mantenimiento

La Tabla 4.7 resume la actividad I0 de Planificación del Proceso.

Actividad I0 **Planificación del Proceso**						
	Entradas	**Salidas**	**Técnicas**	**Roles**	**Interfaces con otros Procesos**	**Nivel de Servicio**
Tarea I0.1 **Asignar Responsables**	Listado con posibles responsables	Listado con asignación de responsables a los roles		- Propietario del producto - Responsable de mantenimiento		Básico
Tarea I0.2 **Adquirir Conocimiento de la Aplicación**	Producto Software en operación o desarrollo	Lista de elementos software	- Estudio de la documentación - Observación y entrevistas	- Equipo de mantenimiento - Usuarios		Básico
Tarea I0.3 **Preparar Entornos de Prueba**	Elementos Software del sistema en operación	Copias de los elementos Software	Instalación de herramientas	Equipo de mantenimiento	Adaptación	Intermedio
Tarea I0.4 **Definir Procedimientos de la Petición de Modificación**	Plan de manteni - miento	Documento petición de modificación. Procedimientos		- Equipo de mantenimiento - Usuario	Gestión de la Configuración	Intermedio

Tabla 4.7. Resumen de la Actividad Planificación del Proceso.

4.2.2.3.2 Atención de la petición de modificación

El propósito de esta actividad es recibir una petición de modificación, a la cual se le asigna un tipo de mantenimiento y su prioridad, además se almacena en el Registro de peticiones ordenado por prioridad. En esta actividad comienza el ciclo que cada petición de modificación tendrá que seguir para ser atendida y se compone las tareas siguientes:

- **Tarea I1.1. Recibir petición de modificación.** El usuario entrega una Petición de modificación (DOC1), que es recibida y registrada por el Gestor de Peticiones. Este deberá asignar un identificador único a cada Petición de Modificación.

- **Tarea I1.2. Decidir el tipo de mantenimiento.** A partir de la Petición de modificación (DOC1) recibida y registrada en la tarea anterior el Gestor de Peticiones decide aceptar o rechazar la petición. Si la petición es aceptada se decide el tipo de mantenimiento que debe aplicarse y es agregada al Registro de Peticiones en orden de prioridad. Si la petición es rechazada se debe justificar las razones. Finalmente se analiza la relación entre las peticiones que están en el Registro de Peticiones y se decide cuales pueden abordarse en forma conjunta (Lista de Espera). También se realiza la estimación del esfuerzo necesario para llevar a cabo la petición de modificación.

En la Figura 4.7 se representa el diagrama de actividades de Atención de la petición de modificación.

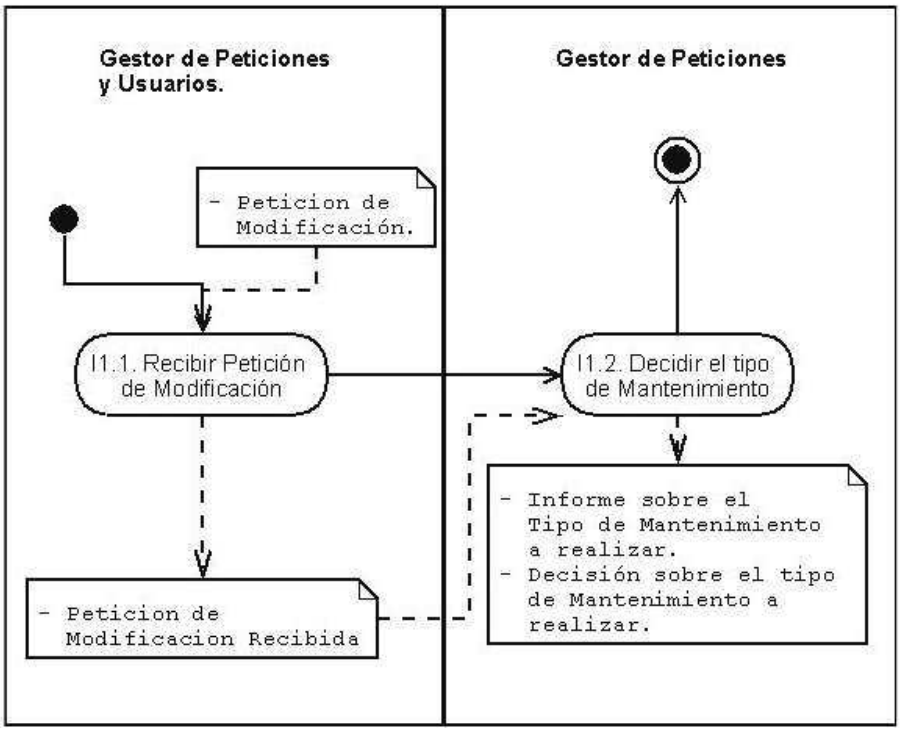


Figura 4.7. Diagrama de actividad de Atención de la petición de modificación

La Tabla 4.8 resume la actividad I1 de Atención de la Petición de Modificación.

Actividad I1 **Atención de la Petición de Modificación**						
	Entradas	**Salidas**	**Técnicas**	**Roles**	**Interfaces con otros Procesos**	**Nivel de Servicio**
***Tarea I1.1*Recibir Petición de Modificación.**	Petición de Modificación	Petición de modificación recibida y registrada		- Gestor de Peticiones. - Usuario	Soporte al cliente	Básico
Tarea I1.2 **Decidir el tipo de Mantenimiento**	Petición de Modificación Registrada	Informe sobre el tipo de mantenimiento necesario. Decisión sobre el tipo de mantenimiento a realizar.	- Grafico de quemado. - Tipos de Mantenimiento de Ágil_ MANTEMA. - Evaluación de Impacto	- Gestor de Peticiones	Aseguramiento de la Calidad	Avanzado

Tabla 4.8. Resumen de la Actividad Atención de la Petición de Modificación.

4.2.2.3.3 SprintM No Planificable

El propósito de esta actividad es brindar atención urgente a las peticiones de modificación que bloquean o interrumpen el funcionamiento del producto software. El Sprint del mantenimiento no planificable se ejecuta cuando se asume un mantenimiento correctivo urgente. Estas actividades se ejecutan cuando el error presentado en la petición de modificación paraliza de manera sería el funcionamiento normal del resto del sistema de información o el de la organización, de forma que la corrección del error deba ser inminente. Se recomienda la ejecución de SprintM cortos de entre uno y siete días (dependiendo del tipo de error) con reuniones todos los días.

Esta actividad está compuesta por dos sub-actividades que se describen a continuación.

- Actividad SNP1. Análisis del error
  - **Tarea SNP1.1. Investigar y Analizar Causas.** El Equipo de mantenimiento analiza la Petición de Modificación (DOC1), verifica el problema con la colaboración del Usuario que realizó la petición y lo reproduce. Además, estudia diferentes alternativas para implementar

la modificación para la corrección del error. También se construye una lista de los elementos software a corregir (módulos, rutinas, documentos, etc.).

- Actividad SNP2. Intervención correctiva urgente
  - **Tarea SNP2.1 Realizar acciones correctivas.** El equipo de mantenimiento ejecuta las acciones necesarias para corregir el problema detectado. Se debe identificar todos los componentes del producto software (rutinas, bases de datos, etc.) afectados por la intervención.
  - **Tarea SNP2.2 Ejecutar pruebas unitarias.** El Equipo de mantenimiento debe comprobar el correcto funcionamiento de todos los cambios realizados. Las pruebas realizadas se deben documentar en el Documento de Pruebas Unitarias Realizadas (DOC2). Esta tarea sirve para comprobar la correcta operación del módulo al que se le han practicado las acciones correctivas.

En la Figura 4.8 se representa el diagrama de actividades del SprintM No Planificable y en la Tabla 4.9 se resume las mismas.

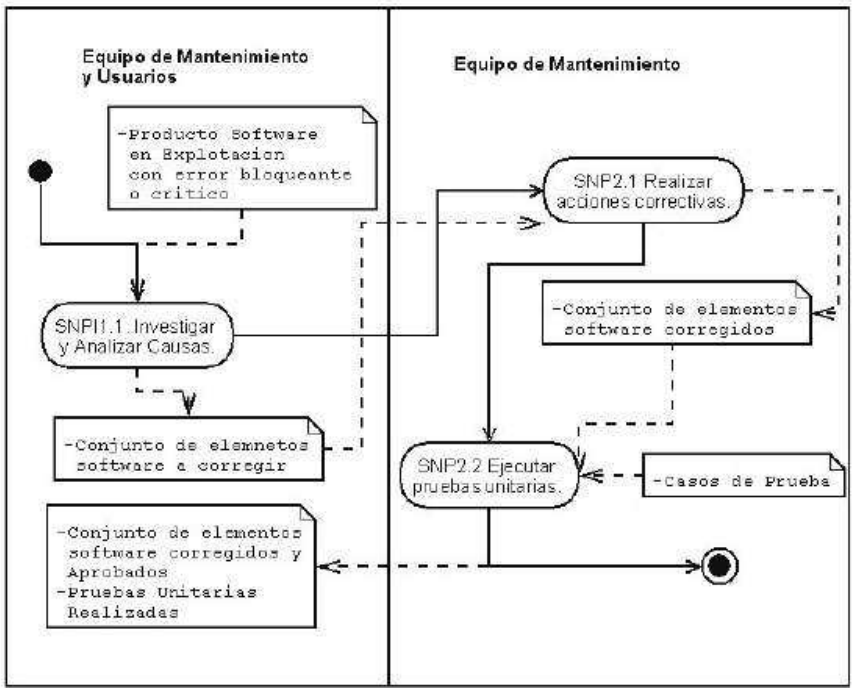


Figura 4.8. Diagrama de actividad del SprintM No Planificable del proceso de mantenimiento

Actividad SNP1 **Análisis del error**						
	Entradas	Salidas	Técnicas	Roles	Interfaces con otros Procesos	Nivel de Servicio
Tarea SNP1.1 **Investigar y Analizar Causas**	Producto Software en explotación con error crítico. Petición de Modificación	Conjunto de elementos Software a corregir	- Estudio de la documentación - Investigar el Producto Software - Observación y entrevistas	- Equipo de Mantenimiento. - Usuario	Soporte al cliente	Básico

Actividad SNP2 **Intervención correctiva urgente**						
	Entradas	Salidas	Técnicas	Roles	Interfaces con otros Procesos	Nivel de Servicio
Tarea SNP2.1 **Realizar Acciones Correctivas**	Conjunto de elementos Software a corregir	Conjunto de elementos software corregidos	Codificación	Equipo de mantenimiento		Básico
Tarea SNP2.2 **Ejecutar Pruebas Unitarias**	Elementos de Software Corregidos. Casos de Prueba	- Elementos de - Software Corregidos y Aprobados. - Documento con las pruebas unitarias realizadas	- Técnicas de Prueba de Software. - Uso de Herramientas como JUnit o SimpleTest. - Pruebas de Regresión	Equipo de Mantenimiento		Básico

Tabla 4.9. Resumen de la Actividad SprintM No Planificable

4.2.2.3.4 SprintM Planificable

El propósito de esta actividad es brindar atención a las peticiones de modificación que afectan de alguna manera el funcionamiento del producto software. El Sprint de mantenimiento planificable se ejecuta cuando se asume los mantenimientos: correctivo no urgente, perfectivo, preventivo y adaptativo. Al ser un Sprint con menor urgencia que el anterior se recomienda la ejecución de SprintM más largos, de entre ocho y quince días (dependiendo del tipo de mantenimiento) con reuniones cada dos días.

Esta actividad está compuesta por dos sub-actividades que se describen a continuación:

- Actividad SP1. Análisis de la petición.
  - **Tarea SP1.1. Analizar y elegir solución.** Si se trata de un mantenimiento correctivo no urgente o perfectivo (CP), se documenta la causa del error y se indican las posibles alternativas de implementación de la solución en el Documento de Diagnóstico del Error y Posibles Soluciones (DOC3). Si se trata de un mantenimiento preventivo (P) o adaptativo (A), solamente se indican las posibles alternativas de implementación en el Documento de Diagnóstico del Error y Posibles Soluciones (DOC3). Luego de analizar cada una de las posibles soluciones de la petición de modificación se elige la alternativa de implementación adecuada, se rellena el Documento de Diagnóstico del Error y Posibles Soluciones (DOC3), indicando la alternativa elegida para la corrección.
- Actividad SP2. Intervención y pruebas.
  - **Tarea SP2.1. Ejecutar intervención (CP/P/A).** El equipo de mantenimiento debe ejecutar las acciones necesarias para ofrecer una solución a la petición de modificación conforme con la alternativa seleccionada, la cual está registrada en el documento Diagnóstico del Error y Posibles Soluciones (DOC3). Se debe identificar todos los componentes del producto software (rutinas, bases de datos, etc.) afectadas por la intervención.
  - **Tarea SP2.2. Ejecutar pruebas unitarias y de integración (CP/P/A).** El Equipo de mantenimiento realiza las pruebas unitarias y de integración sobre el producto software intervenido. Se debe comprobar que la Petición de Modificación (registrada en el DOC1) queda atendida y de que los diferentes elementos software funcionan correctamente en forma conjunta. Una vez finalizadas, se genera el Documento de Pruebas Unitarias Realizadas (DOC2). El propósito es probar el correcto funcionamiento del software tanto en módulos independientes como en todo el sistema.

- **Tarea SP2.3. Ejecutar paralelamente el software antiguo y nuevo.** El equipo de mantenimiento ejecuta acciones reales en el producto software antiguo y en el modificado para detectar y prevenir posibles errores de proceso. Pueden aplicarse pruebas de no regresión, de manera que no se repitan errores anteriores a la intervención.

En la Figura 4.9 se representa el diagrama de actividades del SprintM Planificable.

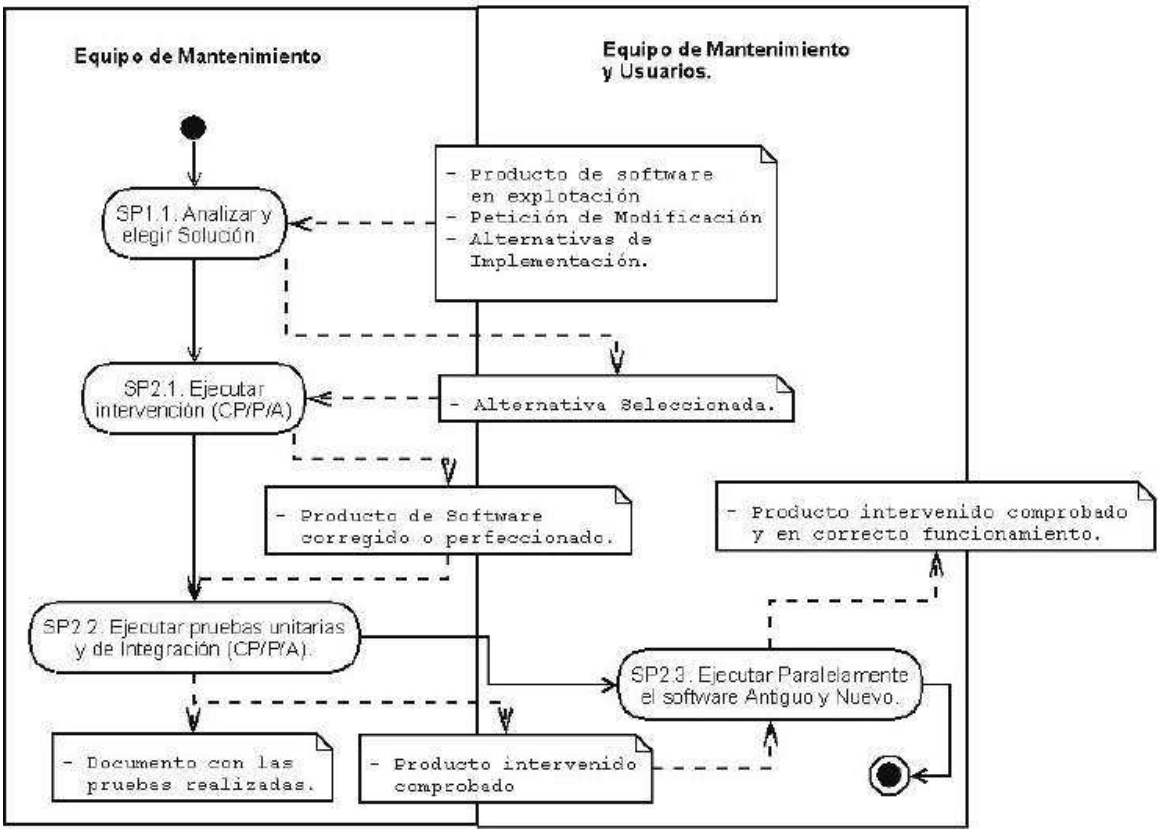


Figura 4.9. Diagrama de actividad del SprintM Planificable del proceso de mantenimiento

La Tabla 4.10 muestra un resumen de esta actividad.

Actividad SP1 **Análisis de la petición**						
	Entradas	**Salidas**	**Técnicas**	**Roles**	**Interfaces con otros Procesos**	**Nivel de Servicio**
Tarea SP1.1 **Analizar y elegir solución**	Producto de software en explotación. Petición de Modificación. Alternativas de Implementación.	Alternativa Seleccionada.		Equipo de Mantenimiento	Soporte al cliente Gestión de la Configuración Gestión de resolución de problemas	Intermedio

Actividad SP2 **Intervención y pruebas**						
	Entradas	**Salidas**	**Técnicas**	**Roles**	**Interfaces con otros Procesos**	**Nivel de Servicio**
Tarea SP2.1 **Ejecutar intervención (CP/P/A)**	Producto de Software en explotación. CP (Diagnóstico del Error) P (Mejora a Realizar) A (Copia del Producto Soft.)	Producto de Software corregido o perfeccionado. A (Copia Adaptada)	Codificación Reestructuración Reingeniería.	Equipo de mantenimiento		Intermedio
Tarea SP2.2 **Ejecutar pruebas unitarias y de integración (CP/P/A)**	Producto intervenido	Producto intervenido comprobado Documento con las pruebas realizadas	Técnicas de pruebas	Equipo de mantenimiento	Aseguramiento de la calidad	Intermedio
Tarea SP2.3 **Ejecutar paralelamente el software antiguo y nuevo.**	Producto intervenido comprobado	Producto intervenido comprobado y en correcto funcionamiento.	- Realización de las operaciones reales en la versión antigua y nueva del producto de Software - Pruebas de Regresión.	- Equipo de mantenimiento - Usuario		Avanzado

Tabla 4.10. Resumen de la Actividad SprintM Planificable

4.2.2.3.5 Seguimiento del SprintM

El propósito de esta actividad es llevar a cabo reuniones de control para hacer un seguimiento del estado de avance y resolver problemas de las intervenciones realizadas mediante los Sprint de Mantenimiento (SprintM). En esta actividad, que es común para los dos tipos de SprintM (planificable y no planificable), se lleva a cabo la revisión del trabajo realizado y se solucionan las dificultades encontradas. Se compone de las siguientes tareas:

- **Tarea SSM1.1. Reuniones habituales.** En estas reuniones de al menos 15 minutos, se reúne todo el equipo de mantenimiento, en el que cada miembro del equipo expone solo los siguientes temas:
  - ¿Qué es lo que hizo desde la última reunión?
  - ¿Qué es lo que va hacer hasta la siguiente reunión? Es muy importante que al salir de la reunión todos los involucrados sepan lo que debe hacer y que todos están alineados en la misma dirección: el mantenimiento del software.
  - ¿Cómo se va a llevar a cabo, te hace falta algo? Todos deben tener claro cómo realizar su trabajo, con esta pregunta deben surgir los problemas que tienen las personas para la realización de la Petición de Modificación.

  Sólo se tratan estos temas para que la reunión sea rápida y no se pierda tiempo, su finalidad es alinear a las personas en la misma dirección y sacar a la luz los problemas e impedimentos que hay para solucionarlos y conseguir el objetivo.

- **Tarea SSM1.2. Seguimientos de los cambios.** Se realiza el control de los cambios producidos en la ejecución del SprintM, este control abarca los siguientes aspectos:
  - Realizar la traza de los cambios que la petición de modificación ha provocado a lo largo de los procesos de desarrollo implicados.
  - Verificar que se han realizado satisfactoriamente las pruebas unitarias, de integración y del sistema que se consideraron necesarias para los componentes a modificar.
  - Comprobar que sólo se ha modificado lo establecido y, en caso contrario, justificar el motivo.
  - Llevar el control de los distintos desarrollos existentes en paralelo sobre un mismo componente, con el fin de coordinar las modificaciones incluidas en cada uno de ellos, y asegurar que en el paso a producción se implantan correctamente.

En la Figura 4.10 se representa el diagrama de actividades de Seguimiento del SprintM.

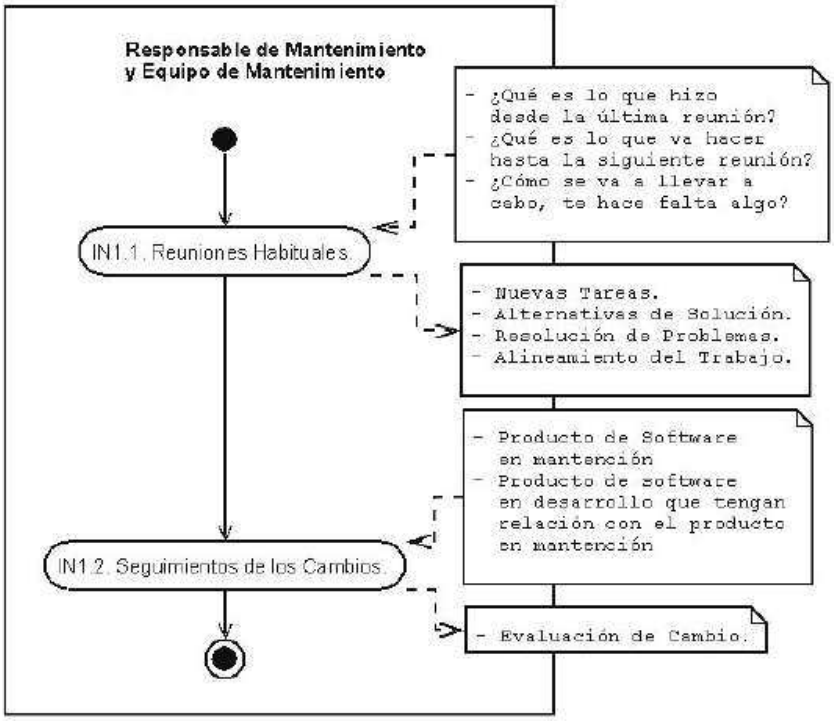


Figura 4.10. Diagrama de actividad del Seguimiento del SprintM

La Tabla 4.11 muestra una recapitulación de esta actividad.

Actividad SSM1 **Seguimiento del SprintM**						
	Entradas	**Salidas**	**Técnicas**	**Roles**	**Interfaces con otros Procesos**	**Nivel de Servicio**
Tarea SSM1.1 **Reuniones habituales.**	Informe ejecutivo de las tareas realizadas. Problemas encontrados	Tareas actualizadas a realizar. Problemas solucionados	Reuniones cortas cara a cara.	Responsable de mantenimiento. Equipo de mantenimiento	Gestión de resolución de problemas	Básico
Tarea SSM1.2 **Seguimientos de los cambios**	Productos Software en mantenimiento. Productos Software relacionados	Validación y control de los cambios realizados		Responsable de mantenimiento. Equipo de mantenimiento	Gestión de cambio de requisitos	Avanzado

Tabla 4.11. Resumen de la Actividad Seguimiento del SprintM.

4.2.2.3.6 Actividades finales

El propósito de esta actividad es socializar los cambios realizados con el fin de que el cliente verifique la corrección de la Petición de modificación. Este conjunto de actividades y tareas es común para todos los mantenimientos. Se denominan finales ya que se ejecutan al terminar el mantenimiento propuesto por una petición de modificación.

Esta actividad está compuesta por dos sub-actividades que se describen a continuación.

◤ Actividad F1. Finalización de la intervención.

- **Tarea F1.1 Verificar y validar corrección con el cliente.** El Equipo de mantenimiento y la organización usuaria del sistema se reúnen para comprobar que el producto intervenido funciona correctamente. En esta tarea se debe haber interacción con el cliente para recabar impresiones, sugerencias, mejoras y su relevancia sobre el producto que se ha entregado. También se evalúan y registran nuevos posibles cambios en el Registro de Peticiones.

- **Tarea F1.2. Pasar a producción.** El Equipo de mantenimiento pasa al entorno de producción el software corregido para su utilización por parte de los usuarios.

- **Tarea F1.3. Documentar manual de usuario.** El Equipo de mantenimiento debe documentar o redocumentar el manual de usuario, si es que ha cambiado el modo de operación del software o si ha agregado nuevas funcionalidades.

- **Tarea F1.4 Registro de la intervención.** La intervención de modificación queda registrada, según los procedimientos de la organización.

- **Tarea F1.5. Reunión de retrospección.** Se deben extraer las mejores prácticas de la última intervención. Aquí el Encargado de Mantenimiento pregunta a su Equipo: ¿Qué fue bien en el último Sprint? ¿Qué se puede mejorar? Toda esta información es tomada para optimizar el próximo SprintM. Una labor muy importante a realizar en esta tarea es la definición y anuncio del próximo Sprint de Mantenimiento a ejecutar.

En la Figura 4.11 se representa el diagrama de actividades de las actividades finales y en la Tabla 4.12 se muestra el resumen correspondiente.

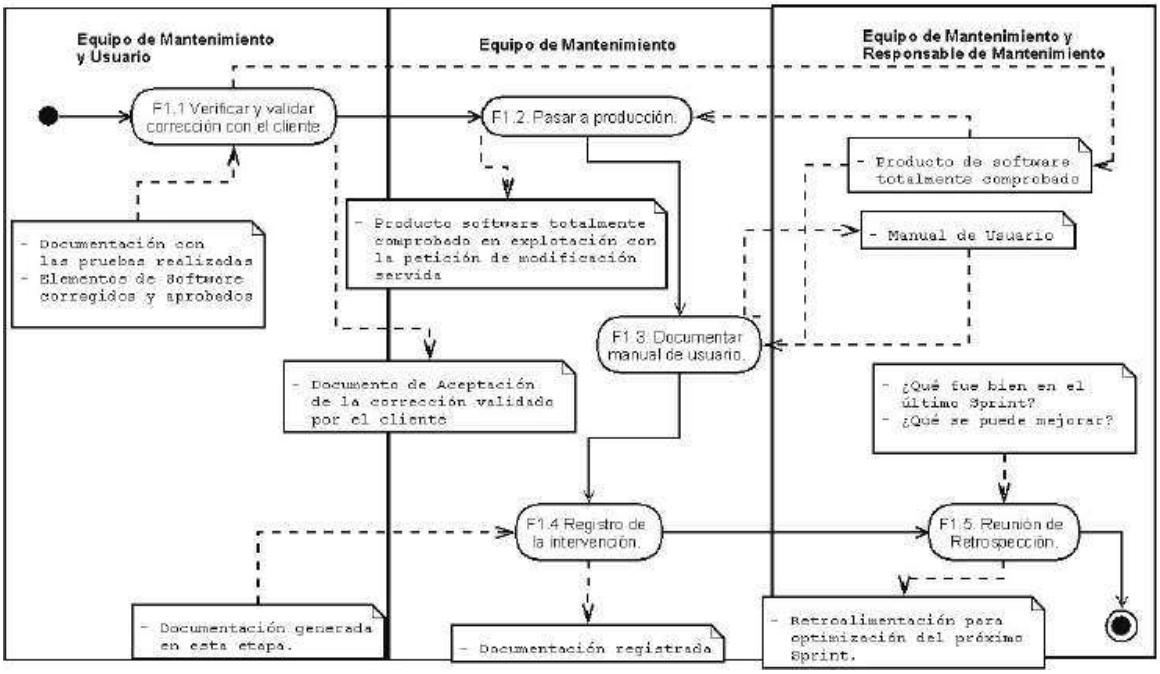


Figura 4.11. Diagrama de actividad de Finalización de la intervención

Actividad F1 **Finalización de la intervención**						
	Entradas	**Salidas**	**Técnicas**	**Roles**	**Interfaces con otros Procesos**	**Nivel de Servicio**
Tarea F1.1 **Verificar y validar corrección con el Cliente**	Documentación con las pruebas realizadas. Elementos de Software corregidos y aprobados	Documento de aceptación de la corrección validado por el cliente. Producto software totalmente comprobado		Equipo de Mantenimiento Usuario	Atención al cliente Otra: Revisión Conjunta	Básico
Tarea F1.2 **Pasar a producción**	Producto de software totalmente comprobado	Producto software totalmente comprobado en explotación con la petición de modificación incorporada		Equipo de Mantenimiento		Básico
Documentar Manual de Usuario	Manual de Usuario. Producto de software totalmente comprobado	Manual de Usuario Actualizado		Equipo de Mantenimiento		Intermedio
Registro de la Intervención	Documentación generada en esta etapa	Información registrada de la intervención		Equipo de Mantenimiento	Otra Documentación	Avanzada
Reunión de Retrospección	Lecciones aprendidas del SprintM	Sugerencias para mejorar el SprintM Anuncio del próximo SprintM	Revisión post-mortem	Responsable de mantenimiento Equipo de Mantenimiento		Básica

Tabla 4.12. Tabla Resumen de la Actividad Finalización de la intervención.

- Actividad F2. Retirada
  - **Tarea F2.1 Desarrollar plan de retirada.** El Equipo de mantenimiento redacta un documento en el que describe cuándo se llevará a cabo la retirada del software.

- **Tarea F2.2 Notificar futura retirada.** El Equipo de mantenimiento notifica al Cliente el momento en el que se ejecutará la retirada del software.
- **Tarea F2.3 Ejecutar paralelo.** El Usuario, con el visto bueno del Cliente y bajo la supervisión del Equipo de mantenimiento, realiza operaciones reales sobre el software que se va a retirar y el software nuevo a implantar (si es que aquél va a ser sustituido).
- **Tarea F2.4 Notificar retirada.** Se notifica la inminencia de la retirada. El producto software es retirado y el nuevo es el que queda en funcionamiento para explotación.
- **Tarea F2.5 Almacenar Datos del Software Antiguo.** Los datos del software antiguo son almacenados.

En la Figura 4.12 se representa el diagrama de actividades de Retirada y un resumen en la Tabla 4.13.

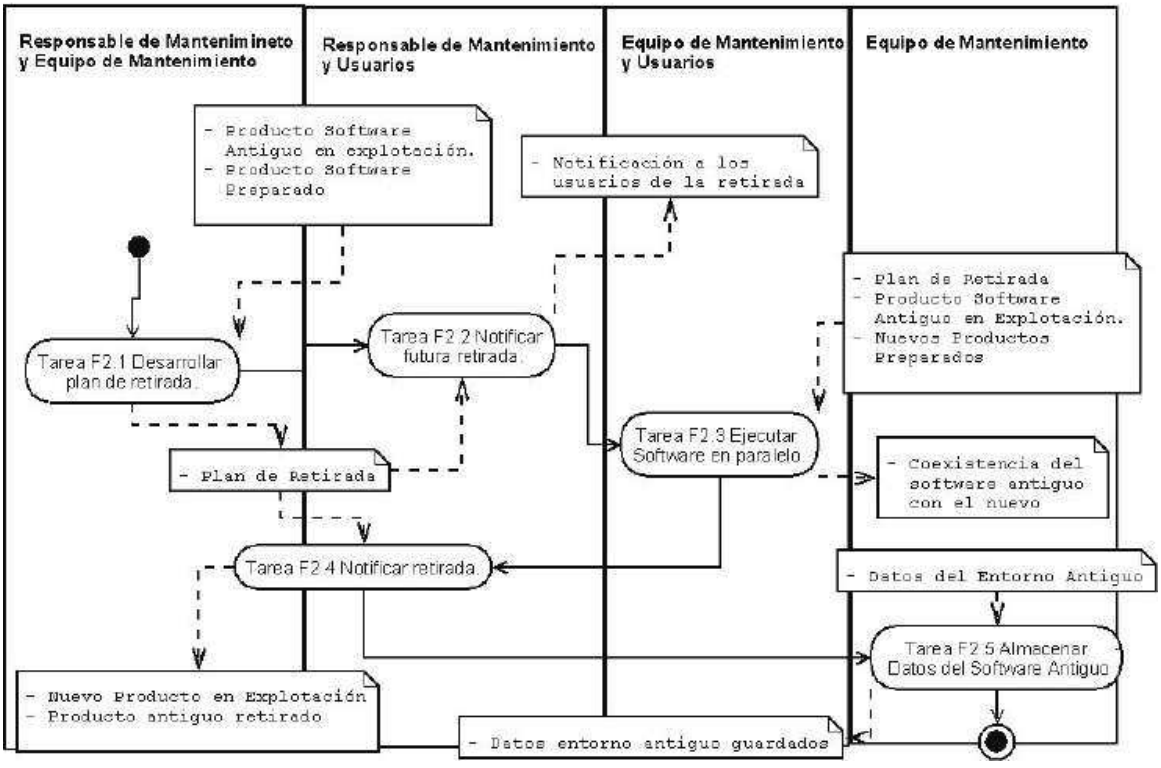


Figura 4.12. Diagrama de actividad de Retirada del software

Actividad F2 Retirada						
	Entradas	**Salidas**	**Técnicas**	**Roles**	**Interfaces con otros Procesos**	**Nivel de Servicio**
Tarea f2.1 **Desarrollar plan de retirada**	Producto software antiguo en explotación. Nuevo producto software	Plan de retirada		Equipo de mantenimiento		Avanzado
Tarea f2.2 **Notificar futura retirada**	Plan de retirada	Notificación a los usuarios de la retirada		Equipo de mantenimiento. Usuario		Avanzado
Tarea f2.3 **Ejecutar paralelo**	Plan de retirada. Producto software antiguo en explotación. Nuevos productos a implantar.	Coexistencia del software antiguo con el nuevo		Equipo de mantenimiento. Usuario		Avanzado
Tarea f2.4 **Notificar retirada**	Plan de retirada	Nuevo producto en explotación. Producto antiguo retirado		Equipo de mantenimiento. Usuario		Avanzado
Tarea f2.5 **Almacenar datos software antiguo.**	Datos del entorno antiguo	Datos entorno antiguo guardados		Equipo de mantenimiento		Avanzado

Tabla 4.13. Tabla Resumen de la Actividad Retirada del software.

4.2.2.3.7 Finalización del servicio

El propósito de esta actividad es dar por finalizado de manera formal el servicio de mantenimiento de software al cliente. Esta actividad se realiza cuando el mantenedor deja de prestar sus servicios a la organización cliente, y se compone de dos tareas:

- **Tarea F3.1 Entrega del inventario y de la documentación.** Se entregan los productos de software generados y modificados al cliente.

- **Tarea F3.2 Cesión definitiva del servicio.** La Organización de mantenimiento deja de prestar sus servicios al Cliente con carácter definitivo.

La Figura 4.13 representa el diagrama de actividades de finalización del servicio.

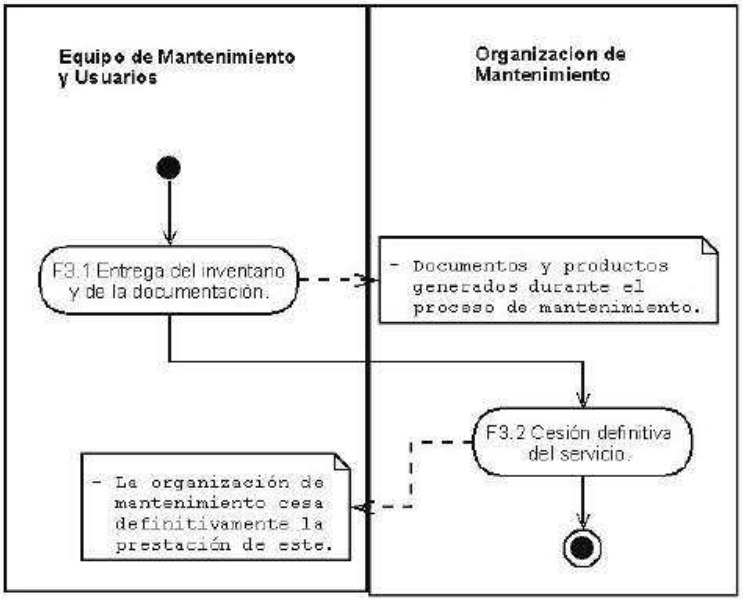


Figura 4.13. Diagrama de actividad de Finalización del servicio de mantenimiento

La Tabla 4.14 muestra una recapitulación de esta actividad.

Actividad F3 **Finalización del servicio**						
	Entradas	**Salidas**	**Técnicas**	**Roles**	**Interfaces con otros Procesos**	**Nivel de Servicio**
Tarea F3.1 **Entrega de Inventario y Documentación**		Documentos y productos generados durante el proceso de mantenimiento.		Equipo de Mantenimiento Usuario		Intermedio
Tarea F3.2 **Cesión definitiva del servicio**		Documento que expone que cesa definitivamente la prestación del servicio.		Equipo de Mantenimiento Usuario		Avanzado

Tabla 4.14. Resumen de la Actividad Finalización del servicio

4.2.3 Interfaces con otros procesos

Este apartado define, a manera de ejemplo, dos interfaces con los cuales el proceso de desarrollo está involucrado en los niveles de servicio intermedio y avanzado. Estas interfaces son los procesos de Aseguramiento de la Calidad y Gestión de la configuración, con las cuales el proceso de mantenimiento tiene relación y que debe implementar (entre otras) para brindar el nivel de servicio de mantenimiento correspondiente.

Las interfaces que se presentan a continuación se basan en las descritas en la metodología Métrica Versión 3 [MAP, 2007], adaptádas a las necesidades de Ágil_MANTEMA. En las siguientes líneas se muestran en mayor detalle.

4.2.3.1 ASEGURAMIENTO DE LA CALIDAD

El responsable de mantenimiento intervendrá durante el proceso de mantenimiento, efectuando revisiones de seguimiento periódicas, más o menos frecuentes según los casos, que sirvan para constatar que el mantenimiento establecido para que la petición de modificación se realice de forma correcta.

En algún caso, según las implicaciones del cambio, puede ser necesario revisar puntualmente:

- El contenido del plan de pruebas de regresión.
- La ejecución de las pruebas de regresión según la normativa acordada en el plan de aseguramiento de calidad.
- Las verificaciones y casos de prueba que se hayan incluido en el plan de pruebas para los cambios producidos por una petición.
- Las incidencias detectadas con el fin de determinar si puede verse afectada alguna propiedad de calidad.

En caso de revisar la ejecución de las pruebas de regresión, se registrará la aprobación de las pruebas por el responsable de mantenimiento.

En la Figura 4.14 se muestran las actividades de Aseguramiento de la Calidad durante el proceso de Mantenimiento.

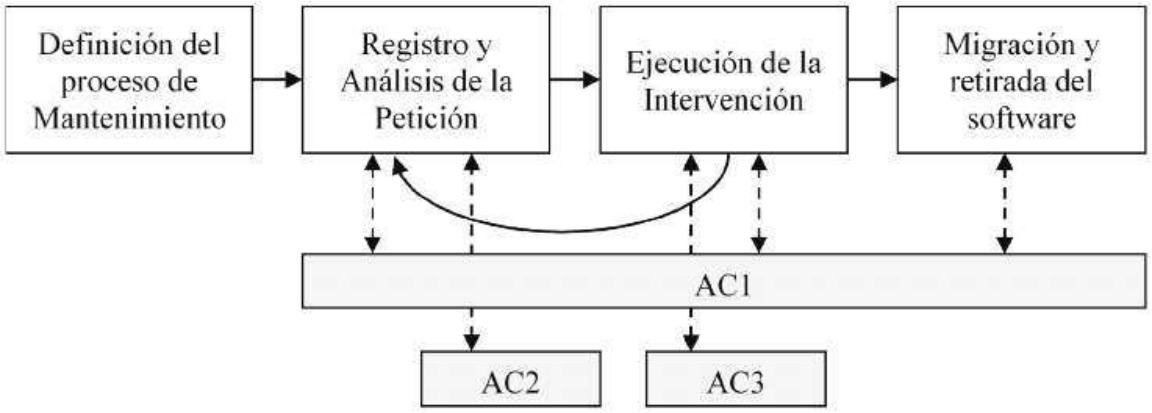


Figura 4.14. Interfaz del proceso de mantenimiento con el Aseguramiento de la Calidad.

- **Tarea AC1: Revisión del Mantenimiento**

  Se realiza una revisión periódica del Registro de Peticiones comprobando que se mantiene actualizado. Asimismo, se revisa que el usuario acepta o rechaza la solución propuesta para dar respuesta a su petición y que aprueba formalmente el cierre de la petición. Esta tarea de la interfaz de aseguramiento de calidad se aplica a todas las actividades del proceso Mantenimiento.

- **Tarea AC2: Comprobación de la Existencia del Plan de Pruebas de Regresión**

  Se revisa que se ha establecido un plan de pruebas de regresión de acuerdo con los criterios establecidos en el plan de aseguramiento de calidad, con el objetivo de determinar qué métodos se van a aplicar para la ejecución de las pruebas, cuáles van a ser los criterios de aceptación, cómo se van a realizar las actividades de verificación y cómo se van a emitir los resultados. Se revisa la existencia de una normativa para la gestión de los resultados de las pruebas.

- **Tarea AC3: Revisión de la Realización de las Pruebas de Regresión**

  Se comprueba que se han realizado las pruebas de regresión y se lleva a cabo la revisión de las verificaciones y casos de prueba que se hayan determinado para la correcta implantación del cambio. Para todo esto, se tendrá en cuenta la normativa establecida para la gestión de los resultados de dichas pruebas. En el caso de existir casos de prueba adicionales, incorporados como consecuencia de las medidas correctoras tomadas para solventar los errores detectados, el encargado de mantenimiento

revisará que se han resuelto de forma correcta. Igualmente, se revisarán las incidencias no resueltas con el fin de valorar hasta qué punto se ven comprometidas las propiedades de calidad establecidas inicialmente. Se registra la aprobación por parte del responsable de mantenimiento.

4.2.3.2 GESTIÓN DE LA CONFIGURACIÓN

El objetivo de la interfaz de gestión de configuración con el proceso de Mantenimiento es conservar la integridad del sistema de información cuando se producen cambios en el mismo.

El beneficio de una buena gestión de configuración en el proceso de mantenimiento es muy elevado, teniendo en cuenta la reducción del tiempo de localización de los problemas, la reproducción de errores y el control y seguimiento de los estados por los que va pasando la petición de mantenimiento. De esta manera se puede conocer en cada momento la situación en la que se encuentra cada cambio en particular y el sistema de información en general.

La interfaz de gestión de configuración en el proceso de mantenimiento es fundamental, al realizarse el control del cambio desde que se produce la notificación del mismo o de la incidencia, momento en el que se registra la solicitud de mantenimiento en el sistema de gestión de la configuración, hasta que la solución es aceptada por el usuario.

Para realizar el análisis de la petición es conveniente solicitar información al sistema de gestión de la configuración para identificar las versiones de los sistemas de información afectados por la petición.

Una vez que ha sido aceptada la propuesta de solución se realiza un registro del cambio en el sistema de gestión de configuración con la información obtenida del mismo relativa a las versiones de los sistemas de información y productos afectados por el cambio. Este registro constituye el nexo de unión entre las peticiones de mantenimiento y los cambios que se van a realizar sobre los sistemas de información afectados. Recoge datos referentes a las versiones de los sistemas de información de los que se parte y cuáles van a ser las nuevas versiones generadas, así como las versiones de los productos concretos afectados por el cambio y cuál será la nueva versión de dichos productos. También debe registrarse en el sistema de gestión de la configuración la nueva versión de los sistemas de información y de los productos según el criterio de versionado establecido en el plan de gestión de la configuración.

Una vez que el cambio ha sido realizado y aceptado se registra dicha aceptación en el sistema de gestión de la configuración.

En la Figura 4.15 se aprecian las actividades de Gestión de Configuración durante el proceso de Mantenimiento.

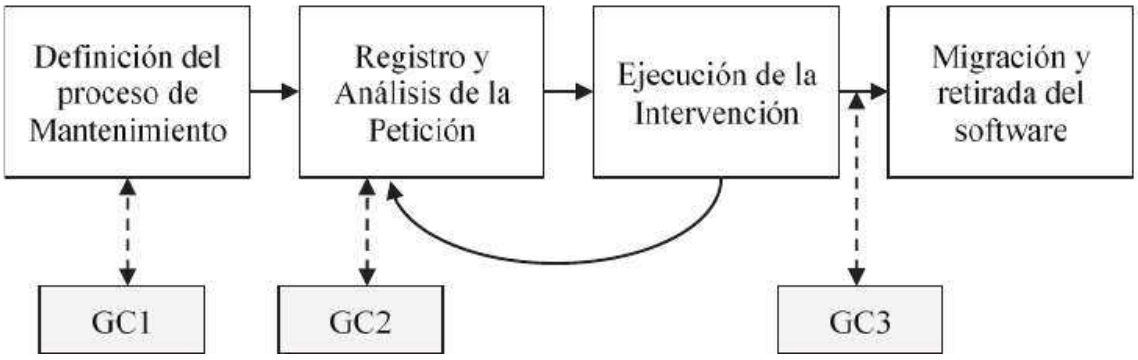


Figura 4.15. Interfaz del proceso de mantenimiento con la Gestión de Configuración.

- **Tarea GC1: Implementar proceso de Gestión de Configuración.**

  Se establece una interfaz con este proceso de soporte para controlar y registrar los cambios del software mantenido, también para ver el estado en que se encuentra. Todo esto para reducir errores, aumentar la calidad y la productividad y evitar los problemas que pueda acarrear una incorrecta sincronización en dichos cambios, al afectar a otros elementos del sistema o a las tareas realizadas por otros miembros del equipo de mantenimiento.

- **Tarea GC2: Registro del Cambio en el Sistema de Gestión de la Configuración**

  Una vez aprobada la propuesta de solución se registra el cambio en el sistema de gestión de la configuración. Este registro refleja las peticiones de mantenimiento que van a ser atendidas con la realización del cambio. Debe indicarse cuáles son las versiones de los sistemas de información y de los productos de las que parte el cambio, y siguiendo el criterio de versionado, cuáles son las nuevas versiones de los mismos que se van a generar como consecuencia de la realización del cambio. La información mantenida en este registro permite en todo momento efectuar una traza de la evolución del sistema y los productos que lo integran desde su puesta en producción como consecuencia de la realización de cambios.

- **Tarea GC3: Registro de la Nueva Versión de los Productos Afectados por el Cambio en el Sistema de Gestión de la Configuración**

  Los productos que hayan sido modificados o creados con motivo de la realización del mantenimiento deben registrarse en el sistema de gestión

de la configuración en la versión correspondiente. La nueva versión de estos componentes comienza su ciclo de estados, de manera que deben registrarse en el estado que establezca el plan de gestión de la configuración.

4.2.4 Comparativa con MANTEMA

En esta sección se mostró la aplicación de Scrum a una metodología robusta como MANTEMA, con el objetivo de crear una metodología de mantenimiento ligera que pueda ser aplicable a las pequeñas y medianas empresas, el resultado ha sido Ágil_MANTEMA.

Los beneficios de esta metodología para las organizaciones que la implementen son:

- Metodología clara, bien definida, que cubre todos los aspectos del mantenimiento en cuanto a procesos, tareas y actividades, técnicas, documentación y medición del proceso.
  - Capacidad de respuesta a cambios en el negocio.
  - Entrega continua y en plazos breves de software funcional.
  - Trabajo continuo entre el cliente y el equipo de desarrollo.
  - Importancia de la simplicidad, eliminando trabajo innecesario.
  - Disminución de costos del proceso de mantenimiento.
  - Mejora continua de los procesos y el equipo de desarrollo.
  - Mejorar la organización interna, logrando comunicación más fluida estableciendo responsables y objetivos.
- Por otro lado, al basarse en la norma estándar ISO 12207, posibilita la ejecución del mantenimiento conforme a lo indicado en esa norma.
  - Mejorar productividad y competitividad de la empresa.
  - Mejorar imagen.
  - Aumenta la satisfacción y fidelidad de los clientes.
  - Incrementa la rentabilidad.

La Tabla 4.15 muestra una comparación entre MANTEMA y Ágil_MANTEMA.

Ágil_MANTEMA	MANTEMA
Número de Roles: 5 Número de Actividades: 10 Número de Tareas: 27 Número de Productos: 3 Número de Técnicas: 1 Número de Relaciones con otros Procesos: 11 Número de Métricas propuestas: 1	Número de Roles: 8 Número de Actividades: 14 Número de Tareas: 46 Número de Productos: 11 Número de Técnicas: 2 Número de Relaciones con otros Procesos: 10 Número de Métricas propuestas: 11
Proceso menos controlado, con pocos principios	Proceso mucho más controlado, con numerosas políticas y normas.
Especialmente preparada para el cambio	Cierta resistencia a los cambios
El cliente es parte del equipo de mantenimiento	El cliente interactúa con el equipo de mantenimiento mediante reuniones
Tiene un punto de equilibrio entre la No-documentación y Demasiada-documentación	Proceso disciplinado sobre el mantenimiento de software
Evita la burocracia y brinda resultados	Detalla procesos con énfasis en la planeación
Brinda cambios y resultados continuos	Producto solo en la finalización del proceso

Tabla 4.15. Comparación entre Ágil_MANTEMA y MANTEMA

4.2.5 OTRAS METODOLOGÍAS

No existen muchas metodologías especializadas en el proceso de mantenimiento del software puesto que la mayoría de las metodologías abarcan el ciclo de vida completo de desarrollo software y dedican parte de la metodología generalista al mantenimiento software.

No obstante, a parte de MANTEMA y Ágil MANTEMA existen otras metodologías de mantenimiento software en la literatura. Por ejemplo, [Hyland-Wood et al., 2008] propone una metodología de mantenimiento usando técnicas de la web semántica y modelado de documentación paradigmática. Los autores discuten teorías de descripción de sistemas software en términos de aplicación de metadatos software funcional y no funcional como documentación de requisitos, métricas, casos de prueba pasados o fallidos, etc. Esta metodología trata de registrar estos metadatos y llevar la traza de ellos a través del proceso de mantenimiento. Haciendo uso de esta metainformación la metodología aboga por usar técnicas de la web semántica para navegar por los sistemas software para facilitar su entendimiento y mantenimiento. La Figura 4.16 muestra el esquema general de esta metodología.

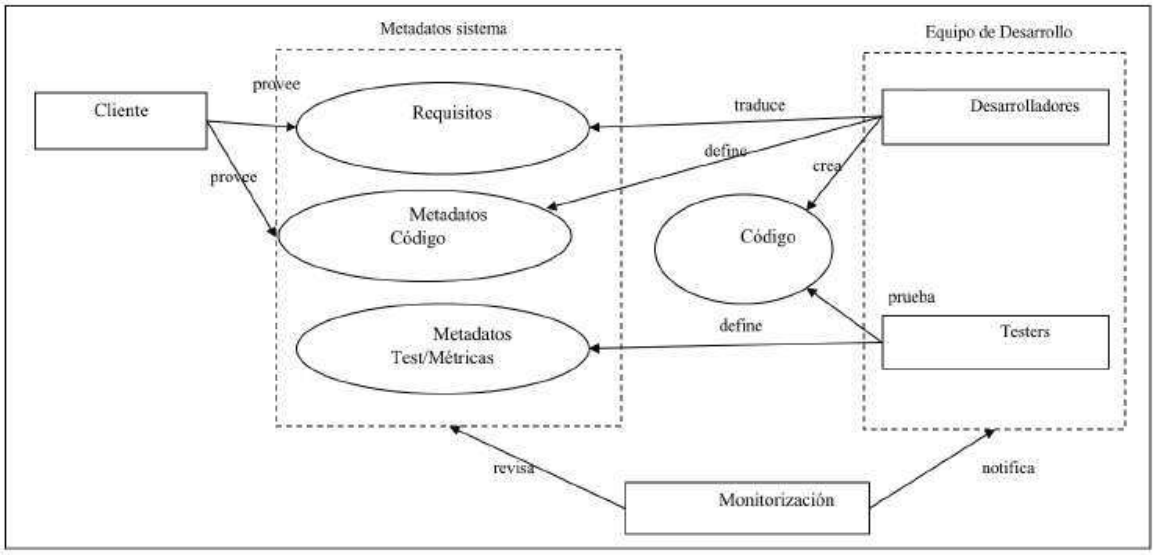


Figura 4.16. Metodología de mantenimiento según [Hyland-Wood et al., 2008].

De forma similar a Ágil MANTEMA, otros autores han propuesto también la aplicación de principios ágiles como los de Scrum para definir aproximaciones a metodologías del proceso de mantenimiento software [Cardoso de Mello, 2012].

Otra metodología ágil que se ha aplicado para definir una metodología de mantenimiento ha sido XP (eXtreme Programming) [Choudhari y Suman, 2005]. En este caso, los autores proporcionan una metodología iterativa basada en XP y realizan un experimento en el que concluyen que este tipo de metodología mejora la productividad de los ingenieros de mantenimiento y mejora además la mantenibilidad futura del sistema software.

4.3 LECTURAS RECOMENDADAS

✓ Cardoso de Mello, M. (2012). Ágile processes for the maintenance cycle. A smarter work cycle for a Smarter Planet. IBM DeveloperWorks.

Se trata de una metodología ágl para el mantenimiento del software que ofrece algunos aspectos complementarios a Ágil MANTEMA.

4.4 SITIOS WEB RECOMENDADOS

- ✓ *https://alarcos.esi.uclm.es/competisoft/web/completo/index.htm* (Metodología)
- ✓ *https://alarcos.esi.uclm.es/ipsw/doc/agilmantema.pdf* (Informé técnico)

  Sitios web de la metodología de mantenimiento AGIL-MANTEMA e informe técnico que la describe en destalle. Ambos recursos son de gran útilidad para comprender a fondo las características de esta metodología.

4.5 EJERCICIOS

Ejercicio 1

Proponga los contenidos que deberían tener los documentos 6 a 13 de los listados en la tabla 3.3.

Ejercicio 2

¿A qué puede deberse que en el mantenimiento no planificable (Tabla 3.2) no se realicen pruebas de integración que sí se hacen en el planificable (Figura 3.3)?

Ejercicio 3

En la figura 11.2 se muestra que debe establecerse interfaz con el proceso de Formación de ISO/IEC 12207. ¿En qué actividad o conjunto de actividades del proceso de mantenimiento definido en MANTEMA piensa que debería utilizarse dicho proceso? ¿Por qué?

Ejercicio 4

Indique los niveles de servicio y capacidad de la metodología Ágil MANTEMA

Ejercicio 5

Enumere las actividades en la metodología Ágil MANTEMA

Ejercicio 6

¿Cuáles son las principales diferencias entre MANTEMA y Ágil MANTEMA?

Ejercicio 7

Compare las actividades y tareas de las tres metodologías ágiles de mantenimiento citadas en el apartado 3.5.

Ejercicio 8

Analice cómo se podría soportar la metodología Ágil MANTEMA en herramientas que soportan SCRUM

Ejercicio 9

¿Qué tipo de formación y experiencia serían convenientes para participantes en el proceso de mantenimiento que aparecen en el apartado 2.2.1.?

Ejercicio 10

Analice cómo podría adaptarse la metodología Ágil MANTEMA a un entorno DevOps.

5

MANTENIBILIDAD DEL SOFTWARE

De todas las propiedades del software, la que tiene una influencia más importante y directa en el mantenimiento del software es la *Mantenibilidad* o *Facilidad de Mantenimiento*. En este capítulo se analiza en detalle esta propiedad, señalando los diversos factores que influyen en ella.

Puesto que existe una relación directa entre los costes necesarios para mantener un producto software y su mantenibilidad (a menor mantenibilidad mayores costes), la cuantificación de esta propiedad es muy útil para poder realizar presupuestos de dichos costes de mantenimiento, lo que resulta muy interesante para las empresas de servicios informáticos.

Aunque un mayor esfuerzo en mejorar la mantenibilidad de un producto software siempre redunda en una reducción de los costes futuros de mantenimiento, algunos autores estiman que no siempre la reducción en el esfuerzo de mantenimiento compensa el incremento en el esfuerzo para mejorar la mantenibilidad [Burton, 1999]. Sin embargo, y aunque, por ejemplo, para un único proceso de mantenimiento quizás no interesa hacer un esfuerzo por mejorar la mantenibilidad, si se tiene en cuenta los costes superiores de mantenimiento acumulados a lo largo del tiempo sí se puede acabar descubriendo que el esfuerzo por mejorar la mantenibilidad hubiese sido beneficioso.

5.1 CONCEPTO DE MANTENIBILIDAD DEL SOFTWARE

Existen varias definiciones de mantenibilidad. A continuación, se relacionan algunas de las más conocidas:

- El Gobierno Federal de EEUU la define como la facilidad con la cual el software puede ser mantenido, mejorado, adaptado o corregido para satisfacer los requerimientos especificados [FIPS, 1984].

- Card y Glass [1990] establecen que la mantenibilidad significa que los cambios tienden a ser confinados en áreas localizadas del sistema (módulos) y son fáciles de llevar a cabo.
- El IEEE la define como la facilidad con que un sistema o componente software puede ser modificado para corregir defectos, mejorar el rendimiento u otros atributos, o adaptarse a un cambio de entorno [IEEE, 1990].
- ISO define la mantenibilidad como el grado de efectividad y eficiencia con el que se puede modificar un producto o sistema por el personal de mantenimiento previsto. [ISO/IEC, 2017].

Actualmente, se intenta que los desarrolladores construyan sistemas software mantenibles a través de técnicas como la integración continua y monitorización de métricas de manera continua. Sin embargo, los costes y los planes de trabajo, siempre con urgencias y prisas, guían el trabajo de los desarrolladores en otra dirección. Pero en el proceso de desarrollo del software deben proveerse alicientes de forma que la mantenibilidad sea un objetivo real si se quiere reducir los costes del ciclo de vida del software (la gran mayoría de los costes se producen en la fase de mantenimiento tal como se comentó en el capítulo 1).

En Pigoski [1996] se resume esta problemática planteando que el objetivo de la mantenibilidad del software debe establecerse durante las fases de análisis de requisitos, diseño y desarrollo. Este autor señala que las técnicas de diseño y desarrollo del software deben utilizar estándares que incorporen dicho objetivo de mantenibilidad.

5.2 ASPECTOS QUE INFLUYEN EN LA MANTENIBILIDAD

Existen varios factores que afectan directamente a la mantenibilidad, de forma que, si alguno de ellos no se satisface adecuadamente, ésta se resiente. Los más significativos son el proceso de desarrollo, la documentación y la comprensión de los programas [Pigoski, 1996].

- **Proceso de Desarrollo**: la mantenibilidad debe formar parte integral del proceso de desarrollo del software. Las técnicas utilizadas deben ser lo menos intrusivas posible con el software existente [Lewis y Henry, 1989]. Los problemas que surgen en muchas organizaciones dedicadas al mantenimiento son de doble naturaleza: mejorar la mantenibilidad y convencer a los responsables de que la mayor ganancia se obtendrá únicamente cuando la mantenibilidad esté incorporada intrínsecamente en los productos software.

- **Documentación**: En múltiples ocasiones, ni la documentación ni las especificaciones de diseño están claras y libre de ambigüedades, o incluso no están disponibles; y, por tanto, los costes del mantenimiento del software se incrementan debido al tiempo requerido para que un mantenedor entienda el diseño del software antes de poder ponerse a modificarlo. Las decisiones sobre la documentación que debe desarrollarse son muy importantes cuando la responsabilidad del mantenimiento de un sistema se va a transferir a una organización nueva [Marciniak y Reifer, 1990].
- **Comprensión de Programas**: La causa básica de la mayor parte de los altos costes del mantenimiento del software es la presencia de obstáculos a la comprensión humana de los programas y sistemas existentes. Estos obstáculos surgen de tres fuentes principales [Chapin, 1987]:
  - La información disponible es incomprensible, incorrecta o insuficiente.
  - La complejidad del software, de la naturaleza de la aplicación o de ambos.
  - La confusión, mala interpretación o olvidos sobre el programa o sistema.

Como ya se ha comentado, muchos de los problemas del mantenimiento se derivan de unos hábitos inadecuados en el proceso de desarrollo del software. Existen estudios que demuestran dicha relación (producida a través de los efectos que dichas prácticas tienen sobre la complejidad del software). Por ejemplo, en [Slaughter y Banker, 1996] se realiza un estudio comparativo a partir de unos modelos de esfuerzo de mantenimiento del software y de prácticas de desarrollo del software.

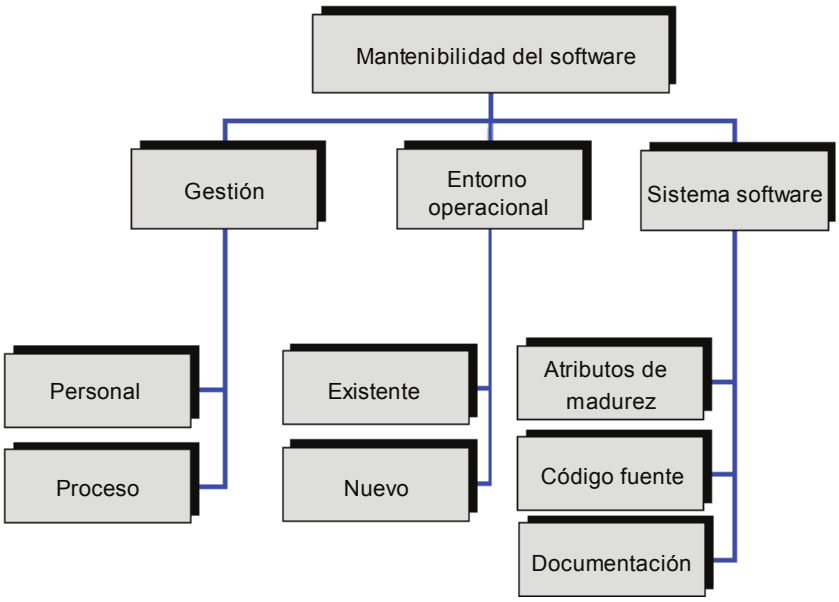


Figura 5.1. Jerarquía en la Mantenibilidad

5.3 ATRIBUTOS DE MANTENIBILIDAD DEL CÓDIGO FUENTE

La mantenibilidad se puede estudiar desde diferentes puntos de vista y en cada caso los aspectos a considerar son diferentes. En la Figura 5.1se muestra una taxonomía propuesta por Oman *et al.* [1991]. Esta taxonomía es bastante detallada especialmente en cuanto a los atributos del código fuente que afectan a la mantenibilidad, ya que entre todos los elementos de un sistema software, establecen que el factor predominante es el propio código fuente. En la Tabla 5.1 se presenta una relación de todos ellos.

Estructuras de control		Estructuras de la información		Tipografía, identificación y comentarios	
Sistema	**Componente**	**Sistema**	**Componente**	**Sistema**	**Componente**
Modularidad	Complejidad	Tipos de datos globales	Tipos de datos locales	Formato del conjunto de programas	Formato de sentencias
Complejidad	Uso de construcciones estructuradas	Estructuras de datos globales	Estructuras de datos locales	Comentarios en el conjunto de programas	Espaciado vertical
Consistencia	Uso de bifurcaciones incondicionales	Acoplamiento del sistema	Acoplamiento de datos	Separación entre módulos	Espaciado horizontal
Nivel de anidamiento	Nivel de anidamiento	Consistencia del flujo de datos	Integridad de la inicialización	Identificadores	Comentarios intramódulos
Control de acoplamiento	Ámbito de las estructuras de control	Consistencia de los tipos de datos	Ámbito de los datos	Símbolos	
Encapsulación	Cohesión	Nivel de anidamiento			
Reutilización de módulos		Complejidad de la E/S			
Consistencia del flujo de control		Integridad de la E/S			

Tabla 5.1. Atributos de mantenibilidad del código fuente

El entorno de programación y, en particular, el paradigma de programación utilizado, puede influir considerablemente en la mantenibilidad de los programas al cambiar aspectos claves como la complejidad y la comprensibilidad del código fuente. Por ejemplo, posibilidades nuevas de la programación orientada a construir aplicaciones web de página única (*single-page application* (SPA) son aplicaciones

o sitios web que interaccionan con el usuario y dinámicamente reescriben la página en vez de cargar nuevas páginas completamente desde un servidor. Por ejemplo, basadas en lenguajes de programación como Node, Angular, etc. pueden hacer menos comprensibles los programas y, por tanto, reducir su mantenibilidad.

5.4 PROPIEDADES DE LA MANTENIBILIDAD

La mantenibilidad se puede considerar como la combinación de dos propiedades diferentes: reparabilidad y flexibilidad. El software es reparable si se pueden eliminar los defectos de forma efectiva y eficaz; y es flexible (evolucionable) si permite cambios para que satisfagan nuevos requerimientos.

5.4.1 Reparabilidad

Un sistema software es reparable si permite la corrección de sus defectos con una cantidad de trabajo limitada y razonable y lo hace de forma eficaz, es decir, se elimina realmente el error evitando que se reproduzca ya que se ataca a la causa raíz del problema. En algunas disciplinas de ingeniería la reparabilidad es un objetivo básico, por ejemplo, el diseño de automóviles se realiza teniendo en cuenta que las piezas que se estropean más a menudo deben ser fácilmente cambiables.

La reparabilidad también se ve afectada por la cantidad y tamaño de los componentes o piezas: un producto software que consiste en módulos bien diseñados es más fácil de analizar y reparar que uno monolítico. Pero el incremento del número de módulos no implica un producto más reparable, ya que también aumenta la complejidad de las interconexiones entre módulos. Se debe buscar un punto de equilibrio con la estructura de módulos más adecuada para garantizar la reparabilidad facilitando la localización y eliminación de los errores en unos pocos módulos.

También puede mejorarse la reparabilidad utilizando herramientas adecuadas. Por ejemplo, empleando un lenguaje de alto nivel en vez de ensamblador (en el cual es mucho más difícil encontrar y corregir los errores) o utilizando depuradores.

La reparabilidad de un producto software está influida por su fiabilidad, ya que al incrementarse esta última, disminuye la necesidad de reparaciones.

5.4.2 Flexibilidad

El software es de naturaleza muy maleable, ya que, debido a su carácter inmaterial, resulta mucho más fácil cambiar o incrementar sus funciones que en

productos de naturaleza física (equipos hardware). Habitualmente, un producto software sufre múltiples modificaciones a lo largo de su tiempo de vida, pasando por diversos estados (versiones). Si el software es diseñado con cuidado, y si cada modificación es realizada cuidadosamente, dicho software tendrá un grado de flexibilidad satisfactorio.

Muchos sistemas software empiezan siendo flexibles, pero después de varios años de evolución llegan a un estado en el cual cualquier modificación supone el riesgo de afectar negativamente a funcionalidades existentes. La razón de esta situación es que, aunque la flexibilidad del software se mejora con la modularización, los cambios sucesivos tienden a reducir la modularidad del sistema original. La situación todavía puede empeorar si se aplican las modificaciones sin un estudio cuidadoso del diseño original y sin una descripción precisa de los cambios en las especificaciones de requerimientos y de diseño.

En la práctica, todos los estudios realizados con grandes sistemas software muestran que la flexibilidad disminuye con cada nueva versión de un producto software. Cada versión complica la estructura del software y, por tanto, las futuras modificaciones serán más difíciles. Para vencer esta dificultad, el diseño inicial del producto, así como cualquier cambio posterior, deben ser realizados con la idea de flexibilidad en mente.

La flexibilidad es una característica tanto del producto software como de los procesos relacionados. En términos de estos últimos, los procesos deben poderse acomodar a nuevas técnicas de gestión y organización, a cambios en la forma de entender la ingeniería, etc.

Un riesgo a la flexibilidad es en muchos casos las propias políticas internas de mantenimiento. En muchas organizaciones para limitar el riesgo de cambios en el software se establece que sólo el módulo afectado puede ser modificado. Aunque por una parte puede parecer buena idea, en muchas modificaciones se requiere retocar o refactorizar otros módulos que consigan que el sistema siga siendo flexible a futuros cambios y no sólo se añada una modificación en un cierto modulo.

5.5 ESTÁNDAR ISO/IEC 25000

La familia de normas ISO/IEC 25000 se organiza en seis apartados principales (véase Figura 5.2).

Figura 5.2. Organización de la familia de normas ISO 25000

- ISO/IEC 2500n – División de Gestión de la Calidad. Las normas que forman este apartado definen todos los modelos, términos y definiciones comunes referenciados por todas las otras normas de la familia 25000.
- ISO/IEC 2501n – División de Modelo de Calidad. Las normas de este apartado presentan modelos de calidad para productos software y sistemas, calidad en uso y datos.
- ISO/IEC 2502n – División de Medición de Calidad. Estas normas incluyen un modelo de referencia de la medición de la calidad de productos software y sistemas, definiciones de medidas de calidad (interna, externa y en uso) y guías prácticas para su aplicación.
- ISO/IEC 2503n – División de Requisitos de Calidad. Estas normas ayudan a especificar requisitos de calidad que pueden ser utilizados en el proceso de licitación de requisitos de calidad de un producto a desarrollar o como entrada del proceso de evaluación.
- ISO/IEC 2504n –División de Evaluación de la Calidad. Este apartado incluye normas que proporcionan requisitos, recomendaciones y guías para la evaluación de productos.
- ISO/IEC 25050-25099 –División de Extensiones. Este apartado incluye normas o informes técnicos que abordan dominios de aplicación específicos o que complementan a otras normas.

5.5.1 Modelo de calidad: ISO/IEC 25010

La norma ISO/IEC 25010 [ISO/IEC, 2011] define dos modelos: un modelo de calidad de producto compuesto por características relacionadas con las propiedades estáticas y dinámicas de un sistema informático; y un modelo de calidad en uso que

propone características relacionadas con el resultado de la interacción cuando un producto se utiliza en un contexto determinado.

El modelo de calidad de producto distingue, como se puede ver en la Figura 5.3, ocho características de calidad de un producto software. Entre ellas se encuentra la mantenibilidad del software.

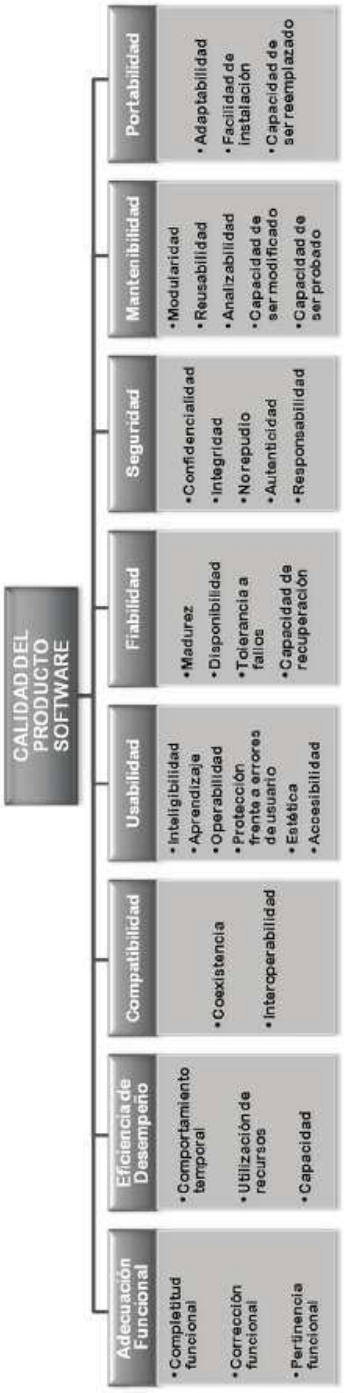


Figura 5.3. Modelo para la calidad del producto software

Según este modelo, la mantenibilidad se subdivide a su vez en:

- Modularidad (modularity). Grado en que un sistema o programa de ordenador está compuesto de componentes discretos de forma tal que un cambio en un componente tenga un impacto mínimo en los demás.
- Analizabilidad (analysability). Grado de efectividad y eficiencia con el cual es posible evaluar el impacto en un producto o sistema de un determinado cambio en una o más de sus partes, o diagnosticar las deficiencias o causas de fallos en un producto, o identificar las partes a modificar.
- Reusabilidad (reusability). Grado en que un activo puede ser utilizado en más de un sistema software o en la construcción de otros activos.
- Modificabilidad (modifiability). Grado con el que un producto o sistema puede ser modificado de forma efectiva y eficiente sin introducir defectos o degradar la calidad de producto existente.
- Capacidad para ser probado (testability). Grado de efectividad y eficiencia con el cual se pueden establecer los criterios de prueba para un sistema, producto o componente y con la que se pueden llevar a cabo las pruebas para determinar si se cumplen dichos criterios.

5.5.2 Evaluación de la calidad: ISO/IEC 25040

La norma ISO/IEC 25040 [ISO/IE, 2011b] propone un modelo de referencia para la evaluación, que considera tanto las entradas al proceso de evaluación (requisitos para la evaluación, especificación de requisitos de calidad, productos a evaluar, etc.), como las restricciones (necesidades, planificación, etc.) y los recursos disponibles (personal, herramientas, equipos informáticos, etc.) para obtener las correspondientes salidas (plan de evaluación, medidas, criterios de decisión, resultados e informe de evaluación, etc.).

5.5.2.1 TAREAS DEL PROCESO DE EVALUACIÓN

En la Figura 5.4 se resumen las tareas del proceso de evaluación que se agrupan en cinco actividades.

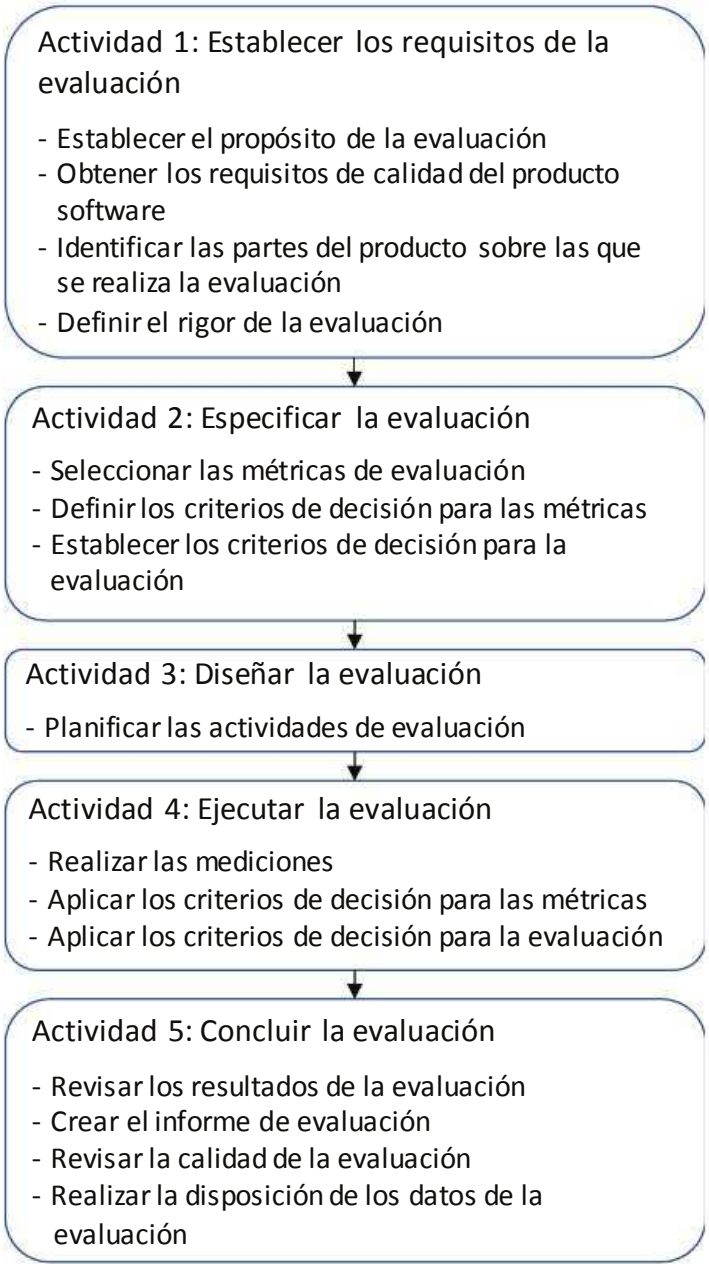


Figura 5.4. Proceso de evaluación de la calidad del producto software [ISO/IE, 2011b]

Establecer los requisitos de la evaluación

Según la norma, esta actividad incluye:

- Establecer el propósito de la evaluación, ya sea la aceptación de productos intermedios o finales, predecir la calidad del producto software final,

seleccionar un producto entre alternativas, decidir la liberación de un producto, etc.

- Obtener los requisitos de calidad del producto software, identificando los stakeholders del mismo.
- Identificar las partes del producto incluidas en la evaluación, especificación de requisitos, documentación de diseño o de pruebas, etc.
- Definir la rigurosidad de la evaluación, es decir, el alcance que cubre la evaluación de calidad respecto a los requisitos de calidad del software, teniendo en cuenta el presupuesto de la evaluación, la fecha objetivo de la evaluación, etc.

Especificar la evaluación

Según la norma, esta actividad se compone de las siguientes tareas:

- Seleccionar medidas de calidad (módulos de evaluación).
- Definir criterios de decisión para medidas de calidad, teniendo en cuenta que los procedimientos de medición deberían medir las características de calidad con suficiente precisión.
- Establecer criterios de decisión para la evaluación, de forma separada para cada característica de calidad bien a base de criterios individuales para las sub-características o una combinación ponderada de las mismas.

Diseñar la evaluación

Esta actividad consiste en planificar las actividades de evaluación teniendo en cuenta los recursos disponibles.

Ejecutar la evaluación

Esta actividad se compone, según la norma, de las siguientes tareas:

- Realizar las mediciones, de acuerdo al plan de evaluación.
- Aplicar los criterios de decisión a las medidas.
- Aplicar los criterios de decisión para la evaluación, identificando cualquier deficiencia en los requisitos de la evaluación, evaluaciones adicionales necesarias, etc.

Concluir la evaluación

Actividad que consiste en:

- Revisar el resultado de la evaluación, entre el evaluador y el solicitante de la evaluación, incluyendo los comentarios sobre la evaluación en la versión final del informe.
- Eliminar los datos de la evaluación, según lo acordado con el solicitante de la evaluación.

5.5.2.2 RECURSOS PARA EL PROCESO DE EVALUACIÓN

Como se puede intuir, todas las tareas del proceso de evaluación de la calidad del producto software requieren una gran cantidad de recursos: herramientas de medición, recursos humanos especializados, un sistema de información para la evaluación del producto, así como una base de conocimiento sobre las métricas, umbrales, etc.

De todos estos recursos resulta fundamental disponer de herramientas que faciliten tanto la toma de las mediciones como la aplicación de los criterios de medición y evaluación de una forma automatizada, evitando errores humanos y facilitando poder medir grandes cantidades de información. Afortunadamente, en los últimos años se han desarrollado varias herramientas de este tipo, ya sean entornos de software libre, como entornos de software propietario (véase capítulo 7).

Por otra parte, también hay que destacar la existencia de laboratorios de evaluación independientes, que realizan una evaluación especializada sobre la calidad del producto, como AQCLab, que es el primero acreditado por ENAC (Entidad Nacional de Acreditación), con el reconocimiento de ILAC (*International Laboratory Accreditation Cooperation*), para la evaluación de la calidad del producto software en base a la ISO/IEC 25000. La existencia de laboratorios de evaluación acreditados ha sido común desde hace varios años en otros sectores como el industrial, el químico o el acústico. Sin embargo, hasta ahora eran desconocidos en el mundo de la ingeniería del software y más concretamente en el de la calidad de los productos software. Los laboratorios de este tipo se acreditan con la norma ISO/IEC 17025 que confirma la competencia técnica del laboratorio y garantiza la fiabilidad en los resultados de los ensayos realizados. El laboratorio AQC Lab cuenta con tres elementos principales, que fueron auditados durante su acreditación y que utiliza durante la evaluación de la calidad de un producto software [Rodríguez Ríos et al., 2014]:

- El Proceso de Evaluación, que adopta la norma ISO/IEC 25040 y la completa con los roles concretos del laboratorio y los procedimientos e instrucciones de trabajo desarrollados.
- El Modelo de Calidad, que define las características y métricas para evaluar el producto software.
- El Entorno de Evaluación, que permite automatizar en gran medida las tareas de la evaluación, de manera que, a partir de mediciones básicas sobre el producto software, permite ir escalando los valores y asignar unos niveles de calidad para las subcaracterísticas y características del modelo.

5.5.3 CERTIFICACIÓN DE LA CALIDAD DE PRODUCTOS SOFTWARE

Al igual que ocurría con la acreditación de laboratorios para otros sectores, la certificación del producto también ha sido desde hace tiempo una práctica muy común, existiendo certificados energéticos en electrodomésticos, certificados de seguridad en vehículos, etc. Sin embargo, en el mundo del software, aunque existen desde hace años certificados para la calidad de los procesos de desarrollo, con modelos como CMMI o normas como ISO/IEC 33000, no ha existido tradicionalmente una certificación para la calidad del propio producto software. Por ello en el año 2013, AENOR (Asociación Española de Normalización y Certificación) decidió incluir en su modelo para el gobierno de las TICs con normas ISO, la certificación del propio producto software en base a la familia de normas ISO/IEC 25000 (Fernández et al., 2013).

Para llevar a cabo esta certificación AENOR colabora con el laboratorio AQCLab, que es el responsable de la evaluación técnica del producto software y AENOR, en base a los resultados de la evaluación, se encarga de la inspección de la viabilidad, recursos y capacidad técnica de la empresa que ha desarrollado el producto.

Actualmente la evaluación de la calidad software se centra en las características de Adecuación Funcional (Rodriguez et al., 2016) y Mantenibilidad.

En la Figura 5.5 se presentan las actividades del proceso de certificación que se describen a continuación:

Figura 5.5. Proceso para la certificación de calidad del producto software

- Actividad 1. El proceso comienza cuando la organización interesada en la calidad del producto software solicita una evaluación a un laboratorio acreditado, como AQCLab. Para ello debe rellenar un formulario con las características del producto software que se quieren evaluar, que es analizado por el laboratorio para emitir un contrato de evaluación con las condiciones del servicio. Aceptado este contrato, la organización hace entrega al laboratorio del producto software a evaluar o le da acceso desde sus instalaciones, en el caso de que la organización no desee que el producto salga de sus sistemas. A partir de aquí, el laboratorio realiza la evaluación haciendo uso del entorno (modelo, proceso y herramientas) basado en ISO/IEC 25000 y acreditado por ENAC.

- Actividad 2. El resultado del paso anterior es un informe de evaluación con los resultados obtenidos, que es entregado a la organización solicitante. En este paso, puede ocurrir que el nivel de calidad obtenido por el producto software no sea suficientemente bueno, en cuyo caso la organización solicitante deberá llevar a cabo las acciones necesarias para mejorar el nivel de calidad, por ejemplo, refactorizar el código, mejorar la especificación de los requisitos o ampliar la cobertura de las pruebas. En este caso, el tiempo que puede transcurrir dependerá el número de defectos que se deben solucionar y de la cantidad de recursos que la organización pueda dedicar para tal fin. Una vez realizadas las mejoras necesarias, la organización deberá repetir el paso 1 del proceso para volver a obtener un informe de evaluación favorable.

- Actividad 3. Cuando el producto software ha obtenido en la evaluación un nivel de calidad 3 o superior (en una escala del 1 al 5), la organización podrá contactar con AENOR, solicitando la certificación del producto e indicando la referencia previa de la evaluación que ha pasado realizada por un laboratorio acreditado.
- Actividad 4. AENOR contacta con el laboratorio evaluador para solicitar los resultados de la evaluación con la referencia indicada por la organización solicitante. Así, la entidad certificadora confirmará la veracidad de la evaluación y los resultados indicados por la organización solicitante.
- Actividad 5. El laboratorio facilita a AENOR el informe de evaluación, a fin de que lleven a cabo el contraste y continúan certificación.
- Actividad 6. Finalmente, AENOR analiza el informe de evaluación facilitado por el laboratorio y realiza una auditoría a la organización solicitante (que puede ser in situ u on-line) para, siguiendo con su reglamento interno de auditoría definido para el producto software, revisar el producto y las características del mismo, la viabilidad de la empresa y sus capacidades técnicas. Como resultado de este proceso de auditoría de certificación, AENOR emite un informe y entrega a la organización un certificado que acredita la calidad del producto software evaluado y la característica o características de software evaluadas [Rodríguez Ríos et al., 2013]. En la Figura 5.6 se muestra un ejemplo del logotipo de certificación para la mantenibilidad y en la Figura 5.7 un ejemplo de certificado.

Figura 5.6. Ejemplo de logotipo AENOR para certificación de mantenibilidad

Este nuevo esquema de certificación basado en la Norma ISO/IEC 25000 ha permitido que durante los últimos años varias decenas de empresas a nivel nacional e internacional hayan evaluado la calidad de sus productos (Rodríguez et al., 2015). En el portal www.iso25000.com se puede encontrar un listado de aquellas empresas que además de haber evaluado su producto, han conseguido también un nivel de mantenibilidad adecuado que les ha permitido alcanzar la certificación de AENOR.

Certificado de Conformidad
Calidad de Producto Software

AENOR
conform

Mantenibilidad
Software

PS-YYYY/NNNN

AENOR, Asociación Española de Normalización y Certificación, certifica que el Producto Software denominado

XXXX, versión X.Y

de la empresa

ZZZZZZZ.

Con domicilio social: YYYYY

Conforme con: La noma ISO/IEC 25000 Ingeniería de Software. Requisitos de calidad y evaluación de producto software (Square)

Para las características: MANTENIBILIDAD SOFTWARE

Sistema de Certificación: Este certificado solo es válido considerando el informe de Evaluación ID:XXXX con fecha AAAA-MM-DD del Laboratorio AQC Lab acreditado por ENAC y el informe de auditoría de certificación del producto software de AENOR XX-YY con fecha AAAA-MM-DD.

Fecha de emisión: AAAA-MM-DD.
Fecha de expiración: AAAA-MM-DD.

Avelino BRITO MARQUINA
Director General de AENOR

AENOR Asociación Española de Normalización y Certificación | Génova, 6. 28004 Madrid. España Tel. 902 102 201 – www.aenor.es

Figura 5.7. Ejemplo de Certificado AENOR para producto software

Entre los testimonios de estas empresas, se destacan los siguientes beneficios de adoptar la Norma ISO/IEC 25000 para mantenibilidad:

- Reducción de hasta un 30% los tiempos dedicados a mantenimiento software.
- Simplificación del producto software hasta en un 40% de líneas de código.
- Reducción de más de un 50% las incidencias correctivas.
- Mejora en la cobertura de las pruebas de hasta un 60%.
- Asegurar al cliente la calidad del producto que se le entrega con evidencias objetivas.
- Establecer acuerdos de nivel de servicio claros, fáciles de comprobar y de cumplir.
- Poder hacer comparativas de mercado entre la calidad de soluciones software para una misma funcionalidad o sector.

5.6 EFECTOS DE LOS CAMBIOS EN EL SOFTWARE

Cuando se le da el visto bueno a un cambio solicitado en el software, el equipo de mantenimiento deberá implementar el cambio. Las tareas a realizar son:

- Modificar los documentos y el código.
- Revisar los documentos y código modificados.
- Probar el código modificado.

Los resultados de estas tareas son un Informe de Modificación del Software (SMR, *Software Modification Report*) y uno o varios elementos de configuración modificados. En el SMR se definen los siguientes aspectos:

- Nombres de los elementos de configuración que han sido modificados.
- Número de versión de cada elemento de configuración modificado.
- Cambios que han sido implementados.
- Fecha de comienzo, fecha de final y esfuerzo requerido (personas-hora).

Los sistemas software pueden llegar a resultar una sangría económica por los costes de mantenimiento. Para evitarlo, los responsables deberían evaluar los efectos de cada cambio, verificar por completo todas las modificaciones en el software, y tener actualizada la documentación; teniendo en cuenta su repercusión sobre las siguientes características del software [Mazza *et al.*, 1994]:

- Eficiencia.
- Consumo de recursos.
- Cohesión.
- Acoplamiento.
- Complejidad.
- Consistencia.
- Transportabilidad.
- Fiabilidad.
- Mantenibilidad.
- Seguridad (security).
- Seguridad de funcionamiento (safety).

Estos efectos pueden evaluarse con la ayuda de herramientas de ingeniería inversa y herramientas de documentación. Las primeras pueden identificar los módulos afectados por un cambio en el nivel de diseño de los programas (por ejemplo, identificando cada módulo que utiliza cierta variable global). Las segundas cuentan con facilidades de referencias cruzadas que pueden rastrear dependencias en el nivel del código fuente.

Muy a menudo, existe más de una opción o manera de cambiar el software para resolver un problema. Los ingenieros de software deberían examinar estas opciones, comparar sus efectos y seleccionar la mejor.

A continuación, se comentan los efectos sobre la complejidad y la mantenibilidad, ya que ambos son los que tienen una relación más directa sobre los costes del mantenimiento (un aumento en la complejidad implica una mayor dificultad de mantenimiento, es decir, una reducción de la mantenibilidad).

5.6.1 Efectos sobre la complejidad

Durante las actividades de mantenimiento la complejidad del software tiende a aumentar, ya que sus estructuras de control tienen que ser extendidas para cumplir nuevos requerimientos [Lehman y Belady, 1985]. Si este fenómeno no se corrige, se llegará a una situación en la que un eventual cambio será impracticable porque requiere más esfuerzo que el disponible para implementarlo. La reducción de la complejidad del software redunda en un aumento de su fiabilidad y mantenibilidad.

La complejidad del software puede ser medida con diversas métricas. Entre las múltiples métricas de complejidad de producto, en McCabe [1982] se propone una «métrica de complejidad esencial» que sirve para medir la distorsión que un cambio produce en las estructuras de control de un módulo.

5.6.2 Efectos sobre la mantenibilidad

Ya se ha comentado que la mantenibilidad es un indicador de la facilidad con que el software puede ser mantenido. La métrica más común para evaluar la mantenibilidad es el «tiempo medio que se tarde en reparar».

Algunos cambios en el software pueden reducir la mantenibilidad. Los que producen este efecto con más asiduidad son:

- Violar los estándares de codificación
- Reducir la cohesión
- Incrementar el acoplamiento
- Incrementar la complejidad esencial

Los costes y beneficios de los cambios que hacen que el software sea más mantenible deberían ser evaluados antes de llevar a cabo dichos cambios. Puesto que todo el código modificado debe ser probado, el coste de volver a probar el código cambiado podría superar la reducción del esfuerzo requerido en futuros cambios.

5.7 MEJORA DE LA MANTENIBILIDAD DE CÓDIGO

5.7.1 Eliminación de Code Smells

En la actualidad, los ingenieros software tratan de mejorar la manenibilidad del código fuente a través de la reducción de lo que se conoce como *code smells*. Un *code smell* también conocido como *bad smell* (mal olor) es una indicación superficial que normalmente se corresponde con un problema de mantenibilidad más severo [Folwer, 2006]. Algunos ejemplos de *code smell* definidos en la literatura son por ejemplo ‘clase Dios’ que es una clase extremadamente larga que hace gran parte de la funcionalidad del sistema. La longitud de una clase en sí no es un problema, pero es un indicador que la mantenibilidad del sistema respecto a dicha clase será peor. Otro ejemplo de *code smell* es ‘envídia de característica’ que indica que una clase realiza numerosas llamadas a funcionalidad de otra clase. Aunque a priori esto puedo no suponer problema alguno, puede esconder un problema de acoplamiento entre clases que empeorará la mantenibilidad.

Además, los malos olores no solo afectan la mantenibilidad, sino que generalmente tienen un impacto directo sobre otras características de calidad del software [Fontana et al., 2013a]. Es por esto, que otros autores consideran los malos

olores como un mecanismo para predecir la aparición de defectos en un software [Palomba et al., 2016].

Se ha realizado numerosos trabajos para realizar la definición y detección de *code smells*. En [Tufano et al., 2015] recogen algunas lecciones aprendidas sobre 'cuando y porqué un programa empieza a "oler".

- **Lección 1**. En la mayoría de las ocasiones el código fuente se ven afectado por malos olores desde su creación. Esto contradice la opinión general de que los malos olores generalmente se deben a efectos colaterales de la evolución del software. Esto implica que la introducción de la mayoría de los olores se puede evitar simplemente realizando controles de calidad durante el desarrollo. De esta forma, en lugar de ejecutar el detector de *code smells* de vez en cuando en todo el sistema, estas herramientas podrían usarse durante actividades críticas (como por ejemplo un nuevo paso a producción) para evitar o al menos limitar la introducción de estos males olores.

- **Lección 2**. Los artefactos de código que se vuelven malolientes como consecuencia de las actividades de mantenimiento y evolución se caracterizan por ciertas tendencias en la evaluación de métricas concretas, que son diferentes de las de los artefactos limpios. Tales resultados fomentan el desarrollo de recomendaciones capaces de alertar a los desarrolladores de software cuando los cambios aplicados a los artefactos del código resultan en tendencias preocupantes para una métrica concreta, que generalmente se convertirán en artefactos afectados por un olor.

- **Lección 3**. Aunque la implementación de nueva funcionalidad y mejora de la existente son principalmente el mecanismo por el que se introducen nuevos olores, existen muchos casos en el que las propias operaciones de refactorización de código sorprendentemente añade nuevos *code smells*.

- **Lección 4**. Los desarrolladores de software recién incorporados al equipo de desarrollo o mantenimiento no son necesariamente los responsables de introducir los malos olores, sino que son aquellos desarrolladores con altas cargas de trabajo. Este resultado pone de relieve que las prácticas de inspección del código deberían fortalecerse cuando los desarrolladores trabajan bajo condiciones estresantes.

Por otra parte, existen trabajos que definen como reducir o eliminar tales *bad smells*. El principal mecanismo para reducir los *bad smells* (y por tanto aumentar la mantenibilidad) es la refactorización de código [Hegedus et al., 2017], y de hecho

se han propuesto operadores de refactorización especificos para cada uno de los *code smells* (véase Capítulo 4). La mayoría de las técnicas de detección de *code smells* se basan en el cálculo de diferentes métricas y observaciones directas sobre el código fuente, pero el tamaño y otras características de diseño no son comúnmente consideradas. [Fontana et al., 2013b] proponen un enfoque basado en algoritmos de *machine learning* para detectar malos olores considerando otros aspectos de diseño. Otros autores como [Aniche et al., 2016] tienen en cuenta aspectos de la arquitectura, tales como la aplicación del patrón Modelo-Vista-Controlador (MVC), para definir *code smells* concretos.

Otro mecanismo para limitar los efectos de los *code smells* sobre la mantenibilidad es prevenirlos. Por ejemplo, [Walter y Alkhaeir, 2016] analizan como el uso de patrones de diseño conducen a un menor número de code smells.

Los *code smells* son potencialmente la causa de baja comprensión y problemas de mantenibilidad. De ahí que han sido estudiados en muchos lenguajes como C, Java o C#, aunque han sido considerados de forma superficial en otros lenguajes como Python [Chen et al., 2018]. A pesar de que los desarrolladores conocen y están preocupados por los efectos de los code smells, sólo un número muy bajo de ingenieros software son capaces de detectarlos y de evaluar sus efectos en la mantenibilidad [Taibi et al., 2017]

5.7.2 Gestión de la Clonación

Un *code smell* concreto es el porcentaje de líneas de código clonadas en un cierto software. Los clones son fragmentos de código iguales o similares que se encuentran en el código fuente de los programas y que entorpecen la realización efectiva de modificaciones (efectos colaterales, repetición de la misma modificación por cada clon, encarecimiento de las pruebas, etc.). La presencia de clones se debe a diferentes razones:

- Copiar y pegar por comodidad
- Estilos de codificación
- Operaciones sobre T.A.D.'s que actúan sobre diferentes tipos de datos
- Mejora del rendimiento (evitar llamadas a funciones, por ejemplo)
- «Accidente»

[Harder, 2013] añade el desarrollo de un sistema software por parte de múltiples desarrolladores como una causa de introducción de clonación. Esto se debe a entornos distribuidos de desarrolladores y deficiencias en la comunicación.

De acuerdo a [Saini et al., 2016] existen cuatro tipos de clones en código fuente de acuerdo a la naturaleza de su similitud basada en el texto o su significado:

- Tipo 1 (clones exactos): dos fragmentos de código son copias exactas una respecto a la otra exceptuando caracteres en blanco, saltos de línea y comentarios.
- Tipo 2 (renombrados): dos fragmentos de código se consideran clones cuando son copias (como en el caso anterior) excepto por el nombre de variables, tipos, literales y nombres de funciones.
- Tipo 3 (clones de grafo): dos fragmentos de código similares, pero con modificaciones tales como sentencias añadidas o eliminadas y el uso de diferentes identificadores como en el caso anterior. El grafo de llamadas sigue siendo el mismo en los dos clones.
- Tipo 4 (clones semánticos): dos fragmentos de código sintácticamente diferentes y con un grafo de llamadas diferentes, pero semánticamente similares, es decir, tienen la misma funcionalidad.

Como ejemplo, esta subsección presenta el método de detección de clones de tipo 3 propuesto por [Baxter *et al.,* 1998]. A primera vista, la primera solución que uno estudiaría para encontrar clones en dos módulos diferentes de un mismo programa consistiría en realizar una comparación del código fuente de ambos módulos. Sin embargo, como se muestra en el ejemplo (escrito en un lenguaje ficticio) de la Figura 5.8, existen multitud de posibles dificultades para automatizar el proceso utilizando este tipo de solución:

- Distinta ubicación de los tokens.
- Uso de distintos identificadores.
- Uso de distintos tipos de datos (en el caso ya mencionado de los T.A.D.'s)
- Colocación de comentarios.

El método propuesto por los autores, en lugar de inspeccionar el código fuente a nivel léxico o sintáctico, construye y revisa el árbol de sintaxis abstracta del programa, así como sus correspondientes subárboles.

Fragmento 1 en módulo A	Fragmento 2 en módulo B

`void Anadir(int e, lista l) {`
`int i:=1; boolean enc:=false;`
`while (not enc) {`
`if l[i].ocupado then`
`i++;`
`else enc=true;`
`}`
`l.info:=e;`
`l.ocupado:=true;`
`}`
	`void Insertar(lista l, int info)`
	`{`
	`int cont; boolean enc;`
	`Elemento e;`
	`cont:=1;`
	`enc:=false;`
	`while (enc==false) {`
	`e:=l[cont];`
	`if (not e.ocupado) then {`
	`e.ocupado:=true;`
	`e.info:=info;`
	`l[cont]:=e;`
	`enc:=true;`
	`} else cont:=cont+1;`
	`}`
	`}`

Figura 5.8. Dos fragmentos de código semánticamente muy similares, pero sintáctica y léxicamente bien diferentes

En un árbol de sintaxis abstracta, cada nodo puede representar una instrucción diferente, un identificador, etc., y puede llevar asociado un tipo de datos. A su vez, de cada nodo puede colgar un subárbol representando un conjunto de instrucciones, como se ilustra en la Figura 5.9.

El algoritmo, cuya versión más básica se detalla en la Figura 5.10, busca subárboles de sintaxis abstracta iguales o parecidos lo más grandes posible. Mediante el primer bucle *Para cada*, el algoritmo decide si considerar el subárbol «s» como un subárbol susceptible de ser considerado un clon (la instrucción *i=1*, por ejemplo, es un subárbol de sintaxis abstracta presente en muchos programas, pero el hecho de que aparezca diez o cien veces en un mismo programa no significa que deba ser considerado un clon). Esta decisión la toma comparando el valor de la función *Hash* con el valor prefijado que se habrá almacenado en *PesoMínimo*. En caso afirmativo, el subárbol *s* se introduce en la estructura *TablaHash*, la cual es procesada a continuación en el segundo bucle.

```
  ...
if a=b then
   for i=1 to f(n)
     aux = aux +
f(i, i-1)
   end for
   a=0
end if
  ....
```

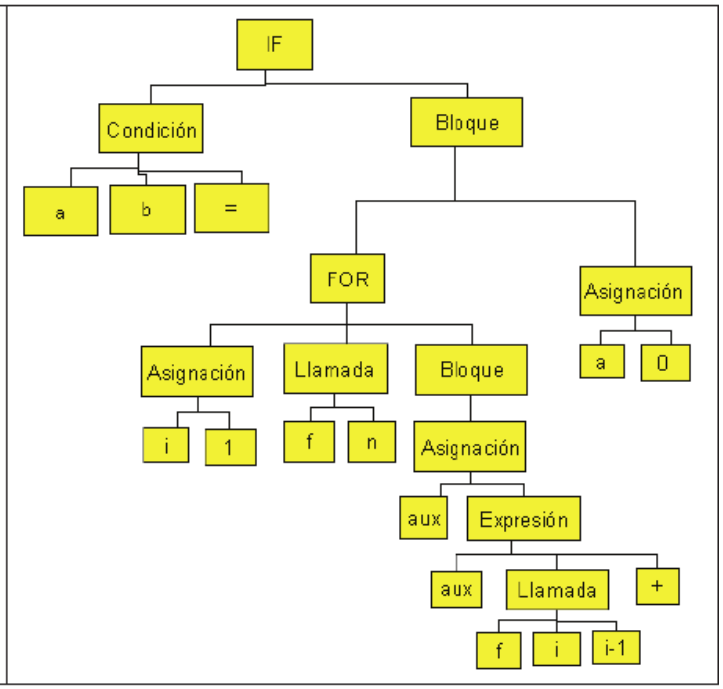


Figura 5.9. Un fragmento de programa y su posible representación como un árbol de sintaxis abstracta

El segundo bucle "*Para cada*" compara, dos a dos, los subárboles almacenados en la misma entrada de la *TablaHash*, determinando que dos subárboles son clones si su similitud es mayor que el valor predeterminado *UmbralDeSimilitud*. En caso afirmativo, comprueba si los subárboles *s* de cada subárbol s_1 y s_2 que acaba de identificar como clones ya estaban clasificados como clones. Nuevamente en caso afirmativo, saca los subárboles *s* del conjunto de clones, pues s_1 y s_2 son clones más grandes, en los cuales ya estaban incluidos sus subárboles *s*.

```
Clones=∅

Para cada s ∈ Subárboles
          Si Hash(s) ≤ PesoMínimo entonces
          TablaHash = TablaHash ∪ {s}
Para cada (s1, s2) que están en la misma entrada de TablaHash
          Si Similitud(s1,s2)≥UmbralDeSimilitud entonces
          Para cada s ∈ Subárboles(s1)
          Si s ∈ Clones entonces
          Clones=Clones - {s}
          Para cada s ∈ Subárboles(s2)
          Si s ∈ Clones entonces
          Clones=Clones - {s}
          Clones=Clones ∪ {s2} ∪ {s2}
```

Figura 5.10. Algoritmo básico para la detección de clones

5.7.2.1 OTRAS TÉCNICAS DE DETECCIÓN DE CLONES

A demás de la técnica anteriormente explicada a modo de ejemplo, existen otras muchas técnicas de detección de clones. Por ejemplo, [Haque et al., 2016] propone una técnica genérica para detectar clones desde diferentes artefactos de código mediante la segmentación de sub-programgmas, módulos y funciones. Esta técnica puede detectar los cuatro tipos de clones. Otros autores van más allá e incluso detectan clones en el código compilado, ya que aseguran que dos fragmentos de código diferentes pueden conducir en muchos casos a clones a nivel de código compilado [Kononenko et al., 2014].

Aunque los investigadores han intentado detectar clones por décadas, muchas de las propuestas fallan al intentar escalarlas a sistemas enormes. En este sentido, [Kim et al., 2017] proponen un enfoque para la detección escalable de clones con un nivel de granularidad a nivel de función y un filtrado de la longitud que reduce el número de comparaciones entre signaturas de funciones y métodos. También, [Patil et al., 2015] proponen una técnica de detección de clones basada en la descentralización de la computación y reducción de código.

A parte de esta solución, en la actualidad la mayoría de los esfuerzos van encaminados en la detección de clones de tipo 4, clones semánticos. Por ejemplo, [Sheneamer y Kalita, 2016] proponen una técnica de Machine Learning (basada en técnicas de Inteligencia Artificial) para la detección de clones sintácticos y semánticos analizando los árboles sintácticos abstractos y los gráfos de dependencia entre programas.

[Saha et al., 2010] desarrollaron una técnica que tiene en cuenta la genealogía de clones y permite analizar desde diferentes dimensiones como evolucionan los clones durante la evolución de los sistemas de información.

La mayor parte de técnicas de detección de clones propuestas han sido automatizadas con multitud de herramientas. En un ejercicio de evaluar dichas herramientas varios autores han realizado estudios de eficacia y eficiencia basados en benchmarks [Svajlenko y Roy, 2014][Svajlenko y Roy, 2015].

5.7.2.2 REFACTORIZACIÓN Y MANEJO DE CLONES

La detección de clones es importante y el primer paso para mejor la mantenibilidad. Una vez se han detectado los clones en el código fuente hay dos posibles tratamientos, refactorizar el código para eliminarlos, o bien simplemente manejarlos y establecer una trazabilidad que permita evolucionar el sistema sin propagar errores entre clones.

Por ejemplo, *JDeodorant* [Mazinanian et al., 2016] es una herramienta que permite eliminar los clones detectados en el código fuente. [Krishnan y Tsantalis, 2013] definen un algoritmo de optimización para el proceso de emparejamiento de sentencias comunes entre clones cuando los clones van a ser refactorizados. [Mondal et al., 2017] ha investigado cuáles de los clones detectados tienen una alta probabilidad de contener errores y de esta forma priorizar la refactorización. [Tsantalis et al., 2015] proponen un enfoque para la evaluar clones dos a dos y decidir si pueden ser refactorizados sin añadir errores o defectos adicionales. De forma similar, [Chen et al., 2017] definen una técnica para manejar la refactorización de clones basada en el reconocimiento de ciertos patrones en el código fuente. De esta forma, se presenta un análisis agrupado de los clones que se han refactorizado y aquellas copias que no han podido ser adecuadamente refactorizados. A parte de la refactorización, en cuanto al manejo de clones, [Duala et al., 2007] proponen realizar la traza de clones durante la evolución del código fuente considerando el concepto de regiones de clones que abstrae la localización exacta de clones.

5.8 DEUDA TÉCNICA

5.8.1 Introducción

De acuerdo con la DRAE[6], el concepto de "Deuda" significa "Obligación que alguien tiene de pagar, satisfacer o reintegrar a otra persona algo, por lo común dinero", es decir, la necesidad de satisfacer un compromiso económico con respecto a una persona u entidad. Partiendo de esta base, podríamos extrapolar este concepto al de "Deuda técnica" ampliando la definición como la obligación que alguien (empresa de desarrollo o el propietario del software, según el contexto) tiene de pagar, satisfacer o reintegrar a otra persona algo, por lo común dinero, con motivo del desarrollo y uso de un software que tiene ciertas carencias de calidad que deberán ser solventadas en algún momento del ciclo de vida del software. De otro modo, la deuda técnica se puede ver como las "actividades que un equipo o miembros de un equipo decide no realizar correctamente y que afectarán al desarrollo en el futuro si no son resueltas" [Sterling, 2010], o tal y como indica [Lim et al., 2012], "desarrollar código inmaduro o incorrecto para entregar, con el objetivo de hacer negocio, un producto lo antes posible".

6 Diccionario de la Real Academia de la Lengua

La deuda técnica es un compromiso, ya que puede considerarse como un resultado invisible de decisiones tomadas en el pasado que afectan al futuro [Kruchten et al., 2013]. La deuda técnica, puede entenderse de forma intuitiva como el coste de refactorizar (resolver) todos los defectos tecnológicos (*technological flaws*) entre los que podemos considerar los *malos olores*, antipatrones o la falta de documentación entre otros, tal y como resume la siguiente ecuación:

$$Deuda\ Técnica = \Sigma Refactorizar(Defecto\ Tecnológico)$$

En [OMG, 2017], un estándar de reciente creación dirigido a la automatización de la medición de la deuda técnica, el propio concepto la "deuda" es una metáfora de carácter "económico" que pretende dar un valor a la calidad del software, y que puede dividirse en los distintos elementos que impactan a varios niveles (ver Figura 5-11):

- Deuda técnica, que se resume como el coste de reparar los problemas y debilidades estructurales existentes en el código, y que deben ser resueltos durante la fase de mantenimiento.
- Principal, que es el coste, en términos de horas, de remediar los problemas existentes en el código, teniendo en cuenta el coste en diseño, implementación y testeo del software.
- Interés, debido al coste excesivo de tener que modificar un código complejo e innecesario, el uso de recursos adicionales por el un código ineficiente (lo cual también está relacionado con la propia sostenibilidad del software), etc.
- Riesgo de negocio, derivado de los posibles efectos que puede tener el sistema en producción y cuyos defectos técnicos podrían provocar dañar la integridad de la información, degradación del rendimiento, problemas de seguridad, etc.
- Responsabilidad, que son los costes a nivel de negocio que ocurren cuando, un código con problemas técnicos, produce problemas operativos en la organización.
- Coste de oportunidad, que hace referencia a los nuevos proyectos que dejarían de abordarse por tener que abordar la resolución de la deuda técnica, lo que supone un compromiso a la hora de decidir en qué medida se dedicarán esfuerzos para eliminar la deuda técnica de un sistema.

No obstante, y a pesar de la connotación negativa del concepto de la deuda técnica, hay estudios revelan que ésta es inevitable y necesaria, por lo que lo que el punto de equilibrio más importante es realizar una correcta gestión de la misma [Lim et al., 2012b][Fernández-Sánchez et al., 2017].

Figura 5.11. Factores de la Deuda Técnica y niveles de influencia

5.8.2 Tipos de deuda técnica

De acuerdo a la taxonomía de la deuda técnica de MacConnell [7], se pueden identificar los siguientes tipos de deuda técnica:

7 *http://www.construx.com/10x_Software_Development/Technical_Debt/*

1. Deuda no intencionada: producida de forma no intencionada por el desarrollo de un trabajo de baja calidad.
2. Deuda intencionada: realizada de forma intencionada, y que puede clasificarse en:
   - Deuda a corto plazo, realizada por razones tácticas, de forma reactiva.
     - Deuda a corto plazo focalizada, que consiste en "atajos" (o defectos) individuales perfectamente identificables en el software desarrollado.
     - Deuda a corto plazo no focalizada, que consiste en la introducción de numerosos pequeños "atajos" (o defectos).
   - Deuda a largo plazo, realizada por razones estratégicas, realizada de forma proactiva.

5.8.3 Patrones de aparición de la deuda técnica

Tal y como se documenta en [Sterling, 2010], existen ciertos patrones de trabajo por parte de los equipos de desarrollo de software que implican un aumento de la deuda técnica en el producto final. Dichos patrones son:

- Presión en la planificación: Pueden ser muchas las razones por las cuales un equipo de desarrollo se acerca a una fecha límite dentro de su planificación, pero llegado ese momento, resulta común encontrarse con que "hay que darse prisa" o "hay que terminarlo". Esto implica que el equipo de desarrollo debe "recortar" por algún sitio para cumplir el calendario, lo que provoca un incremento de la deuda técnica.
- Duplicación: Duplicar código o crear clones a sido una (mala) práctica común en el desarrollo de software. Dicha práctica tiene como origen la falta de experiencia por parte de los miembros del equipo, programación utilizando el "copiar-pegar", presión de la dirección del equipo por entregar el producto, etc. Aunque existen herramientas automatizadas que detectan el código duplicado, como los analizadores estáticos, pero existen otras formas de evitar la duplicidad de código, como puede ser las metodologías basadas en *pair programming*, evitar la tentación de realizar el "copiar-pegar", y la mejora continua del diseño del software, detectando las copias de código y refactorizando las mismas en soluciones más óptimas.

- Hazlo "bien" a la primera: En el lado opuesto de la duplicación de código, está el esfuerzo excesivo por hacer bien el software a la primera, por lo que se pone un esfuerzo excesivo en la planificación y el diseño. Sin embargo, es muy probable que el diseño del software tenga que cambiar debido a múltiples factores (requisitos de usuario cambiantes, pruebas de aceptación, etc.), por lo cual, un esfuerzo excesivo en ese "primer intento puede resultar contraproducente. Es por ello, importante realizar un diseño flexible que permita abordar y asumir la naturaleza cambiante de un software en desarrollo.

5.8.4 Medición de la deuda técnica

Medir la deuda técnica implica medir aquellos defectos del software que, pudiendo haberse evitado, se encuentran presentes en el código del mismo. Tal y como se propone en el [OMG, 2017], la medición de la deuda técnica se puede llevar a cabo en base a un conjunto de medidas que actúan sobre unas características en las que se organizan los distintos defectos técnicos. Dichas características se corresponden con los tipos de problema que puede causar la deuda técnica, y son:

- Seguridad, entre los que se puede encontrar la declaración de cláusulas *catch*, el uso de excepciones genéricas en cláusulas *Throws*, uso de credenciales codificadas en el mismo código fuente para la autenticación remota, etc.
- Fiabilidad, como son las dependencias inter-módulo, configuraciones de red codificadas directamente en el código fuente, etc.
- Rendimiento/eficiencia, como ocurre con las operaciones implementadas mediante bucles demasiado costosos, uso de memoria que no se libera, realización de consultas a base de datos con demasiadas columnas, etc.
- Mantenibilidad, demasiados elementos de código en comentarios, niveles de complejidad ciclomática demasiado altos, métodos con demasiados parámetros, árboles de herencia con demasiados niveles de profundidad, etc.

La medición de todos estos tipos de defectos técnicas o patrones, puede automatizarse mediante entornos de análisis estático que detectan la gran parte dichas violaciones o defectos en el código [Marinescu, 2012], ofreciendo una estimación de la deuda en base a las horas que, de acuerdo con las heurísticas, implican la resolución de cada uno de los problemas en el software.

5.9 LECTURAS RECOMENDADAS

- ✓ *Mistrik, I., Soley, R., Ali, N., Grundy, J. y Tekinerdogan, B. eds. (2016). Software Quality Assurance in Large Scale and Complex Software-Intensive Systems. Amsterdam, Morgan Kaufmann.*

  Esta recopilación presenta diferentes temas de gran interés para el aseguramiento de la calidad del software, enfocándose a sistemas complejos.

- ✓ *Rodríguez, M. y Piattini, M. (2018). Calidad de productos software. Madrid, Ra-Ma.*

  Esta obra profundiza en todos los aspectos tratados en este capítulo, ofreciendo métricas y técnicas concretas para la evaluación y mejora de la calidad de los productos software.

5.10 SITIOS WEB RECOMENDADOS

- ✓ *www.ISO25000.com*

  En este portal se reúne información relativa a la mejora de la calidad de los productos software y a la utilización de la norma ISO 25000.

- ✓ *www.aqclab.es*

  Es el sitio web del laboratorio AQC Lab, el primero acreditado por ENAC para la evaluación de la manteniblidad del producto software.

5.11 EJERCICIOS

Ejercicio 1

Enumere diez recomendaciones prácticas concretas que pueden mejorar la mantenibilidad del código fuente.

Ejercicio 2

Explique las principales dificultades que existen para medir la mantenibilidad de un sistema software.

Ejercicio 3

Indique características de la programación orientada a crear aplicaciones web de página única podrían reducir la mantenibilidad. Explicar las causas.

Ejercicio 4

Señale cinco factores que influyen en la mantenibilidad del software.

Ejercicio 5

Enumere y describa brevemente los tipos de clonación de código que existen.

Ejercicio 6

Analice las propuestas que se han realizado sobre técnicas de detección de clones en las principales revistas y conferencias de mantenimiento.

Ejercicio 7

¿Cuántas certificaciones de mantenibilidad de software se han emitido? Consulte el portal iso25000.com

Ejercicio 8

¿Cómo estimaría el proceso necesario para mejorar la mantenibilidad del software?.

Ejercicio 9

Describa las interacciones entre las diferentes subcaracterísticas de la mantenibilidad: modularidad (modularity), analizabilidad (analysability), reusabilidad (reusability), modificabilidad (modifiability) y capacidad para ser probado (testability).

Ejercicio 10

¿Cómo puede afectar la metodología de desarrollo (p.ej. tradicional, ágil, etc.) a la mantenibilidad del código desarrollado?.

6

MÉTRICAS PARA EL MANTENIMIENTO

Las organizaciones se enfrentan, cada día más, a la necesidad de estimar y controlar el mantenimiento de software utilizando medidas más rigurosas que las obtenidas a partir de la simple experiencia personal o la analogía del proyecto con otros anteriores.

En los últimos años se han desarrollado un conjunto de métricas que pueden utilizarse en el mantenimiento con diferentes fines: estimar los costes de las intervenciones, controlar objetivamente el proceso o detectar los módulos del sistema con más propensión a fallos.

6.1 CONCEPTOS GENERALES

Para proporcionar mayor mantenibilidad a un producto software (y menores costes de mantenimiento, por tanto), necesitamos conocer los factores que influyen en ella, entre los cuales -como vimos- se encuentran los siguientes:

- Comprensibilidad, o facilidad de comprensión del producto.
- Modificabilidad, o facilidad de modificación.
- Facilidad de prueba.

Estos tres factores se ven afectados, sobre todo, por la *complejidad* [Li y Cheng, 1987]. En la misma línea, Banker *et al.* [1993] y Gill y Kemerer [1991] muestran cómo los costes y la productividad durante el mantenimiento aumentan o disminuyen en función de que aumente o disminuya la complejidad del producto software.

La medida del coste de mantenimiento será entonces una medida *indirecta*, pues para conocerlo necesitamos saber, previamente, el valor de otros atributos del producto, tal y como ilustramos en la Figura 6.1.

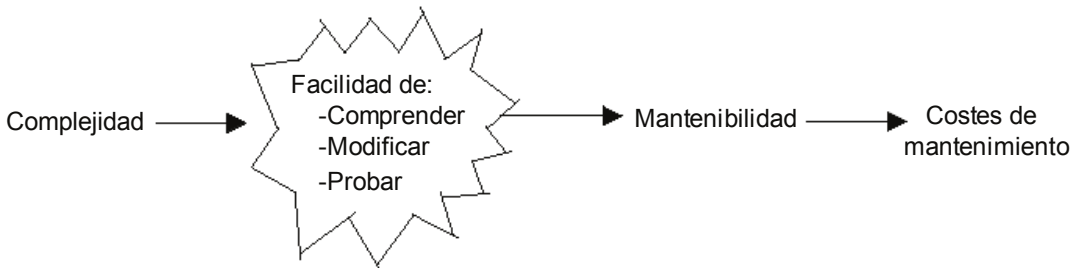


Figura 6.1. Relación entre la complejidad y la mantenibilidad

En Ingeniería de Software, los instrumentos de medida utilizados para medir atributos se conocen con el nombre de *métricas*. Además, dependiendo del propósito de la medida que realicemos, distinguiremos entre:

- Medición del proyecto, basado en la gestión de proyectos.
- Medición del producto, centrado en su calidad y aspectos técnicos.
- Medición del proceso, basado en el estudio y control de la capacidad de los procesos, así como en la gestión de los cambios en el proceso.

El proceso software constituye la base a partir de la cual se realiza el trabajo dentro de una organización. Dichos procesos se aplican en la práctica en forma de proyectos. Como resultado de la ejecución de proyectos concretos se utilizan recursos y se obtienen productos. Por lo tanto, para establecer un marco de medición dentro de una organización es necesario definir, recoger y analizar métricas sobre el proceso, el proyecto y recursos asociados, así como del producto software.

Para Henderson-Sellers [1996], una *medida de producto* es aquella que representa una característica del producto en un instante de tiempo claramente determinado. La medida puede realizarse tanto en la fase de diseño como durante el mantenimiento, pero no contiene sino una descripción instantánea del atributo medido. El mismo autor afirma que no se puede entender el proceso (en nuestro caso, de mantenimiento) tomando sólo una o unas pocas medidas instantáneas. Una *medida de proceso* sirve para evaluar diferentes aspectos del proceso de mantenimiento, permitiendo a sus responsables efectuar previsiones de costes y esfuerzo, detectar anomalías en el proceso de mantenimiento o mejorar las actividades.

Sin embargo, la implantación de un programa de métricas en una organización, aunque ayuda a estimar los costes de desarrollo y mantenimiento, también la hace incurrir en otros costes que antes no existían, y que Card y Glass [1990] sitúan entre el 5-15% de los costes totales.

6.2 MÉTRICAS DE PRODUCTO

En la sección anterior vimos que la complejidad es el atributo que más influye en la mantenibilidad de los productos software y, por tanto, en su coste de mantenimiento.

Existen, sin embargo, otros atributos que poseen también influencia directa sobre el coste de mantenimiento: el tamaño del producto, la cantidad y calidad de los comentarios y otro tipo de documentación, el grado de familiaridad del personal de mantenimiento con el producto, etc.

En esta sección comentamos algunos de estos atributos y mostramos métricas que nos permitirán calcular sus valores.

6.2.1 Complejidad

La definición de complejidad más ampliamente aceptada es la de McCabe [1976], que define la *complejidad ciclomática* de un programa como el número de caminos linealmente independientes que podemos contar en el grafo de flujo que lo representa.

Un camino es linealmente independiente cuando introduce un nuevo conjunto de sentencias de proceso o una nueva condición [Pressman, 2014].

En la Figura 6.2 representamos un fragmento de código en un lenguaje ficticio, que podría ser un programa completo o una subrutina, y su grafo de flujo.

```
1  begin
 n=1
2   while n<100 do
3      i=1
4      while i<n do
5          if n mod 2 =0
then
6                KP = KP +
1
7          else
                 KN = KN -
1
           end if
8          i = i + 1
9      end while
A      n = n + 1
B   end while
C  end
```

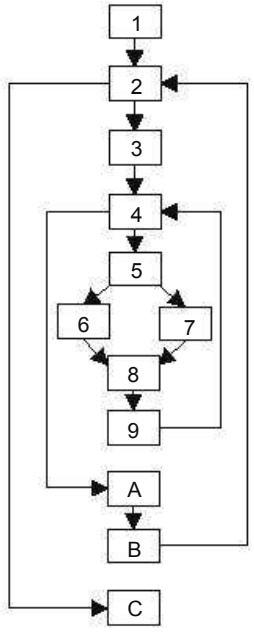


Figura 6.2. Un programa y su grafo de flujo

En el grafo de la Figura 6.2 podemos encontrar los siguientes caminos linealmente independientes:

- 1-2-C
- 1-2-3-4-A-B-2-C
- 1-2-3-4-5-6-8-9-4-A-B-2-C
- 1-2-3-4-5-7-8-9-4-A-B-2-C

Estamos, por tanto, ante un fragmento de código de complejidad ciclomática (representada por *v(G)*) de valor 4. Sin embargo, *v(G)* puede calcularse de cualquiera de las siguientes otras maneras:

v(G)=e-n+2 ... siendo *e* el número de arcos y *n* el número de nodos del grafo de flujo.
v(G)=1+d ... donde *d* es el número de nodos de decisión presentes en el grafo.
v(G)=nº de zonas del grafo

Apliquemos estos métodos al ejemplo anterior:

- Según el número de arcos y nodos: v(G)= 14 - 12 + 2 = 4.
- Según el número de nodos de decisión: hay 3 nodos de decisión en G (los nodos 2, 4 y 5), con lo cual: v(G)=1 + 3 = 4.
- Según el número de zonas del grafo: en la Figura 6.3 reproducimos el mismo grafo de la Figura 6.2 con las cuatro zonas marcadas en diferentes tonos.

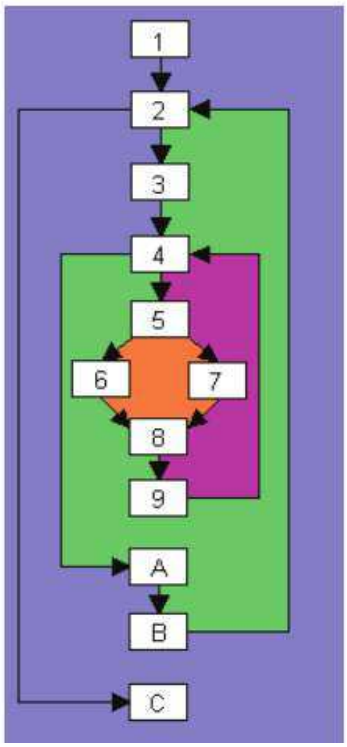


Figura 6.3. Zonas del grafo para el cálculo de v(G)

Además de servir para el cálculo de la complejidad ciclomática, los caminos linealmente independientes de un grafo de flujo representan todos los caminos del programa que deben ser sometidos a pruebas de funcionamiento. Es decir, recorriendo cada uno de los caminos linealmente independientes, tenemos la garantía de haber recorrido al menos una vez todas las sentencias ejecutables del programa.

A la hora de calcular la complejidad ciclomática de un producto, en los niveles superiores de abstracción procedimental damos el valor 1 a la complejidad ciclomática de los componentes del siguiente nivel de abstracción. Ilustramos esta idea en el ejemplo de la Figura 6.4, en la que el procedimiento C consta únicamente de dos llamadas a los procedimientos A y B. La complejidad de C es 1, independientemente de cuál sea el valor de v(A) y v(B).

```
Procedimiento A(f : Fichero;
          Variable fSalida : Fichero);
Variables i : integer;
                r : registro;
inicio
  i=0
  mientras i<Tamaño(f) hacer
          Leer(r)
          Si r.saldo<100000 entonces
          si r.edad>21 entonces
             Escribir(fSalida, r)
          fin si
          si no
 si r.saldo<1000000 entonces
                Escribir(fSalida, r)
 Fin si
          Fin si
          i = i+1
  fin mientras
fin A              v(A)=5
```

```
procedimiento B(variable
     f : Fichero);
inicio
         OrdenacionPorMezcla(f)
fin B
           v(B)=1
procedimiento C(f : Fichero)
variables aux : Fichero;
inicio
       A(f, aux);
       B(aux);
fin C
           v(C)=1
```

Figura 6.4. La complejidad ciclomática depende del nivel de abstracción

McCabe [1976] considera *muy alta* la complejidad ciclomática de un grafo de flujo G (y del fragmento de código al que representa) con v(G)>10, ya que valores superiores hacen que las pruebas del módulo sean extremadamente difíciles de realizar. Otros autores [Grady, 1994] sitúan en 15 este valor límite. En cualquier caso, los módulos con alta complejidad ciclomática deben ser reconstruidos por completo, quizás dividiéndolos en módulos de menor complejidad.

6.2.2 Tamaño

Podemos medir el tamaño de un producto software utilizando unidades de muy diverso tipo: número de líneas de código, número de módulos, número de páginas de listado del código fuente o número de rutinas, por citar algunas.

Algunos autores han propuesto conjuntos de propiedades que deben cumplir ciertas métricas para ser consideradas de determinados tipos. Así, por ejemplo, Weyuker [1988] publicó un conjunto de axiomas, posteriormente muy discutidos y sobre cuya validez no existe unanimidad, que debía cumplir cualquier métrica para ser considerada como «de complejidad». Para esta sección son muy útiles las siguientes propiedades, propuestas por Briand *et al.* [1996], y que deben ser cumplidas, según estos autores, por cualquier métrica «de tamaño»:

1. El tamaño de un sistema es mayor o igual a cero.
2. El tamaño de un sistema es cero si el número de atributos de tamaño del sistema es cero.
3. El tamaño de un sistema es igual a la suma de los tamaños de sus módulos.

Líneas de código

El tamaño en líneas de código (LDC) —o en miles de líneas (abreviadamente KLDC, de *kilolíneas de código*)— del producto es un atributo objetivo y de fácil medición.

Para que el resultado sea efectivamente objetivo es preciso, sin embargo, definir previamente el concepto de línea de código, de manera que todos nos refiramos a lo mismo. Según Fenton y Pfleeger [1997], la definición más extendida es la proporcionada por Hewlett-Packard: «cualquier sentencia del programa, excepto las de comentario y las líneas en blanco».

No obstante, la inclusión de líneas de comentario (LDCC) en el código fuente de un programa requiere ciertos costes y esfuerzos por parte del programador o del mantenedor, que no deben dejar de ser considerados al medir diferentes aspectos del producto o del personal del equipo: dedicar tiempo a la inclusión de comentarios en el código implica dejar de escribir sentencias ejecutables y productivas, pero mejora la comprensibilidad de los programas e influye positivamente en su facilidad de mantenimiento. De este modo, podemos utilizar el número de líneas comentadas, el número total de líneas y tal vez algún otro atributo para obtener una métrica relacionada con la mantenibilidad del producto. Fenton y Pfleeger [1997] definen la *Densidad de comentarios* como el cociente entre el número de líneas de código

comentadas entre el número total de líneas de código. No obstante, más que la cantidad de comentarios importa su calidad ya que uno de los principales problemas de los comentarios es que se quedan desactualizados debido a cambios en el código subyacente sin que se actualice dicho comentario. Lo que lleva al efecto contrario, es decir, no solo no mejora la comprensibilidad si no que podría empeorarla.

Por otra parte, el número de LDC es altamente dependiente del lenguaje de programación en que se encuentre implementado el producto. Por ello, para realizar operaciones como comparar el tamaño de dos productos de funcionalidad similar, pero construidos en diferentes lenguajes de programación, o para utilizar los datos de tamaño con el fin de realizar medidas de predicción, no debe utilizarse como estimador el número de LDC. Sí es posible, sin embargo, normalizar el número de LDC entre diferentes lenguajes, de lo cual nos ocupamos a continuación.

Puntos función

La métrica de los puntos función fue propuesta por Albretch [1979], y pretende medir la *cantidad de funcionalidad* de un programa en función de diversos parámetros. En su forma más básica, el número de puntos de función de un producto software se calcula mediante la Tabla 6.1.

En la columna «Nº» de la tabla anotamos el número de apariciones de cada atributo, asociándole un factor de peso según consideremos que su complejidad es baja, media o alta. A continuación, anotamos la suma de la última columna en la casilla PF, obteniendo el total de «Puntos de función sin ajustar».

El número de puntos de función ajustados (PFA) del producto software se calcula según la expresión siguiente:

$$PFA = PF \times \left[0.65 + 0.01 \times \sum_{i=1}^{14} F_i \right]$$

Cada elemento F_i (los cuales se relacionan a continuación), representa un *valor de ajuste de la complejidad*. A cada uno se le da un valor entre 0 y 5, según las características del producto que se va a construir:

- Comunicación de datos.
- Rendimiento.
- Frecuencia de transacciones.
- Requisitos de manejo del usuario final.
- Procesos complejos.

- Mantenibilidad.
- Instalación en múltiples lugares.
- Funciones distribuidas.
- Carga de trabajo.
- Entrada de datos on-line.
- Actualizaciones on-line.
- Utilización con otros sistemas.
- Facilidad de operación.
- Facilidad de cambio.

Parámetro				Peso		Total
Entradas de usuario	Compl. baja			3		
	Compl. media			4		
	Compl. alta			5		
Salidas de usuario	Compl. baj			4		
	Compl. media			5		
	Compl. alta			7		
Peticiones de usuario	Compl. baja			3		
	Compl. media			4		
	Compl. alta			6		
Archivos	Compl. baja			7		
	Compl. media			10		
	Compl. alta			15		
Interfaces externas	Compl. baja			5		
	Compl. media			7		
	Compl. alta			10		
					PF:	

Tabla 6.1. Tabla para el cálculo de los puntos de función

El uso de los puntos de función en Ingeniería del Software ha sido, sin embargo, muy discutido por Dolado y Fernández [1999]. En el caso particular del mantenimiento, diversos estudios han mostrado resultados negativos en cuanto a su adecuación [Niessink y VanVliet, 1997]. Así, por ejemplo, se indica que no son aplicables a la mayoría de las intervenciones de mantenimiento correctivo, ya que éstas afectan por lo general a sólo unas pocas líneas de código, que no llegan a constituir ni siquiera un punto-función. No obstante, son muchas las organizaciones de software que utilizan puntos-función para estimar el coste de las intervenciones, sobre todo de mantenimiento perfectivo.

Equivalencia entre líneas de código y puntos de función

Para estimar el número de puntos de función de una aplicación hemos considerado los cinco parámetros que aparecen en la tabla de la Tabla 6.1. Es claro, sin embargo, que el desarrollo del código que gestione, por ejemplo, una petición de usuario de complejidad alta, requiere diferente nivel de esfuerzo según el lenguaje de programación en que lo estemos desarrollando.

La siguiente tabla muestra la equivalencia entre líneas de código y puntos de función, según Behrens [1993] y Jones [1991]:

Lenguaje	LDC / PF	
	Behrens [1983]	Jones [1991]
Ensamblador	320	300
C	150	128
Cobol	106	105
Fortran	106	105
Pascal	91	90
Ada	---	70
Lenguajes orientados a objetos	21	30
Lenguajes de 4ª generación (4GL)	40	20
Generadores de código	---	15

Tabla 6.2. Correspondencia entre LDC y PF.

6.2.3 Ecuación de Putnam

Putnam [1978] construyó, a partir de la recogida y estudio de datos empíricos, un modelo para la estimación de tamaños y esfuerzos de desarrollo de productos software.

La siguiente ecuación permite estimar el tamaño (T) del producto en líneas de código:

$$T = C \times K^{1/3} \times t_d^{4/3}$$

Donde C es una constante que mide el nivel tecnológico de la organización, K representa el esfuerzo total de desarrollo (incluyendo mantenimiento) del producto en personas-año y t_d es el tiempo de desarrollo en años.

C toma valores aproximadamente en el intervalo [2000, 11000]. El valor inferior indica que el entorno de desarrollo es pobre y carente de metodologías; el superior significa que el entorno es excelente, con uso de herramientas automáticas y muy buenas metodologías. El valor de C se obtiene de datos de desarrollo de productos anteriores.

El esfuerzo de desarrollo en personas-año podemos obtenerlo despejando K:

$$K = \frac{T^3}{C^3 * t_d^4}$$

Como se puede observar, la dependencia del esfuerzo respecto del tiempo de desarrollo no es lineal: si el cliente accede a aumentar el plazo inicial de entrega del producto, la organización consigue reducir los esfuerzos necesarios de forma altamente significativa.

6.2.4 Medición de la Mantenibilidad

Imaginemos que tenemos que elegir entre dos sistemas diferentes, ambos desarrollados con el mismo lenguaje y que tienen el mismo tamaño. La elección vendría determinada por el más fácil de mantener (lo que implica menores costes de mantenimiento), pero para saberlo tenemos que conocer su mantenibilidad.

Por otro lado, los gestores aborrecen las sorpresas, especialmente si éstas significan un aumento imprevisto de los costes. Un método de medir la mantenibilidad ayuda a los responsables a tomar una decisión de mantenimiento, por ejemplo, elegir si un componente deberá ser mantenido o completamente reescrito para reducir los costes de mantenimiento futuros.

Por tanto, en orden a determinar mejor los costes de mantenimiento del software, los mantenedores deberían medir la mantenibilidad de los sistemas desarrollados y utilizar la mantenibilidad de los sistemas como un factor en la determinación de costes.

Las métricas durante el desarrollo ayudan a determinar la cuantía en que se está incorporando en el software el objetivo de mantenibilidad. Una vez el software ha sido desarrollado, las métricas pueden guiar durante el proceso de mantenimiento, bien para evaluar el impacto de un cambio (mantenibilidad de la nueva configuración obtenida), o bien para realizar un análisis comparativo entre varias propuestas o aproximaciones diferentes para realizar una modificación requerida por los usuarios.

En los párrafos anteriores surge una pregunta: ¿Cómo medir la mantenibilidad? La respuesta a esta pregunta se encuentra en la estrecha relación que existe entre los conceptos de calidad del software y de mantenibilidad. Por esta razón, los estándares ISO e IEEE proponen métricas de calidad para medir la mantenibilidad del software.

También se ha investigado sobre modelos de regresión polinomial que predicen la mantenibilidad del software. Para ello, se utiliza una combinación de variables de predicción en una ecuación polinomial para definir un índice de mantenibilidad (MI) [Welker y Oman, 1995].

Las métricas de mantenibilidad no pueden medir el coste de realizar un cambio particular al sistema software, sino que miden aspectos de la complejidad y la calidad de los programas, ya que, como se comentó anteriormente, existe una alta correlación entre la complejidad y la mantenibilidad.

La mantenibilidad no está restringida únicamente al código; es también un atributo de otros elementos de un producto software, incluyendo los documentos de especificación y diseño, y los documentos del plan de pruebas. En suma, deberán existir medidas de mantenibilidad para todos los elementos software que están o estarán sometidos a mantenimiento.

Existen dos aproximaciones diferentes para medir la mantenibilidad, según se tomen en cuenta los aspectos externos o internos de los atributos considerados [Fenton y Pfleeger, 1997]. La mantenibilidad es claramente un atributo de producto externo porque no depende únicamente del producto, sino también de la persona que realiza el mantenimiento, del soporte documental, de las herramientas disponibles, y de la utilización real del software. La aproximación externa más directa para medir la mantenibilidad consiste en medir el proceso de mantenimiento; si el proceso es efectivo, entonces se asume que el producto es mantenible.

La aproximación alternativa se utiliza para identificar atributos internos de producto y determinar cuáles de ellos son predictivos de las medidas de proceso. Aunque esta aproximación es más práctica puesto que las medidas pueden realizarse mucho más fácilmente, es importante recordar que la mantenibilidad nunca se puede definir sólo en términos de medidas internas (por ejemplo, como las que se mostraron: líneas de código, complejidad ciclomática, etc.).

6.2.4.1 MEDIDAS EXTERNAS DE LA MANTENIBILIDAD

Desde este punto de vista externo, se buscan medidas que caracterizan la facilidad de aplicar el proceso de mantenimiento a un producto software. Por simplicidad, se trabaja con el concepto de implementar un cambio, que es válido para los cuatro tipos de mantenimiento (correctivo, adaptativo, preventivo y perfectivo).

La característica clave de la mantenibilidad será la velocidad de implementar un cambio una vez que la necesidad de su realización está definida. Por esta razón, se define una medida llamada tiempo medio para reparación (MTTR), que es el tiempo medio que necesita el equipo de mantenimiento para implementar un cambio y poner de nuevo operativo el sistema software modificado. Para calcular esta medida es necesario registrar cuidadosamente la siguiente información:

- 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 pueden utilizarse otras medidas dependientes del entorno si, previamente, ha sido registrada y está disponible la información necesaria:
- Ratio entre el tiempo total para implementar los cambios y el número total de cambios implementados.
- 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. Al fin y al cabo, hay que tener en cuenta que las medidas externas de la mantenibilidad sobre cómo funciona dicho proceso y la calidad del mismo.

6.2.4.2 MEDIDAS INTERNAS DE LA MANTENIBILIDAD

Se han propuesto numerosas medidas de atributos internos como indicadores de la mantenibilidad. Para determinar las medidas (y sus atributos internos relacionados) que más afectan a la mantenibilidad, la selección se debe realizar en combinación con medidas externas de la mantenibilidad. Por ejemplo, en [Porter y Selby, 1990] se han utilizado técnicas estadísticas (análisis de árboles de

clasificación) para identificar cuáles son las mejores medidas de producto que son las mejores para predecir los errores de interfaz con probabilidad de aparecer durante el mantenimiento.

El número ciclomático [McCabe, 1976] o cualquier otra medida sencilla resultan habitualmente insuficientes por sí mismas como indicadores de la mantenibilidad, ya que capturan una visión muy reducida de la estructura y complejidad del software. Por esta razón se han realizado múltiples estudios para determinar valores límites para otras medidas más sofisticadas [Fenton y Pfleeger, 1997].

Existen diversas propuestas para intentar describir las interrelaciones entre los atributos estructurales internos y la mantenibilidad. Uno de dichos atributos es la legibilidad, que se considera uno de los aspectos claves para la mantenibilidad. A su vez, los atributos internos que determinan la estructura de los documentos se consideran unos indicadores importantes de la legibilidad. Un ejemplo de medida de la legibilidad de programas es la propuesta realizada por De Young y Kampen [1979]. Estos investigadores hicieron un análisis de regresión a partir de datos subjetivos de evaluación de la legibilidad y establecieron una interrelación entre la legibilidad y tres atributos internos del código fuente: longitud media de los nombres de las variables, número de líneas que contienen sentencias, y número ciclomático de McCabe.

En Coleman et al. [1994] se encuentra una propuesta para definir modelos de esfuerzo de mantenimiento utilizando análisis de regresión sobre medidas de atributos internos.

6.2.5 Medida del Envejecimiento Software

[Visaggio, 2001] presenta un estudio que generaliza algunos de los síntomas del envejecimiento de un sistema de información heredado. El envejecimiento software puede verse como la perdida de ciertas características de calidad como consecuencia del mantenimiento evolutivo del sistema. Entre estas características de calidad, por supuesto, también se encuentra la mantenibilidad. Cada síntoma de envejecimiento presentado en el estudio especifica un conjunto de métricas y la interpretación de estos resultados sugieren ciertas acciones de mantenimiento.

El estudio realizado por [Visaggio, 2001] se basa en un análisis retrospectivo de los datos recopilados durante la ejecución de grandes proyectos de mantenimiento de sistemas de información heredados con cierta complejidad. Cada síntoma de envejecimiento es definido como una descripción conceptual, las causas que lo generan y los efectos sobre el proceso de mantenimiento.

6.2.5.1 POLUCIÓN

Se refiere a artefactos software (código, documentación, configuración) que no sirven para la explotación del software, bien porque ya no son usados, o bien porque siendo usados están desactualizados, o bien porque hay clones de ciertos artefactos software.

- **Programas duplicados**, se refiere a clones de código, que no necesariamente son exactamente igual. Como se vio en capítulos anteriores existen varias técnicas para detectarlos.
- **Programas obsoletos**. Programas que disponen de código fuente, pero no artefactos ejecutables. Se puede medir con búsquedas simples.
- **Programas sin código fuente**. Programas ejecutables de los cuales no hay código fuente. Se puede medir con búsquedas simples.
- **Componentes no usados**. Son componente software no usados y por tanto no útiles para el sistema de información.
- **Código muerto**. Es el código que no es alcanzable en ejecución y por tanto no es útil. Existen técnicas de análisis estático (sintáctico) y dinámico (en ejecución) que determinan y por lo tanto cuentan el número de líneas de código muerto.
- **Datos muertos.** Similar a lo anterior detecta los datos (variables de programa) que nos son alcanzables por un programa en ejecución.

6.2.5.2 CONOCIMIENTO EMBEBIDO

Hace referencia a la lógica de negocio que está presente en el código fuente del sistema de información y que no está representado en ningún otro sitio (por ejemplo, en la documentación). Este conocimiento complica la tarea de la retirada del software y abordar la implementación de uno nuevo, ya que es complicado extraer una traza del conocimiento de negocio embebido en el código.

- **Módulos y datos incomprensibles**. Módulos cuyo significado y utilidad no puede derivarse de la documentación presente.
- **Capacidades perdidas**. Estas son funciones de aplicación que contiene el sistema heredado, pero que no se pueden localizar con precisión en los componentes de software. Ambas métricas pueden medirse con técnicas de depuración, *profiling* y *slicing*.

6.2.5.3 EMPOBRECIMIENTO LÉXICO

Hace referencia a los nombres de variables, funciones y módulos que realmente no representan un significado acorde con la semántica de dichas variables, funciones y módulos.

- **Inconsistencias en nombre de datos y módulos**. Esta métrica tiene que ser calculada mediante la lectura del código fuente y con intervención humana.

6.2.5.4 ACOPLAMIENTO

Hace referencia a la fuerte dependencia entre módulos del sistema heredado. No solo a través del flujo de control sino a través de los datos.

- **Ficheros patológicos**. Ficheros que son creados y/o modificados por diferentes programas, lo que hace difícil su sustitución o borrado durante el mantenimiento. Es fácil contarlos, pero no así detectarlos puesto que habría que considerar todos los programas de una organización o aquellos que potencialmente pueden modificarlos.

- **Datos de control**. Son datos usados para controlar el flujo de ejecución y el comportamiento de un sistema de información heredado. Hay analizadores sintácticos de código que ofrecen una buena aproximación para detectar este tipo de datos.

- **Complejidad de módulo.** Determina la complejidad de los módulos basándose en la computación de las sentencias condicionales (ya sean explicitas o implícitas) [Visaggio, 2001].

6.2.5.5 ARQUITECTURAS DE DATOS SUPERPUESTAS

En muchas ocasiones, en los diferentes procesos de mantenimiento se imponen ciertas arquitecturas de datos que entran en conflicto con la arquitectura y diseño con la cual se comenzó el desarrollo original del sistema. Esto hace que en muchas ocasiones se detecten conflictos y colisiones debido a la convivencia de dos o más arquitecturas diferentes.

- **Ficheros no útiles**. Ficheros no usados por ningún programa, o usados por programas no usados.

- **Ficheros obsoletos**. Un fichero usado por al menos un programa, pero no existen programas que escriban en dicho fichero.
- **Ficheros temporales**. Aquellos creados y leídos, pero nunca actualizados ni eliminados.
- **Ficheros permanentes**. Aquellos creados, leídos, modificados, pero nunca eliminados.
- **Ficheros anómalos**. Ficheros leídos, modificados, pero que no son creados por ningún programa.
- **Datos semánticamente redundantes**. Dos datos se describen como redundantes en lo que respecta al dominio semántico si el dominio de definición de uno está contenido o es igual al del otro, y si cada valor igual en los dos dominios de definición puede interpretarse de la misma manera para ambos datos.
- **Datos computacionalmente redundantes**. Son datos redundantes producidos por un mismo programa.
- **Datos estructurales**. Metadatos que soportan la estructura de los datos del sistema de información.
- **Estructura de datos solapadas.** Hace referencia a tablas y espacios de datos compartidos por varios programas.

6.3 MÉTODOS DE ESTIMACIÓN DEL ESFUERZO DE MANTENIMIENTO

La estimación del esfuerzo de las intervenciones de mantenimiento puede resultar muy importante para los gestores, cuando planifican actividades de mantenimiento y realizan análisis de costes/beneficios [Jorgensen, 1995]. De acuerdo con este mismo autor, los investigadores han dedicado más tiempo a la elaboración de modelos de estimación de esfuerzo para nuevos desarrollos que para mantenimiento, aunque esta tendencia ha comenzado a cambiar desde que el artículo referido fue publicado.

En esta sección presentamos un resumen de algunos métodos de estimación del esfuerzo de mantenimiento propuestos. La mayoría de los métodos de estimación del esfuerzo se basan en la predicción de tiempo de acuerdo al cálculo de una serie de métricas [Wu et al., 2016].

6.3.1 Estimación por analogía

La estimación por analogía se basa en el principio de comparar las características del sistema que vamos a modificar con el mismo conjunto de características del mismo u otros sistemas, que previamente hemos modificado. Si se dispone de datos relativos al esfuerzo empleado en estas intervenciones pasadas, se podrá realizar una estimación del esfuerzo necesario para la futura intervención.

El conjunto de características a utilizar debe estar restringido a la información disponible en el momento de hacer la predicción. Una vez seleccionadas las características, se realiza algún tipo de análisis, bien matemático, bien subjetivo (juicio experto), que enfrente los valores de los atributos históricos con los valores del producto que se va a intervenir.

Shepperd *et al.* [1996] presentan un método y una herramienta para estimación de costes basada en la minimización de la Distancia Euclídea en un espacio n-dimensional, siendo *n* el número de atributos que se comparan. La elección de los atributos debe limitarse a la información disponible en el momento en que se requiere la predicción. El esfuerzo computacional necesario para encontrar las mejores analogías y calcular el esfuerzo aumenta exponencialmente en función del número de atributos seleccionados (20 atributos a comparar de 21 proyectos necesitaron 42 horas de análisis, mientras que 12 atributos utilizaron 10 minutos para el mismo número de proyectos).

6.3.2 Modelo COCOMO para mantenimiento

Granja y Barranco [1997] realizan una serie de modificaciones al conocido modelo COCOMO de estimación de costes, de tal manera que obtienen una versión aplicable a la previsión de los costes de mantenimiento.

En el modelo COCOMO original se define la *Tasa de Cambios Anual* (TCA) de la siguiente manera:

TCA= (NLN + NLM) / NLT siendo NLN el número de líneas nuevas, NLM el número de líneas modificadas y NLT el número total de líneas.

Conociendo la expresión anterior, y teniendo en cuenta el coste del desarrollo (CD), el *Coste de Mantenimiento* (CM) se calcula con esta expresión:

$$CM=TCA * CD$$

A partir de este punto, los autores introducen un nuevo parámetro, el *Índice de Mantenibilidad* (IM), que mide la facilidad de mantenimiento del producto

considerado. Este índice, cuya forma de cálculo mostramos más adelante, influye directamente en el coste de mantenimiento, de manera que la expresión de éste queda de esta forma:

$$CM=TCA*CD*IM$$

Para estos autores, toda acción de mantenimiento puede dividirse en tres tareas:

- Comprensión de los cambios que deben hacerse.
- Realización de las modificaciones necesarias.
- Pruebas de los cambios realizados.

El coste de mantenimiento (CM) puede expresarse como la suma de los costes empleados en cada una de ellas:

$CM = CM_C + CM_R + CM_P$, siendo CM_c los costes de la comprensión, CM_R los costes de la realización de los cambios y CM_P costes de las pruebas.

Esta expresión se puede desglosar de esta manera:

$CM_C = TCA * CD * I_C$ I_C : índice de comprensibilidad

$CM_R = TCA * CD * I_R$ ...donde: I_R : IM de la realización de cambios

$CM_P = TCA * CD * I_P$ I_P : IM de las pruebas

El coste de mantenimiento en personas-mes queda, finalmente, de esta forma:

$$CM = CM_C + CM_R + CM_P = TCA * CD * (I_C + I_R + I_P)$$

De aquí quedan por conocer los diferentes índices de mantenibilidad para cada una de las tres etapas que Granja y Barranco distinguen en el mantenimiento, así como la tasa de cambios anual, que es redefinida en este modelo.

Los índices se calculan en función de las tres métricas siguientes:

- X_C : porcentaje de líneas comentadas por cada cien, para calcular I_C.
- X_R : porcentaje de líneas sin datos constantes por cada cien, para calcular I_R.
- X_P : número de errores comprobados por cada cien líneas de código, para I_P.

Deben utilizarse datos históricos, procedentes de proyectos ya mantenidos, para obtener las funciones que nos den el valor de los índices a partir de las métricas anteriores. Así mismo, de la misma tabla histórica se extraen diferentes matrices que nos permitirán el cálculo de la tasa de cambios anual (TCA).

6.3.3 Modelado del mantenimiento como un sistema dinámico

Calzolari *et al.* [1998] proponen un original método para modelar el esfuerzo de mantenimiento utilizando sistemas dinámicos. Se basan en el modelo predador/presa introducido en 1972 por May.

La idea básica del modelo de May es que, en ausencia de presas, los predadores se extinguen y, en ausencia de predadores, la población de presas alcanza y sobrepasa la capacidad del entorno para alimentarlas. En los casos intermedios se alcanza un equilibrio estable en el sentido de que un incremento en el número de presas permite a los predadores cazar y reproducirse, lo cual causa que decrezca el número de presas; esto, a su vez, provoca que se reduzca el número de predadores.

Los autores realizan la siguiente comparación del modelo dinámico de May con el esfuerzo de mantenimiento correctivo: cuando se corrigen defectos, la actividad de correctivo disminuye. Sin embargo, a diferencia del modelo biológico-dinámico de May, los defectos no tienen capacidad para reproducirse. La única forma de que aparezcan nuevos errores en el software es mediante la entrega de una nueva versión del producto (esto no es realmente una reproducción de defectos —como cabría esperar de una población en sentido estricto—, sino una inserción instantánea de defectos; esto es, de presas).

La similitud del esfuerzo de mantenimiento con el modelo dinámico de May se basa en que:

1. El mantenimiento correctivo es esencialmente predador de defectos software, y el esfuerzo de mantenimiento se alimenta de errores descubiertos por el usuario.

2. Los mantenimientos perfectivo y adaptativo se alimentan de necesidades del usuario, y el esfuerzo de estos dos tipos de mantenimiento se adapta a la cantidad de solicitudes de estos dos tipos de mantenimiento.

En el artículo referenciado, se expone el método aplicándolo a mantenimiento correctivo. Tras la adaptación del modelo original al mantenimiento, los autores concluyen lo siguiente:

1. Se prevén aumentos de correctivo inmediatamente después de cada nueva versión de la aplicación. Tras este incremento, el esfuerzo de mantenimiento disminuye, debido a que el número de presas ha disminuido suficientemente. Se espera que este comportamiento se repita tras cada entrega. Además, el ajuste en la predicción resulta muy bueno.
2. La duración del esfuerzo de mantenimiento dependerá del grado en que se haya modificado el producto software.

6.3.4 Estimación del esfuerzo de mantenimiento con puntos-función

Niessink y van Vliet [1997] realizan ciertos análisis comparativos entre el esfuerzo de mantenimiento de varias intervenciones y el número de puntos-función modificados, contados según métodos diferentes. No obtienen, sin embargo, resultados satisfactorios para ningún modelo predictivo, pero sí que observan que el esfuerzo de mantenimiento es mucho más dependiente del tamaño del componente que se va a cambiar que del tamaño del propio cambio (medidos ambos parámetros en puntos-función). Es decir, que:

$$Esfuerzo \approx K \times Tamaño\ del\ componente \times (1 + \square \times Tamaño\ del\ cambio)$$

... en vez de:

$$Esfuerzo \approx K \times Tamaño\ del\ cambio$$

La falta de adecuación de los puntos-función para realizar estimaciones ha sido también discutida en Dolado y Fernández [1999].

6.3.5 Análisis de métodos de Jorgensen

Este autor [Jorgensen, 1995] compara la fiabilidad en la predicción de diferentes métodos de estimación, de los tipos que listamos más adelante. El estudio es realizado aplicando cada método a un conjunto de 100 intervenciones de mantenimiento. De cada intervención de mantenimiento se recogen los siguientes datos:

- *Causa*, que se corresponde con el tipo de mantenimiento (correctivo, adaptativo, perfectivo o preventivo).
- Prioridad de la tarea (alta, media o baja).
- Años de experiencia como mantenedor.

- *Confianza* del mantenedor, o grado de conocimiento del mantenedor, en lo relativo a si sabe resolver la petición inmediatamente después de haber leído u oído la especificación. El grado de conocimiento puede ser *Sabe cómo resolver la tarea*, *Puede saber resolverla* o *No sabe cómo resolverla.*
- Nivel de estudios del mantenedor.
- Horas dedicadas a la intervención.
- Tamaño de la intervención (medida como la suma del número de LDC añadidas, modificadas y borradas, incluyendo líneas de comentario).
- Lenguaje de programación
- Tipo del cambio (introducción o borrado de módulos, cambio de interfaz, cambio del flujo de control, cambio de declaraciones de datos, cambio de datos o sentencias de asignación).
- Fuentes de información utilizadas (comunicación con los usuarios, documentación del usuario, documentación del lenguaje o la herramienta, comunicación con personal del sistema, comunicación con otros mantenedores y uso de la documentación del sistema).
- Edad de la aplicación cambiada.
- Tamaño de la aplicación cambiada.
- Descripción informal de la intervención.

Una vez caracterizadas con estas informaciones, el autor realizó un análisis de correlación entre cada atributo y el esfuerzo dedicado a la intervención, de manera que deja de considerar algunos atributos y redefine otros. Finalmente, las variables que utilizará son las siguientes:

Atributo	Observaciones
Causa	0→ correctivo; 1→ cualquier otro
Cambio	0→ más del 50% del esfuerzo se empleó en actualización de código, comparado con la inserción y borrado. 1→ otro caso
Modo	0→ más del 50% del esfuerzo se empleó en desarrollar nuevos módulos; 1→ otro caso
Confianza	0→ el mantenedor está seguro de cómo resolver la tarea inmediatamente después de haber leído u oído la especificación por primera vez. 1→ otro caso

Tabla 6.3. Atributos considerados en el análisis.

El resto de variables dejaron de ser consideradas, pues no había una correlación suficientemente significativa. Los modelos analizados toman como parámetros los valores de los atributos mostrados en la Tabla 6.3, y son los siguientes:

- Modelos simples: Esfuerzo = Tamaño / Productividad media
- Modelos de regresión simple.
- Modelos de regresión múltiple.
- Modelos que utilizan redes neuronales, tomando como muestra la proporcionada por la herramienta PlaNet versión 5.6.
- Modelos que utilizan reconocimiento de patrones.

Para cada modelo, el autor calcula diferentes medidas, basadas en la *Magnitud del Error Relativo*:

- *MMRE*, que es la media de la magnitud del error relativo.
- *MdMRE*, que es la mediana de la magnitud del error relativo.
- *PRED1(0.25)*, que es el porcentaje de intervenciones con un error relativo menor que 0.25.
- *PRED1(0.50)*, que es el porcentaje de intervenciones con un error relativo menor que 0.50.
- *PRED2(0.25, 0.50)*, que es el porcentaje de intervenciones con un error relativo menor que 0.25, o con |Esfuerzo real - Esfuerzo predicho|≤ 0.5 días

Tras el cálculo de estas medidas, el autor concluye que los modelos más fiables son:

- Log(Esfuerzo) = K + a × Modo + b × Modo × log(Tamaño) + c × Confianza, minlog(Esfuerzo actual - Esfuerzo predicho)) ^{2}
- Log(Esfuerzo) = K + a × log(Tamaño) + b × Causa + c × log(Tamaño) × Confianza + d × Cambio + e × log(Tamaño) × Cambio + f × Modo + g × log(Tamaño) × Modo + h × Confianza + i × log(Tamaño) × Confianza, minlog(Esfuerzo actual - Esfuerzo predicho)) ^{2}
- Un método híbrido, que utiliza regresión y reconocimiento de patrones.

6.4 CALIDAD EN PROYECTOS DE MANTENIMIENTO

Así como medimos productos y procesos de mantenimiento, previa identificación de los atributos que influyen en ellos, podemos también identificar y medir los atributos de proyectos y en concreto del departamento de mantenimiento de una organización con objeto de conocer su nivel de calidad.

La medición del proyecto y sus recursos asociados constituye el elemento principal sobre el que se basa el estudio de las métricas del proceso software. Cuando se mide el proyecto el objetivo fundamental que se pretende es el de reducir el coste total y el tiempo de desarrollo del mismo. Los indicadores de proyecto permiten al administrador de software [Pressman, 2014]:

- Evaluar el estado del proyecto en curso.
- Realizar un seguimiento de los riesgos potenciales.
- Detectar las áreas de problemas antes de que se conviertan en "críticas".
- Ajustar el flujo y las tareas de trabajo.
- Evaluar la habilidad del equipo del proyecto en controlar la calidad de los productos de trabajo de la ingeniería del software.

En relación con las métricas de proceso, las mediciones del proyecto de software se consideran a un nivel táctico, es decir, las métricas de proyectos y los indicadores derivados de ellos son utilizados por un administrador de proyectos y por un equipo de mantenimiento software para adaptar el flujo de trabajo del proyecto y las actividades técnicas de mantenimiento.

Para medir la calidad de un equipo de mantenimiento software y sus proyectos, debemos identificar las variables de las cuales depende este atributo y utilizar una métrica para conocer su valor.

En muchas organizaciones de software se intenta medir el rendimiento de sus miembros mediante relaciones entre atributos de tamaño y de tiempo. Así, por ejemplo, se dice que el programador A es mejor que B porque aquél ha mantenido una media de 2000 LDC al mes y éste sólo 1500. Las métricas de este tipo carecen de validez desde el momento en que A trabaje en ensamblador y B en java Puede pensarse, sin embargo, que sí son aplicables en el caso de que ambos programadores utilicen el mismo lenguaje, pero volvemos a encontrar dudas razonables si el programador A mantiene un fragmento de un programa de control del tráfico aéreo de un aeropuerto, y B una pequeña aplicación de contabilidad. Como afirman Fenton y Pfleeger [1997], «las medidas de longitud no capturan información sobre la calidad y utilidad del software, ni hacen medidas de funcionalidad ni productividad».

Sneed [1997] identifica los siguientes atributos como utilizables para evaluar el rendimiento de un departamento de mantenimiento, y propone métricas para calcularlos:

- Productividad.
- Fiabilidad.
- Grado de adherencia a los estándares de documentación.
- Consistencia de la documentación.

La medida de la *productividad* viene dada por la siguiente métrica:

$$\text{Productividad} = \frac{\text{Tamaño} \times \text{Tasa de cambios}}{\text{Esfuerzo de mantenimiento}}$$

Este autor mide la fiabilidad en función de los errores debidos a cambios realizados por el equipo de mantenimiento:

$$\text{Fiabilidad} = \frac{\text{Errores}}{LDC_t + LTC_c}$$

En la expresión de la fiabilidad, LDC_t representa el número de líneas de código totales y LDC_c el número de líneas de código cambiadas, bien por adición, supresión o modificación.

El grado de adherencia a los estándares de documentación se mide según el número de deficiencias de documentación encontradas y el tamaño del producto:

$$Adherencia = 1 - \frac{Deficiencias}{Tamaño}$$

Por otra parte, entendemos por *documentación inconsistente* aquella que es incompatible con el código existente en el sistema. Por ejemplo, será inconsistente la documentación de una función que ya no existe, o aquella que indica que una rutina toma dos parámetros enteros como entrada y en el código se ve que sólo toma uno.

Sneed [1997] no propone una métrica para este último atributo; pero sí lo utiliza, junto a los anteriores, para obtener dos métricas: una para lo que él llama

degradación del departamento de mantenimiento y otra para la *nota objetiva del departamento de mantenimiento* que, respectivamente, se calculan mediante estas expresiones:

*Degradación = Productividad * Fiabilidad * Adherencia * Consistencia*
Nota objetiva = Media (Productividad, Fiabilidad, Adherencia, Consistencia)

La opinión de los clientes respecto del departamento de mantenimiento es también un factor muy importante que debe tenerse en cuenta ([Pigoski, 1996]; [Sneed, 1997]). Para la medición de este atributo, Sneed propone la entrega de cuestionarios a los clientes, con preguntas puntuables entre 0 y 10 puntos. La nota que el usuario da al departamento se coloca en el intervalo [0, 1] mediante la siguiente expresión:

Nota del usuario = Total puntos obtenidos / Total puntos posibles

Finalmente, la calidad del departamento se evalúa multiplicando la nota objetiva y la nota del usuario:

*Calidad del departamento = Nota del usuario * Nota objetiva*

Otros recursos relevantes que considerar son los equipos de trabajo y las herramientas [Fenton et al., 1997]. La productividad de los miembros del equipo de trabajo depende de muchos factores, como por ejemplo la estructura del equipo, las herramientas y los métodos utilizados. Por otro lado, también se considera la experiencia como un factor importante de productividad, aunque es difícil de medir. También es importante considerar de forma separada la experiencia de los individuos y la experiencia del equipo, ya que, aunque los integrantes de un equipo puedan tener buena experiencia, el equipo puede que no funcione bien y la productividad sea baja. En general la experiencia individual se puede medir con escalas ordinales, pero es importante considerar que la experiencia se actualiza con mucha frecuencia. En relación con las herramientas, es importante establecer su relación con otras variables del proyecto como la eficacia de las herramientas en términos de la calidad obtenida, tiempo de entrega y la productividad del personal que las utilizó. De hecho, los modelos de estimación consideran el uso de las herramientas a la hora de estimar el tamaño o coste de desarrollo de software.

Tal y como indica [Pressman, 2014], se pueden obtener otras métricas de proyectos una vez que comienza el desarrollo del producto propiamente dicho y en este aspecto es importante realizar un seguimiento de los errores detectados durante todas las tareas de Ingeniería del Software. A medida que el software va evolucionando desde la especificación al diseño, se recopilan las métricas técnicas para evaluar la calidad del diseño y para proporcionar indicadores que influirán en el

enfoque a seguir para la generación de código y para las pruebas. El uso de métricas para los proyectos tiene dos características fundamentales [McDermid, 1991]: estas métricas se utilizan para minimizar la planificación de mantenimiento guiando los ajustes necesarios que eviten retrasos y atenúen problemas y riesgos potenciales; y se utilizan para evaluar la calidad de los productos en el momento actual con el fin de poder mejorarlos.

6.5 MÉTRICAS PARA ENTORNOS ESPECÍFICOS

En esta sección presentamos algunas métricas diseñadas expresamente para determinados entornos.

6.5.1 Métricas para programas COBOL

En este apartado presentamos algunas métricas para programas COBOL, debido fundamentalmente a que sigue siendo todavía el lenguaje para transacciones de negocio más utilizado a nivel internacional, y en el que se encuentra implementada la mayoría de los sistemas que se mantienen en la actualidad.

Exponemos los métodos de control del mantenimiento de Natale [1995] y Rossi *et al.* [1991], y un método que ayuda a mejorar el código que se mantiene mediante «trampas defensivas», debido a Vesely [1997].

En Natale [1995] se propone, con el fin de controlar el mantenimiento de programas COBOL, llevar a cabo dos niveles de control:

- Control de la dimensión y complejidad, que tiene en cuenta ciertos valores no superables de elementos tales como: número de líneas de código, instrucciones ALTER y PERFORM, GO TO a la misma etiqueta LABEL, nodos, arcos, IF anidados, MULTI/ENTRY/EXIT. Si un programa supera cualquiera de los valores máximos fijados para estas características, deberá ser descompuesto hasta presentar valores dentro de los límites aceptables.
- Control de la estructuración y distribución de la complejidad dentro de cada unidad de programación, que consiste en valorar el grado de estructuración de los programas mediante los siguientes parámetros:
  - Porcentaje de no estructuración.
  - Modularidad.

- Número de unidades de programa.
- Máximo número de caminos linealmente independientes por cada unidad.
- Máximo número de líneas de código por cada unidad.
- Nivel de anidamiento de las PERFORM.

En otro estudio, Rossi *et al.* [1991] identifican los atributos de programas COBOL que citamos a continuación:

- Número total de líneas de código (LDC) y número de LDC en la PROCEDURE DIVISION, que es un primer indicador macroscópico, aunque no muy preciso, de la dificultad de implementar cualquier modificación sobre el código.
- Número de LDC ejecutables, que es también otro identificador ampliamente usado, aunque también solamente aproximado.
- Número de LDC ejecutables «destructivas», como ENTRY y ALTER. Estas instrucciones no deberían utilizarse.
- Presencia de código «muerto» dentro del programa. Es causa posible la falta de estructuración y de poca calidad del desarrollo. Este tipo de código debería ser eliminado.
- Presencia de bloques recursivos de PERFORM y subrutinas llamadas cíclicamente. No deberían utilizarse.
- Densidad de instrucciones condicionales, medida como el número de instrucciones que comprueban una o más condiciones (IF, SEARCH, EVALUATE, GO TO... DEPENDING ON, READ... AT END, PERFORM... UNTIL, etc.) dividido por el número total de sentencias ejecutables. No debería exceder el 15%.
- Densidad de instrucciones de modificación del flujo de control: número de instrucciones que alteran el flujo de control dividido por el número total de sentencias ejecutables. Debería estar comprendida entre el 5% y el 15%.
- Densidad de PERFORM, que es el cociente entre el número de PERFORM y el número de LDC en la PROCEDURE DIVISION. Debería estar comprendida entre un 5% y un 12%.

- Número de subrutinas llamadas mediante PERFORM. Debería ser aproximadamente el 5% del número de LDC en la PROCEDURE DIVISION.
- Densidad de comentarios, que es el número de líneas de comentario entre el número total de instrucciones. Debería estar comprendida entre un 10% y un 20%.
- Corrección de la indentación, que se refiere al grado de cumplimiento del código del programa del estándar establecido en la organización. La alta corrección facilita la lectura del código.

En cuanto a la mejora de la calidad del código COBOL que se mantiene, Vesely [1997] propone la adición de «trampas defensivas»; esto es, sentencias que se incorporan en el programa fuente con el fin de detectar y manejar datos fuera de dominio u operaciones ilegales. Las «trampas defensivas» pueden clasificarse en:

- Pasivas: tienen menos impacto en el rendimiento, ya que sólo se disparan cuando ocurre una anomalía. Por ejemplo: ON SIZE ERROR en sentencias aritméticas, WHEN OTHER en sentencias EVALUATE, AT END en sentencias SEARCH, etc.
- Activas: implican ejecución de código adicional de manera frecuente: prueba IF NEGATIVE en datos sin signo involucrados en sentencias MOVE o SUBSTRACT; contadores para detectar invocaciones incorrectas de transferencias de control (por ejemplo, utilizando PERFORM).

6.5.2 Métricas para Orientación a Objetos

Aunque las métricas anteriores también se pueden usar para medir sistemas orientados a objetos (por ejemplo, el número de líneas de código de un programa escrito en java), el software desarrollado siguiendo el paradigma OO difiere del desarrollado utilizando enfoques tradicionales. Ello planteó la necesidad de definir nuevas métricas adaptadas a las características particulares de este paradigma, de las cuales se presentan a continuación algunas propuestas representativas.

Las tres primeras propuestas, las métricas MOOSE, MOOD y las de Loren y Kidd, fueron definidas a nivel de diseño, considerando un nivel de diseño detallado. Las propuestas de métricas para diagramas UML se definieron a nivel conceptual.

6.5.2.1 MÉTRICAS MOOSE

Las métricas MOOSE o también conocidas como métricas CK fueron propuestas por [Chidamber y Kemerer, 1994] y son las más difundidas en orientación a objetos. De hecho, existen numerosos trabajos empíricos sobre la relación de estas métricas y, por ejemplo, la propensión a errores, o la mantenibilidad de las clases, etc. Este conjunto de métricas, definidas a nivel de clases, está compuesto por las siguientes seis métricas:

- **Métodos ponderados por clase (WMC, *Weighted Methods per Class*)**, que mide la complejidad de una clase. Si todos los métodos son igualmente complejos, entonces WMC es igual al número de métodos definidos en una clase. Sea la clase C_i que tiene los métodos M_1, ..., M_n siendo su complejidad respectiva c_1,...., c_n, es posible definir la fórmula:

$$WMC = \sum_{i=1}^{n} c_i$$

  Si consideramos en el ejemplo de la Figura 6.5 que todos los métodos tienen complejidad 1, entonces: WMC(Persona) = 8, WMC(Cliente)= 4 (no considerando los métodos heredados) y WMC(Empleado)=10 (no considerando los métodos heredados).

- **Profundidad del Árbol de Herencia de una Clase, (DIT, *Depth of Inheritance Tree*)**. La métrica DIT mide el máximo nivel en la jerarquía de herencia. Se trata de la cuenta directa de los niveles de la jerarquía de herencia, considerando que en el nivel cero de la jerarquía se encuentra la clase raíz. DIT se considera como una métrica del número de clases antecesoras que una clase podría potencialmente afectar, debido a que cuanto mayor sea el nivel de profundidad de herencia de una clase mayor es el número de métodos y atributos que hereda de otras clases.

- **Número de Hijos (NOC, *Number of Children*).** NOC es el número de subclases subordinadas a una clase en la jerarquía, es decir, la cantidad de subclases que pertenecen a una clase. Según [Chidamber y Kemerer, 1994], NOC es un indicador del nivel de reutilización, de la posibilidad de haber creado abstracciones erróneas, y del nivel de pruebas requerido.

  Como ejemplo de cálculo de las métricas DIT y NOC, considérese el diagrama de UML de la Figura 6.5. Los valores de la métrica DIT serían: DIT(Clase Persona)=0 (clase persona es la clase raíz o nivel 0), Clase Cliente=1, Clase Empleado=1, Clase Empleado Fijo=2, y Clase Empleado

Temporal=2. En cuanto a la métrica NOC, los valores resultantes serían: NOC(Clase Persona=2), Clase Cliente=0 y Clase Empleado=2.

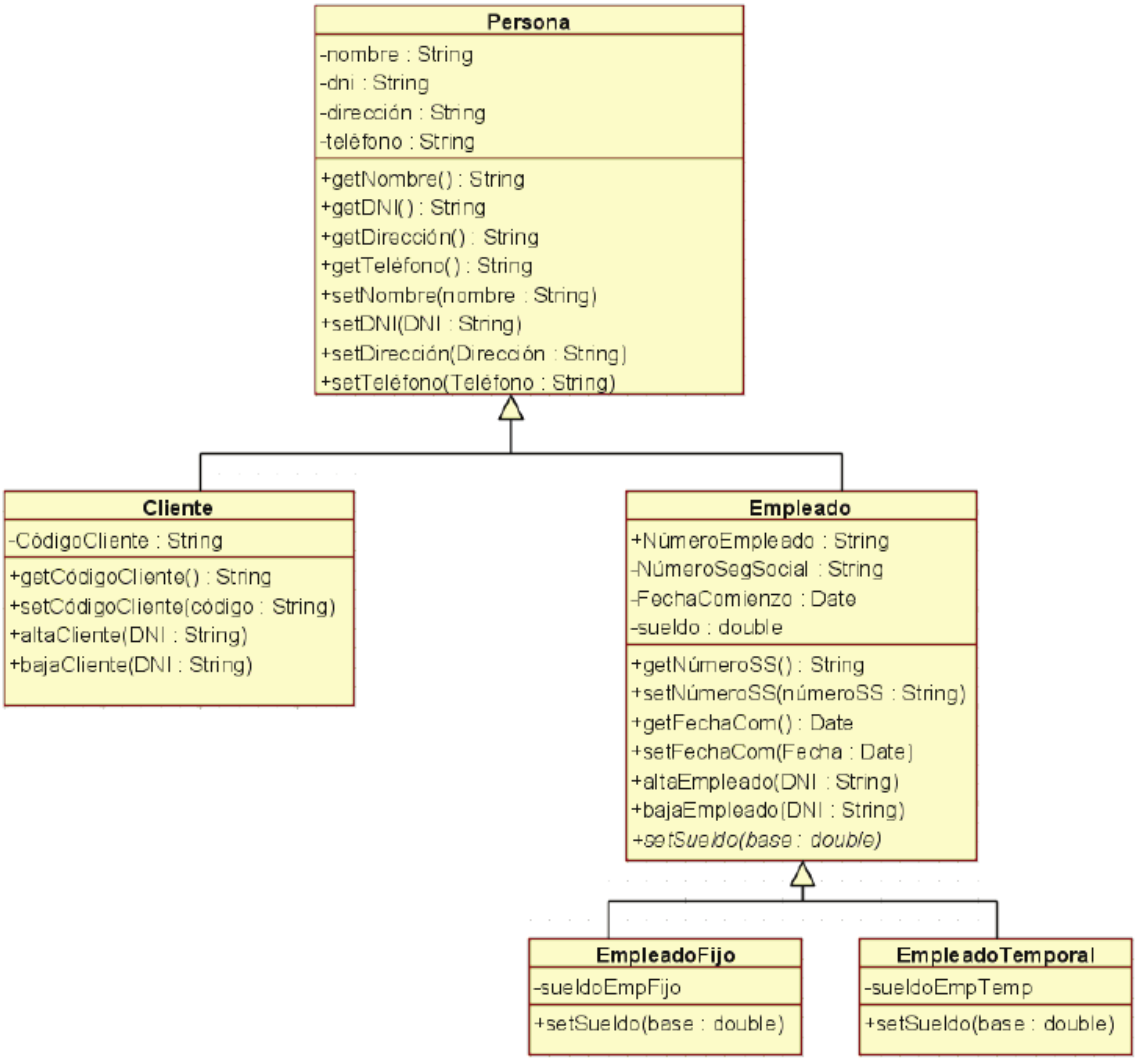


Figura 6.5. Diagrama de clases UML de ejemplo

- **Acoplamiento entre Objetos (CBO, *Coupling Between Objects*)**. La métrica CBO indica para una clase el número de otras clases con las que está acoplada. Se considera que un objeto está acoplado a otro cuando actúa sobre ese otro objeto, por ejemplo cuando un método de un objeto utiliza un método de otro objeto. Esta métrica se considera útil para predecir el esfuerzo necesario para el mantenimiento y las pruebas. Para ilustrar el cálculo de la métrica CBO considérese el siguiente ejemplo ilustrado con un diagrama de clases UML en la Figura 6.6.

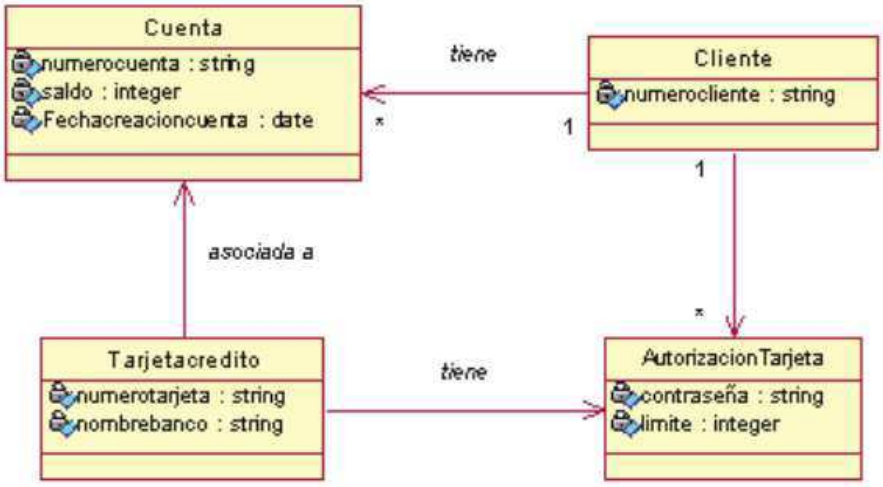


Figura 6.6. Diagrama de ejemplo para el cálculo de CBO

El acoplamiento entre objetos para cada clase sería: CBO (Cuenta) = 0, CBO (Tarjeta Crédito)= 2 (Usa métodos de las clases Cuenta y Autorización de Tarjeta) y CBO (Cliente)=2 (Usa métodos de las clases Cuenta y Autorización de Tarjeta).

- **Respuesta de una clase (RFC, *Response For a Class*)**. RFC indica el número de métodos que pueden ser ejecutados potencialmente como respuesta a un mensaje recibido por un objeto de esa clase. RFC por lo tanto se calcula contando las ocurrencias de llamadas a otras clases de una clase particular. La fórmula para calcular esta métrica es la siguiente: RFC=|RS|, donde RS es el conjunto respuesta para la clase. El conjunto respuesta para la clase se puede expresar de la siguiente manera: RS= Ui {Ri}, donde {Ri} es el conjunto de métodos llamados por el método i y {M} es el conjunto de todos los métodos de la clase.

  Para una mayor comprensión del modo de cálculo de RFC, considérese que se tiene un sistema compuesto por tres clases A, B y C, de modo que la clase A tiene 4 métodos f1,f2,f3,f4; la B tiene 4 métodos B::f1,f2,f3,f4 y la clase C tiene 5 métodos. C::f1..f5. Las invocaciones de los métodos de A son según este esquema:

```
Clase A con cuatro métodos:
    A::f1( ) invoca B::f1( ), B::f2( ) y C::f3( )
    A::f2( ) invoca B::f1( )
    A::f3( ) invoca A::f4( ), B::f 3( ), C::f1( ) y C::f 2( )
    A::f4( ) No llama a otros métodos
```

```
Entonces
    RS= { A::f1, A::f2, A::f3, A::f4 }
          U {B::f1, B::f2, C::f3 }
          U (B::f1}
          U {A::f4, B::f3, C::f1, C::f2 }
        ----------------------------------------
         = {A::f1, A::f2, A::f3, A::F4, B::f1, B::f2, B::f3,
            C::f1, C::f2, C::f3}
    Resultando RFC(A)=10
```

La métrica RFC se ha calculado para la clase A, la cual tiene 4 métodos locales y llama desde esos métodos a 6 métodos remotos (B::f1, f2, f3 y C::f1,f2,f3). La métrica RFC (Respuesta para una clase) se considera por tanto como la suma de los métodos locales a una clase, más los métodos remotos (métodos invocados de otras clases desde los métodos locales a la clase)

- **Falta de cohesión en los métodos (LCOM, *Lack of Cohesion in Methods*)**. LCOM establece en qué medida los métodos hacen referencia a los atributos. Se calcula como el número de pares de funciones sin variables compartidas de instancia menos el número de pares de funciones con variables de instancia compartidas. LCOM es una métrica de la cohesión de una clase en base al número de atributos comunes usados por diferentes métodos. Un valor alto en LCOM implica falta de cohesión, es decir, escasa similitud entre los métodos siendo siempre deseable un alto grado de cohesión en los métodos de una clase.

  Para ilustrar el cálculo de LCOM, considérese la Figura 6.7 donde los óvalos representan los métodos de una clase y los puntos representan los atributos de la clase. Un punto estará dentro de un óvalo perteneciente a un método, si en el mismo se hace referencia al atributo representado por el punto.

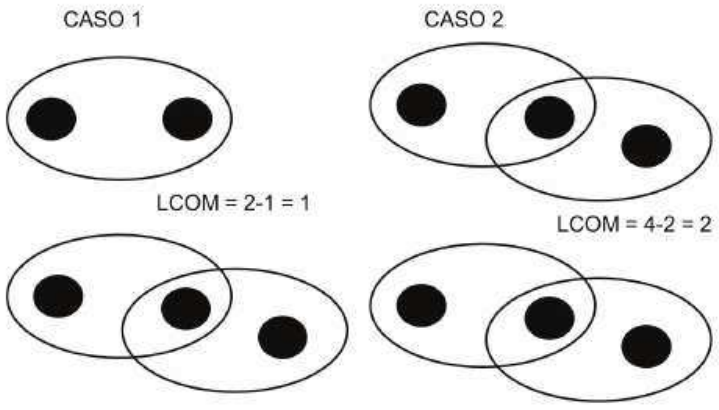


Figura 6.7. Diagrama de ejemplo para el cálculo de LCOM y LCOM*

En la Figura 6.7 se ve como LCOM se incrementa según se va incrementando el número de eslabones en la cadena. Esto no es deseable, ya que LCOM mide la cohesión y no debe depender del número de métodos en la cadena, sino de en qué medida los métodos afectan a los atributos de la clase. En el ejemplo de la Figura 6.7 con 3 métodos en la cadena LCOM=0. En cambio, con 5 métodos en la cadena LCOM=2. Debido a estos problemas [Henderson-Sellers, 1996] propone LCOM* como medida de cohesión, que se calcula mediante la siguiente fórmula, siendo M(A*i*) = número de métodos que accede al atributo *i*; *m* = número de métodos de la clase.

$$LCOM^* = \frac{PROMEDIO\left(\forall_i M(A_i)\right) - |m|}{|1 - m|}$$

En el caso 1 de la Figura 6.8 el cálculo de esta métrica sería: i=6 (n° atributos), m=3 (n° métodos), M(A1)= M(A2)=M(A3)= M(A4)= M(A6)=1, M(A5)=2, LCOM*= [1/6 * (1+1+1+1+2+1)]-3 / (1-3), siendo LCOM*=0’916.

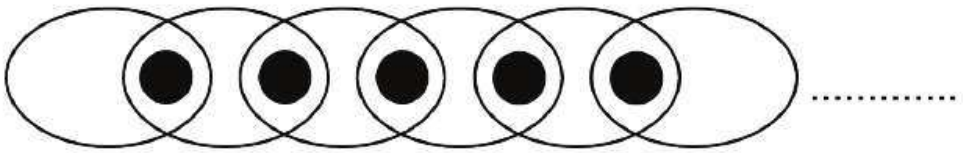


Figura 6.8. Incremento de LCOM

En [Arvanitou et al., 2017] se presenta un mapeo sistemático que señala que la mantenibilidad es la característica (atributo de calidad) más estudiada, que normalmente la evaluación de las características de calidad se lleva a cabo por una sola métrica, en lugar de una combinación de varias, y resume las principales métricas utilizadas (con su frecuencia) para diferentes características de calidad como se muestra en la Tabla 6.4. Se destaca también que las métricas que mayor validez empírica presentan son DIT, NOC, RFC, LCOM1 y WMC.

Atributo de calidad	Métrica de calidad	Frec.
Mantenibilidad	Depth of Inheritance Tree (DIT)	6
	Lines of Code (LOC)	6
	Weighted Methods per Class (WMC)	6
	Cyclomatic Complexity (CC-VG)	5
	Lack of Cohesion of Methods-1 (LCOM1)	5
	Tight Class Cohesion (TCC)	4
	Number of Children (NOC)	4
	Response for Class (RFC)	4
	Message Passing Coupling (MPC)	4
	Data Abstraction Coupling (DAC)	4
	Number of Methods (NOM)	4
Reusabilidad	Lack of Cohesion of Methods-1 (LCOM1)	3
	Lines of Code (LOC)	2
	Coupling Between Objects (CBO)	2
	Response for Class (RFC)	2
	Message Passing Coupling (MPC)	2
	Weighted Methods per Class (WMC)	2
	Number of Children (NOC)	2
	External Class Complexity (ECC)	2
	External Class Size (ECS)	2
Propensión al cambio	Depth of Inheritance Tree (DIT)	3
	Number of Children (NOC)	3
	Coupling Between Objects (CBO)	3
	Response for Class (RFC)	3
	Lack of Cohesion of Methods-1 (LCOM1)	3
	Data Abstraction Coupling (DAC)	3
	Number of Attributes (NA)	3
Comprensibilidad	Lines of Code (LOC)	3
	Depth of Inheritance Tree (DIT)	2
	External Class Complexity (ECC)	2
	External Class Size (ECS)	2
Facilidad de prueba (*testability*)	Response for Class (RFC)	4
	Coupling Between Objects (CBO)	3
	Lack of Cohesion of Methods-1 (LCOM1)	3
	Lines of Code (LOC)	3
Modificabilidad	Depth of Inheritance Tree (DIT)	2
	Halstead n1	2
	Halstead n2	2
Estabilidad	Weighted Methods per Class (WMC)	2
	Lines of Code (LOC)	2
	System Design Stability (SDI)	2

Tabla 6.4. Características y métricas de calidad

6.5.2.2 MÉTRICAS MOOD

Este conjunto de métricas fue propuesto por [Brito e Abreu et al., 1994] y su objetivo es medir los principales mecanismos del paradigma OO, tales como encapsulación, herencia, polimorfismo y paso de mensajes, así como polimorfismo y su consecuente influencia sobre la calidad del software y la productividad en el desarrollo. Las métricas MOOD se pueden utilizar en las fases de diseño y se definieron para ser aplicadas a nivel de diagrama de clases (Tabla 6.5).

Nombre	Definición
MHF	El Factor de Ocultamiento de los Métodos (*Meted Hiding Factor*) mide la proporción entre los métodos definidos como protegidos o privados y el número total de métodos. MHF se propone como una medida de **encapsulación**, cantidad relativa de información oculta.
AHF	El Factor de Ocultamiento de los Atributos (*Attribute Hiding Factor*) se define como el cociente entre la suma de las invisibilidades de todos los atributos definidos en todas las clases y el número total de atributos definidos en el sistema considerado. La invisibilidad de un atributo es el porcentaje del total de clases desde las cuales los atributos son invisibles. AHF se definió como una medida de **encapsulación.**
MIF	El Factor de Herencia de los Métodos (*Method Inheritance Factor*) se define como el cociente entre la suma de los métodos heredados en todas las clases del sistema considerado y el número total de métodos existentes (tanto los definidos localmente como los heredados) en todas las clases. MIF se define como una medida de herencia y, por lo tanto, como una medida del nivel de **reutilización.**
AIF	El Factor de Herencia de los Atributos (*Attribute Inheritance Factor*) se define como el cociente entre la suma de los atributos heredados en todas las clases del sistema considerado y el número total de atributos existentes (tanto los definidos localmente como los heredados) en todas las clases. Al igual que MIF, AIF se considera como un medio para expresar la capacidad de **reutilización** en un sistema.
PF	El Factor de Polimorfismo (*Polymorphism Factor*) se define como el cociente entre el número actual de posibles diferentes situaciones de polimorfismo, y el número máximo de posibles situaciones distintas de polimorfismo para la clase Ci. PF es una medida del **polimorfismo** y una medida indirecta de la asociación dinámica en un sistema.

Tabla 6.5. Métricas MOOD

A continuación, se ilustra con el siguiente ejemplo de código escrito en C++ el cálculo de las métricas anteriores.

```
   Class FormaGeométrica {
   Protected:
   Double posicionX;
   Double posicionY;
Void Dibujar();
```

```
   Public:
        Void Cortar();
      Void Borrar();
      Void Mover(double DesplazX, double DesplazY);
      Void Desagrupar();
           Virtual void Posicionar (double posX, double
                                        posY); //constructor
      Virtual void Escribir (int color); //llama a
                                                  //dibujar
              Virtual double Area();
   }
Class cuadro:public  FormaGeométrica{
   Protected:
       Double anchura;
       Double altura;
     Double DameAnchura();
           Double DameAltura();
   Public:
     Void Establecerdimensiones (double altura, double anchu-
ra);
     Void Posicionar (double posX, double posY);
     Void Escribir( int color);
     Double Area();
  }
Class círculo:public  FormaGeométrica{
     Protected:
       Double radio;
     Public:
          Void EstablecerRadio (double radio);
          Void Posicionar (double posX, double posY);
          Void Escribir( int color);
          Double Area();
   }
```

◤ Cálculo de la métrica MHF

Clase	M_H	M_V	M_D
FormaGeométrica	1	7	8
Cuadro	2	4	6
Círculo	0	4	4

M_H: FormaGeométrica=1 (Dibujar); Cuadro=2 (DameAnchura, DameAltura); Círculo=0

M_V: FormaGeométrica=7 (Cortar, Borrar, Mover, Desagrupar, Posicionar, Escribir, Area); Cuadro=4 (EstablecerDimensiones, Posicionar, Escribir, Area); Círculo=4 (EstablecerRadio, Posicionar, Escribir, Área)

$M_D = M_H + M_V$

MHF= 3/18= 0,1666666

Cálculo de la métrica AHF

Clase	A_H	A_V	A_D
FormaGeométrica	2	0	2
Cuadro	2	0	2
Círculo	1	0	1

A_H: FormaGeométrica=2 (posicionX, posicionY); Cuadro=2 (Anchura, Altura); Círculo=1 (Radio);

A_V: FormaGeométrica=0; Cuadro=0; Círculo=0

$A_D = A_H + A_V$

AHF= 5/5= 1

Cálculo de la métrica MIF

Clase	M_N	M_O	M_I	M_D	M_A
FormaGeométrica	8	0	0	8	8
Cuadro	3	3	5	6	11
Círculo	1	3	5	4	9

M_N: FormaGeométrica=8 (Todos los métodos); Cuadro=3 (DameAnchura, DameAltura, EstablecerDimensiones); Círculo=1 (Radio)

M_O: FormaGeométrica=0; Cuadro=3 (Posicionar, Escribir, Área); Círculo=3 (Posicionar, Escribir, Área)

M_I: FormaGeométrica=0; Cuadro=5 (Cortar, Borrar, Mover, Desagrupar, Dibujar); Círculo=5 (Cortar, Borrar, Mover, Desagrupar, Dibujar).

$M_A = M_D + M_{I;} M_D = M_N + M_{O;}$

MIF= 10/28=0,3571428

- **Cálculo de la métrica AIF**

Clase	A_N	A_O	A_I	A_D	A_A
FormaGeométrica	2	0	0	2	2
Cuadro	2	0	2	2	4
Círculo	1	0	2	1	3

A_N: FormaGeométrica=2 (posicionX, posicionY); Cuadro=2 (Altura, Anchura); Círculo=1 (Radio)

A_O: FormaGeométrica=0; Cuadro=0 ; Círculo=0

A_I: FormaGeométrica=0; Cuadro=2 (posicionX, posicionY); Círculo=2 (PosicionX, PosicionY)

$A_D = A_N + A_O$; $A_A = A_D + A_I$;

AIF= 4/9=0,444444..

- **Cálculo de la métrica PF**

Clase	M_O	M_N	DC
FormaGeométrica	0	8	2
Cuadro	3	3	0
Círculo	3	1	0

D_C: FormaGeométrica=2 (Cuadro, Círculo); Cuadro=0; Círculo=0;

PF= 6/16=0,375

Después de llevar a cabo un estudio empírico, [Brito e Abreu et al., 1996] sugirieron que:

- Cuando el valor de MHF aumenta, la densidad de defectos y el esfuerzo requerido para corregirlos tendrían que disminuir.

- Idealmente, el valor de la métrica AHF sería 100%; por ejemplo, todos los atributos estarían ocultos y solo podrían ser accedidos desde los métodos de las clases correspondientes.
- A primera vista podría resultar tentador pensar que la herencia debería ser usada extensivamente. Sin embargo, la excesiva reusabilidad a través de la herencia hace a los sistemas más difíciles de comprender y mantener.
- En relación con la métrica PF, en algunos casos los métodos redefinidos pueden contribuir a reducir la complejidad e incluso a hacer el sistema más comprensible y fácil de mantener.

6.5.2.3 MÉTRICAS DE LORENZ Y KIDD

[Lorenz y Kidd, 1994] propusieron un conjunto de métricas llamadas "*métricas de diseño*", que se refieren a características estáticas del diseño de un producto software. Estos autores clasificaron las métricas en: métricas de tamaño, métricas de herencia y métricas de características internas de las clases (Tabla 6.6):

	Nombre	Definición
Métricas de tamaño	PIM	El Número de Métodos de Instancia Públicos de una clase es el número total de métodos públicos de instancias. Los métodos públicos son aquellos que están disponibles como servicios para otras clases.
	NIM	El Número de Métodos de Instancia es la suma de todos los métodos de una clase, considerando tanto los métodos públicos como los protegidos y privados.
	NIV	El Número de Variables de Instancia es el número total de variables a nivel de instancia que tiene una clase, considerando las variables privadas y protegidas disponibles en una clase.
	NCM	El Número de Métodos de Clase es el número total de métodos a nivel de clase, por ejemplo, aquellos métodos que son globales a todas las instancias que tiene una clase.
	NVV	El Número de variables de Clase es el número total de variables a nivel de clase que tiene una clase.

Métricas de herencia	NMO	El Número de Métodos Sobrecargados es el número total de métodos sobrecargados por una subclase.
	NMI	El Número de Métodos Heredados es el número total de métodos que una clase hereda.
	NMA	El Número de Métodos Añadidos es el número total de métodos que se definen en una subclase. Esta métrica se definió para medir la calidad del uso de la herencia, ya que examina la relación de herencia entre subclases y superclases.
	SIX	El Índice de Especialización para cada clase se define así: $\frac{NúmerodeMétodosSobreescritos * NiveldeAnidamientoEnLaJerarquía}{NúmeroTotalDeMétodos}$ Esta métrica mide hasta qué punto una subclase redefine el comportamiento de una superclase.
Métricas de características internas de las clases	APPM	El Promedio de Parámetros por Método se define así: $\frac{NúmeroTotalDeParámetrosPorMétodo}{NúmeroTotalDeMétodos}$
	LOC	Líneas de Código por Método. Es el número de líneas ejecutables en un método
	NOM	Número de mensajes enviados en un método

Tabla 6.6. Métricas de [Lorenz y Kidd, 1994]

6.5.3 Métricas para bases de datos

A nivel lógico, el único criterio que se ha venido aplicando tradicionalmente es la teoría de la normalización para las bases de datos relacionales. Parece sorprendente que, existiendo centenares de métricas de software, las métricas para bases de datos hayan sido descuidadas. Esta situación puede ser explicada ya que, hasta hace no demasiado tiempo, las bases de datos jugaban un papel relativamente secundario dentro de los sistemas de información siendo los programas los verdaderos protagonistas. Ello justifica la gran presencia de métricas orientadas a código que podemos encontrar en la literatura. Sin embargo, aunque las bases de datos se han convertido en el corazón de los Sistemas de Información (SI) más relevantes para el funcionamiento de la sociedad, su diseño sigue siendo una tarea larga, difícil y costosa. Además, el tamaño y la naturaleza de los datos pueden influir, en gran medida, en muchos aspectos de un SI como el esfuerzo de desarrollo. Ello motiva la importancia de la evaluación de la calidad de las bases de datos, y sus modelos lógicos juegan un papel destacado.

En relación a las propuestas de métricas para modelos lógicos de datos, cabe destacar la propuesta de métricas de [Calero et al., 2001] para evaluar la mantenibilidad de los modelos relacionales, que se resumen en la Tabla 6.7.

Métrica	Notación	Definición
Número de Atributos de una Tabla	NA(T)	definida como el número de atributos de una tabla T
Número de Claves Ajenas	NFK(T)	definida como el número de claves ajenas de una tabla T
Profundidad del Árbol Referencial de una Tabla	DRT(T)	definida como la profundidad máxima de todos los caminos referenciales del grafo que se forma, tomando la tabla T como el nodo raíz del grafo y todas las tablas relacionadas con T mediante integridad referencial como el resto de nodos y siendo las relaciones de integridad referencial los arcos del mismo
Ratio de Claves Ajenas de una Tabla	RFK(T)	definida como el porcentaje de atributos de la tabla T que son claves ajenas $$RFK(T) = \frac{NFK(T)}{NA(T)}$$
Número de Tablas	NT	definida como el número total de tablas que hay en el esquema
Cohesión del Esquema	COS	definida como la suma del número de tablas al cuadrado que hay en cada componente no conexa del grafo del esquema, siendo los nodos de este grafo las tablas del esquema y los arcos las relaciones de integridad referencial $$COS = \sum_{i=1}^{|US|} NT_{USi}$$
Ratio de Normalidad	NR	definida como la relación entre el número de tablas en tercera forma normal (o superior) entre el número total de tablas $$NR = \frac{NT3NF}{NT}$$ Siendo NT3NF es el número de tablas en 3NF
Número de Atributos	NA	definida como el número total de atributos que hay en el esquema $$NA = \sum_{i=1}^{NT} NA(T_i)$$
Número de Claves Ajenas	NFK	definida como el número total de claves ajenas que hay definidas en el esquema $$NFK = \sum_{i=1}^{NT} NFK(T_i)$$

Profundidad del Árbol Referencial	DRT	definida como la profundidad máxima de todos los caminos referenciales del grafo que se forma tomando las tablas del esquema como los nodos y las relaciones de integridad referencial como los arcos del mismo $$DRT = \max_{i=1}^{NT}(DRT(T_i))$$
Ratio de Claves Ajenas	RFK	definida como el porcentaje de atributos del esquema que son claves ajenas $$RFK = \frac{NFK}{NA}$$

Tabla 6.7. Resumen de métricas para el modelo relacional

6.6 LECTURAS RECOMENDADAS

✓ Calidad de Sistemas de Información 4º edición. Piattini, M., García, F., García, I. y Pino, F.J. (2018), Madrid, Ra-Ma.

Este libro es muy útil para aquel lector que desee ampliar sus conocimientos sobre métricas y calidad de sistemas de información.

6.7 EJERCICIOS

Ejercicio 1

Explique los siguientes conceptos y ponga un ejemplo de cada uno, en el contexto del mantenimiento de software: atributo interno, atributo externo, medida directa, medida indirecta, medida de predicción.

Ejercicio 2

Suponga que los proyectos enumerados en la figura 8.8 están realizados en lenguaje Cobol, y que se desea estimar, con los datos de esa tabla, el coste de mantenimiento de un proyecto que consta de 20.000 LDC en lenguaje C. ¿Qué modificaciones se necesitan realizar en dicha tabla?

Ejercicio 3

¿Entre qué rango de valores puede oscilar la métrica *Calidad del departamento*, de Sneed?

Ejercicio 4

Si trabaja con algún lenguaje de programación no recogido en la tabla de la Tabla 6.2, recopile información de sus últimos proyectos y trate de hallar una correspondencia entre puntos de función y líneas de código.

Ejercicio 5

Discútase la siguiente afirmación: *la productividad de los programadores y mantenedores puede calcularse dividiendo el número de puntos de función creados o mantenidos entre el número de horas de trabajo*.

Ejercicio 6

Obtenga el lector los nueve axiomas de Weyuker [1988] y compruebe si la métrica de complejidad de McCabe puede ser considerada, según estos axiomas, una métrica de complejidad.

Ejercicio 7

Proponga tres posibles factores internos (propiedades internas) que puedan influir en la mantenibilidad de un producto software, y piense una métrica interna para cada uno de ellos.

Ejercicio 8

Piense tres posibles factores externos que puedan influir en la mantenibilidad de un producto software, y piense una métrica interna para cada uno de ellos.

7

HERRAMIENTAS PARA EL MANTENIMIENTO DEL SOFTWARE

Puesto que las tareas de mantenimiento representan la mayor carga de trabajo durante el ciclo de vida del software, la disponibilidad de herramientas para automatizar estas actividades ayudará a reducir notablemente el coste global del software. Entre las herramientas para mantenimiento del software tradicionales, se pueden señalar las siguientes:

- Generadores de referencias cruzadas.
- Generadores de organigramas.
- Controladores de código fuente.
- Analizadores automáticos de interfaces.
- Gestores de ficheros.
- Decompiladores.
- Evaluadores del impacto de las modificaciones.
- Detectores de componentes afectados por los cambios.

Estas herramientas, junto con otras nuevas propuestas, se pueden agrupar en tres categorías [Mazza *et al.*, 1996]:

- Herramientas de navegación
- Herramientas para perfeccionamiento del código
- Herramientas de ingeniería inversa

7.1 HERRAMIENTAS DE NAVEGACIÓN

Las herramientas de navegación permiten al ingeniero de software buscar rápida y fácilmente las partes del software que le interesan. Las capacidades típicas de estas herramientas son:

- Identificación de dónde se usan las variables.
- Identificación de los módulos que son utilizados por un módulo (y los módulos que utilizan a dicho módulo).
- Visualización de un árbol de llamadas.
- Visualización de estructuras de datos.

El conocimiento de dónde son utilizados los módulos y las variables es crítico para comprender el impacto y los efectos de un cambio. La visualización del árbol de llamadas y de las estructuras de datos ayuda a comprender los flujos de control y de datos.

En entornos de *orientación a objetos* (OO) las herramientas de navegación tienen en cuenta las características de este paradigma. Para ello incluyen funcionalidades como, por ejemplo, distinguir el significado correcto de un símbolo con sobrecarga; o permitir la localización de funciones y atributos heredados. Además, surgen otros aspectos que hay que tener en cuenta como por ejemplo los métodos polimórficos, que sólo puede conocerse en tiempo de ejecución cuando los tipos de los argumentos son conocidos. Este hecho hace más difícil entender lo que el código hará mediante los análisis e inspecciones estáticos y, por tanto, habrá que recurrir obligatoriamente al análisis dinámico del programa durante su ejecución para comprender lo que está haciendo [Kirchmayr et al., 2016].

7.2 HERRAMIENTAS PARA PERFECCIONAMIENTO DEL CÓDIGO

Las herramientas más importantes de este tipo son:

- *Reformateadores de código fuente*: a partir del código fuente generan una salida con formato y presentación mejorados. Son muy utilizados para convertir código antiguo a nuevos estándares de codificación.
- *Reestructuradores (refactorizadores) de código fuente*: sirven para estructurar programas poco o mal estructurados. La reestructuración se realiza construyendo un diagrama de flujo a partir del programa original,

estructurando el diagrama obtenido y generando el nuevo programa. Casi siempre, y en la medida de lo posible, las estructuras de control se reducen a las tres básicas (secuencia, selección e iteración).

7.3 HERRAMIENTAS DE INGENIERÍA INVERSA

Las herramientas de ingeniería inversa procesan el código fuente para producir otro tipo de elemento software. Dada la naturaleza de esta técnica, las herramientas disponibles comercialmente se han desarrollado para las aplicaciones heredadas (*legacy code*) y, especialmente, para entornos COBOL, FORTRAN, java y SGBD relacionales. Estas herramientas son muy útiles cuando la documentación sobre el código es inexistente o está desfasada. Dentro de esta categoría, se encuentran varios grupos de herramientas, según su funcionalidad:

- *Recuperadores de diseño*: son herramientas que recuperan diseños desde el código fuente, examinan las dependencias modulares y las representan en términos de una cierta metodología de diseño. Se subdividen en dos clases, según se apliquen a los datos o a los procesos:
  - *Recuperadores para ingeniería inversa de datos*: capaces de extraer la información del código fuente que describe la estructura de los datos. Por ejemplo, desde código COBOL se extraen las definiciones de ficheros y se genera el diagrama E/R. Lo mismo se puede realizar a partir de un esquema relacional de base de datos.
  - *Recuperadores para ingeniería inversa de procesos*: permiten obtener la descripción lógica de las entidades y las reglas de negocio a partir del código de los programas.
- *Redocumentadores*: a partir del código fuente, permiten generar diagramas, gráficos, listas de utilización y otras informaciones de diseño que sirven para comprender mejor el código. En esta categoría estaría la tecnología JavaDoc que desde ciertas anotaciones en código fuente en cada método y clase java permite generar documentación HTML automáticamente.
- *Analizadores de código*: incluyen diversas ayudas para facilitar el estudio del código. Sus funcionalidades coinciden en parte con las herramientas de navegación, ya que suelen incluir posibilidades como la indentación automática del código fuente y la visualización dinámica de las llamadas (a procedimientos o funciones).

- *Decompiladores*: la decompilación traduce código compilado a código fuente. Sus capacidades son útiles algunas veces para diagnosticar defectos de la compilación, por ejemplo, por optimizaciones erróneas. Otras veces son útiles simplemente porque no se dispone del código fuente de ciertos ejecutables.

Muchas de las herramientas anteriores están limitadas a lenguajes de programación específicos (aun cuando se abarcan la mayoría de los lenguajes principales) y requieren un cierto grado de interacción con el ingeniero de software.

Las herramientas de ingeniería inversa y directa de la próxima generación harán un uso mucho mayor de técnicas de **inteligencia artificial**, empleando una base de conocimientos específica del dominio de la aplicación (es decir, un conjunto de reglas de descomposición que se aplicarían a todos los programas de un cierto ámbito de aplicación tal como el control de producción o la gestión económica). El componente de inteligencia artificial asistirá en la descomposición y reconstrucción del sistema software, aunque seguirá siendo necesaria la intervención del ingeniero de software a lo largo del ciclo de la reingeniería y en general del proceso de mantenimiento software.

7.4 LECTURAS RECOMENDADAS

- *www.infogoal.com/cbd/cbdtol.htm*

  En esta página web se encuentra un listado bastante preciso de herramientas para mantenimiento de sistemas COBOL dividido en diferentes categorías.

7.5 EJERCICIOS

Ejercicio 1

Analice el soporte al mantenimiento que ofrecen los entornos generales de desarrollo, estilo Eclipse.

Ejercicio 2

Indique todos los requisitos posibles en una herramienta para análisis de código fuente COBOL.

Ejercicio 3

Implemente el algoritmo para compilar todos los módulos de una aplicación en un sistema de gestión de versiones incremental en el que se guarda entera la última versión de cada módulo y sólo las diferencias con la versión anterior en las restantes.

Ejercicio 4

Consiga información sobre tres herramientas de reingeniería y realice un estudio comparativo entre ellas.

Ejercicio 5

Estudie las repercusiones que sobre una herramienta de GCS puede tener el uso de un SGBD orientado a objetos frente a uno relacional.

Ejercicio 6

Sugiera los costes ocultos de utilizar herramientas CASE en la fase de mantenimiento del software.

Ejercicio 7

Indique cinco tipos de herramientas CASE y su utilidad para mantenimiento.

Ejercicio 8

Explique las posibles utilidades de un procesador de textos como Word o WordPerfect para análisis de código fuente.

Ejercicio 9

Señale todas las posibles opciones de incorporar a una herramienta de navegación para lenguaje SQL.

Ejercicio 10

Explique las instrucciones del lenguaje Java que deben ser utilizadas por una herramienta de ingeniería inversa de datos.

PARTE II

TEMAS AVANZADOS

8

MANTENIMIENTO DE SOFTWARE GREEN

La sostenibilidad *("green"*) es un concepto que está adquiriendo cada vez mayor importancia en el ámbito del software, cuyo objetivo principa es conseguir un software que sea cada vez más sostenible y respetuoso con el medio ambiente contribuyendo, por ejemplo, en la reducción del consumo energético. Como hemos visto, el mantenimiento de software permite no solamente resolver los problemas del software y mejorar su calidad, también hace que sea posible refactorizarlo mediante transformaciones para así mejorar ciertas características del software. Este capítulo proporciona una visión general del tipo de técnicas, herramientas y prácticas que podrían resultar ser útiles en el mantenimiento de software en ese esfuerzo por mejorar la calidad de los sistemas, pero ahora tratando de que el software sea más "*Green"*. En este sentido, esa propiedad o atributo que tendría un software de ser más o menos "verde" se denomina *Greenability* [Calero y Piattini, 2015], por lo que en este sentido, estudiaremos cómo el mantenimiento puede contribuir a mejorar la capacidad de un software para ser sostenible.

8.1 INTRODUCCIÓN

Hay tres niveles de impacto sobre el medio ambiente causados por las TI [Nauman et al., 2011]: *"los del primer grado son los efectos medioambientales que son el resultado de la producción y del uso de las TIC; es decir, el uso de los recursos en la minería, y la contaminación causada por esta actividad, la fabricación de hardware, el consumo energético durante el uso de las TI, y la eliminación de residuos procedentes de equipos electrónicos; los impactos de segundo grado son los efectos causados por el uso de las TIC de forma indirecta, como la conservación de energía y recursos por la optimización de proceso (los efectos de la desmaterialización), o de la conservación de recursos mediante la sustitución de productos materiales por*

los correspondientes productos no-materiales (los efectos de la sustitución);y los impactos del tercer grado son los efectos indirectos a largo plazo sobre el medio ambiente y que son el resultado del uso de las TIC. Éstos pueden incluir unos estilos de vida en proceso de constante evolución y que provocan un crecimiento económico más rápido; en el peor de los casos, estos efectos son más fuertes que el ahorro conseguido con anterioridad (nos referimos a los efectos rebote)".

Por lo tanto, el interés en diseñar e implementar las soluciones de Green IT ha aumentado considerablemente a lo largo de estos últimos años [Murugesan, 2008]. Un motor principal en este movimiento verde es el hecho de que las demandas energéticas de las TI están creciendo a gran velocidad, debido a que se están adoptando los servicios TI con cada vez más intensidad a nivel global [Murugesan, 2013]. En la búsqueda de producir un software sostenible [Nauman et al., 2011],van apareciendo unos modelos y propuestas nuevos, pero hay un asunto aquí que no se puede ignorar: la enorme cantidad de sistemas de información en operación hoy en día en todas las organizaciones. La mayoría de ellos seguramente se han desarrollado siguiendo un enfoque *clásico* (no de *Green Software Engineering* [Nauman et al., 2011]). Como consecuencia, estos sistemas de software no serían sensibles en cuanto a cualquier impacto sobre el medio ambiente, y como tales, no se les consideran como *Green* TI.

Las actividades del mantenimiento clásico permiten la mejora de las características de la calidad de software [ISO/IEC, 2013]; el mantenimiento, a su vez, podrían ser también de ayuda para la mejora de la "greenability" para aquellos sistemas heredado que no han sido desarrollados siguiendo los enfoques "green". Este capítulo se propone dar una definición del mantenimiento *green*, y considera las posibles maneras de llevar a cabo la mejora de *software greenability* en el proceso de mantenimiento. El primer paso que se ofrece en este trabajo, por lo tanto, consiste en explorar los efectos que tienen sobre la *greenability* los problemas que surgen en el mantenimiento clásico (defectos en la refactorización de software), sin considerar el mantenimiento *green* como un mantenimiento completamente separado del que no lo es.

Asimismo, puesto que la deuda técnica impacta en el mantenimiento clásico de software, se propone un concepto innovador: la deuda ecológica. Este tipo de deuda tiene como objetivo medir el coste de no llevar a cabo un mantenimiento de software ecológico para mejorar la característica de la calidad de la *greenability* de un sistema de software. Se analiza la relación entre la deuda ecológica y la deuda técnica, juntamente con su relación tanto con el mantenimiento clásico como con el mantenimiento *green.*

8.2 MANTENIMIENTO DE SOFTWARE MÁS ECOLÓGICO

Los sistemas de software actuales se tienen que *"mantener"* para obtener nuevas versiones con un nivel más alto de *greenability*; desafortunadamente, el mantenimiento tradicional de software no aborda este tipo de mejora de la calidad. A pesar de que el mantenimiento de software es una área de investigación que está en auge, el concepto de "mantenimiento de software *verde*" o "mantenimiento sostenible" se menciona en solamente unos pocos artículos [Nauman et al., 2011] [Shenoy et al., 2011]. En [Nauman et al., 2011], los autores presentan el modelo *GREENSOFT,* en el cual se describe el software *green* (sostenible), juntamente con el correspondiente proceso de ingeniería. Centrándose en la etapa del desarrollo del software, el modelo *GREENSOFT* contempla una *fase de uso,* en la cual se evalúa el impacto que se produce como resultado de ejecutar, utilizar y mantener el producto software. Como describe la propuesta, se llevan a cabo distintas actividades en esta etapa: la instalación de parches o actualizaciones del software, la configuración de los sistemas software, la formación de los empleados en cuanto al uso adecuado de software (los trabajadores bien formados desempeñan su tarea con más rapidez, lo cual implica un consumo de energía más bajo), la configuración del sistema software para que consuma menos energía, o simplemente dejar sus ordenadores en modo de suspensión cuando sus equipos están inactivos. Hay otros asuntos relacionados con la energía y el consumo de recursos que se tratan en esta fase de uso (es decir, del mantenimiento), incluyendo la conexión a los servicios provistos por otros servidores, o el remplazo del hardware desfasado (porque la actualización de los sistemas de software instalados requiere unos equipos más potentes).

El *Green Software Development Model* [Shenoy et al., 2011] propone un enfoque nuevo para el ciclo de vida del desarrollo de software. Esta propuesta se centra en lo que se debe considerar en cada fase del proceso de desarrollo (la recopilación de requisitos, el diseño, la implementación, las pruebas, la implantación y el mantenimiento) para que se produzca el software de una manera no dañina al medioambiente. El enfoque tiene en cuenta algunos de los temas clásicos de mantenimiento, los cuales incluyen cómo corregir los errores, además de cómo afinar el sistema, implementar las funciones nuevas, etc. Desde el punto de vista ecológico, los autores proponen: (i) utilizar una documentación electrónica en vez de una que emplee papel; (ii) no usar la ingeniería inversa, ni ninguna técnica basada en tener que desmontar el sistema software, puesto que éstas son actividades que consumen mucho tiempo y no poca energía. Para reducir el impacto del punto anterior, hemos de intentar (iii) involucrar al equipo de desarrollo en el mantenimiento. De esta manera se acelera el mantenimiento, porque así el equipo de mantenimiento se familiarizará más rápido con el sistema que ha de ser mantenido. Los autores también proponen: (iv) que se intente evitar la migración del software, puesto que esto conduce a la

eliminación de hardware heredado y de algunos dispositivos, aparte de una retirada de la tecnología antigua; y (v) que se construya el sistema de tal manera que se adapte sin problema a las nuevas tecnologías, los nuevos dispositivos y equipos de hardware.

Como podemos observar, el modelo inicial del *mantenimiento de software ecológico (verde)* parece haberse distorsionado ligeramente. Las ideas que se presentan en esta sección no mejoran la *greenability* del software, ni reducen el consumo de energía. Las propuestas antes mencionadas son unas buenas prácticas que ayudan para que haya una etapa de mantenimiento más respetuoso con el medio ambiente (una reducción del consumo de papel, la formación de los usuarios para que lleven a cabo sus tareas con más velocidad y a la vez consumiendo menos recursos, y evitar tener que emplear la refactorización y la reingeniería). El hecho es que la reducción del consumo energético no se ve afectada por ninguno de estos enfoques beneficiosos. "Hacer que el mantenimiento del software sea más *verde*" no es lo mismo que "un mantenimiento para hacer que el software sea más *verde"* La segunda idea encaja mejor con el concepto de mantenimiento establecido en la norma ISO/IEC 14764 [ISO/IEC, 2006], donde se afirma que el mantenimiento es el proceso por el cual se pretende mejorar la calidad de un producto software. Partimos de la idea de que la *greenability* es una característica de la calidad del software cuyo propósito es medir hasta qué punto un producto software tiene un consumo adecuado de energía. Por lo tanto, el mantenimiento del software es la única vía posible para poder alcanzar ese tipo de mejora mediante la transformación del software. Es con esta idea en mente que hablamos del *Mantenimiento de Software Ecológico (Green)*. O nos referimos a aquellas tareas que se llevan a cabo para obtener versiones nuevas de un sistema en particular, a la vez que se optimiza el consumo de energía.

8.2.1 El Mantenimiento de Software Ecológico ("Green")

Cuando se ha desarrollado un sistema bajo los principios de la *ingeniería de software ecológico* [Nauman et al., 2011], el mantenimiento ecológico de software tiene sentido, puesto que conserva tanto la funcionalidad como la *característica de la greenability* del sistema de software. Dicho esto, debemos considerar qué es lo que ocurre con la gran cantidad de sistemas software que no han sido desarrollados de acuerdo con los principios de *greenability*. En este caso, el mantenimiento ecológico (como faceta de la etapa del mantenimiento del software) tiene que llevar a cabo todos los cambios necesarios para mejorar la calidad de la característica de la *greenability* del sistema. De esta manera, el mantenimiento "verde" del software que se realiza en este tipo de sistemas iría dirigido a la mejora del consumo de energía, por ejemplo.

La definición que se propone a continuación se inspira en la que se propone en [ISO/IEC, 2006]:*el mantenimiento ecológico (green) se realiza a lo largo de todo el ciclo de vida del software, y termina con la retirada del producto de software*, *en este punto llevando a cabo todas las actividades que se requieren para reducir el impacto medioambiental del software retirado. Incluye la modificación de código y la documentación para que se resuelva cualquier posible desvío de los requisitos de"greenability"(o de la implementación de un requisito nuevo), sin modificar la funcionalidad original del código fuente*" Esta definición tiene en cuenta que el mantenimiento "verde" ha de (i) llevarse a cabo a lo largo de todo el ciclo de vida del software; (ii) conservar la funcionalidad original del sistema de software; (iii) mantener los requisitos sostenibles del sistema, y (iv) incluir unos requisitos nuevos (por ejemplo, para los sistemas que no habían sido desarrollados inicialmente de acuerdo con los principios de la ingeniería ecológica y sostenible).

Ahora es posible considerar un proceso nuevo de mantenimiento en el cual se tienen en cuenta tanto los temas de mantenimiento clásico como los requisitos de *greenability* (véase la Figura 8.1).

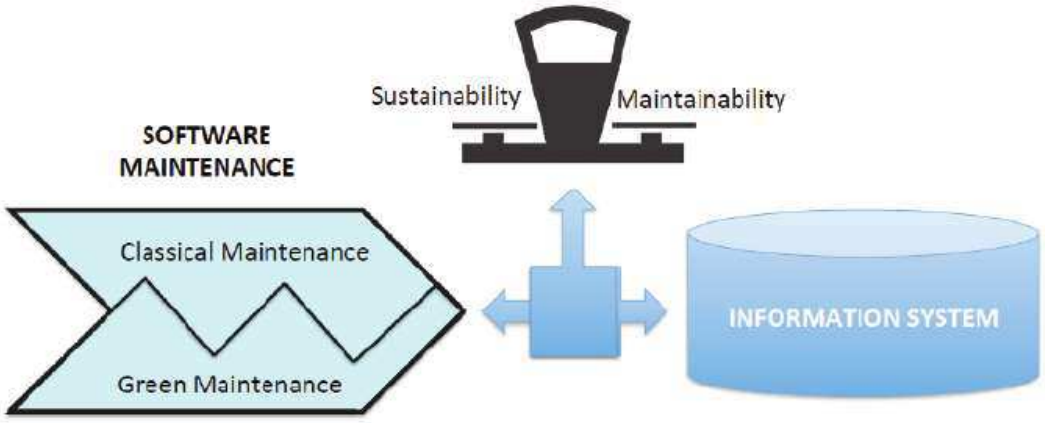


Figura 8.1. Concepto unificado del Mantenimiento de Software

Desafortunadamente, el problema no es tan fácil como considerar un proceso con dos subconjuntos de actividades, cada uno tratando una cuestión diferente. Que sepamos, no hay estudios que traten este tema en concreto, pero es posible encontrar algunos trabajos que abordan el tema de la relación entre los requisitos ecológicos y la mantenibilidad del software. En [Pérez-Castillo y Piattini, 2014], se analizan las consecuencias de aplicar la refactorización para resolver el antipatrón God-Class. Aunque se habla en profundidad de este estudio de caso en las secciones a continuación, la conclusión más importante es el hecho de que la mejora del atributo de la mantenibilidad del software [ISO/IEC, 2013] puede empeorar la característica de calidad de la *greenability* de un sistema, y viceversa. (Véase la figura adjunta).

A pesar de que el mantenimiento clásico y el mantenimiento *green* de software deben considerarse como un solo proceso, en realidad hay una encrucijada donde se puede determinar el equilibrio más adecuado entre la mantenibilidad y la *greenability.*

En la fase de mantenimiento, la característica más significativa de calidad es la mantenibilidad *(nivel de eficacia y eficiencia con el cual un producto o sistema puede ser modificado por los que se pretende que realicen el mantenimiento"* [ISO/IEC, 2013]). Por otro lado, es posible considerar que la *greenability* mide hasta qué punto/grado un sistema de software es respetuoso o no con el medioambiente, basada esta medida en el consumo energético de dicho sistema. Un valor alto para esta característica apuntaría a un consumo bajo de energía.

Esta encrucijada entre el mantenimiento clásico y el mantenimiento *verde* de software a la cual se ha hecho referencia en párrafos anteriores puede interpretarse también como un problema de cómo tener un equilibrio entre la mantenibilidad y la *greenability*. Cuando se mejora una subcaracterística de la mantenibilidad esta acción puede afectar la *greenability* bien de forma positiva, o bien negativamente [Pérez-Castillo y Piattini, 2014]; o puede que no le afecte en absoluto. Por otro lado, si la *greenability* de un sistema software se mejora durante el mantenimiento, puede que la mantenibilidad empeore. Es esencial que se realicen estudios para que se averigüe qué tipo de efecto ejerce sobre la *greenability* la mejora de una sub-característica de mantenibilidad. En estos momentos, no se puede hacer más que predecir una implicación entre la mantenibilidad y la *greenability* (véase la Tabla 8.1).

La Tabla 8.1 presenta la relación hipotética entre la *greenability* y la mantenibilidad ("X" para un posible relación; "?" cuando no está clara la relación). Puesto que la mantenibilidad es un concepto de granularidad gruesa, se establece una relación entre la *greenability* y las subcaracterísticas de la mantenibilidad. Para cada sub-característica, se considera si existe o no una posible relación con la *greenability;* sin embargo, no está claro hasta qué punto la relación es positiva o negativa. Por ejemplo, es probable que la subcaracterística de modularidad *("capacidad de un sistema o programa de ordenador (compuesto de componentes discretos) que permite que un cambio en un componente tenga un impacto mínimo en los demás"* [ISO/IEC, 2013]*)* esté relacionada con la *greenability,* por dos razones:

- La modularización implica que hay más elementos para construir el sistema, y por lo tanto que hay más elementos para intercomunicar. Este hecho supone una cantidad más alta de consumo energético, porque un mayor número de mensajes requiere más energía.

- Por otro lado, un grado adecuado de modularización significa que el diseño será mejor; como consecuencia, el mantenimiento será más rápido (y con un consumo energético más bajo).

Otra subcaracterística que seguramente tiene un impacto positivo sobre la *greenability* es la reusabilidad ("La capacidad de un activo que permite que sea utilizado en más de un sistema software o en la construcción de otros activos" [ISO/IEC, 2013]). Cuanto más se use un componente, mayor será su nivel de optimización. Si consideramos el nivel de optimización en términos de su *greenability*, es probable que cuando más optimizado esté, menor sería su consumo energético.

		Greenability	
		Relación	**Implicación**
Mantenibilidad	Modularidad	X	- Mayor cantidad de módulos significa que habrá más líneas de comunicación. - Un mejor diseño implica menos energía y menos tiempo para llevar a cabo cualquier otro tipo de mantenimiento.
	Reusabilidad	X	- Unos activos altamente reusables están propensos a optimizarse, como lo es también su *greenability*
	Analizabilidad	?	
	Modificabilidad	X	-Si un activo es fácilmente modificable, es más probable que se mantenga (y que no empeore) su *greenability.*
	Facilidad de prueba	?	

Tabla 8.1. Relación entre las subcaracterísticas de la mantenibilidad y la greenability

8.3 IDENTIFICANDO NUEVAS TÉCNICAS PARA LA MEJORA DE LA GREENABILITY EN EL MANTENIMIENTO GREEN DEL SOFTWARE

En este momento, solamente las propuestas como [Nauman et al., 2011] [Shenoy et al., 2011] sugieren qué se podría hacer durante la etapa de mantenimiento y uso para mejorar/conservar la *greenability* del software. La literatura no apunta a ningún tipo de modificación del código fuente/diseño que se tendría que llevar a

cabo durante el mantenimiento a fin de conseguir una mejora de la *greenability.* Eso significa que es posible que las técnicas ya existentes para mejorar la mantenibilidad del software también pudieran ser útiles para mejorar la *greenability.*

Partiendo de este supuesto, es importante averiguar: (i) ¿cómo podemos tratar el código fuente para que lo mejoremos sin que a la vez modifiquemos su funcionalidad? y (ii) ¿qué estructuras/problemas deben detectarse cuando se busca mejorar la mantenibilidad y (quizás) la *greenability?*

La primera pregunta tiene una respuesta muy sencilla. En la etapa de mantenimiento, la *refactorización* es el proceso para mejorar los problemas del código fuente [Mens et al., 2004] [Fowler, 1999], para que de forma simultánea se mejore la calidad del sistema. La refactorización es uno de los pasos principales en la reingeniería de software. La segunda pregunta se refiere a todas las cosas que deterioran el código fuente (a la vez que deterioran el sistema de software). En el mantenimiento, el sistema puede refactorizarse para reparar los errores, además de resolver una programación malhecha o unas prácticas de diseño mal realizadas (los "malos olores" y los "anti-patrones").

La refactorización de software es la mejor opción para tratar el tema del mantenimiento "verde" de software, puesto que es la única manera de tratar el código si hemos de dotar el sistema de software con unas capacidades ecológicas. El asunto clave ahora es descubrir qué ha de ser refactorizado, y qué tipos de mejoras reducen el consumo de energía. Aplicar la refactorización con el fin de darle a un sistema una serie de capacidades *verdes* puede entenderse como una reingeniería de software *verde*, o *Software Greengineering*.

8.3.1 Malos olores (bad smells) en el software

Los malos olores (*bad smells*) se pueden concebir como unas estrategias de diseño deficientes que aparecen en el código fuente como resultado de un desarrollo rápido y descontrolado. En [Fowler, 1999] el autor recoge un conjunto de malos olores que se dan con frecuencia en los sistemas software. La Tabla 8.2 resume el conjunto de olores en el código [Fowler, 1999], además de dar una breve definición de cada uno. Para cada mal olor, el autor también considera un conjunto de posibles maneras de acercarse a la refactorización, pero esto queda fuera del alcance de este capítulo.

Bad Smell	Definición	Impacto
Duplicated Code	Un fragmento de código se repite una o varias veces.	-
Long Method	Unos métodos con demasiados variables, parámetros y código.	0
Large Class	Una clase con demasiados variables de instancia. Demasiadas responsabilidades.	-
Long Parameter List	Un método con una lista larga e incomprensible de parámetros	0
Divergent Change	El código es parecido, pero no es igual.	+
Shotgun Surgery	Para un cambio en concreto, se requieren muchos cambios en otros sitios.	+
Feature Envy	Un método en una clase se interesa más en otra clase que en su propia clase. El método le tiene "envidia" a los datos de otro objeto.	+
Data Clumps	El mismo conjunto de datos se repite en varios sitios (clases, métodos, parámetros, código, etc.).	-
Primitive Obsession	Uso de tipos primitivos en vez de objetos, reduciendo la comprensibilidad.	+
Switch Statements	La misma sentencia switch se encuentra repartida en distintos lugares.	+
Parallel Inheritance Hierarchies	Cuando se forma una sub-clase de una clase, se tiene que formar una sub-clase de otra.	+
Lazy Class	Una clase que no hace más que incurrir gastos para su mantenimiento y para su comprensión.	+
Speculative Generality	Se incluyen demasiados tipos, vínculos y casos especiales en el código para una necesidad del futuro todavía sin determinarse.	+
Temporary Field	Un variable de instancia en una clase no siempre se instancia. Eso hace que sea difícil que se pueda comprender.	0
Message Chains	Una larga cadena de mensajes acopla el cliente del mensaje a una estructura específica de navegación	+
Middle Man	Una clase delega mucho de su comportamiento a otra segunda clase.	+
Inappropriate Intimacy	Dos clases están muy estrechamente acopladas.	+
Alternative Classes with Different Interfaces	Dos métodos que hacen lo mismo, pero se llaman de forma distinta. Dos clases haciendo algo muy parecido.	0
Incomplete Library Class	Las librerías no suelen contener toda la funcionalidad que necesitamos.	0
Data Class	Las clases sin responsabilidad. Solamente campos con métodos getters y setters	+
Refused Bequest	Una clase no necesita todos los campos y métodos que hereda de su clase padre.	+

Tabla 8.2. Malos olores del código y su posible impacto sobre la greenability

En el mantenimiento clásico, la refactorización de software se lleva a cabo para resolver el problema de calidad que subyace en el fenómeno de los malos olores. Como propone el autor, para cada mal olor es posible aplicar una solución de refactorización que transforme el estado actual del sistema a uno equivalente pero de mejor calidad, y sin la presencia de un mal olor. La cuestión del asunto es que en el mantenimiento clásico tales transformaciones son deseables y necesarias, pero no está claro hasta qué punto exista algún tipo de correlación entre el mantenimiento clásico y el mantenimiento "verde" que asegure una mejora de la *greenability* del sistema. En este sentido, se requieren estudios experimentales para que se establezca cuáles son los malos olores que se deban quitar, y cuáles son las soluciones de refactorización que han de aplicarse. La encrucijada mencionada en la conclusión de la sección del estudio de caso deja claro que hay algunas mejoras en la mantenibilidad que son inversamente proporcionales a la mejora de la *greenability.*

Aparte del mal olor y la correspondiente definición de cada uno, la Tabla 8.2 incluye una columna llamado *Impacto.* "Impacto" hace referencia al efecto que la refactorización de un mal olor tiene sobre la *greenability.* Como hemos dicho anteriormente, no se ha investigado el efecto de la refactorización sobre la *greenability,* así que se ha formulado una hipótesis para los valores de la columna de *Impacto* (es decir, se requiere experimentación para que se validen). Se han designado tres valores para predecir el impacto de una refactorización: (i) "+", cuando la refactorización puede tener un efecto positivo, mejorando la *greenability;*(ii)"0", cuando no está claro si la refactorización tiene un efecto positivo o negativo sobre la *greenability; y "*-*"* cuando la refactorización empeora la *greenability.*

8.3.2 Antipatrones (antipaterns)

Según [Brown et al., 1998], los antipatrones son *"una forma literaria que describe una solución a un problema que ocurre con frecuencia y que genera unas consecuencias claramente negativas. El antipatrón puede ser el resultado de un gestor o un desarrollador que no sepa hacer su trabajo de otra manera mejor, o que no tenga los conocimientos o la experiencia suficientes para saber resolver un tipo de problema en concreto, o que haya aplicado un patrón perfectamente bueno en el contexto equivocado".* Este tipo de antipatrones ejercen un impacto negativo sobre la calidad de software, y tienen la capacidad de empeorar la mantenibilidad del software.

Por otro lado, los antipatrones podrían ser una manera de tratar la *greenability* del software. Encontrar y resolver los antipatrones mejora la calidad del software, pero el reto ahora es descubrir hasta qué punto la refactorización (para eliminar los antipatrones) pueda también afectar la *greenability.*

En [Brown et al., 1998] los autores abordan el tema de los antipatrones desde tres perspectivas diferentes: (i) los antipatrones del desarrollo de software, o los problemas técnicos y las soluciones que se introducen por la acción de los programadores; (ii) los antipatrones de la arquitectura; es decir, los problemas comunes en cuanto a cómo los sistemas se estructuran; y (iii) los antipatrones de gestión, o los problemas en los procesos de software y en las organizaciones de desarrollo. Desde el punto de vista de la ingeniería de software, el tipo más interesante de antipatrón de software es el primer tipo, es decir los antipatrones de desarrollo, porque representan los problemas que deberían ser abordados en el mantenimiento de software. Los antipatrones de arquitectura también podrían ser considerados como significativos, puesto que la mayor parte de las consecuencias de estos antipatrones podrían tratarse en el código fuente (aliviando sus síntomas, por lo menos). Por otro lado, no es útil el antipatrón de gestión si nos acercamos al tema de la *greenability* desde un punto de vista de la Ingeniería del Software. Los antipatrones de gestión identifican unos problemas relacionados con los recursos humanos que intervienen en los proyectos de software. Aunque ciertos antipatrones de gestión (aquellos relacionados con la manera de trabajar) pueden asociarse de forma directa con la *greenability,* no son el enfoque central de este capítulo. La Tabla 8.3 resume los patrones de desarrollo más comunes.

Antipatrón	Definición	Impacto
The Blob (God Class)	Una clase contiene la mayor parte de las responsabilidades, mientras que las otras contienen solamente datos y proceso menor.	-
Continuous Obsolescence	La evolución de la tecnología hace que les sea difícil a los desarrolladores mantener una buena inter-operación del software con otros productos.	0
Lava Flow	Código muerto y diseño olvidado se congelan dentro de un diseño que está en constante evolución.	0
Functional Decomposition	Los sistemas OO producidos por los desarrolladores no orientados a objetos. El código orientado a objetos se parece al lenguaje estructural.	+
Poltergeist	Unas clases con un ciclo de vida de muy corto plazo. Este tipo de clase suele tener la responsabilidad de iniciar los procesos.	+
Boat Anchor	Un artefacto de software o hardware que, a pesar de ser muy costoso, no cumple ningún propósito útil.	+
Golden Hammer	Los equipos de desarrollo aplican vez tras vez unas soluciones en las tienen gran experiencia, en vez de buscar explorar unas soluciones nuevas.	0
Dead End	Se modifica un componente reutilizable apoyado por un vendedor o un suministrador, lo que complica la integración de este tipo de modificaciones en los nuevos productos.	0

Spaghetti code	La estructura de software se elabora de una manera *ad hoc*, una circunstancia que hace que sea difícil de mantener y extender.	-
Input Kludge	Los algoritmos de tipo ad hoc gestionan la entrada del programa.	+
Walking through a Minefield	Los productos se entregan demasiado pronto, y un número muy significativo de errores están (con toda probabilidad) en el código.	+
Cut-and-Paste Programming	Unos bloques de contenido de código fuente cortados y pegados ocasionan unos problemas de mantenimiento.	-
Mushroom Management	Los desarrolladores están aislados del usuario final del sistema, Los requisitos se reciben de manera indirecta, por medio de otros intermediarios.	0

Tabla 8.3. Los antipatrones del desarrollo de software

La Tabla 8.4 presenta los antipatrones más comunes en la arquitectura. Según [Mowbray, 1998], los antipatrones en la arquitectura se *"enfocan en algunos problemas y errores comunes en la creación, implementación y gestión de arquitectura."*

Antipatrón	Definición	Impacto
Autogenerated Stovepipe	Un sistema local es migrado a una arquitectura distribuida. Si el diseño no varía y se mantiene igual, aparecen los problemas, como los que tienen que ver con la manera en la que se trasladan los datos.	+
Jumble	Los elementos verticales y horizontales se mezclan. El software es complejo y difícil de desarrollar y reutilizar.	-
Stovepipe System	No hay ni abstracción ni documentación de sub-sistemas. Su integración tiene que hacerse de manera *ad hoc*.	+
Cover Your Assets	Los requisitos se extienden por "toneladas" de documentos. Los desarrolladores no tienen ninguna idea sobre qué hacer con este caos de información.	NA
Vendor Lock-In	Un producto adopta una tecnología en concreto y llega a ser dependiente de las condiciones que impone el vendedor. Los problemas surgen cuando se actualiza el producto.	+
Wolf Ticket	Un producto que cumple con los requisitos de software, pero cuyas interfaces en realidad pueden que no cumplan los estándares publicados.	0

Architecture by Implication	Unos arquitectos con un exceso de confianza creen que cierta documentación sobre la arquitectura no es necesaria. Tienen gran experiencia, y consideran que pueden prescindir de algo, puesto que ellos sí que lo tienen muy claro en su mente. El desarrollo no es posible sin que tal información esté documentada.	0
Warm Bodies	Se asignan muchos programadores al proyecto, pero solo unos cuantos son desarrolladores de la calidad.	NA
Design by Committee	Un diseño complejo de software suele ser desarrollado por un comité de expertos. Una diversidad de opinión en las decisiones democráticas conduce a unos diseños que son complicados de implementar.	NA
Swiss Army Knife	Una interfaz excesivamente compleja. La clase intenta servir demasiados propósitos.	-
Reinvent the Wheel	No hay un traslado tecnológico entre un proyecto y otro nuevo. Se aprovecha al máximo la ventaja de los conocimientos de diseño que ya se poseen.	0
The Grand Old Duke of York	No se tiene en cuenta el talento humano cuando se define la arquitectura del sistema. Las aptitudes para comunicarse con la gente son importantes a la hora de decidir en qué etapa del desarrollo del software deba trabajar un participante en particular.	0

Tabla 8.4. Los Antipatrones en la Arquitectura de Software

La Tabla 8.3 y la Tabla 8.4 incluyen una columna llamada *Impacto,* cuyo propósito es representar el posible impacto de aplicar la refactorización de software para resolver los antipatrones. En este caso, hay cuatro posibles valores: "+" si la refactorización tiene un impacto positivo sobre la *greenability, "-"* si la refactorización tiene un impacto negativo sobre la *greenability* y "NA" cuando no hay refactorización de software para resolver el antipatrón (es decir, está asociado más con el proceso que con el producto). Como hemos señalado con los malos olores del código, la predicción del impacto de la refactorización de los antipatrones no es más que una mera hipótesis. Es en este punto que se han de llevar a cabo experimentos con cada antipatrón (y con las combinaciones de éstos), para así averiguar cuáles de ellos serían candidatos adecuados para mejorar la *greenability* en una etapa de mantenimiento (es decir para un mantenimiento *verde*).

8.4 LA DEUDA ECOLÓGICA

Aunque la deuda técnica es de hecho la ausencia de unos requisitos funcionales o no-funcionales esenciales (una ausencia que ha ocurrido de manera intencionada,

o no intencionadamente), es posible describir una situación parecida con respeto al desarrollo *verde* de software: el concepto de la *deuda ecológica*. Los requisitos de *greenability* (como requisitos no-funcionales) no serían un valor absoluto (por ejemplo, el tiempo de respuesta cuando se hace una consulta), sino que se permitirían estar en cualquier ubicación dentro de un abanico de valores. Evidentemente, cuanto más *verde* sea el sistema, menor será su consumo de recursos. Pero una vez más, los problemas de tener que encontrar un equilibrio, deben ser analizados: si los costes a largo plazo de hacer un software más sostenible (pero con un nivel aceptable de *greenability*) son inferiores a los que supondría hacer un sistema con un alto nivel de *greenability*, entonces sí que sería factible asumir cierto nivel de *deuda ecológica*. Es importante hacer un seguimiento de esta deuda, porque es posible que se cambie una estipulación legal, o algún requisito; y que se tuviera que refactorizar los sistemas, con el consiguiente desperdicio de recursos, aparte de incurrir los inevitables costes adicionales. Eso significa: (i) una deuda técnica del código y la documentación que ha de ser refactorizada, y el (ii) consumo excesivo (pero aceptado) de recursos hasta el momento en el que ha ocurrido la modificación de las reglas o los requisitos.

Sin embargo, no toda la *deuda ecológica* se debe a unas decisiones tomadas gracias a los requisitos de *greenability* (o de dejar de cumplir los requisitos totalmente). Otros tipos de *deuda ecológica* son casi inevitables y hasta necesarios; por ejemplo, en la estrategia a seguir para actualizar un producto software influye en muchos aspectos de ese producto, como el traslado/migración de datos, la infraestructura de procesamiento y del hardware, etc., lo que puede incurrir en un consumo mayor de recursos y energía [Nauman et al., 2011], implicando una disminución de la *greenability* del software. Si sometemos nuestros sistemas a estas políticas o a estas estrategias, estamos aceptando una deuda ecológica que ha de ser reconocida y cuantificada.

Por tanto, la *deuda ecológica* puede ser considerada como "*el coste (en términos del uso de recursos) de entregar un sistema software con un nivel de greenability por debajo del nivel de los requisitos no funcionales establecidos por los interesados, más el coste incurrido que se requiere para refactorizar el sistema en el futuro"*

$$\text{Deuda\_Ecológica} = \Sigma Coste(recurso_i) + \Sigma Refactorización(Defecto\_Ecológico_i)$$

La deuda ecológica y la tecnológica tienen un factor en común; la necesidad de arreglar los defectos. Como se propuso en la sección anterior, un posible punto de partida para clasificar qué defectos afectan la *greenability* sería el análisis de aquellos defectos relacionados con el mantenimiento clásico, validando su impacto sobre la *greenability* de software. La ecuación anterior apunta a una consideración muy importante cuando se habla de la deuda ecológica: se trata del coste fijo (y no recuperable) de los recursos sobreutilizados. Un recurso

sobreutilizado está considerado como un recurso de software o de hardware que está sobredimensionado para la necesidad real del software. Por ejemplo, un recurso sobredimensionado muy común que está relacionado con la *greenability* es el consumo energético. El consumo de energía se puede expresar como coste económico ($, €, £, etc.), huella de carbono (CO_2) o energía eléctrica. El coste de estos recursos es el tipo de inversión que no se puede recuperar o reparar (a diferencia de la deuda técnica). Es posible hacer una correspondencia entre el consumo de energía malgastada y un concepto económico: el coste no-recuperable (es un desembolso que no se puede recuperar).

No obstante, es importante señalar que en la deuda ecológica este gasto no-recuperable debe ser tenido en cuenta en el contexto del mantenimiento de la estrategia de mantenimiento de software para el portafolio o cartera de aplicaciones del software. La deuda técnica (así como la refactorización de los defectos ecológicos de la deuda ecológica) existe a lo largo de todo el tiempo durante el cual la organización decide no resolverla, pero el factor del gasto no-recuperable de la deuda ecológica es un gasto continuo que nunca se recuperará. Este es el motivo más importante que nos impulsa a querer definir y formalizar el mantenimiento *green* de software que se ha presentado en este capítulo.

Anteriormente en este capítulo, avanzamos nuestra propuesta de que se considere el mantenimiento *green* y el mantenimiento clásico bajo el mismo proceso; lo mismo debe hacerse con la deuda técnica y la deuda ecológica. Ambos tipos de deuda tienen la misma necesidad de arreglar los defectos. De hecho, podemos afirmar que el sistema de software tiene un conjunto de defectos que producen sus deudas, y que ese conjunto está compuesto de la unión de los defectos, tanto los ecológicos como los técnicos.

Para cualquier sistema en particular, si no existiera un conjunto de defectos (técnicos y ecológicos) que afecta la mantenibilidad y la *greenability* claramente, de manera positiva, y en el mismo grado, sería preciso hacer un análisis de compromisos, puesto que la refactorización de los defectos tecnológicos afectaría negativamente a la deuda ecológica, y viceversa. Por otro lado, si hubiera unos defectos en común, sería posible aplicar una serie de transformaciones de refactorización que beneficien tanto la deuda ecológica como la tecnológica (mejorando la *greenability* y la mantenibilidad). Finalmente, (si los defectos técnicos y ecológicos fueran los mismos (aunque no es muy probable que esto ocurra), entonces la refactorización de todos ellos reduciría ambos tipos de deuda, mejorando así la *greenability* y la mantenibilidad.

8.5 ESTUDIO DE CASO

El estudio de caso que se presenta aquí es un extracto de otro más extenso [Pérez-Castillo y Piattini, 2014], consistente en un estudio exploratorio que considera el antipatrón de software del tipo *God Class (Clase Dios)* [Smith y Williams, 2000], además de investigar los efectos secundarios de aplicar una posible refactorización (véase la Figura 8.2). Este antipatrón consiste en la clase en cuestión (una clase dios) que (i) lleva a cabo la mayor parte del trabajo del sistema; (ii) desempeña el papel de controlador y, (iii) que se encuentra rodeada de clases sencillas (contenedores de datos).

Los autores ilustran el problema central por medio del siguiente ejemplo sencillo y minimalista. Consideremos un sistema de pagos en el cual es necesario realizar unos pagos y reembolsos. Por un lado, la Figura 8.2 (a) muestra un fragmento del diagrama de clase para una posible arquitectura de este sistema. La clase *CreditCard* puede considerarse una clase dios, puesto que contiene casi toda la inteligencia. Recupera cada operación y comprueba el estatus (aceptado o rechazado) y, según el estatus, realiza el pago o el reembolso. Este diseño de la arquitectura tiene una cohesión deficiente, y está muy acoplada a las clases de datos. Por otra parte, la Figura 8.2 (b) aporta una sencilla solución de refactorización, de la cual se ha extraído una clase para la inteligencia de la operación. La clase *Operation* no hace más que informar sobre su estatus (aceptado o rechazado), y responde a las invocaciones de *pay()* y *refund()*. La clase *CreditCard* hace todo el trabajo: le pide la información necesaria a la *Operation,* toma decisiones, y le dice a la *Operation* qué es lo que debe hacer.

Tal y como se puede observar en la Figura 8.2, el problema principal que surge después de refactorizar el sistema fuente reside en el hecho de que la comunicación (el intercambio de mensajes) entre la clase dios, las clases secundarias, y las nuevas clases (creadas después de refactorizar el antipatrón) aumentan de forma considerable. Esta consecuencia establece el equilibrio entre la mantenibilidad (la calidad de diseño) y la *greenability* (consumo de energía).

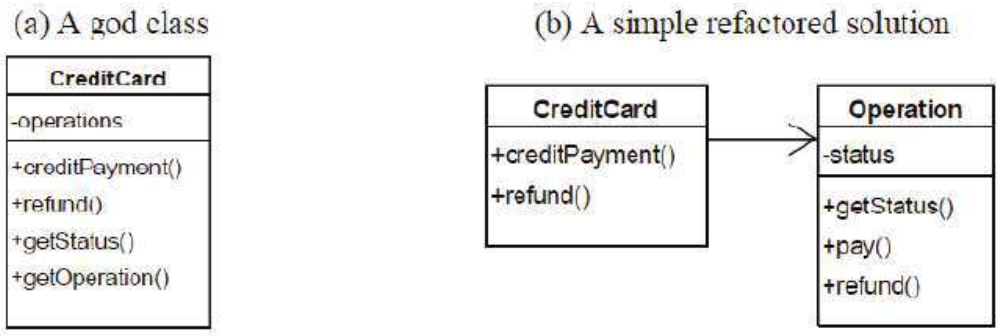


Figura 8.2. (a) Ejemplo de Clase Dios, y (b) posible refactorización

Se propone la siguiente hipótesis de investigación: *el consumo de energía disminuye como resultado de haber reducido el tráfico de mensajes entre objetos*, de manera que el objetivo de la investigación es demostrar que cuando se aplican patrones típicos de refactorización para detectar antipatrones conocidos, se produce un tráfico excesivo de mensajes entre objetos. Por consiguiente, el consumo energético también es más alto.

La hipótesis fue evaluada por dos estudios de caso industriales (y de código abierto) Informa [Informa, 2007], (que ofrece una librería RSS basada en la plataforma Java), y *NekoHTML* [CyberNeko, 2009] (un escáner HTML para analizar los documentos HTML y habilitar el acceso a la información por medio de las interfaces XML).

Para realizar el estudio de caso, se siguieron los pasos que se exponen a continuación:

1. Se analizan los dos sistemas bajo estudio, con el fin de detectar posibles incidencias del antipatrón de la clase dios. Los análisis fueron realizados con el *plug-in* JDeodorant eclipse [JDodorant, 2012], el cual, a su vez, propone una posible refactorización para resolver el problema. La refactorización se aplica a continuación, y se obtiene una nueva versión para cada sistema: *Informa* ^{R}y *NekoHTML*^{R}.

2. El siguiente paso consiste en la medición del tráfico entre objetos. Para llevar a cabo una comparación estricta entre los sistemas de información iniciales y los que han sido refactorizados, se utiliza el mismo escenario de ejecución; está basado en los casos de prueba de los dos sistemas. Para cuantificar las invocaciones de la operación de objetos, el código fuente de ambos sistemas fue rastreado y perfilado (instrumentado) utilizando el *Eclipse Test & Performance Tools Platform* (TPTP) [Eclipse, 2013].

3. El consumo de energía también se mide para ambas versiones de cada sistema. Asimismo, el escenario de ejecución se establece, utilizando los casos de prueba aportados por los desarrolladores del software. Para medir el consumo de energía, los sistemas ahora se ejecutan sin ninguna instrumentación, procurando hacer todo lo posible para evitar cualquier sesgo. La medición se hace por el registrador de energía *Volcraft Energy Logger 4000*. Este artefacto mide el consumo de energía por segundo en vatios (W). Se mide también el uso del procesador.

4. Una vez recopilados todos os datos, los resultados se analizan y se proporcionan unas interpretaciones, con el propósito de verificar la hipótesis inicial.

Los datos recogidos por los estudios de caso se resumen en la Tabla 8.5, que proporciona la mayor parte de las métricas de la arquitectura/diseño del sistema original y refactorizado bajo estudio, además de mostrar la diferencia entre las dos versiones. La parte superior de la Tabla 8.5 muestra (i) el número de líneas de código fuente (ii) el número de clases; (iii) el número de métodos; (iv) el acoplamiento aferente como el promedio del número de clases fuera de un paquete que dependen de unas clases dentro del paquete; (v) el acoplamiento eferente como el número de clases dentro de un paquete que dependen de unas clases fuera del paquete; y finalmente (vi) la complejidad ciclomática de McCabe, el cual calcula la cantidad de flujo que hay en un fragmento de código.

	Métrica	Informa	Informa^R	Dif. (%)	NekoHTML	NekoHTML^R	Dif. (%)
Arquitectura	#Líneas de código	9739	9891	1.56%	7938	8179	3.04%
	#Clases	116	127	9.48%	60	74	23.33%
	#Métodos	996	1024	2.81%	473	523	10.57%
	Acoplamiento Aferente	10	10.5	5.00%	5.29	5.43	2.71%
	Acoplamiento Eferente	7.21	7.57	4.95%	5.57	7.29	30.78%
	Complejidad Ciclomática	1.87	1.84	-1.18%	3.44	3.23	-6.24%
Refact.	#Clases Dios	21	0		10	0	
	Ratio Clase Dios	18.1%	0%		17%	0%	
	#Extrac. Clases	49	0		26	0	
Ejecución	#Casos de Prueba	337	337	0.00%	4201	4201	0.00%
	# Errores	71	71	0.00%	0	0	0.00%
	# Fallos	18	19	5.56%	1800	2200	22.22%
	# Mensajes	6221	97846	1473%	1550848	7900600	409%
	Tiempo (s)	57	60	5.26%	22	27	22.73%
Potencia	Vatios totales	2052.6	2207.7	7.56%	743.9	893.4	20.10%
	Vatios	36.7	37.4	1.91%	33.8	34.4	1.62%

Tabla 8.5. Métricas de la arquitectura, el tráfico de mensajes, y el consumo energético durante la ejecución

Aunque hay una enorme cantidad de datos incluidos en la Tabla 8.5, lo que es más importante destacar para nuestro propósito es el incremento del intercambio de mensajes, además del aumento en el consumo de energía. Por un lado, los datos revelan que la arquitectura refactorizada produce entre 4 y 14 veces más mensajes para ambos sistemas. Por otro lado, la Figura 8.3 presenta la evolución del consumo activo de energía (en vatios) durante la ejecución de los sistemas originales y la de los sistemas refactorizados.

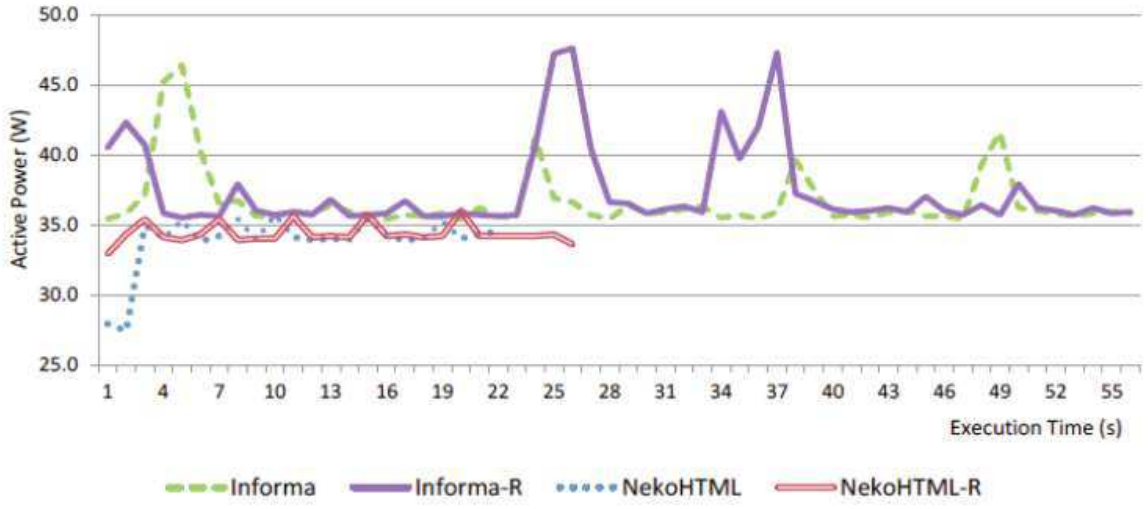


Figura 8.3. Consumo energético durante la ejecución

La diferencia en el consumo de energía, como promedio, fue de 1,91%. En el caso de *NeckoHTML*, el sistema original consumía unos 33,8 vatios por segundo, mientras que el consumo de *NeckoHTML* ^{R} tenía un promedio de 34,4. Esto supone un aumento de 1,62%. Además, puesto que el tiempo para la ejecución fue mayor para los sistemas refactorizados, los aumentos en el consumo de energía (sus valores absolutos) fueron de 7,6% y 20,1%, respectivamente. Es probable que los aumentos en el consumo energético se deban a los diferentes escenarios de ejecución (que se basan en los casos de prueba) para cada sistema. Estos escenarios podrían provocar que, como resultado de la refactorización de la clase dios, la herencia y estructuras generadas en la refactorización, afecten la ejecución provocando ese incremento en el consumo.

A pesar de las varias limitaciones y amenazas a la validez de este estudio, los resultados del experimento han demostrado que una arquitectura en la cual las clases dios han sido refactorizadas puede empeorar en cuanto al consumo de energía. Eso se debe al excesivo tráfico de mensajes que se deriva de la refactorización de la arquitectura, la cual a su vez produce un efecto dañino sobre el consumo energético de los sistemas refactorizados.

8.6 LECTURAS RECOMENDADAS

✓ Calero, C. y Piattini, M. (eds.) (2015). Green in Software Engineering, Alemania, Springer.

En este libro, se presenta una amplia panorámica sobre los diferentes esfuerzos para conseguir un software más sostenible.

8.7 EJERCICIOS

Ejercicio 1

Enumere al menos cuatro antipatrones típicos de los sistemas orientados a objetos.

Ejercicio 2

Dado un software *open source* o un desarrollo propio, utilice una herramienta para la detección de antipatrones, como *JDeodorant* [8], para identificar el antipatrón *God Class*.

Ejercicio 3

Empleando un entorno para la refactorización de *bad smells*, trate de eliminar las ocurrencias del antipatrón *God Class* identificado en el Ejercicio 2.

Ejercicio 4

Dada la refactorización del patrón *God Class* llevada a cabo en el Ejercicio 3, trate de justificar si a su juicio, el incrementará en número de mensajes intercambiados por las clases de la nueva versión del software.

Ejercicio 5

Compare los conceptos de deuda técnica y deuda ecológica.

8 *https://marketplace.eclipse.org/content/jdeodorant*

9

TÉCNICAS PARA EL MANTENIMIENTO

En este capítulo introducimos los conceptos de Ingeniería Inversa, Reingeniería y Modernización, presentando algunos métodos y técnicas para llevarlos a cabo. Además, se introducen y comentan otra serie de soluciones técnicas, destinadas a facilitar las labores del mantenimiento de software.

9.1 INTRODUCCIÓN

Bajo el concepto de «soluciones técnicas» englobamos el conjunto de operaciones que se realizan directamente sobre el producto software propiamente dicho (código fuente, sistema de ficheros, bases de datos) con el fin de modificarlo. Como sabemos, tales modificaciones pueden ir encaminadas a la corrección de errores, a la adición de nuevas funcionalidades, a mejorar su rendimiento u otras propiedades, o a su adaptación a un cambio de entorno.

En cualquiera de estas situaciones, se exige que el programador que va a implementar el cambio posea al menos un mínimo conocimiento del producto software y de su dominio de aplicación, para luego pasar a localizar —en el caso de, por ejemplo, el mantenimiento correctivo— el error y su causa, realizar la necesaria modificación y prueba, y pasar el producto intervenido al entorno de funcionamiento.

Una de las técnicas que pueden utilizarse para facilitar la comprensión del producto software es la Ingeniería Inversa que, como señala [Arnold, 1992], es «el proceso de construir especificaciones formales abstractas del código fuente de un sistema heredado (*legacy*), de manera que estas especificaciones puedan ser utilizadas para construir una nueva implementación del sistema usando Ingeniería *hacia delante*».

No obstante, otros autores no indican que el nivel de partida del proceso de Ingeniería Inversa tenga que ser siempre el código fuente, sino que puede ser cualquier nivel dado de abstracción [Piattini *et al.*, 1996]. Estos mismos autores afirman que la Ingeniería Inversa «recrea modelos pertenecientes a niveles superiores, ya sean orientados a datos o a procesos», que podemos asociar, respectivamente, a bases de datos y a programas. En este capítulo abordamos ambos tipos de orientación.

Parafraseando a [Biggerstaff *et al*., 1994], al final del proceso de Ingeniería Inversa se debe poder explicar qué hace el programa, cuál es su estructura, qué efectos tiene en su contexto operacional y cuáles son sus relaciones con su dominio de aplicación. El ingeniero de mantenimiento software, entonces, en sus trabajos de Ingeniería Inversa, no modifica la funcionalidad ni las características de un sistema, sino que, simplemente, actúa como un notario que toma nota exacta de lo que ve, aunque a un nivel más alto de abstracción. Un ejemplo de esto sería la generación de un diagrama de clases a partir de un fragmento de código fuente.

Si, tras la aplicación de Ingeniería Inversa decidimos utilizar técnicas de «Ingeniería directa» para reconstruir el sistema, estaremos utilizando el concepto de Reingeniería, que [Arnold, 1992] define como «Cualquier actividad que:

- Mejore la comprensión del software.
- Prepare o mejore el propio software, normalmente para incrementar su facilidad de mantenimiento, reutilización o evolución».

Hilando esta definición con la de Ingeniería Inversa vista más arriba, también de la misma referencia, podemos definir de forma válida la Reingeniería como «la modificación de un producto software, o de ciertos componentes, usando para el análisis del sistema existente técnicas de Ingeniería Inversa y, para la etapa de reconstrucción, herramientas de Ingeniería Directa, de tal manera que se oriente este cambio hacia mayores niveles de facilidad en cuanto a mantenimiento, reutilización, comprensión o evolución».

Sin olvidar que estamos dentro de un área de la Ingeniería del Software, debemos tener presente que la reconstrucción de componentes del sistema puede consistir tanto en reprogramarlo, en redocumentarlo, en rediseñarlo, como, en general, en rehacer algunas o todas las características del producto que sean necesarias. En particular, y teniendo presentes los objetivos de la Reingeniería arriba expuestos, es de especial importancia realizar una correcta Redocumentación para cualquier cambio que se realice. Según [Arnold, 1992], la Redocumentación es «la creación de información correcta y actualizada del software» (no olvidemos que el concepto de «software» es algo más que el producto final): si observamos la Figura 9.1 (entre

paréntesis se indica el orden en que se realiza cada tarea), vemos cómo realizamos redocumentación en cada nivel de abstracción del proceso de Reingeniería. Profundizando en esta idea, [Pressman, 2014] menciona tres opciones respecto a qué y cómo redocumentar:

1. Si el sistema funciona y la redocumentación consume muchísimos recursos, es posible que sea más correcto dejar el tema como está y no redocumentarlo.

2. Si es preciso actualizar la documentación, pero los recursos disponibles para hacerlo son limitados, puede seguirse la idea de «documentar cuando se modifica». Con el tiempo, se irá construyendo una colección de documentación útil e importante.

3. Si el sistema es fundamental para la organización, es preciso volver a documentarlo por completo. En este caso, una opción inteligente es reducir la documentación al mínimo imprescindible.

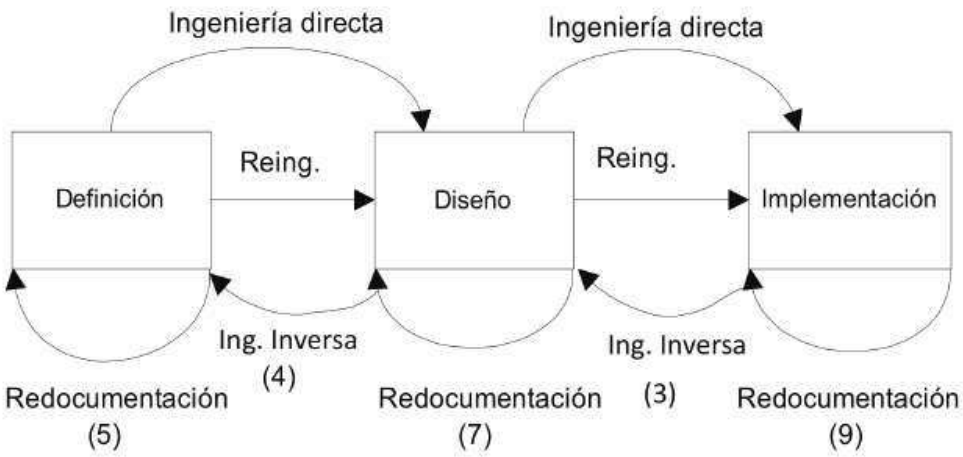


Figura 9.1. Ingeniería directa, Ingeniería inversa, Reingeniería y Redocumentación.

Otro concepto mencionado anteriormente es el Rediseño, que consiste en «consolidar y modificar los modelos obtenidos añadiendo nuevas funciones requeridas por los usuarios» [Piattini *et al.*, 1996].

Queremos terminar este apartado insistiendo en la idea de que es la Reingeniería la que modifica el sistema; la Ingeniería Inversa no modifica nada de lo que hay: sólo refleja la realidad en un nivel más alto de abstracción.

9.2 INGENIERÍA INVERSA DE PROGRAMAS

Si estamos trabajando en un proyecto de Ingeniería Inversa sobre un producto de software escrito en C y constantemente nos encontramos con fragmentos de código como el mostrado en la Figura 9.2, es evidente que, aunque la complejidad sea pequeña, nos veremos constantemente obligados a mirar con lupa código prácticamente ilegible.

```
Char *f(char *c) {
   unsigned long i, lon;
   char *x=(char *) malloc(100);
   lon=strlen(c);
   if (c[0]==’\’’) {
        for (i=0; i<lon-1; i++) x[i]=c[i+1]; x[i]=’\0’;
    } else
    if (c[0]==’\”’) {
         for (i=0; i<lon-1; i++) x[i]=c[i+1];  x[i]=’\0’;
    } else for (i=0;i<lon;i++)

   x[i]=c[i];
   if ((x[strlen(x)-1]==’\’’) || (x[strlen(x)-1]==’\”’))

   x[strlen(x)-1]=’\0’;
   return x;
}
```

Figura 9.2. Código funcionalmente correcto, pero poco legible

Deducimos, de un primer vistazo, que las líneas de código de la Figura 9.2 toman una cadena como parámetro y devuelven también una cadena. Podemos suponer que el valor devuelto por la función va a ser una transformación sobre la cadena que se le pasa. Esta deducción, que en este caso es correcta, se complica un poco si intentamos conocer el tipo de transformación que *f* realiza sobre *c*. Si disponemos del código fuente y de un compilador apropiado, podremos hacer una traza de su ejecución usando el depurador. Si alguna de las dos condiciones no se cumple, la traza la realizaremos a mano, construyendo posiblemente una tabla en la que iremos reflejando, además, los cambios sufridos por las variables. Probablemente comprobaremos la función aplicándole diversos ejemplos y, también probablemente, tras unos minutos, pensaremos que el trabajo habría resultado mucho más fácil si hubiéramos leído el código tal y como se muestra en la Figura 9.3.

```
#define COMILLA_SIMPLE  '\''
#define COMILLA_DOBLE   '\"'
   ..........
   /*Autor: Juan Gómez Montijo
   Entradas: una cadena.
   Devuelve: la misma cadena sin comillas ni al principio ni
al
             final, si es que las tenía. */
  char *QuitaComillas(char *Cadena) {
   unsigned long i, l;
   char *Resultado=(char *) malloc(100);
   /* Quitamos la comilla final, si la hay */
   if ((Cadena[strlen(Cadena)-1]==COMILLA_SIMPLE) ||
       (Cadena[strlen(Cadena)-1]==COMILLA_DOBLE))

   Cadena[strlen(Cadena)-1]='\0';
   /* Pasamos la cadena a una auxiliar, quitando la comilla
ini-
       cial si la hay */
   if ((Cadena[0]==COMILLA_SIMPLE) ||

   (Cadena[0]==COMILLA_DOBLE)) {

   for (i=0; i<strlen(Cadena)-1; i++) {

   Resultado[i]=Cadena[i+1];

   }
   } else

   for (i=0; i<strlen(Cadena); i++)

   Resultado[i]=Cadena[i];
   Resultado[i]='\0';
   return Resultado;
  }
```

Figura 9.3. Código equivalente al de la Figura 9.2, pero legible y bien documentado

Durante el análisis del gran programa en el que podría estar incluido el segundo fragmento es casi seguro que, leído el significativo nombre de la función y los comentarios que lleva en su cabecera, no nos detendríamos a investigar el mecanismo por el cual quita o deja de quitar las comillas de una cadena. Es decir,

estaríamos evitando el estudio de lo que resulta accesorio para centrarnos en lo que realmente resulta práctico, que es, precisamente, saber que la función quita las comillas a una cadena.

Podemos profundizar en este concepto señalando que, en los proyectos de Ingeniería Inversa y Reingeniería de productos terminados, la primera ocupación del ingeniero se centra en asignar o asociar valor semántico a los elementos sustanciales del código fuente. [Biggerstaff, 1994] y [Weide, 1995] indican las dos tareas necesarias para esta labor:

1. Identificación y recopilación de los *componentes funcionales* del sistema que hacen que éste se comporte de la manera en que nosotros podemos describirlo a alto nivel. Aquí englobamos rutinas, variables, constantes o grupos de ellas; tipos de datos, tipos abstractos de datos, objetos, clases, llamadas a funciones o a procedimientos y cualesquiera otros elementos.

2. Asignar significado, dentro del contexto de la aplicación, a los componentes funcionales recopilados en el punto anterior, de tal manera que cumplamos el doble objetivo de explicar cómo cada componente funcional desempeña su papel en el comportamiento conjunto del sistema, y cómo funciona el sistema a partir del comportamiento de cada componente funcional.

Retomando el ejemplo de la Figura 9.3, identificaremos la función *QuitaComillas* como un componente funcional si el hecho de quitar las comillas a una cadena es una actividad del programa realmente sustancial. En caso afirmativo, tras concluir la segunda tarea deberemos poder responder a preguntas como «¿En qué situaciones se llama a esta función?» o «¿Por qué es necesario procesar esta cadena sin comillas?».

9.2.1 Identificación y recopilación de componentes funcionales

Todo proceso de descubrimiento e investigación precisa, además del conocimiento del entorno de trabajo, dosis altas de intuición y perspicacia, cualidades éstas que podríamos calificar como *herramientas informales* y que son de gran utilidad.

[Biggerstaff, 1994] afirma que, para las tareas de identificación y recopilación, usamos conocimiento genérico para inferir, por ejemplo, que un bloque de código realiza una actividad de importancia dentro de la aplicación. Es decir, que aunque no hallemos una causa objetiva que justifique por qué hemos deducido que un módulo es imprescindible para la comprensión a alto nivel de un sistema,

sí que existe una serie de factores, ligados más o menos a nuestro conocimiento, a nuestra experiencia, a nuestra intuición, que permiten identificar el módulo como un componente funcional.

La ausencia en un sistema de un componente funcional impide de manera seria (su reparación lleva consigo costes elevados) el funcionamiento de la aplicación, dificulta la legibilidad del código o del documento (según en qué etapa del proceso de Ingeniería Inversa nos encontremos), impide la comprensión del proyecto en general o de otro componente funcional, o bien hace caer a niveles inadmisibles los índices de calidad, fiabilidad, rendimiento, etc.

Algunas ideas que pueden ayudar a la identificación de componentes funcionales son las siguientes:

- Cada componente funcional puede o bien ocupar un módulo distinto de código fuente (por ejemplo, dispondremos de una biblioteca distinguida con la definición del tipo de datos *Lista* y sus operaciones asociadas), o bien los componentes funcionales aparecen próximos unos a otros (serie declaraciones de constantes que definen los tamaños de los campos de una tabla, por ejemplo).

- Las series de componentes funcionales suelen aparecer junto a muchos comentarios, agrupados entre límites formados por cadenas de asteriscos u otros símbolos.

- Los identificadores de los componentes funcionales suelen constar de muchos caracteres. Recordemos que, en el ejemplo de la Figura 9.3, la función se llamaba *QuitaComillas*.

Por supuesto, de gran ayuda resulta el hecho de que posiblemente la documentación del proyecto se encuentre ordenada y agrupada cronológica o funcionalmente, si es que en el desarrollo del producto se aplicaron adecuadamente las técnicas de la Ingeniería del Software.

En nuestra inmersión en el estudio del código fuente para identificar componentes funcionales encontraremos, además de las dificultades naturales inherentes a este trabajo, otras más detallistas y específicas como:

- La sinonimia, o uso de distintos identificadores para referirse a un mismo componente funcional.

- La polisemia, o uso de un mismo identificador para referirse a componentes funcionales diferentes.

Otro problema muy corriente es la existencia de comentarios no actualizados en el código fuente, que dificultan el trabajo del ingeniero de software despistándolo y llevándolo a conclusiones erróneas o contradictorias. Ténganse en cuenta situaciones como la del sencillo ejemplo de la Figura 9.4, donde ni el comentario de cabecera ni el nombre del método *TotalConIVA* son ya válidos, puesto que, como se puede observar, se ha eliminado la ejecución de la instrucción *return r*(1+mIVA)*, tal vez durante las pruebas realizadas por algún programador, que ha olvidado deshacer el cambio.

```
Class Factura {
    Private Cliente mCliente;
    Private Vector mLineas;
    Private double mIVA;
    Date mFecha;
    Private String mNumero;
     ......

/* Autor: Lázaro Stoller.
   Devuelve el total con IVA de la factura instanciada. */
   Public double TotalConIVA() {
        double r=0;
        LineaDeFactura l;
        for (int i=0; i<=mLineas.size()-1; i++) {
            l=(LineaDeFactura) mLineas.elementAt(i);
            r+=l.getPrecio();
        }
//        return r*(1+mIVA);
        return r;
    }
     ....
}
```

Figura 9.4. Ejemplo de comentario no actualizado

9.2.2 Asignación de valor semántico a los componentes funcionales

Si bien con los métodos anteriores podemos confeccionar una lista de candidatos a componentes funcionales, la observación minuciosa del código nos permite valorar la importancia real de cada candidato recopilado para filtrar nuestra lista.

En la práctica, estas dos tareas (identificación y asignación de significado) no se realizan secuencialmente, sino más bien de forma paralela, una a la vez que la otra o una durante la otra. Aunque la idea que se cita a continuación es utilizada por [Premerlani *et al.* 1994] en el contexto de la Ingeniería Inversa de bases de datos, entendemos que también es plenamente aplicable en estas dos tareas: «Proponemos un proceso informal que requiere buen juicio. Nuestras etapas están débilmente ordenadas (en la práctica hay mucha iteración, vuelta atrás (*backtracking*) y reordenación de etapas). La presentación lineal es un gran artificio [...]». Realmente, para identificar un bloque de código como componente funcional debemos hacer un recorrido de sus sentencias que confirme su cualidad de componente funcional, y para recorrer un bloque de código de interés que despierta nuestro interés, previamente habremos debido identificarlo como componente funcional.

En [Piattini *et al.* 1996] se citan dos tipos de análisis del código fuente:

- *Análisis estático*: se realiza una comprobación del código sin ejecutarlo. Podemos construir con ello diagramas de flujo de muy alto nivel que ayudarán a la comprensión y a la lógica del problema, así como a identificar los componentes funcionales.
- *Análisis dinámico*: ejecutando el programa detectaremos aquí errores de la lógica del programa al encontrar resultados diferentes de los esperados.

Con ambos tipos de análisis encontraremos bucles infinitos, código inalcanzable o innecesario (código muerto) y otro tipo de defectos de los cuales, aunque no sea nuestra labor corregirlos en esta fase, sí que deberemos tomar buena nota.

9.3 RECONSTRUCCIÓN DE PROGRAMAS

En la reconstrucción de programas partimos de los productos elaborados en la fase de Ingeniería Inversa para, aplicando técnicas de Ingeniería directa, reconstruir el programa objeto del estudio.

En la Ingeniería Inversa habremos conseguido una Recuperación del diseño del programa en la que, según [Piattini *et al.*, 1996], «se recrean abstracciones de diseño partiendo de una combinación del código, documentación, experiencia personal y conocimientos generales sobre los dominios del problema y la aplicación».

9.3.1 Reestructuración

La reestructuración es la transformación de un producto software a otra forma de representación, pero sin cambiar de nivel de abstracción. Actualmente, muchas organizaciones se ven obligadas a modificar sus aplicaciones para aprovechar las oportunidades que permiten tecnologías emergentes como Internet, la arquitectura cliente/servidor o la computación distribuida [Jahnke y Wadsack, 1999]. Antes de realizar los cambios, las aplicaciones suelen ser sometidas previamente a un proceso de reestructuración, de manera que se genere un producto software equivalente al original, en el mismo nivel de abstracción, pero con mayor comprensibilidad y mantenibilidad, y menos propenso a la introducción de errores. Estas características de la nueva representación del sistema facilitan y abaratan los costes ulteriores de la modificación.

A continuación, presentamos un ejemplo de reestructuración de C++. [Fanta y Rajlich 1998] presentan un conjunto de herramientas para reestructurar código C++ de una aplicación CAD desarrollada en C++ con 200.000 líneas de código, 80 clases y 50 funciones globales. Las herramientas desarrolladas permitían:

- **Inserción de funciones**

  La inserción de funciones consiste en insertar una función global dentro de una clase y convertirla en un método público.

  Los pasos necesarios son:

  - Insertar la cabecera en la declaración de la clase.
  - Actualizar cuerpo de la función (acceso directo a los miembros de la clase).
  - Actualizar la cabecera de la función (preceder con el identificador de clase y eliminar el parámetro).
  - Actualizar llamadas a la función (añadir instancias de clase sobre las que ejecutar las llamadas).
  - Eliminar declaraciones de la función.

  La parte izquierda de la Figura 9.5 muestra dos fragmentos de código: uno de ellos es una clase, mientras que el otro es una función global que toma como parámetro un objeto de esa clase. Dicha función puede reconvertirse en el método *f* que se muestra a la derecha.

Encapsulación de código en funciones

En este caso, el usuario selecciona un bloque de código que considera susceptible de ser reconvertido en una función y una llamada a función. La herramienta debe decidir si el bloque seleccionado es sintácticamente completo. Ésta, en caso afirmativo, coloca el bloque de código en una nueva función y sustituye la aparición original del bloque por una llamada a la función. Posteriormente, la función puede pasar a ser miembro de una clase.

La sustitución del bloque por la función y llamada equivalentes requiere la ejecución de diferentes acciones respecto a las variables utilizadas en el bloque y en sus aledaños:

- Se genera una variable local por cada variable del bloque que no posea información para el resto del bloque.
- Las variables globales permanecen como globales.
- Se pasan por valor las variables locales que se usan posteriormente sin haber cambiado su valor.
- El resto, se pasan por referencia.

Class A	Class A

`{`
`  public:`
`  int i;`
`  protected:`
`  char c;`
`};`
`...`
`int f(A& a)`
`{`
`  a.i=4;`
`  ...`
`}`
	`{`
	`  public:`
	`  int i;`
	`  int f();`
	`  protected:`
	`  char c;`
	`};`
	`...`
	`int A :: f() {`
	`  this->i=4;`
	`  ...`
	`};`

Figura 9.5. Inserción de una función dentro de una clase

En la Figura 9.6 hemos enmarcado el bloque de código que el usuario desea encapsular en una función. Como se puede observar a la derecha, la herramienta ha considerado que el bloque es sintácticamente completo y lo sustituye por una llamada a la función *newfun*.

Expulsión de funciones

Consiste en expulsar un método de la clase de la que es miembro, de manera que se reconvierte en una función aislada. La clase será pasada a la función como parámetro.

Los pasos que deben seguirse son:

- Quitar la cabecera de la función de la declaración de clase.
- Generar, en caso necesario, métodos públicos para acceder a los miembros privados de la clase a los que accedía la función.
- Actualizar la cabecera de la función.
- Actualizar el cuerpo de la función.
- Actualizar las llamadas.
- Incluir la declaración temprana de la función en el fichero de cabecera correspondiente.

```
void f(char c)
{
   int i, count, len;
   char str[max;
   cin >> str;
   len=strlen(str);
      count=0;
      for (i=0; i<=len; i++)
            if (str[i]==c) {
              count++;
          str[i]='\n';
            }
  cout << str << count << «\n»;
}
```

```
void f(char c)
{
  int i, count, len;
  char str[max;
  cin >> str;
  len=strlen(str);
  newfun(count, len, str, c);
  cout << str << count << «\n»;
}
newfun(int &count, int len,
char *str, char c)
{
  int i;
  count=0;
  for (i=0; i<=len; i++)
    if (str[i]==c) {
            count++;
            str[i]='\n';
    }
}
```

Figura 9.6. Encapsulación de un bloque de código en una función

En general todos estos tipos de operaciones se conoce como **refactorización** de código fuente, el cual puede ser definido como todo cambio realizado en la

estructura interna del código fuente para hacerlo más entendible y más mantenible sin alterar su comportamiento externo [Fowler y Beck; 1999]. La refactorización de código fuente se ha aplicado extensamente a lenguajes de programación orientados a objetos donde se ha comprobado la mejora de diferentes características de calidad [Dallal y Abdin, 2017], aunque no todas las refactorizaciones se apliquen a nivel de código, existiendo la posibilidad de realizar estas mejoras a nivel de modelo [Cunha, 2016]

No obstante, existen varias barreras por parte de los equipos de desarrollo y mantenimiento que previenen la aplicación de los diferentes operadores de refactorización de código fuente [Tempero et al. 2017]:

- Recursos: muchas veces los equipos de mantenimiento reportar que no hay tiempo para refactorizar el código fuente.
- Riesgos: se percibe como una tarea que puede introducir errores y fallos adicionales como consecuencia del cambio
- Dificultad: se considera una tarea compleja, cuando en realidad la mayoría de los operadores de refactorización están automatizados.
- Retorno de la inversión (ROI): Los beneficios de la refactorización no están claros frente a los costes de refactorizar, reejecutar las pruebas, etc.
- Restricciones técnicas: gran variedad de problemas técnicos, dependencias, etc. Reportadas por el equipo de mantenimiento.

9.4 INGENIERÍA INVERSA Y REINGENIERÍA DE BASES DE DATOS

La aplicación de un proceso de Ingeniería Inversa a una base de datos surgió en su momento por la necesidad o el deseo de migrar desde modelos de datos antiguos (jerárquico, red) al modelo relacional; posteriormente al cambiar de un sistema gestor a otro, posiblemente los dos relacionales; también por la migración de un sistema relacional a otro orientado al objeto, e incluso más recientemente de bases de datos relacionales a bases de datos NoSQL.

Como sabemos, el diseño de bases de datos tradicional puede describirse como una secuencia de los siguientes tres procesos:

1. **Diseño conceptual**, mediante el cual se obtiene el llamado «esquema conceptual», que es una forma de representar la información independiente del sistema sobre el que se vaya a almacenar.

2. **Diseño lógico,** en el que se transforma el esquema conceptual en un esquema lógico optimizado que ahora sí depende del sistema real y que tiene las siguientes características:
   - Es equivalente al esquema conceptual: es decir, representa el mismo universo del discurso.
   - Depende del sistema real de almacenamiento de la información, pues sigue su modelo de datos.
   - Es eficiente en cuanto a espacio de almacenamiento y tiempos de respuesta.
3. **Diseño físico,** en el que se traducen las estructuras estáticas obtenidas en el diseño lógico a código del lenguaje de definición de datos del sistema real, así como las dinámicas a secciones de procedimientos y funciones que las implementen.

Al recuperar el diseño de una base de datos relacional, intentaremos obtener, a partir de su diseño en un soporte físico, los esquemas lógico y conceptual del universo del discurso que están representando, tal y como mostramos en la Figura 9.7:

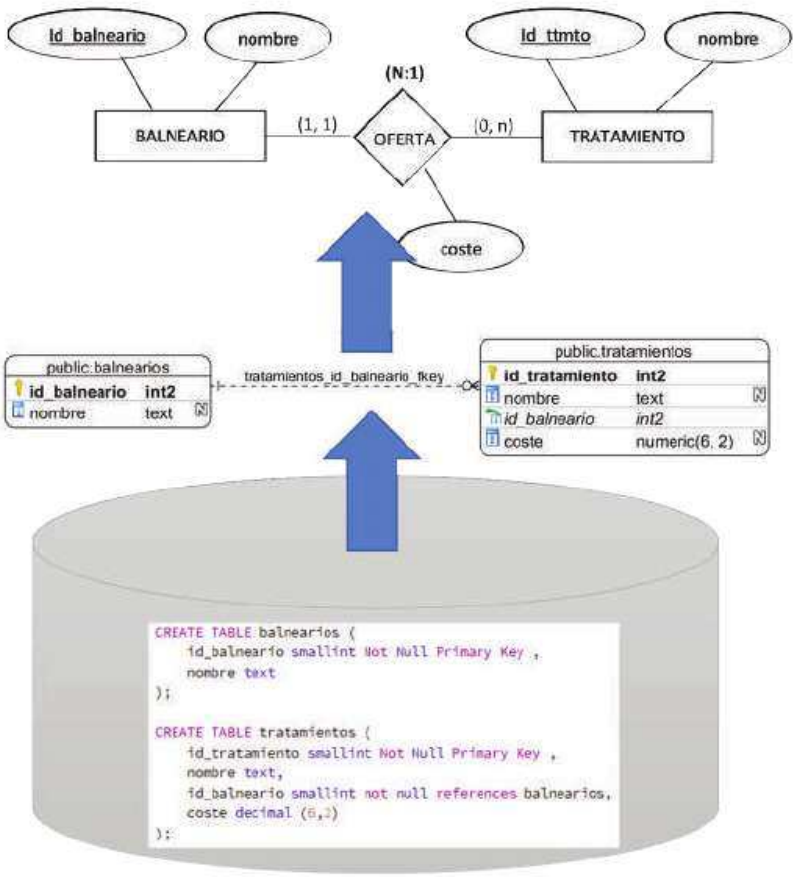


Figura 9.7. Fases en la ingeniería inversa de una BD relacional

[Pedro de Jesús y Sousa, 1999] presentan una interesante comparación entre diferentes métodos aplicables a la Ingeniería Inversa de bases de datos relacionales. En la Tabla 9.1 mostramos resumidamente los diferentes métodos analizados, las condiciones necesarias para su aplicación, las entradas que toman y el tipo de resultado que producen.

Evidentemente, en una misma base de datos relacional podemos encontrarnos tablas que se adecuen más a un método que a otro. En este sentido, [Sousa *et al.*, 1999] proponen un algoritmo para clasificar las tablas de una misma base de datos en diferentes conjuntos, de manera que pueda aplicarse a cada uno de éstos el método de Ingeniería inversa más apropiado.

El algoritmo consta de las tres siguientes fases:

Fase 1. Agrupación de tablas

1. Identificación de claves primarias

   Determinar los atributos que forman parte de la clave primaria (PK) de cada tabla y resolver los posibles conflictos potenciales de nombre que existan entre ellos.

2. Agrupar las tablas en entidades abstractas e interrelaciones

   Las tablas se agrupan según los atributos comunes de sus claves primarias, según el siguiente método:

   - Seleccionar las tablas cuyas PK: (1) no contienen la PK de otra tabla; (2) son disjuntas o iguales entre sí. Bajo estas dos condiciones, y en el caso de que tuviéramos el conjunto de tablas mostrado en la parte izquierda de la Figura 9.8 (en donde R_i representa una tabla y K_j un atributo de su clave primaria), podríamos realizar uno de los dos agrupamientos de tablas mostrados a la derecha. Nótese que, por ejemplo, R_5 no podría ser incluida en ninguno de los dos grupos *a* y *b* porque incumple la primera condición, ya que su clave primaria contiene la clave primaria de la tabla R_4.

Método	Entradas	Salidas	Precondiciones
Chiang *et al.*	Datos Relaciones Claves primarias	EER	3FN Consistencia de nombres Ausencia de errores en PK
Johannesson	Relaciones Dependencias funcionales Dependencias de inclusión	Par (L, IC)	3FN
Markowitz y Makowsk	Relaciones Dependencias clave Restricciones de integridad referencial	EER	Relaciones en FN de Boyce-Codd
Navathe y Awong	Relaciones	EER	Relaciones en 3FN o FN de Boyce-Codd Consistencia en los nombres de los atributos Ausencia de ambigüedades u homónimos en FK Especificación de todas las claves candidatas
Petit *et al.*	Relaciones con restric. de unicidad no nulas Datos Código	EER	Ninguna
Premerlani y Blaha	Relaciones Datos	Modelo de clases OMT	Ninguna
Signore *et al.*	Relaciones Código	ER	Ninguna

Tabla 9.1. Diferentes métodos para la ingeniería inversa de BB.DD. relacionales

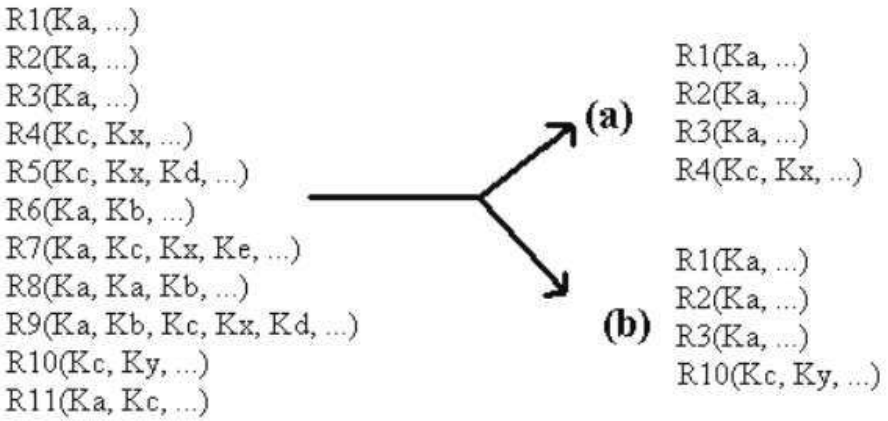


Figura 9.8. Selección de grupos de tablas según sus claves primarias

- Colocar las tablas con las mismas PK en el mismo grupo (entidad abstracta), de manera que no haya grupos que contengan tablas con PK disjuntas. En este paso, podemos seleccionar cualquiera de los dos agrupamientos mostrados en la Figura 9.8 y asignarlo a una «entidad abstracta». Si seleccionamos la opción *a*, tendremos en principio dos entidades abstractas, cada una compuesta por tablas con las mismas PK; por tanto, habrá dos entidades abstractas: $AE1=\{R_1, R_2, R_3\}$ y $AE2=\{R_4\}$.

- Añadir cada tabla restante a una entidad abstracta, si al menos un atributo de la PK pertenece a la entidad, y los atributos restantes de la PK no aparecen en ninguna otra entidad. En este paso debemos fijarnos en las tablas que no pertenecen a ninguna entidad abstracta, como por ejemplo R_5: los dos atributos K_C y K_X de la PK de esta tabla aparecen en la PK de las tablas que hemos colocado en AE2, y el atributo restante de su PK (K_d) no se encuentra incluido en la PK de la entidad abstracta AE1, por lo que podemos ubicar R_5 en AE2. Repitiendo esta operación con las tablas restantes, tendremos que:

  $$AE1=\{R_1, R_2, R_3, R_6, R_8\}\ ;\ AE2=\{R_4, R_5, R_{10}\}$$

  Nótese que las tablas R_7, R_9 y R_{11} han quedado sin asignar a ninguna de las dos entidades abstractas identificadas hasta el momento. Fijándonos en R_7, encontramos la justificación en que el atributo K_a forma parte de la clave primaria de *AE1* y los atributos K_c y K_X pertenecen a la clave primaria de *AE2*, por lo que, de acuerdo al algoritmo, no debe ser incluida dentro de ninguna entidad abstracta.

  – Crear un nuevo grupo de tablas cuyos atributos de la PK pertenezcan a las mismas entidades abstractas. Estos grupos se llaman interrelaciones abstractas. En el ejemplo, crearíamos la interrelación abstracta $AR1=\{R_7, R_9, R_{11}\}$ porque los atributos que aparecen en *AE1* y *AE2* no aparecen en ninguna otra entidad abstracta. A partir de aquí, tenemos el diagrama entidad-interrelación de elementos abstractos mostrado en la Figura 9.9.

Fase 2. Refinamiento de elementos abstractos

La Ingeniería inversa de las tablas de una entidad abstracta depende sólo de esa entidad, y a cada entidad abstracta se le puede aplicar un proceso de Ingeniería inversa diferente.

Por tanto:

... para detectar...	*... puede usar la técnica de...*
Generalizaciones	Petit *et al.*
Entidades fuertes	Chiang *et al.*
Entidades débiles	Chiang *et al.*
Interrelaciones	Christian
Agregaciones	Chiang *et al.*

Tabla 9.2. Métodos propuestos

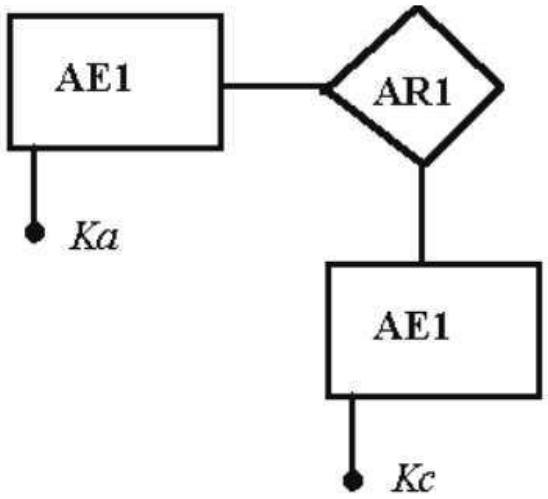


Figura 9.9. Diagrama de elementos abstractos

Fase 3. Obtención del esquema final

Se integran aquí los diferentes esquemas conceptuales intermedios en un esquema final, que se completa con lo que pudiera faltar.

El proceso completo puede representarse como la secuencia mostrada en la Figura 9.10: se crea el *esquema de elementos abstractos* de acuerdo con el método presentado por [Sousa *et al.*, 1999]; a continuación, se aplica a cada elemento abstracto el método de ingeniería inversa más adecuado, obteniéndose los *esquemas conceptuales intermedios*, que son integrados en un único *esquema conceptual final*, que representa al *sistema físico* original.

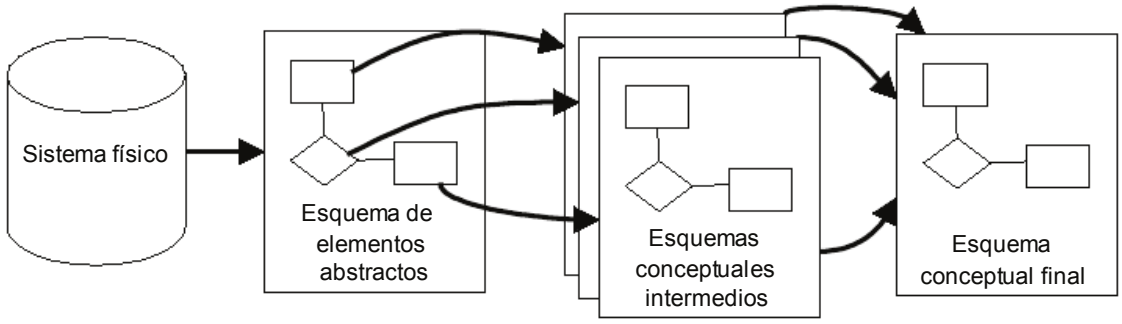


Figura 9.10. Vista general del proceso

Sin embargo, en la actualidad es posible encontrar enfoques sobre reingeniería de bases de datos que abordar las nuevas tendencias, como por ejemplo la migración a bases de datos "NoSQL", y especialmente las que soportan el *Big Data* [Zhu et al, 2016]. También es posible observar cómo paradigmas como el MDE son de utilidad a la hora de plantear estrategias de reingeniería de bases de datos [Bermudez et al., 2017] .

9.5 INGENIERÍA INVERSA Y REINGENIERÍA DE INTERFACES DE USUARIO

En muchas ocasiones, los productos software han sido construidos para proporcionar gran fiabilidad y altos rendimientos a los procesos que realizan, dedicando a estas tareas grandes cantidades de recursos y esfuerzos durante su desarrollo que han podido dejar en el olvido al usuario final y que han ido en detrimento de la facilidad de uso del programa. En estas situaciones, los equipos de mantenimiento deben dedicarse a adaptar aplicaciones en explotación a las necesidades y preferencias de los usuarios, respetando su lógica interior.

Es en estos casos cuando hablamos de reingeniería de las interfaces de usuario, con la que pueden conseguirse avances importantes en poco tiempo. Antes de comenzar el proceso inverso, el equipo de trabajo debe conocer perfectamente tanto el producto como los defectos del sistema existente y las preferencias y expectativas de los usuarios. [Plaisant *et al.,* 1997] especifican las siguientes tareas:

1ª) Recopilación y estudio de toda la documentación disponible

Por documentación entendemos tanto las especificaciones del sistema y del diseño como los manuales del usuario, ayuda en línea, etc.

2ª) Entrevistas a los diferentes grupos de usuarios y observación de sus métodos de trabajo

- Entrevistas a la dirección para identificar los objetivos, recursos y plazos.
- Entrevista con los diseñadores y jefes de mantenimiento para asignar recursos técnicos, concretar los humanos e identificar los puntos débiles del sistema, en especial cuando se haya subcontratado el mantenimiento.
- Entrevista con los usuarios finales, para conocer los aciertos y errores del sistema, la frecuencia de cada tipo de operación, sus formas de uso, etc.

3ª) Uso del sistema por el propio equipo de mantenimiento

Así se consigue un conocimiento profundo del flujo de procesos y datos del sistema. En esta etapa tomaremos nota de gran cantidad de detalles que habían pasado inadvertidos en las fases previas.

Del mismo modo, podemos hacer pequeñas alteraciones en el código fuente del sistema original para añadir, por ejemplo, mecanismos de auditoría que nos ayuden a conocer el número de accesos a cada pantalla u operación.

En el proceso de reingeniería de interfaces de usuario, como en cualquier proceso de Ingeniería del Software, se debe construir la documentación adecuada para el equipo de mantenimiento. Pero, además, cuando la interfaz haya sido reconstruida, debe también reconstruirse la documentación para los usuarios (manuales, sistema de ayuda y tutorial en línea).

9.6 MODERNIZACIÓN DE SISTEMAS DE INFORMACIÓN

Durante las últimas dos décadas la reingeniería ha sido considerada el mecanismo principal para llevar a cabo el mantenimiento evolutivo de los sistemas heredados [Bianchi et al., 2003]. La reingeniería mantiene la información de negocio heredada haciendo que los cambios en el software se puedan realizar de forma más fácil, confiable y rápida, dando como resultado costes de mantenimientos asumibles [Bennett y Rajlich, 2000]. Sin embargo, un estudio de [Sneed, 2005] asegura que cerca de la mitad de los proyectos de reingeniería fracasan debido, principalmente, a dos problemas: (i) la reingeniería de sistemas de información grandes y complejos es muy difícil de automatizar [Canfora y Di Penta, 2007], y por tanto los costes crecen significativamente; y (ii) la reingeniería tradicional carece de cierto grado de formalización y estandarización [Kazman et al., 1998], y por tanto existen diferentes herramientas de reingeniería focalizadas en tareas y tecnologías específicas que no

pueden ser integradas o reutilizadas en diferentes proyectos de reingeniería. Por estas dos razones, la industria del software ha demandado procesos de reingeniería que faciliten el mantenimiento evolutivo de los sistemas de información heredados de forma estandarizada y automatizable a gran escala.

9.6.1 Modernización Dirigida por la Arquitectura (ADM)

Como solución a estas demandas, el concepto de reingeniería tradicional ha sido desplazado hacia la Modernización Dirigida por la Arquitectura (ADM, de sus siglas en inglés *Architecture-Driven Modernization*). ADM es el concepto de modernizar sistemas de información existentes centrándose en todos los aspectos de la arquitectura actual de un sistema y la habilidad para transformar su arquitectura actual en otras mejoradas [OMG, 2006]. ADM aboga por llevar a cabo procesos de reingeniería siguiendo el estándar MDA (*Model-Driven Architecture*) [Miller y Mukerji 2003]. Es decir, ADM hace posible modelar todos los artefactos software heredados como modelos y establece transformaciones de modelos entre los diferentes niveles de abstracción que define MDA. Estos principios del desarrollo dirigido por modelos ayudan a solucionar los problemas de estandarización y automatización inherentes a la reingeniería tradicional. En este sentido, ADM aborda el fenómeno de la erosión del software mediante la modernización de los sistemas de información heredados.

El objetivo de esta sección es dar una visión general de los conceptos y estándares relacionados con ADM y mostrar la forma en la cual, estos elementos pueden solucionar un problema tradicional de la industria del software: la modernización de los sistemas de información heredados en empresas y organizaciones.

El paradigma de la modernización del software, y concretamente ADM tal como lo define OMG, puede ser considerado como un mecanismo para llevar a cabo mantenimiento evolutivo. Es decir, ADM favorece la modernización de sistemas de información heredados erradicando, o al menos minimizando, los problemas derivados de la erosión y envejecimiento del software. De acuerdo a [OMG, 2003], ADM es el proceso de comprensión y evolución de los artefactos software heredados para restaurar el valor de los sistemas existentes.

La modernización software tiene por objetivo revitalizar los sistemas de información haciéndolos más ágiles frente a cambios en su entorno. La modernización del software aparece como una necesidad en varios escenarios y contextos bien definidos [OMG, 2007]:

- Mejora de las aplicaciones
- Conversión lenguaje-a-lenguaje
- Migración de plataforma
- Integración no intrusiva entre aplicaciones
- Transformación hacia la Arquitectura Orientada a Servicios (SOA)
- Migración de arquitecturas de datos

El incremento de los costes de mantenimiento junto con la necesidad de preservar el conocimiento de negocio de los LIS (Legacy Information System) ha convertido a la modernización del software en una importante área de investigación [Daga et al., 2005]. La modernización del software combate esos problemas ya que su uso permite conseguir los siguientes beneficios [OMG, 2007]:

- Aumento de la agilidad de negocio en las organizaciones mediante la creación de agilidad en sus sistemas de información.
- Mejora del retorno de la inversión (ROI, *Return On Investment*) relacionada con los LIS.
- Mejora en la productividad del desarrollo software.
- Reducción del esfuerzo y de los costes de mantenimiento
- Extensión de la vida útil de los sistemas de información existentes (LIS)

ADM soluciona los problemas de la reingeniería tradicional llevando a cabo los procesos de reingeniería teniendo en cuenta los principios del desarrollo dirigido por modelos. No obstante, ADM no reemplaza a la reingeniería sino que pretende ser una mejora de la misma.

El proceso genérico de reingeniería en herradura ha sido adaptado a ADM, conociéndose como modelo de modernización en herradura (véase Figura 9.11). En el proceso de modernización, de acuerdo a los principios del desarrollo dirigido por modelos [Miller y Mukerji, 2003], existen tres tipos de modelos:

- **Modelo Independiente de la Computación** (CIM, Computation Independent Model), que es una vista del sistema a alto nivel de abstracción desde un punto de vista independiente de la computación. Un modelo CIM no muestra detalles la estructura del sistema. Estos modelos a veces son llamados también modelos de dominio y juegan el rol de cerrar la brecha conceptual entre los expertos de dominio y los expertos del sistema que conocen su diseño y construcción.

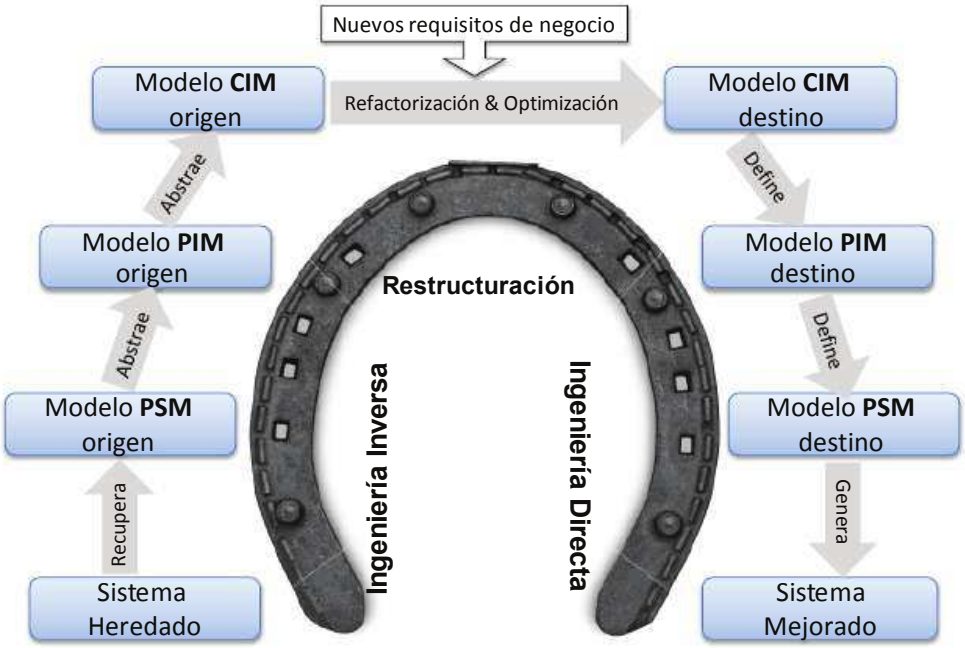


Figura 9.11. Proceso de modernización en herradura

- **Modelo Independiente de la Plataforma** (PIM, Platform Independent Model), que es una vista del sistema desde un punto de vista independiente de la plataforma a un nivel intermedio de abstracción. Un modelo tipo PIM tiene cierto grado de independencia tecnológica para poder ser reutilizado con diferentes plataformas tecnológicas de un tipo similar.
- **Modelo Específico de la Plataforma** (PSM, Platform Specific Model), que representa el sistema a bajo nivel de abstracción desde un punto de vista dependiente de la plataforma tecnológica. Un modelo PSM combina la especificación del modelo PIM junto con detalles específicos de la propia implementación del sistema.

ADM es una iniciativa que permite abordar el problema de formalización gracias a que aboga por la representación como modelos de todos los artefactos software involucrados. Esto significa que ADM también establece el marco para el desarrollo de transformaciones entre los modelos. Dichas transformaciones son formalizadas mediante el estándar QVT (*Query / Views / Transformations*) propuesto por el OMG [OMG, 2008]. Además, los principios del desarrollo dirigido por modelos facilitan la reutilización de los modelos involucrados en proyectos de modernización ADM. Por ejemplo, se podrían generar varios modelos PIM a partir de un único modelo CIM para cubrir diferentes plataformas. A su vez, cada

modelo PIM podría utilizarse para obtener diferentes implementaciones a través de los respectivos modelos PSM. Por consiguiente, el problema de automatización se puede solucionar también debido a la automatización de las transformaciones entre modelos junto con la reutilización de dichos modelos.

Normalmente, los principios de desarrollo dirigido por modelos se aplican a las fases de ingeniería directa, para obtener automáticamente código fuente desde varios tipos de modelos, por ejemplo, en UML. Sin embargo, los principios del desarrollo dirigido por modelos pueden ser aplicados en la realidad también a las fases de ingeniería inversa y de reestructuración.

En la fase de ingeniería inversa, la información recuperada desde los artefactos software puede ser usada para representar modelos de acuerdo a metamodelos específicos. Además, técnicas tradicionales de reestructuración y refactorización pueden ser adaptadas y aplicadas sobre esos modelos. La reestructuración basada en modelos tiene varias ventajas con respecto a la reestructuración tradicional: (i) permite desarrollar técnicas de refactorización independientes de la plataforma y la tecnología; (ii) una transformación de reestructuración podría ser implementada como un modelo en sí misma, así que la transformación podría ser también reutilizada; (iii) la refactorización sobre modelos hace posible definir de forma fácil técnicas de refactorización generales o independientes del dominio de aplicación; y finalmente (iv) mejora la localización de características en el código fuente, ya que se consigue una mejor trazabilidad a lo largo de modelos que se organizan en diferentes niveles de abstracción.

La modernización del software mediante ADM puede ser vista como un mantenimiento de tipo evolutivo, basado en el proceso de reingeniería y siguiendo un enfoque dirigido por modelos. Dado que ADM aboga por llevar a cabo procesos dirigidos por modelos, su automatización y reutilización (basada en modelos abstractos) están aseguradas. Así que los productos de salida de un proceso basado en ADM no son solamente la mejora o evolución de un sistema de información heredado, sino también un conjunto de modelos, representando el sistema a diferentes niveles de abstracción, que podrán ser reutilizados en futuros procesos de mantenimiento. Por lo tanto, el problema de la entropía del software se minimiza con la evolución del software por medio de ADM.

Según el modelo de modernización en herradura, un proceso basado en ADM puede ser clasificado en tres tipos de nivel de actuación [Khusidman y Ulrich, 2007]. Estos tipos dependen del nivel de abstracción alcanzado en la fase de ingeniería inversa en el modelo en herradura y, por lo tanto, cada tipo de proceso de modernización define una curva particular de modernización del software (véase Figura 9.12). Dependiendo del nivel de abstracción alcanzado en cada curva, la información y modelos disponibles para reestructurar el sistema de información

heredado podrían ser muy diferentes. Normalmente, un nivel de abstracción mayor genera mayores posibilidades de reestructuración durante la modernización del software. Los tres tipos de modernización son los siguientes:

- **Modernización Tecnológica.** Este tipo de modernización considera el nivel de abstracción más bajo y ha sido históricamente el más aplicado a sistemas de información heredados. Una empresa lleva a cabo una modernización tecnológica cuando ésta aborda problemas derivados de la obsolescencia del lenguaje o plataforma de sus sistemas, para conformar nuevos estándares o normas, para mejorar su eficiencia, usabilidad o factores similares. Este primer tipo de proceso basado en ADM a veces no se considera estrictamente un proceso de modernización del software porque sólo se centra en modificaciones correctivas y preventivas.

- **Modernización de Aplicaciones o Datos.** Este tipo de modernización considera un nivel de abstracción intermedio, ya que se centra en la reestructuración al nivel del diseño de las aplicaciones y datos. Este tipo de modernización puede estar motivado por factores como la mejora de la reutilización, la reducción de lógica de programa deslocalizada o la complejidad del sistema, la aplicación de patrones de diseño, etc. Existe una delgada línea de separación entre este tipo de modernización y el anterior, que se considera cruzada cuando la modernización conlleva un impacto sobre el diseño del sistema.

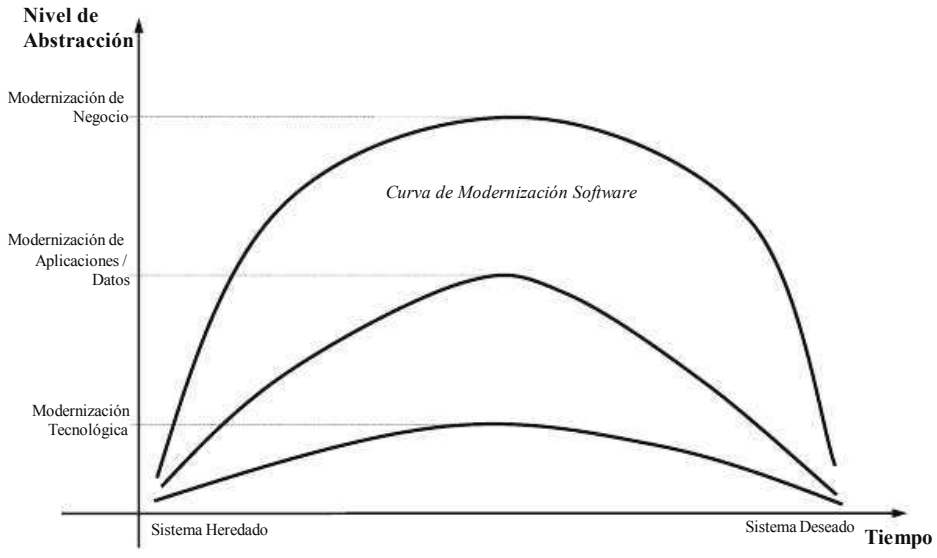


Figura 9.12. Curvas de modernización del software según nivel de abstracción

- **Modernización de Negocio.** Este tipo de modernización incrementa el nivel de abstracción hasta el máximo, ya que la fase de reestructuración es realizada sobre la arquitectura de negocio, es decir, las reglas y procesos de negocio que gobiernan el sistema de información heredado. Este tipo de modernización incorpora, además de los modelos tecnológicos y de aplicaciones/datos, modelos con la semántica del negocio, que son claves para: (i) preservar la información de negocio embebida en el sistema de información heredado, y (ii) alinear los requisitos de negocio de la empresa con los futuros sistemas de información modernizados.

La modernización de negocio es probablemente el tipo más importante de modernización ya que los cambios en los procesos de negocio son uno de los criterios de decisión más recurrentes para llevar a cabo la modernización de sistemas de información heredados [Koskinen et al., 2005]. Sin embargo, pocos intentos de modernización han logrado, hasta el momento, alcanzar el nivel de abstracción de negocio a lo largo de la fase de ingeniería inversa. Esto puede ser debido a que los paradigmas de correspondencia entre el negocio y las tecnologías de la información carecen de estandarización para modernizar los sistemas de información heredados. Por esta razón, los esfuerzos de estandarización de la iniciativa ADM han sido tan importantes. Como resultado, los estándares ADM permiten abordar actualmente este tipo de procesos de modernización.

9.6.2 Estándares ADM

ADM no solo adopta estándares existentes como MDA o QVT, sino que además ha impulsado el desarrollo de un conjunto de estándares relacionados para abordar los diferentes desafíos que aparecen durante la modernización de sistemas de información heredados [OMG, 2009a]:

- *Knowledge Discovery Metamodel* (KDM) [OMG, 2016], presenta un metamodelo para el descubrimiento de conocimiento y es el primer estándar completo de la iniciativa ADM por lo que se convierte en la piedra angular para el resto de estándares propuestos.

- *Abstract Syntax Tree Metamodel* (ASTM) [OMG, 2011], es una especificación construida sobre KDM para la representación de software a nivel procedural por medios de árboles sintácticos abstractos. ASTM complementa a KDM para obtener representaciones abstractas de código y otros artefactos software y facilitar el intercambio de estos modelos abstractos.

- *Structures Metrics Metamodel* (SMM) [OMG, 2016b], que define un metamodelo con el cual representar la información de medidas relacionadas con el software concernientes a artefactos software legados.
- *ADM Pattern Recognition specification* es un estándar en desarrollo pensado para facilitar la inspección de metadatos y definición de patrones en los sitemas heredados.
- *ADM Visualization specification*, es un estándar en desarrollo centrado en la visualización de modelos.
- *ADM Refactoring specification*, especifica mecanismos para refactorización de modelos KDM.
- *ADM Transformation specification*, este estándar en desarrollo define mapeos y transformaciones entre modelos KDM y ASTM.

La Figura 9.13. muestra el conjunto de estándares ADM contextualizados en el modelo de modernización en herradura junto con otros estándares OMG relacionados en el proceso de modernización.

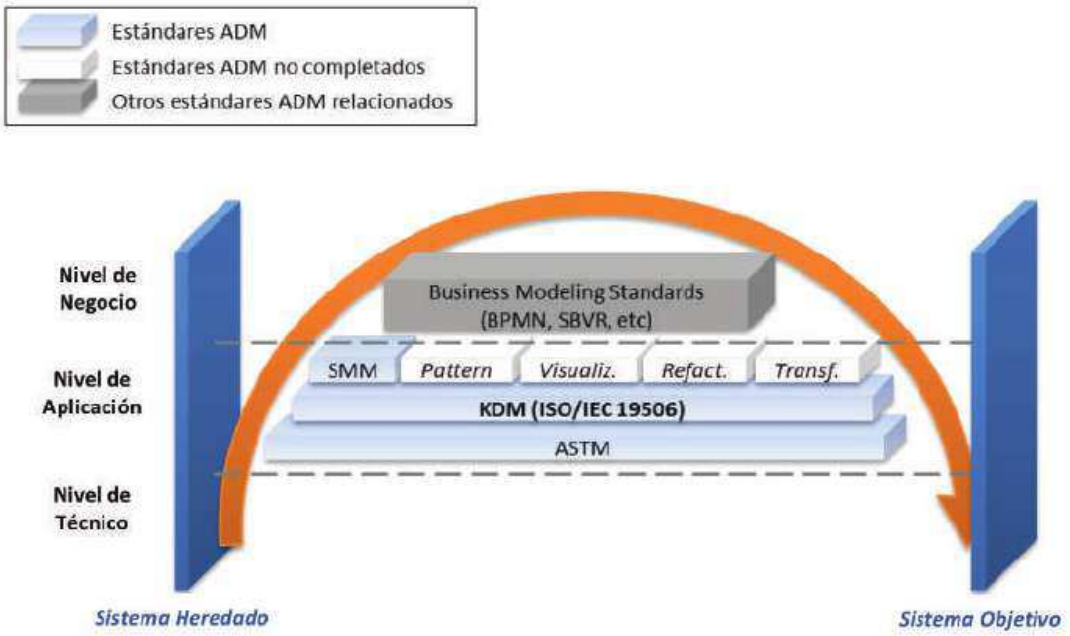


Figura 9.13. Proceso de modernización en herradura

9.6.2.1 EL ECOSISTEMA KDM

Los sistemas de información heredados cada vez son más complejos por lo que su desarrollo y posterior manejo requiere un esfuerzo importante. Los proyectos de modernización del software basados en ADM implican, quizás, un esfuerzo mayor para modernizar sistemas de información heredados o simplemente para entender la complejidad de los mismos.

El estándar KDM está cambiando la forma en la que se construyen y utilizan las herramientas de ingeniería inversa. Las herramientas de ingeniería inversa tradicionales han sido construidas durante años en forma de silo (véase Figura 9.14, izquierda). Por contra, el estándar KDM hace posible la construcción de herramientas de ingeniería inversa en lo que se conoce como el ecosistema KDM (véase Figura 9.14 derecha). En este ecosistema las herramientas de ingeniería inversa recuperarían diferente información de diversos artefactos, aunque esta información se representaría homogéneamente en un repositorio común de acuerdo al metamodelo KDM. Además, futuras herramientas de análisis se podrían enlazar en el repositorio KDM y generar nueva y valiosa información para la modernización de sistemas de información heredados.

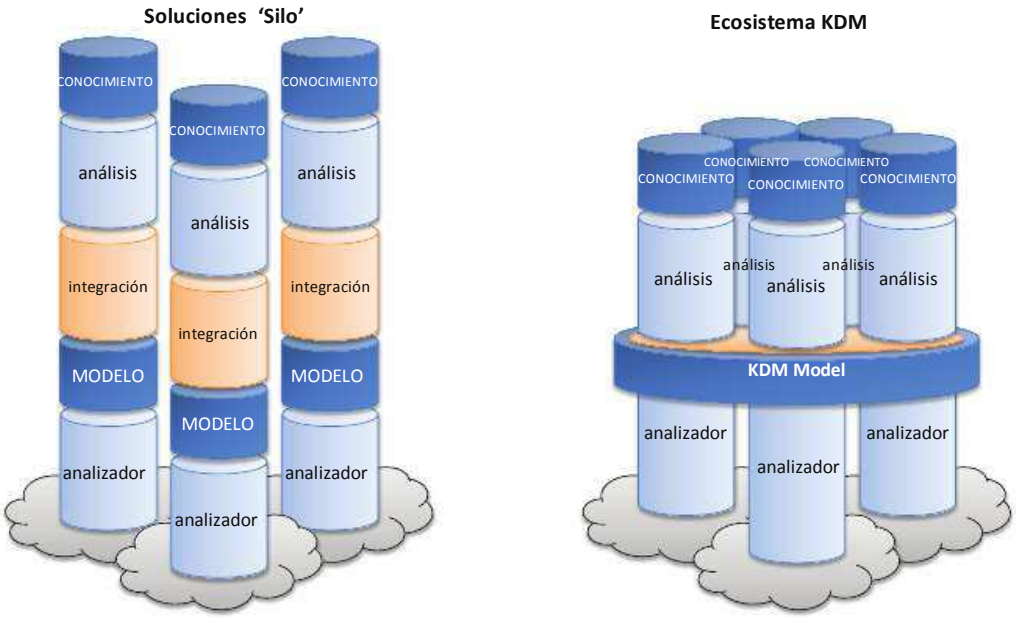


Figura 9.14. Herramientas de ingeniería inversa: soluciones en forma de silo (izquierda) frente a un ecosistema KDM (derecha).

9.6.2.2 EL METAMODELO KDM

El objetivo de la especificación KDM [OMG, 2009a] es la representación de sistemas de información heredados como un todo y no sólo de su código fuente [Khusidman, 2008]. El metamodelo definido por el estándar KDM provee una vista comprensiva de alto nivel acerca del comportamiento, la estructura y datos de un sistema. El metamodelo KDM está dividido en varias capas o niveles de abstracción representando tanto los artefactos software lógicos como físicos. Las capas separan la información de los sistemas heredados en varios aspectos ortogonales que son conocidos en Ingeniería del Software como vistas de arquitectura (véase Figura 9.15).

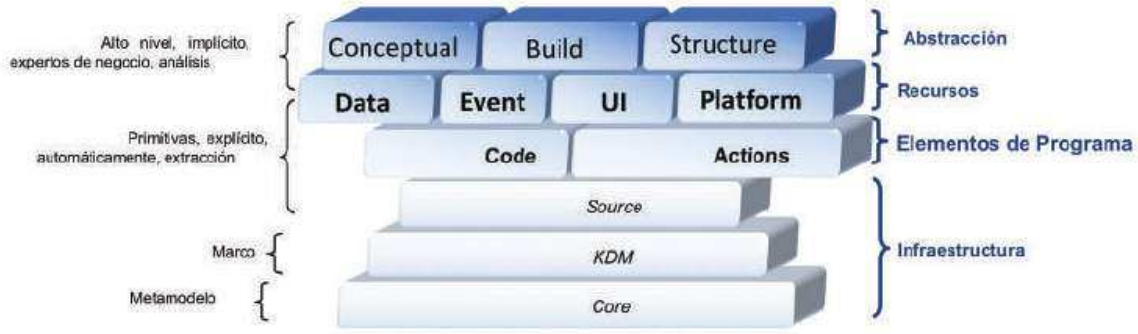


Figura 9.15. Capas, paquetes y aspectos en el metamodelo KDM (adaptada de [OMG, 2016]).

Concretamente, el metamodelo KDM dispone de cuatro capas de abstracción que se basan cada una en su capa anterior. Además, cada capa se organiza en diferentes paquetes que definen un subconjunto de elementos del metamodelo cuyo propósito es representar una faceta independiente de los sistemas de información heredados.

- **Capa de infraestructura** define un pequeño conjunto de conceptos usados de forma sistemática a lo largo del resto de la especificación KDM. Esta capa contiene tres paquetes: *Core*, *KDM* y *Source*. Los paquetes *Core* y *KDM* definen los elementos comunes del metamodelo que dan soporte para el resto de paquetes del metamodelo. El paquete *Source* enumera los artefactos de un sistema de información heredado y define mecanismos de trazabilidad entre los elementos representados usando KDM y su representación original en el sistema de información heredado.

- **Capa de elementos de programa** provee una representación homogénea independiente del lenguaje para varios elementos comunes a varios lenguajes de programación. Esta capa contiene dos paquetes: *Code* y *Action*. El paquete *Code* representa los elementos estructurales del código fuente y varias relaciones entre ellos, mientras que el paquete *Action* se centra en describir el comportamiento y flujo de control y datos entre los elementos del paquete *Code*.

- **Capa de recursos de ejecución** facilita la representación de información de alto nivel a cerca del sistema de información heredado y su entorno de ejecución, es decir, esta capa se centra en todos aquellos elementos que no están estrictamente contenidos en el código fuente. Esta capa contiene cuatro paquetes: *Data*, *Event*, *UI* y *Platform*. El paquete *Data* define aspectos de los datos persistentes (por ejemplo, ficheros de datos y bases de datos) usados por el sistema de información heredado. El paquete *Event* representa el modelo de eventos y acciones que subyace durante la ejecución del sistema. El paquete *UI* representa aspectos relacionados con las interfaces de usuario. Finalmente, el paquete *Platform* se usa para definir los artefactos relacionados con la plataforma y entorno de ejecución del sistema de información heredado.

- **Capa de abstracción** define un conjunto de elementos del metamodelo cuyo propósito es representar información específica del dominio del sistema así como proveer una perspectiva de negocio. La capa de abstracción cuenta con tres paquetes: *Conceptual*, *Structure* y *Build*. El paquete *Conceptual* representa los elementos específicos de dominio. El paquete *Structure* representa los componentes de alto nivel de la arquitectura del sistema de información heredado (por ejemplo, subsistemas, capas o paquetes). El paquete *Build* define elementos relacionados con la propia construcción del programa como roles, actividades de desarrollo, entregables, etc.

9.6.3 Ejemplo de Modernización de Software

Esta sección presenta un caso de estudio que aplica un proceso basado en ADM para modernizar una base de datos heredada hacía un entorno SOA [Perez-Castillo et al, 2009].

En el proceso de modernización se definieron como objetivos el descubrimiento de servicios web a partir de la base de datos relacional existente y su generación automática mediante un conjunto de transformaciones entre modelos. Siguiendo el modelo de modernización de software en herradura (véase Figura 9.11):

- Primero, se obtiene un modelo PSM a partir de la base de datos heredada, de acuerdo al metamodelo del estándar SQL-92 [ISO/IEC 1992].
- Segundo, el modelo PSM es transformado en un modelo PIM abstrayendo el sistema mediante un modelo de objetos. Este modelo PIM es representado en términos del metamodelo del estándar UML2 [OMG, 2007].
- Tercero, la fase de reestructuración puede modificar el modelo de objetos con el fin de añadir nuevos requisitos de negocio.
- Cuarto, el proceso genera mediante ingeniería inversa un modelo PSM concreto que describe los servicios web a un nivel de abstracción.
- Finalmente, el código fuente de los servicios web es generado desde el modelo PSM, así que el nivel de abstracción es reducido nuevamente.

El proceso de modernización propuesto se explica en detalle en las siguientes subsecciones. Adicionalmente, un pequeño ejemplo considerando sólo un fragmento de la base de datos heredada se presenta a continuación (ver Figura 9.16). El fragmento considerado consta de cuatro tablas: *Proceedings*, *Papers*, *Authors* y *Author-Paper*. Las actas de congreso (*proceedings*) tienen varios artículos de investigación (*papers*), los cuales son escritos por varios investigadores (*authors*). Finalmente, la tabla *Author-Paper* relaciona cada artículo con los autores que lo escriben mediante dos claves ajenas a sendas tablas.

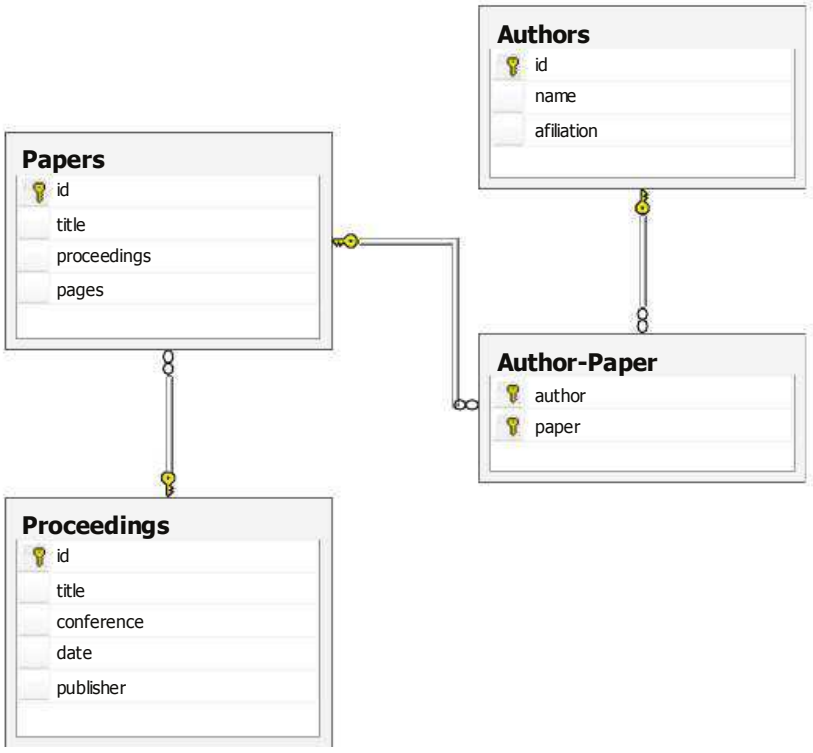


Figura 9.16. Porción del esquema de la base de datos modernizada en el caso de estudio.

9.6.3.1 FASE DE INGENIERÍA INVERSA

La primera fase del proyecto de modernización consistió en la ingeniería inversa de la base de datos heredada, con el objetivo de recuperar el diseño de una base de datos relacional y crear un modelo PSM que representa dicha base de datos. Los metadatos recuperados desde la base de datos fueron representados en un modelo PSM según el metamodelo del estándar SQL-92 (véase Figura 9.17).

Los metadatos necesarios para construir el modelo del esquema de la base de datos fueron extraídos a través del *INFORMATION_SCHEMA* [Melton y Simon, 1993]. Éste es un mecanismo estandarizado (desde la especificación SQL-92) que identifica un conjunto de vistas encargadas de mostrar información acerca del esquema de una base de datos de forma independiente al sistema gestor de base de datos en concreto.

Además, los modelos obtenidos se hicieron persistentes mediante el estándar XMI (*XML Metadata Interchange*) [Grose et al., 2001], tal como se muestra en la Figura 9.18. En este archivo XMI existe un elemento por cada tabla y restricción del esquema de la base de datos según el metamodelo propuesto. Hay cuatro instancias de la metaclase *BaseTable* para representar las cuatro tablas y también las restricciones

(tanto primarias como de integridad referencial) son modeladas respectivamente mediante instancias de las metaclases *PrimaryKey* y *ReferentialConstraint*.

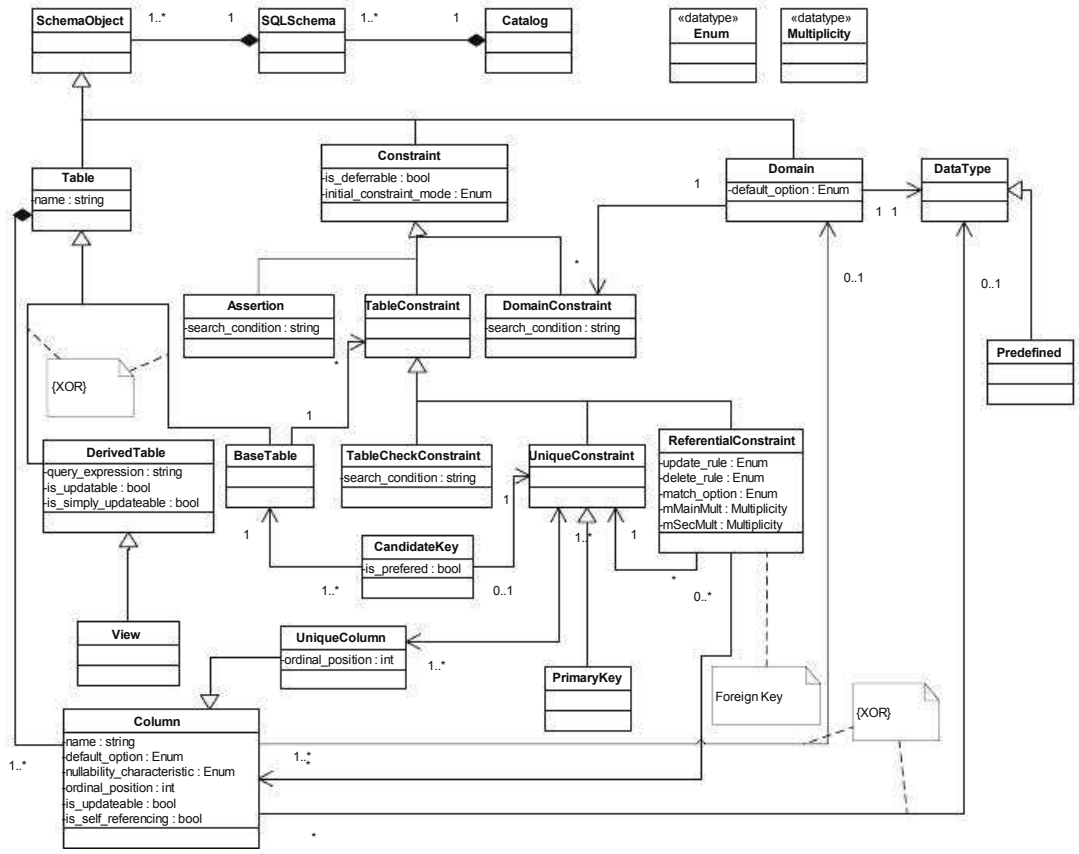


Figura 9.17. Metamodelo del estándar SQL-92 (basado en [Calero et al., 2006]).

Una vez que el diseño de la base de datos fue recuperado, el modelo PSM resultante se transforma en un modelo PIM que representa la información de la base de datos (el esquema de la base de datos) mediante un diagrama de clases. Por consiguiente, en este paso se creó un modelo de objetos que fue considerado en las fases siguientes como la base para generar los servicios Web. El modelo de objetos es representado de acuerdo al metamodelo de UML2 y la transformación desde el modelo PSM anterior fue implementada mediante QVT. La Figura 9.19 muestra una parte de la transformación, la cual presenta dos relaciones: la primera transforma cada esquema de base de datos del modelo PSM en un paquete de código en el modelo PIM, y la segunda relación genera una instancia de la metaclase *Class* por cada instancia de la metaclase *Table* del modelo PSM. Prácticamente, la relación QVT aplica a la inversa el patrón de diseño *'una clase-una tabla'* para modelar la persistencia de sistemas orientados a objetos [Gamma et al., 1995].

```
<?xml version="1.0" encoding="utf-8" ?>
- <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI">
  - <xmi:Documentation>
      <exporter>PRECISO</exporter>
      <exporterVersion>0.5</exporterVersion>
    </xmi:Documentation>
  - <Schema name="Conferences">
    - <SchemaObjects>
      - <Tables xmi:id="_0">
        + <BaseTable name="Papers" xmi:id="_1.0">
        - <BaseTable name="Proceedings" xmi:id="_1.1">
          - <Columns>
            + <Column name="id" xmi:id="_1.1.0" ordinal_position="1"
                is_nullable="NO">
            + <Column name="title" xmi:id="_1.1.1" ordinal_position="2"
                is_nullable="NO">
            + <Column name="conference" xmi:id="_1.1.2"
                ordinal_position="3" is_nullable="NO">
            + <Column name="date" xmi:id="_1.1.3" ordinal_position="4"
                is_nullable="NO">
            + <Column name="publisher" xmi:id="_1.1.4"
                ordinal_position="5" is_nullable="NO">
            </Columns>
          </BaseTable>
        + <BaseTable name="Authors" xmi:id="_1.2">
        + <BaseTable name="AuthorPaper" xmi:id="_1.3">
        + <BaseTable name="sysdiagrams" xmi:id="_1.4">
        </Tables>
      - <Constraints xmi:id="_1">
          <Assertions />
        - <TableConstraints>
          - <UniqueConstraints>
            + <UniqueConstraint name="UK_principal_name" xmi:id="_2.8">
            </UniqueConstraints>
          + <PrimaryKeyConstraints>
          - <ReferentialConstraints>
            + <ReferentialConstraint name="FK_Paper_Proceedings"
                xmi:id="_2.4" update_rule="NO ACTION" delete_rule="NO
                ACTION" match_option="SIMPLE"
                xmlns="FK_Paper_Proceedings">
            + <ReferentialConstraint name="FK_Author-Paper_Author"
                xmi:id="_2.5" update_rule="NO ACTION" delete_rule="NO
                ACTION" match_option="SIMPLE" xmlns="FK_Author-
                Paper_Author">
            + <ReferentialConstraint name="FK_Author-Paper_Paper"
                xmi:id="_2.6" update_rule="NO ACTION" delete_rule="NO
                ACTION" match_option="SIMPLE" xmlns="FK_Author-
                Paper_Paper">
            </ReferentialConstraints>
            <TableCheckConstraints />
          </TableConstraints>
          <DomainConstraints />
        </Constraints>
        <Domains />
      </SchemaObjects>
    </Schema>
</xmi:XMI>
```

Figura 9.18. Modelo de la base de datos del caso de estudio en XMI.

```
transformation rdbms2uml(rdbms:SimpleRDBMS, uml:SimpleUML) {
   -- map each schema to a package
   top relation SchemaToPackage {
     pn : String;
     checkonly domain rdbms s : RdbmsSchema {
       rdbmsName = pn
     };
     enforce domain uml p : UmlPackage {
      umlName = pn
     };
   }
   -- map each table to a class
   top relation TableToClass {
     cn : String;
     checkonly domain uml t : RdbmsTable {
       rdbmsSchema = s : RdbmsSchema {},
       rdbmsName = cn
     };
     enforce domain rdbms c : UmlClass {
       umlNamespace = p : UmlPackage {},
       umlName = cn
     };
     when {
       SchemaToPackage(s, p);
     }
     where {
       -- Call to ColumnToAttribute relation for each column
in each class
       t.rdbmsColumn->forAll(cl:RdbmsColumn |
cl.oclIsUndefined()
         implies ColumnToAttribute(t, c, cl));
     }
   }
   ...
}
```

Figura 9.19. Fragmento de la implementación QVT para transformar el modelo de base de datos (PSM) en un modelo de objetos (PIM).

El modelo de objetos resultante en el ejemplo consiste en cuatro objetos, uno por cada tabla en el modelo del esquema de la base de datos, es decir, *Proceedings*, *Papers*, *Authors* y *Author-Paper*. Estas clases constituyen la denominada *capa o modelo de negocio* para el sistema destino que se desea construir basado en servicios web (véase Figura 9.20). Esta capa de negocio contiene toda la información de

negocio para los futuros servicios web. Las capas restantes fueron creadas en fases posteriores del proyecto de modernización.

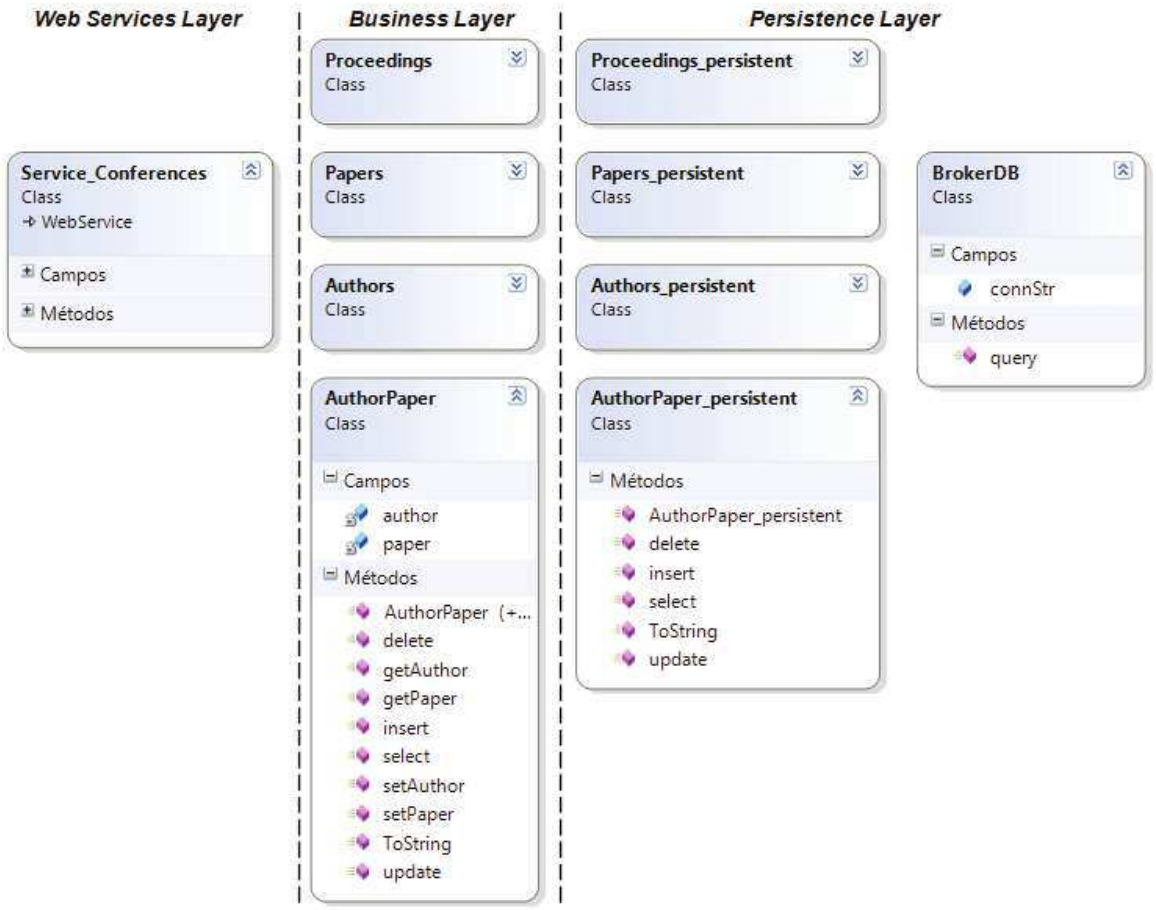


Figura 9.20. Modelo de objetos obtenido durante el caso de estudio.

9.6.3.2 FASE DE REESTRUCTURACIÓN

En general, la fase de reestructuración se encarga de transformar el modelo PIM en otro modelo PIM mejorado al mismo nivel de abstracción. Durante la reestructuración del proyecto de modernización se añadieron objetos auxiliares para soportar la capa de persistencia del modelo de objetos obtenido previamente (véase Figura 9.20).

Además, durante la fase de reestructuración se descubrieron los servicios que potencialmente podrían ser usados para manejar la información de la base de datos heredada. Así pues, para el proceso de modernización se definieron un conjunto de patrones que servían para realizar búsquedas sobre el esquema recuperado de la base de datos. A su vez, cada patrón definía una plantilla para crear una serie de

servicios web predefinidos. La Figura 9.21 recoge los patrones que se buscaron y los servicios asociados a cada uno de ellos. Por un lado, se definieron patrones simples que involucraban solo a una tabla. Estos servicios eran directamente obtenidos del esquema de la base de datos como las operaciones *CRUD* (*Create / Read / Update / Delete*) así como servicios de obtención y almacenamiento de datos para las diferentes columnas de una tabla (operaciones *getter* and *setter*). Por otra parte, se definieron patrones más complejos que involucraban a varias tablas. En este caso, se obtuvieron servicios directamente desde vistas de la base de datos. También se aplicaron patrones relacionados con las claves primarias y ajenas de varias tablas, como: (i) el patrón tabla referenciada, cuando existía una clave ajena entre dos tablas; (ii) el patrón tabla combinada, cuando había dos o más claves ajenas desde una misma tabla a otras tablas; y (iii) el patrón tabla observada, el cual buscaba dos o más claves ajenas en dos tablas diferentes apuntando hacia la clave primaria de una misma tabla. Los servicios candidatos que se descubrían desde cada uno de estos patrones pueden consultarse en la Figura 9.21.

<table>
<tr><td rowspan="2">Servicios Simples</td><td rowspan="2">Tablas</td><td colspan="2">Operaciones CRUD (Insercción, lectura, actualización y borrado)</td></tr>
<tr><td colspan="2">Operaciones Getter y Setter</td></tr>
<tr><td rowspan="4">Servicios Complejos</td><td>Vistas</td><td colspan="2">Queries</td></tr>
<tr><td rowspan="3">Patrones en el esquema de la base de datos</td><td>Referenced Table</td><td>Selecciona_A_de_B (pkB)
Selecciona_B_por_A (pkA)</td></tr>
<tr><td>Tabla Combinada</td><td>Selecciona_A_por_B (pkB)
Selecciona_A_por_B_filtrado (pkB, colsM)
Existe_A_relacionado_con_B(pkA, pkB)
Selecciona_A_por_B_y_C (pkB, pkC)
Selecciona_A_por_B_y_C_filtrado(pkB, pkC, colsM)
Existe_A_relacionado_con_B(pkA, pkB, pkC)</td></tr>
<tr><td>Tabla Observada</td><td>Selecciona_A_por_B (pkB)
Selecciona_B_por_A (pkA)</td></tr>
</table>

Figura 9.21. Patrones y plantillas de servicios usadas en el caso de estudio.

Siguiendo con el ejemplo, en esta fase el proceso de modernización primero añade cuatro objetos a la capa de persistencia: *Procedings_persistent*, *Papers_persistent*, *Author_persistent* and *AuthorPaper_persistent*. Además, varios servicios

candidatos son descubiertos por medio de la aplicación de los patrones presentados. Si nos centramos solo en la tabla *Author-Paper*, la cual tiene dos claves ajenas hacía las tablas *Authors* y *Papers* el patrón *'Tabla Combinada'* sería disparado. Usando las plantillas definidas para ese patrón, los servicios web candidatos que se generarían automáticamente serían los que se muestran en la Figura 9.22. Los servicios de la plantilla para seleccionar tuplas filtradas por algún campo no son instanciados en el ejemplo ya que la tabla *Author-Paper* no tiene columnas adicionales a parte de las que forman parte de la clave primaria.

```
Author selecciona_Authors_por_Papers(int id_paper)
Boolean existe_Authors_relacionado_con_Papers(int id_author,
int id_paper)
Paper selecciona_Papers_por_Authors(int id_author)
Boolean existe_Papers_relacionado_con_Authors(int id_paper,
int id_author)
```

Figura 9.22. Servicios obtenidos en el caso de estudio a partir de la detección del patrón 'Tabla Combinada' para la tabla 'Author-Paper'.

9.6.3.3 FASE DE INGENIERÍA DIRECTA

La tercera fase tuvo como objetivo la obtención, por medio de ingeniería directa, de un conjunto de servicios web que manejaran los datos de la base de datos heredada. Primero, el último modelo PIM (transformado durante la fase de reestructuración) es transformado en un modelo PSM representando las interfaces de los servicios web. Este modelo se representa de acuerdo al metamodelo WSDL (*Web Services Description Language*) [W3C, 2007]. El ejemplo desarrollado se centra nuevamente en la instancia del patrón *'Tabla Combinada'* que fue descubierto, y particularmente se muestra el fragmento del fichero WSDL generado para el servicio *'selecciona_Authors_por_Papers'* (véase Figura 9.23).

El último paso consistió en la generación del código fuente para soportar: (i) los modelos de objetos y de persistencia obtenidos en las fases previas, y (ii) la implementación de los servicios web desde el modelo WSDL. Siguiendo con el ejemplo, el código fuente de las clases mostradas es escrito automáticamente por la herramienta de soporte en ficheros en disco duro de acuerdo a un lenguaje de programación que soporte la definición de servicios web. En concreto, en el proyecto de modernización se empleó C#. Finalmente, los servicios web obtenidos fueron desplegados en un servidor web a fin de ser integrados con el sitio web de la organización. Por lo tanto, los servicios web se convirtieron en servicios totalmente funcionales que envolvieron a la base de datos heredada proveyendo un acceso a los datos dentro de un entorno SOA.

```
<?xml version="1.0" encoding="utf-8" ?>
- <wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:tns="http://Conferences.org/"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  targetNamespace="http://Conferences.org/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">Web
    service for testing access to database by means of Conferences
    objects.</wsdl:documentation>
- <wsdl:types>
  - <s:schema elementFormDefault="qualified"
      targetNamespace="http://Conferences.org/">
    - <s:element name="selectAuthors_for_Papers">
      - <s:complexType>
        - <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="paper"
              type="s:int" />
          </s:sequence>
        </s:complexType>
      </s:element>
    + <s:element name="selectAuthors_for_PapersResponse">
    </s:schema>
  </wsdl:types>
+ <wsdl:message name="selectAuthors_for_PapersSoapIn">
- <wsdl:message name="selectAuthors_for_PapersSoapOut">
    <wsdl:part name="parameters"
      element="tns:selectAuthors_for_PapersResponse" />
  </wsdl:message>
- <wsdl:portType name="Service_ConferencesSoap">
  - <wsdl:operation name="selectAuthors_for_Papers">
      <wsdl:documentation
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">Return the
        object list correspond to the pattern "COMBINED TABLE" to
        relationship "selectAuthors_for_Papers"</wsdl:documentation>
      <wsdl:input message="tns:selectAuthors_for_PapersSoapIn" />
      <wsdl:output message="tns:selectAuthors_for_PapersSoapOut" />
    </wsdl:operation>
  </wsdl:portType>
+ <wsdl:binding name="Service_ConferencesSoap"
    type="tns:Service_ConferencesSoap">
+ <wsdl:binding name="Service_ConferencesSoap12"
    type="tns:Service_ConferencesSoap">
+ <wsdl:service name="Service_Conferences">
</wsdl:definitions>
```

Figura 9.23. Fragmento de fichero WSDL para soportar el servicio web 'selectAuthors_for_Papers'.

9.7 COSTES Y BENEFICIOS DE LA REINGENIERÍA Y LA MODERNIZACIÓN

Antes de reconstruir un sistema en explotación, es altamente recomendable analizar las diversas alternativas disponibles:

- Dejar el producto como está.
- Adquirir uno en el mercado que realice la misma función.
- Reconstruirlo.

Evidentemente, elegiremos la opción que mejor relación coste/beneficio nos ofrezca. Para calcular los costes de un proyecto de Reingeniería, [Sneed, 1995] propone un modelo basado en cuatro etapas:

9.7.1 Justificación del proyecto

Requiere el análisis del software existente, de los procesos de mantenimiento actuales y del valor de negocio de las aplicaciones; todo esto con el objetivo de poder evaluar si se aumentará el valor de estos tres factores.

La mayoría de las organizaciones sólo toman en consideración los procesos de reingeniería cuando el coste de un nuevo desarrollo es demasiado alto. En cualquier caso, y aunque a primera vista parezca la única o la mejor alternativa, es necesario confirmar la necesidad de reconstruir el sistema.

Cuatro operaciones nos pueden dar una idea de los costes del proyecto y del valor de negocio del software actual:

- Introducción de un sistema de evaluación de los costes del mantenimiento. Es recomendable que esta tarea la lleve a cabo la organización anticipándose con suficiente antelación al momento en que se percibe la necesidad de aplicar el proceso de Reingeniería.

- Análisis de la calidad del software actual, para lo cual pueden utilizarse auditores de código automáticos que proporcionan datos del tamaño, complejidad y métricas de calidad del código fuente junto con evaluaciones realizadas por expertos de forma manual. Estos valores son incorporados a una base de datos que es utilizada por otra herramienta para realizar comparaciones y obtener resultados.

- Análisis de los costes de mantenimiento: [Berns, 1984] propone tres métricas para medir los procesos de mantenimiento: «Dominio del impacto» o proporción de instrucciones y elementos de datos afectados por una tarea de mantenimiento con respecto al total de instrucciones y elementos de datos del sistema; «Esfuerzo empleado», que es el número de horas dedicadas a tareas de mantenimiento, con lo que se puede obtener una media del número de horas por tarea de mantenimiento; y

«Tasa de errores de segundo nivel», que es el número de errores causados por acciones de mantenimiento. Si observamos que estas tres medidas se incrementan, es muy probable que los costes de mantenimiento se incrementen con el tiempo.

- Evaluación del valor de negocio del sistema actual, que es realizado por la dirección de la organización con ayuda de expertos técnicos.

9.7.2 Análisis de la cartera de aplicaciones

En esta etapa se cotejan la calidad técnica y el valor de negocio de cada aplicación, con el objetivo de construir una lista de aplicaciones, ordenada según sus prioridades en el proceso de Reingeniería.

La *calidad técnica* de un producto es una medida relativa, dependiente de cada organización, que se calcula en función de diversas características (complejidad ciclomática o errores/KLDC, por ejemplo).

Para cada variable que interviene en la calidad técnica se fijan unos límites inferior y superior (que representan, respectivamente, los valores máximo y mínimo de calidad). Para hallar el nivel de calidad de la variable considerada, [Sneed, 1995] propone la siguiente fórmula:

$$C_i = 1 - \frac{\text{Medida actual} - \text{Límite inferior}}{\text{Límite superior - Límite inferior}}$$

Por ejemplo, si establecemos los valores mínimo y máximo de calidad en 0 y 7 errores por KLDC, y actualmente hay 3, Ci=0,571.

Asociando un punto en un plano para cada aplicación, e interpretando el valor de negocio y la calidad técnica como coordenadas de estos puntos, los representamos en un diagrama como el de la Figura 9.24. Las aplicaciones situadas en el cuadrante superior izquierdo tienen alta calidad y bajo valor de negocio, por lo que no requieren reingeniería; las situadas en el cuadrante inferior izquierdo tienen poco valor en ambos parámetros, por lo que pueden ser desarrolladas de nuevo o reemplazadas por productos comerciales; las del superior derecho tienen gran valor de negocio y alta calidad: se les puede aplicar reingeniería, pero sin excesiva prioridad; las del inferior derecho tienen alto valor de negocio y baja calidad técnica, por lo que serán las primeras candidatas a la reingeniería.

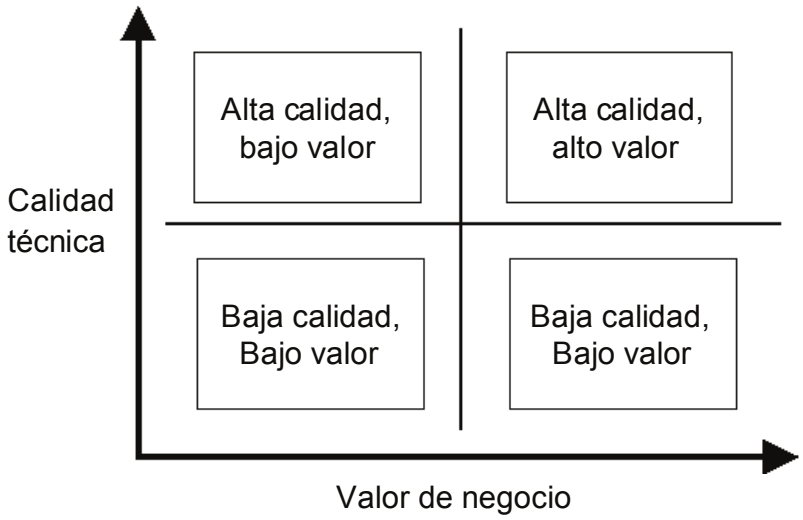


Figura 9.24. Diagrama para el análisis de la cartera de aplicaciones

9.7.3 Estimación de costes

Se realiza identificando y ponderando, mediante métricas adecuadas, todos los componentes del software que se van a modificar.

Se deben considerar los costes de cada proyecto de reingeniería: si éstos son superiores a los beneficios, la Reingeniería no será una alternativa viable y la aplicación deberá ser desarrollada de nuevo o bien adquirirse en el mercado.

Para estimar los costes de la reingeniería, tenemos ciertas ventajas respecto a la misma estimación en proyectos de Ingeniería directa: no debemos calcular factores influyentes como el número de líneas de código, sentencias ejecutables, elementos de datos, accesos a ficheros, etc., ya que son medidas que se pueden tomar directamente de la aplicación bajo análisis.

[Sneed, 1995] aconseja utilizar como variables para calcular los costes las que ofrecemos a continuación, y que deben ser debidamente ponderadas en función de su influencia en el coste total:

- Número de líneas de código no comentadas.
- Coste de los casos de prueba, que se calcula multiplicando el coste medio de cada caso de prueba por el número de éstos, que es función de la complejidad ciclomática del problema.

- Número de accesos a ficheros, bases de datos y campos. En la ponderación de estas entradas/salidas consideraremos la complejidad de las estructuras de información y el grado de independencia de la aplicación respecto de los datos.
- Número de operaciones que realizan los usuarios de la aplicación, número de ventanas, número de informes, etc., para el caso de las interfaces de usuario.

A continuación, se traducen los valores obtenidos a personas-mes usando una variante del modelo COCOMO en el que se divide el número de líneas de código por un valor que depende del lenguaje de programación que se utilice (2 para ensamblador, 3 para C y PL/I y 4 para Cobol). La ecuación de Sneed para el tiempo de desarrollo de un trabajo de Reingeniería es la siguiente:

$$Tiempo\ de\ desarrollo = 2.5 * (esfuerzo)^{0.19}$$

9.7.4 Análisis de costes/beneficios

Una vez que se ha calculado el coste de la Reingeniería, la última etapa es comparar los costes con los beneficios esperados (no es suficiente con examinar los beneficios que aporte la Reingeniería).

La Unión de Bancos Suizos seleccionó los 13 parámetros que mostramos en la Figura 9.25 para tomar la decisión de abandonar, redesarrollar o aplicar reingeniería a aplicaciones heredadas.

El beneficio proporcionado por continuar manteniendo el producto sin reingeniería es el siguiente:

$$B_M = [P_3 - (P_1 + P_2)] * P_{16}$$

Deberá retocarse la fórmula cuando los diversos costes varíen de un año para otro.

Si desarrollamos de nuevo el sistema, obtenemos este beneficio:

$$B_D = [(P_{12} - (P_{10} + P_{11})) * (P_{16} - P_{14}) - (P_{13} * P_{15})] - B_M$$

El beneficio producido por la Reingeniería es:

$$B_R = [(P_6 - (P_4 + P_5)) * (P_{16} - P_8) - (P_7 * P_9)] - B_M$$

P_1=Valor de negocio actual (anual).
P_2=Coste previsto de mantenimiento tras la reingeniería (anual).
P_3=Coste previsto de operaciones tras la reingeniería (anual).
P_4= Valor de negocio previsto tras la reingeniería actual (anual).
P_5=Coste estimado de la Reingeniería.
P_6=Duración estimada de la Reingeniería.
P_7=Factor de riesgo de la Reingeniería.
P_8=Coste previsto de mantenimiento tras el redesarrollo (anual).
P_9=Coste previsto de operaciones tras el redesarrollo (anual).
P_{10}=Valor de negocio previsto del nuevo sistema (anual).
P_{11}=Coste estimado del redesarrollo.
P_{12}= Duración estimada del redesarrollo.
P_{13}=Factor de riesgo del redesarrollo.
P_{14}=Vida esperada del sistema.

Figura 9.25. Parámetros a considerar en proyectos de reingeniería

9.8 LECTURAS RECOMENDADAS

- *Journal of Software: Evolution and Process*

  En esta revista se recogen los principales trabajos relacionados con las principales técnicas de ingeniería inversa y su aplicación a nivel industrial.

- Mens, T. y Demeyer, S. (eds.). Software Evolution. Heidelber, Berlin (Alemania). 2010.

  En este libro, se recogen diferentes propuestas para el análisis y la renigeniería de sistemas heredados.

- Suryanarayana, G., Smarthyam, G. y Sharma, T. Refactoring for Software Design Smells. Amsterdam. Morgan Kaufmann. 2015.

  En este libro se presentan diversas opciones de refactorización de diversos tipos de *smells* aplicables al software.

✓ Birchall, C. Re-Engineering Legacy Software. Manning. 2016.

Este libro presenta diferentes técnicas para la refactorización, así como un framework de soporte a las mismas.

9.9 SITIOS WEB RECOMENDADOS

✓ *http://ieeexplore.ieee.org/Xplore/home.jsp*

En este portal se pueden encontrar decenas de artículos y comunicaciones sobre las técnicas tratadas en este capítulo, y las actas de los congresos más importantes de mantenimiento y evolución de software: *International Conference on Software Maintenance and Evolution, European Conference on Software Maintenance and Reengineering*, y *Working Conference on Reverse Engineering*, estas dos últimas que actualmente se han fusionado en el nuevo congreso SANER (*Software Analysis, Evolution, and Reengineering*).

En estas actas encontraremos interesantísimos artículos de muy variados autores, en los que se muestran las últimas novedades en cuanto a mantenimiento y evolución de software. Muchos de los artículos presentan técnicas específicas para reconstruir, redocumentar o realizar ingeniería inversa de muy variopintos entornos.

9.10 EJERCICIOS

Ejercicio 1

Explique los conceptos de Ingeniería Inversa y Reingeniería.

Ejercicio 2

Explique el concepto de *Componente funcional*. Para observar la importancia de asignar nombres significativos a los componentes funcionales, realice el siguiente experimento: entregue a un compañero un sencillo programa cuyos identificadores den una idea de las tareas que desempeñan, y entregue el mismo programa a otro compañero, pero asignando a todos los identificadores cadenas de no más de un carácter de longitud. A continuación, pídales que expliquen en voz alta qué hace el programa y obtenga sus propias conclusiones.

Ejercicio 3

¿Qué tareas realizaría usted antes de acometer un proyecto de Reingeniería?

Ejercicio 4

¿Qué diferencias existen entre la reingeniería tradicional y la modernización dirigida por la arquitectura o ADM?

Ejercicio 5

Explique cuál es la función del estándar KDM dentro de la Modernización Software

Ejercicio 6

Proponga una serie de parámetros que pudieran servir para medir el esfuerzo de Reingeniería de bases de datos relacionales. Justifique su elección.

Ejercicio 7

Analice las herramientas existentes para la reingeniería. ¿Cuáles siguen el enfoque de ADM?

Ejercicio 8

Compare y complemente el análisis de la cartera de aplicaciones, con el enfoque propuesto para la cartera de proyectos por parte del PMI.

Ejercicio 9

Lleve a cabo un mapeo sistemático sobre las áreas en las que se han propuesto metodologías para la reingeniería tomando como base las publicaciones de la *International Conference on Software Maintenance and Evolution* del IEEE.

Ejercicio 10

¿Qué tipo de método o técnica propondría para estimar el esfuerzo de la reingeniería?

10

ARQUEOLOGÍA DE PROCESOS DE NEGOCIO

Los sistemas de información de las organizaciones son, o deberían ser, un fiel reflejo de sus procesos de negocio (BP por sus siglas en inglés, *Business Process*) [Heuvel, 2006], ya que ofrecen gran parte de la operativa descrita en ellos. Por lo tanto, muchas organizaciones demandan visualizar sus procesos de negocio a fin de poder modificarlos, mejorarlos y ser capaces de hacer frente a los cambios del entorno y así mantener su competitividad intacta [Jeston et al., 2008].

No obstante, los sistemas de información heredados (LIS, *Legacy Information Systems*) suponen un impedimento a lo anterior debido a que estos sistemas hacen que las organizaciones sufran cierta rigidez. Esta rigidez se debe a que los LIS no pueden evolucionar con la misma agilidad que los procesos de negocio [Heuvel, 2006], ya que dependen, en muchos casos de forma rígida, de la infraestructura tecnológica y sistemas de la organización.

Por otra parte, los BP asociados a los LIS no siempre están disponibles (visibles a través de un modelo) o estando disponibles, no están alineados debido a la evolución por separado de los LIS y los BP. De hecho, el mantenimiento de los LIS al margen de los BP hace que los LIS embeban mucho conocimiento a lo largo del tiempo [Mens, 2008]. Por lo tanto, en estos casos puede ser muy útil el descubrimiento de los BP desde los LIS mediante ingeniería inversa, de forma que pueda saberse a ciencia cierta cuáles son los BP reales actuales (modelos AS-IS) y se puedan sincronizar con los LIS. Es importante tener en cuenta que en una organización podrá haber varios LIS, así como varios BP, y las relaciones entre ellos pueden ser *n* a *n*, ya que un BP es soportado por varios LIS, y un LIS puede dar soporte a varios BP.

10.1 CONCEPTOS GENERALES

Esta sección introduce los conceptos de Proceso de Negocio y la manera en que ADM puede facilitar la arqueología de procesos de negocio.

10.1.1 Modelos de Procesos de negocio

Los BP definen la secuencia de actividades necesarias para conseguir los objetivos de negocio definidos por una organización. Un proceso de negocio toma una o varias entradas y genera una salida que aporta un valor para los clientes de la organización [Jeston et al., 2008, Weske, 2007].

En los últimos años, y de cara a optimizar el rendimiento empresarial, se ha observado un incremento del interés de las organizaciones en las tareas específicas de gestión de sus procesos de negocio (BPM, *Business Process Management*) [Weske, 2007]. Esta gestión conlleva definir e incorporar métodos, técnicas y herramientas para apoyar el diseño, gestión y análisis de los procesos de negocio [Aalst et al., 2003b]. Una gestión óptima de los procesos de negocio permite adaptarlos a posibles cambios del entorno con el fin de incrementar la satisfacción del cliente, reducir costes, diferenciar sus productos o servicios, etc. [Jeston et al., 2008]. La relación entre los procesos de negocio y los sistemas de información se resume en la Figura 10.1.

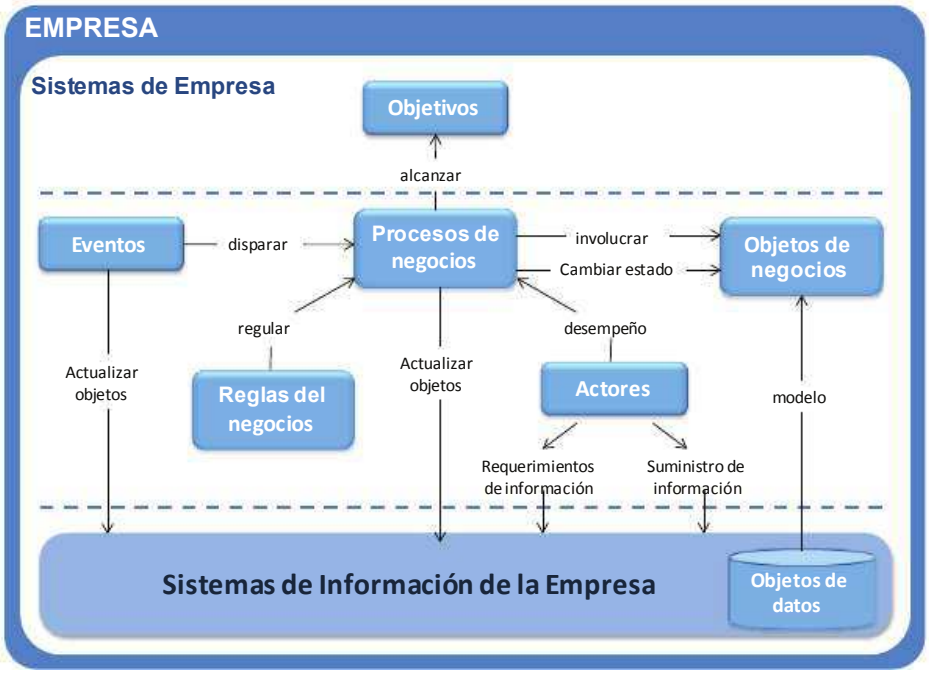


Figura 10.1. Relación de los Procesos de Negocio con los Sistemas de Información.

Como se ha comentado, es importante que las organizaciones realicen una gestión de sus procesos de negocio de forma eficaz y continua a fin de mantener y mejorar su nivel de competitividad. Como parte de la gestión de procesos de negocio, es necesario poder representar los procesos de negocio en una notación entendible por las personas involucradas en su gestión. Entre las notaciones existentes se encuentra la notación BPMN (*Business Process Modeling Notation*) [OMG, 2011, White et al., 2008], una notación gráfica muy extendida para representar los Diagramas de Procesos de Negocio (BPD, *Business Process Diagram*).

BPMN es un estándar desarrollado por la OMG que proporciona una notación para el modelado de la gestión de los procesos de negocio, mostrando las interrelaciones entre los distintos componentes, sus fuentes de información y el personal asociado a ellos. Todo esto es descrito mediante una forma entendible para todos los miembros de la organización, desde los analistas de negocio hasta los desarrolladores. La Figura 10.2 muestra los elementos básicos de BPMN. En ella destacan los objetos de flujo como los eventos (de inicio, intermedios o finales), las actividades o tareas y las compuertas (*gateways*), los conectores como flujos de secuencia, flujos de mensaje o asociaciones, los contenedores (*pool*) y compartimentos (*lane*), y los artefactos como los objetos de datos (*Data Object*).

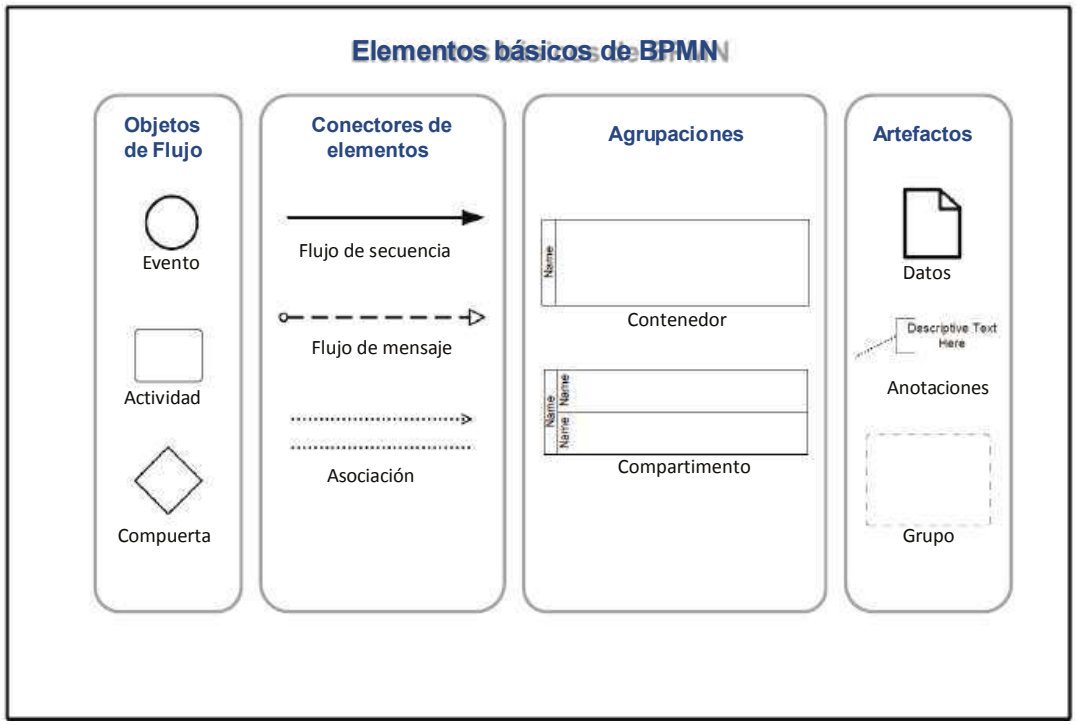


Figura 10.2. Elementos básicos de BPMN

10.1.2 ADM para la Arqueología de Procesos de Negocio

Como se presentó en el capítulo 9, el modelo de modernización en herradura tiene tres vertientes diferentes respecto al alcance del proceso de modernización. La modernización dirigida por la arquitectura de negocio representa el sistema al máximo nivel de abstracción, reestructurando a nivel de reglas de negocio o procesos de negocio. Los riesgos y el impacto del cambio son mayores cuando la fase de restructuración se realiza a mayor nivel de abstracción, por ejemplo, sobre el modelo CIM (Modelo independiente de la computación). No obstante, en este nivel de abstracción las opciones y oportunidades para reestructurar el LIS son mucho mayores. Por ello, la arqueología de procesos de negocio se centra en la extracción de modelos BP en la fase de ingeniería inversa.

10.2 UN MARCO PARA LA ARQUEOLOGÍA DE PROCESOS DE NEGOCIO

Proponemos un marco de trabajo denominado MARBLE (del nombre en inglés *Modernization Approach for Recovering Business processes from LEgacy systems*), para el descubrimiento y extracción de BPs desde LIS, siguiendo el enfoque ADM, centrado en la fase de ingeniería inversa, que ofrece las siguientes ventajas:

- Aumentar la agilidad de cambio en los sistemas (LIS) y/o el negocio (BP) gracias a las transformaciones entre modelos y la generación automática de código.
- Utilizar KDM, de manera que toda la información derivada de la extracción de los BP se representa de forma estándar y puede ser utilizada para obtener nuevos sistemas de información modernizados que están alineados con los BP extraídos, completando así un proceso de reingeniería completo.
- Reducir costes de mantenimiento
- Extender el ciclo de vida de los LIS.

MARBLE se centra en la fase de ingeniería inversa del modelo de reingeniería en herradura y se basa en KDM, el cual permite realizar representaciones conceptuales abstractas de las diferentes vistas de la arquitectura de los LIS. Posteriormente, ese conocimiento se transforma progresivamente en los BPs. Para ello, MARBLE se divide en cuatro niveles de abstracción y definiendo además tres transformaciones entre ellos (véase Figura 10.3).

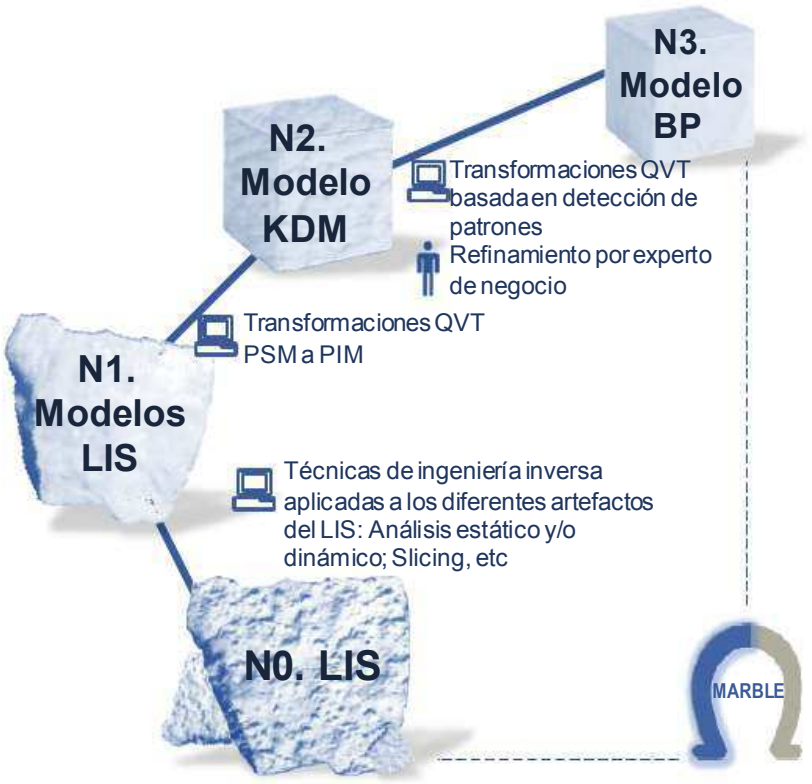


Figura 10.3. Vista general de MARBLE

- **Nivel 0 (N0)**. Este nivel representa al LIS de origen en el mundo real, del cual se pretende descubrir los BPs asociados.
- **Nivel 1 (N1)**. Este nivel agrupa un conjunto de modelos que representan las diferentes vistas o aspectos de la arquitectura del LIS, es decir los diferentes activos software como código fuente, bases de datos, componentes, etc. La transformación N0-a-N1 obtiene modelos PSM desde cada activo software heredado. Para ello se recurre a la extracción de información del LIS por medio de técnicas de ingeniería inversa clásicas como análisis estático/dinámico, *slicing*, etc. [Canfora et al., 2007]. Los diferentes modelos PSM se construyen de acuerdo a metamodelos específicos: por ejemplo, si el LIS es un sistema java se utilizará un metamodelo de java para crear un modelo de código, si la tecnología de bases de datos es la relacional se empleará un metamodelo de SQL para modelar el esquema de la base de datos y así con los demás activos software.
- **Nivel 2 (N2)**. Este nivel se enfoca en la representación integrada de todos los activos software de un LIS de acuerdo al metamodelo de KDM. Durante la transformación N1-a-N2 los modelos específicos se transforman

hacia un único modelo PIM basado en KDM. Estas transformaciones se formalizan mediante QVT. Dentro de este mismo nivel pueden establecerse transformaciones entre diferentes capas de la arquitectura KDM. Por ejemplo, transformaciones entre la capa de elementos de programa o recursos que representan conocimiento explícito del LIS y la capa de abstracción que representa conocimiento implícito. De esta manera se reduce progresivamente la brecha conceptual entre el LIS y los BPs.

- **Nivel 3 (N3)**. El último nivel representa los modelos de BPs que son recuperados desde el modelo KDM. Este modelo equivale al CIM y representa un conjunto de diagramas de BPs. Para la representación de los diagramas BP, MARBLE cuenta con un metamodelo desarrollado ex profeso basado en BPMN. La transformación N2-a-N3 se establece también mediante transformaciones QVT, aunque en este caso están basadas en la detección de patrones de negocio en el modelo KDM. Adicionalmente, un experto de negocio puede refinar los BPD extraídos mediante las transformaciones.

La Tabla 10.1 resume cada una de las tres transformaciones de MARBLE. El objetivo final es completar el camino N0→N1→N2→N3 a fin de obtener los BPs asociados desde un LIS.

<table>
<tr><th colspan="2">Transformación</th><th>Artefactos</th><th>Metamodelos</th><th>Técnica de transformación</th></tr>
<tr><td rowspan="2">N0-a-N1</td><td>In</td><td>Activos software del LIS: código fuente, bases de datos, interfaces de usuario, etc.</td><td>-</td><td rowspan="2">Técnicas de ingeniería inversa: análisis estático y dinámico, program slicing, etc.</td></tr>
<tr><td>Out</td><td rowspan="2">Modelos PSM por cada activo software</td><td rowspan="2">Metamodelos específicos: MM_{JAVA}, MM_{SQL}, MM_{C++}, etc.</td></tr>
<tr><td rowspan="2">N1-a-N2</td><td>In</td><td rowspan="2">Transformaciones entre modelos establecidas mediante QVT Relations.</td></tr>
<tr><td>Out</td><td rowspan="2">Modelos KDM tipo PIM</td><td rowspan="2">Metamodelo de KDM</td></tr>
<tr><td rowspan="2">N2-a-N3</td><td>In</td><td rowspan="2">• Transformaciones entre modelos basadas en la detección de patrones y establecidas mediante QVT.
• Refinamiento por parte del experto de negocio.</td></tr>
<tr><td>Out</td><td>Modelo de Diagramas de Procesos de Negocio</td><td>Metamodelo de BPMN</td></tr>
</table>

Tabla 10.1. Transformaciones entre los niveles de MARBLE.

Hay que tener en cuenta que no todas las partes de los procesos de negocio de una organización se ejecutan o están soportados por los LIS ya que existen tareas que se ejecutan de forma manual con intervención humana. Por esto, es que el proceso descrito por MARBLE considera la intervención humana al final de la última transformación para enriquecer el modelo de proceso de negocio. Así se pueden añadir dichas tareas de ejecución manual, roles adicionales, etc.

10.2.1 Ejemplo para sistemas java

MARBLE está definido como un marco genérico de acuerdo con ADM para la arqueología de procesos de negocio. Sin embargo, este marco se debe instanciar para cada tipo de lenguaje o lenguajes de programación u otras tecnologías involucradas en el LIS. Las siguientes secciones muestran a modo de ejemplo la definición de las tres transformaciones de MARBLE para sistemas java.

10.2.1.1 TRANSFORMACIÓN NO-A-N1

En este caso la técnica usada es el análisis estático de código fuente y el artefacto software es el código java. El análisis estático consiste en el análisis sintáctico de cada uno de los ficheros de código que pertenecen al LIS.

La ventaja de esta técnica es que la construcción de un parseador sintáctico de código es fácil y poco costoso de desarrollar con las técnicas actuales de teoría de compiladores. Además, la importancia del código fuente es incuestionable en los LIS ya que el conocimiento de negocio está real y literalmente "enterrado" en el código fuente [Müller et al., 2000].

No obstante, el análisis estático de código fuente tiene algunas limitaciones. Por ejemplo, esta técnica no puede extraer información sobre la secuencia de código más comúnmente ejecutada o aquellas partes que no son alcanzables nunca o rara vez. Esta información sí se puede extraer por análisis dinámico que se realiza con el programa en ejecución, no obstante, es ciertamente complejo y a veces incluso imposible, ya que algunas de las técnicas de análisis dinámico se basan en la inserción de trazas que recuperan información en ejecución lo que implica modificar ciertamente el programa, y muchas compañías son reacias a modificar el código de programas críticos de negocio.

De esta forma, esta transformación consiste en un parseador que construye un árbol sintáctico abstracto de cada fichero java, es decir, un modelo de acuerdo con el metamodelo java (véase Figura 10.4. Vista simplificada del metamodelo de java.). Para ilustrar la transformación se presenta un ejemplo de un pequeño archivo java

que contiene una clase con dos métodos (véase la parte central de la Figura 10.5). Después de la ejecución de la transformación N0-a-N1 un árbol sintáctico abstracto correspondiente al fichero java aparece en la parte de la derecha de la Figura 10.5.

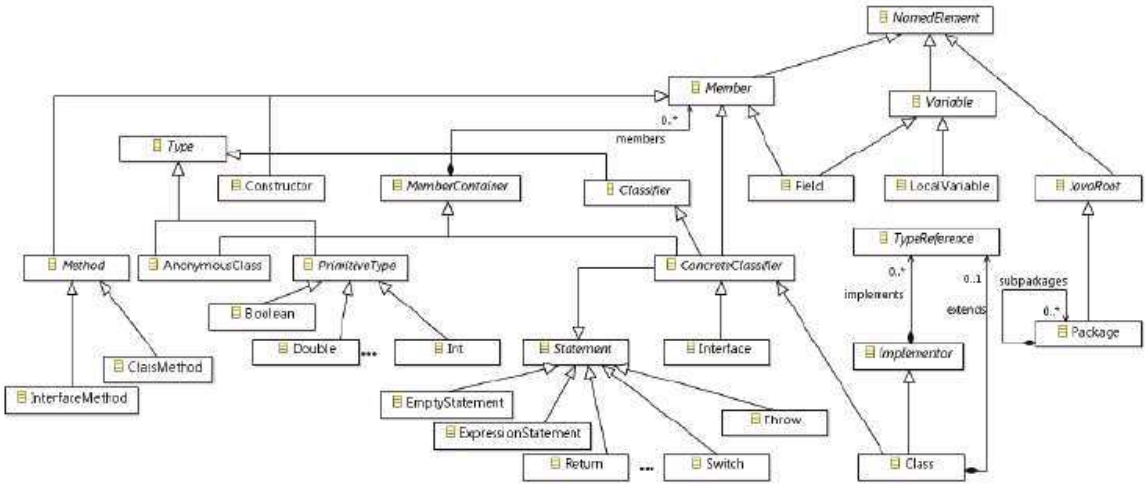


Figura 10.4. Vista simplificada del metamodelo de java.

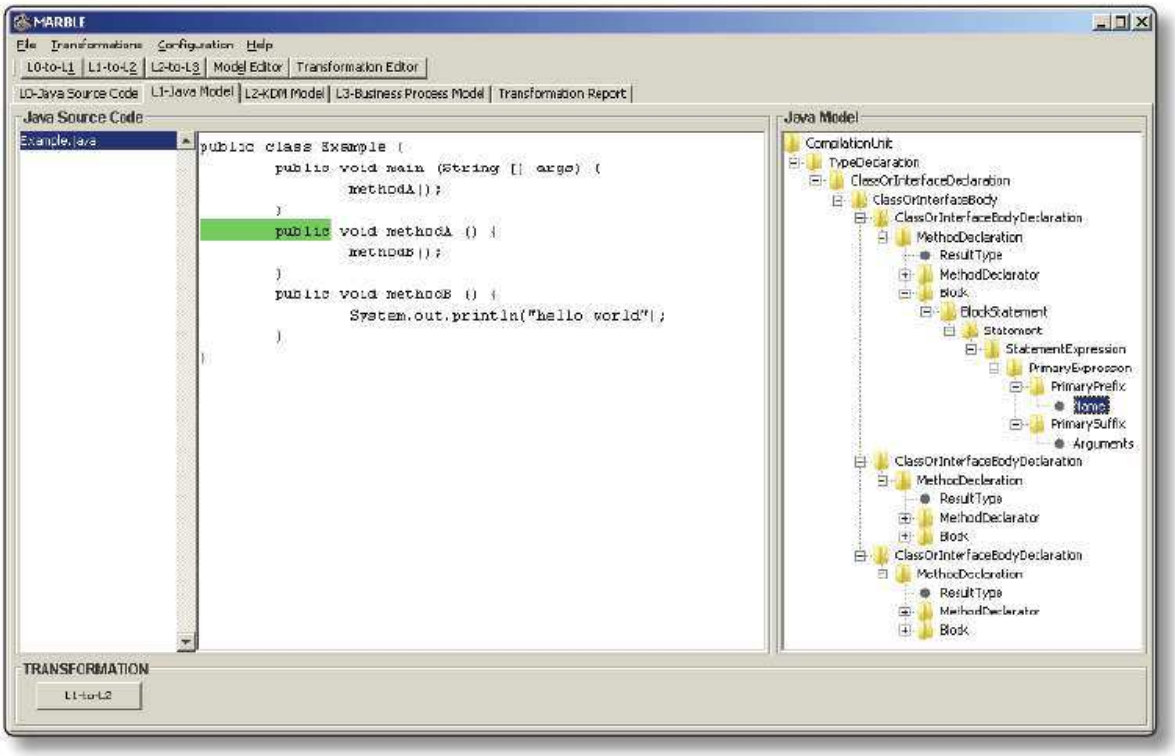


Figura 10.5. Un ejemplo de un pequeño fichero java después de la transformación N0-a-N1.

10.2.1.2 TRANSFORMACIÓN N1-A-N2

Esta transformación obtiene un único modelo PIM de acuerdo con el metamodelo KDM. El modelo KDM en N2 aglutina todos los modelos de código fuente java obtenidos en la transformación previa. Debido a que el código fuente es el único artefacto software considerado, esta transformación usa sólo la parte del metamodelo KDM referida a dicho artefacto, es decir los paquetes *Code* y *Action* dentro de la capa *Program Element*. Esta capa es la segunda capa de abstracción dentro del metamodelo KDM la Figura 10.6 muestra una versión simplificada de los metaelementos más importantes de los paquetes *Code* y *Action* del metamodelo KDM. De acuerdo con el metamodelo KDM, cada LIS analizado se representa como un elemento *CodeModel*, el elemento raíz. A su vez, un *CodeModel* se compone de *AbstractCodeElements*, un metaelemento que representa una clase padre abstracta para todos los tipos de entidades KDM que pueden ser usadas como *CallableUnit*, *StorableUnit,* entre otras. *CodeElements* están también interrelacionados por medio de *AbstractActionRelationships* (paquete *Action*), un metaelemento que representa una clase padre abstracta para elementos como *Flow*, *Calls*, *Reads*, *Writes*.

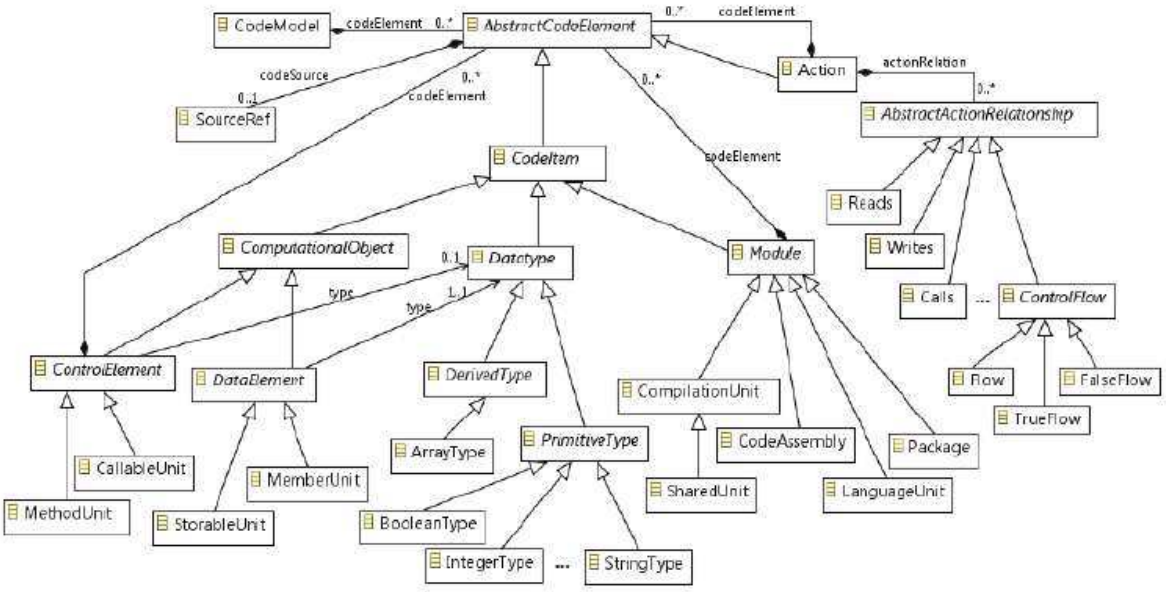


Figura 10.6. Vista simplificada de los paquetes Code y Action del metamodelo KDM.

Esta trasformación se implementa en QVT *Relations*, la parte declarativa del lenguaje de transformaciones QVT, ya que la transformación de modelos en estos niveles es casi directa de forma declarativa puesto que muchas de las transformaciones entre elementos son renombrados o agrupaciones de elementos. La

Figura 10.7 muestra como ejemplo una trasformación para transformar cada paquete java en un elemento *package* de KDM. La segunda relación transforma cada clase java en un elemento *CompilationUnit* en la salida del modelo KDM.

```
transformation Java2KDM (java:java, kdm:KDM){
   top relation package2package {
      packageName : String;
      checkonly domain java jp : java::containers::Package
{
         name = packageName
      };
      enforce domain kdm kp : KDM::code::Package {
         name = packageName
      };
      where {
         jp.compilationUnits->forAll (jc:java::containers::C
ompilationUnit |
            jc.oclIsKindOf(java::classifiers::Class) implies
class2compilationUnit
            (jc.oclAsType(Class), kp));
      }
   }
   relation class2compilationUnit {
      className : String;
      checkonly domain java jc : java::classifiers::Class {
         name = className
      };
      enforce domain kdm kp : KDM::code::Package {
         codeElement = kcu : KDM::code::CompilationUnit {
            name = className
         }
      };
      where {
         jc.members->forAll (jm:java::members::Method |
            jm.oclIsKindOf(java::members::Method) implies
method2callableUnit
            (jm.oclAsType(Method), kcu));
      }
   }
   ...
}
```

Figura 10.7. Dos relaciones QVT en la transformación N1-a-N2.

Para continuar con el ejemplo introducido en la anterior sección, se muestra el modelo KDM (véase Figura 10.8) obtenido desde el modelo de código previo después de ejecutar la transformación N1-a-N2. El modelo KDM contiene una instancia *CodeModel* llamada 'Example' la cual está anidada dentro de una instancia *Segment* con el mismo nombre. La instancia *CodeModel* tiene un elemento *Package* con el nombre 'default', ya que el fichero java original no definió ningún paquete. La instancia *Package* contiene anidados una instancia *CompilationUnit* llamada 'Example' como se definió en el nombre de la clase java original. Posteriormente se pueden ver los elementos *CallableUnit* por cada uno de los métodos java. Finalmente, cada instancia *CallableUnit* contiene diferentes elementos como *EntryFlow, Action,* etc.

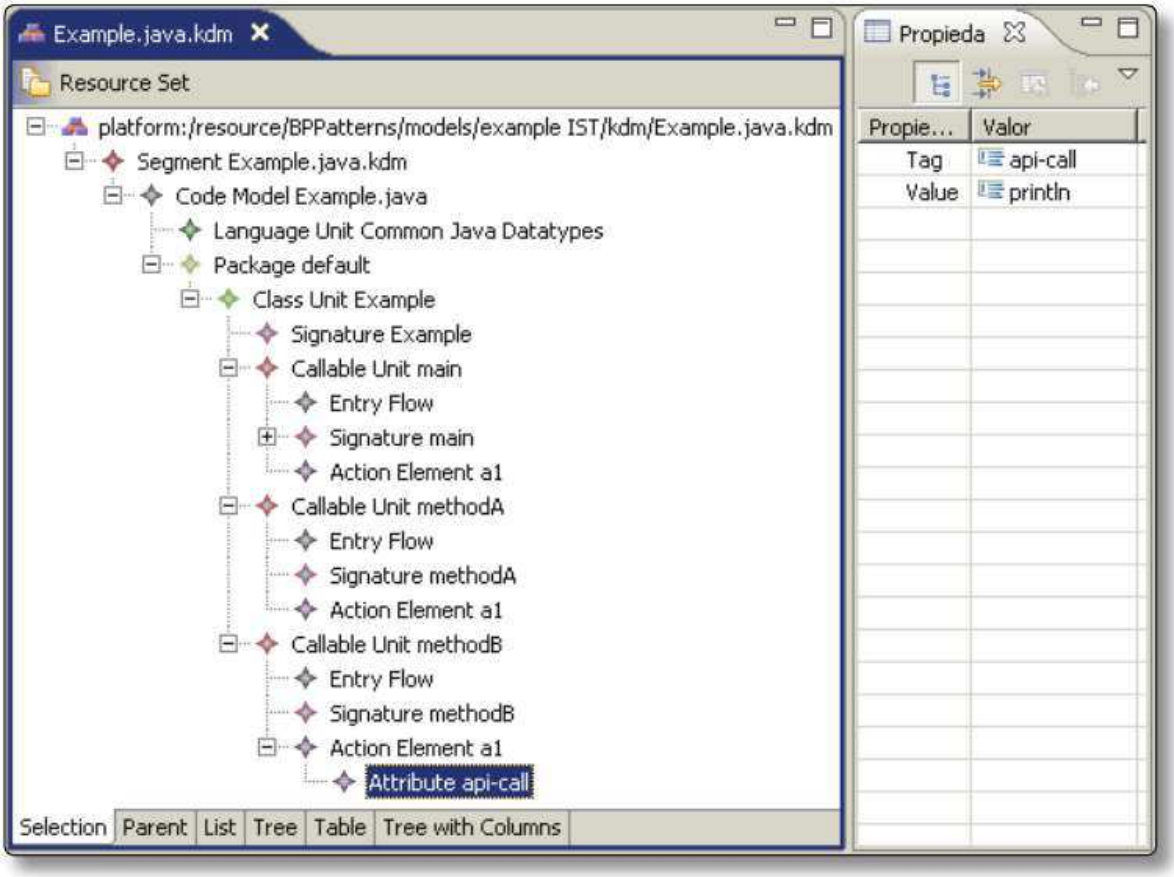


Figura 10.8. Un ejemplo de modelo KDM obtenido después de ejecutar N1-a-N2.

10.2.1.3 TRANSFORMACIÓN N2-A-N3

La última transformación obtiene los modelos de proceso de negocio desde el modelo KDM anterior a través de la aplicación de un conjunto de patrones y modificaciones finales realizadas por expertos. Los modelos de procesos de negocio

en el nivel N3 son representados de acuerdo con el metamodelo BPMN (Figura 10.9).

El conjunto de patrones considera patrones bien conocidos en la literatura que han sido aplicados con éxito por expertos de negocio para el modelado de procesos de negocio en procesos de ingeniería directa "[Aalst et al., 2003a; Pérez-Castillo et al., 2010; Zdun et al., 2007; Zhao et al., 2008]. Los patrones de negocio establecen que estructuras específicas de los sistemas heredados (representados a través del modelo KDM) han de ser transformados en ciertas estructuras en los modelos de procesos de negocio. Por lo tanto, cada patrón consiste en (i) una configuración de origen o estructura de elementos en el modelo de entrada KDM (N2), y (ii) una estructura de elementos de salida en el modelo destino (modelo BPMN, N3). La Tabla 10.2 muestra el conjunto de patrones definidos, la cual provee una aserción para cada patrón en lógica de predicados de primer orden así como una representación gráfica en el modelo de procesos de negocio destino.

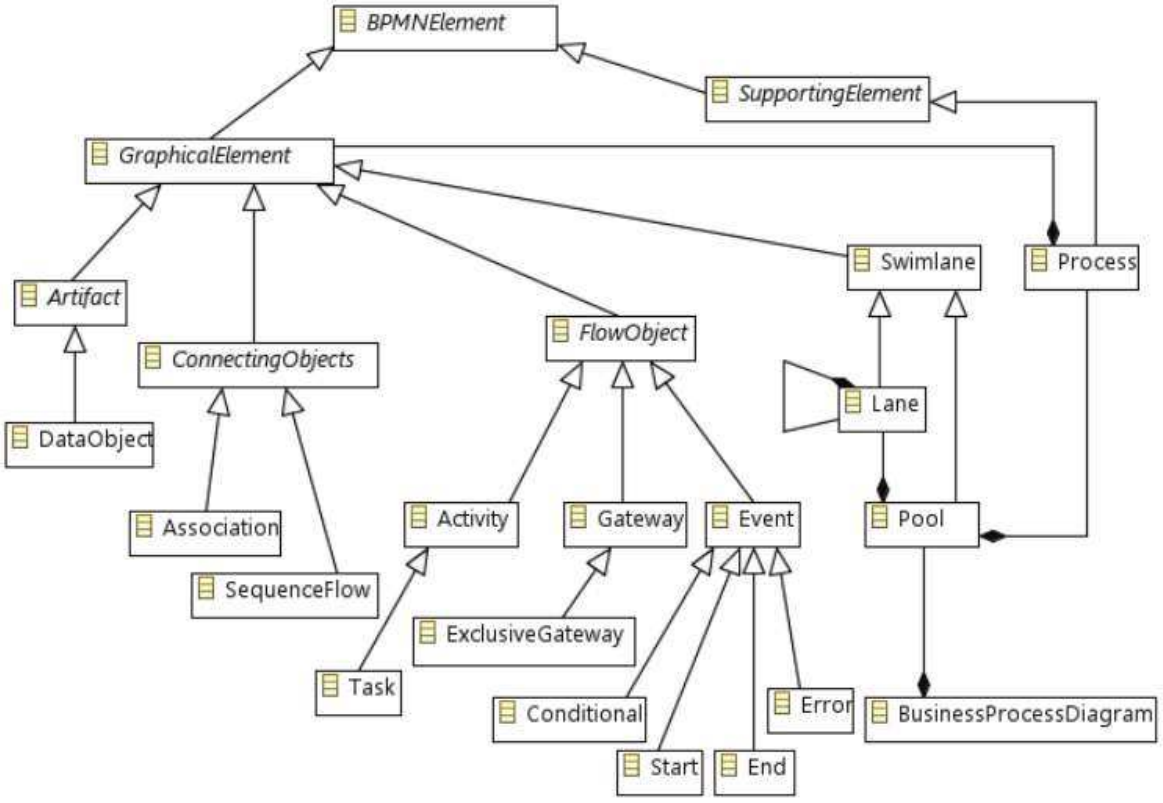
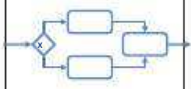
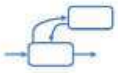
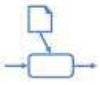


Figura 10.9. Metamodelo BPMN.

Patrón	Elementos detectados en N2	Structura en N3
P1. BPD Skeleton. Este patrón crea la estructura raíz del modelo BP. Crea un diagrama BP por cada modelo de código KDM. Este patrón también construye un elemento *pool* con un proceso anidado en el diagrama BP por cada paquete en el modelo KDM.	$\forall x,y\ CODE\_MODEL(x) \wedge COMPILATION\_UNIT(y) \wedge COMPILATION\_UNIT(y) \in CODE\_MODEL(x) \Rightarrow BPD(x) \wedge POOL(y) \wedge PROCESS(y) \wedge PROCESS(y) \in POOL(y) \in BPD(x)$	
P2. Sequence. Este patrón toma cualquier pieza de código invocable del modelo de código KDM y las mapea con tareas en el diagrama BP. Además, la secuencia de llamadas entre las unidades invocables es transformada en un conjunto de flujos de secuencia con el mismo orden entre las tareas.	$\forall x,\exists y\ CALLABLE\_UNIT(x) \wedge CALLABLE\_UNIT(y) \wedge CALL(x,y) \Rightarrow TASK(x) \wedge TASK(y) \wedge SEQUENCE\_FLOW(x,y)$	
P3. Branching. Este patrón transforma cada salto condicional en el código fuente con dos opciones mutuamente exclusivas en un nodo de decisión (*gateway*) exclusivo en el modelo BP. Típicamente este tipo de puntos condicionales están relacionados con las sentencias de tipo *if ... then ... else* o *switch* en varios lenguajes de programación.	$\forall x,\exists y,z\ CALLABLE\_UNIT(x) \wedge CALLABLE\_UNIT(y) \wedge CALLABLE\_UNIT(z) \wedge TRUE\_FLOW(x,y) \wedge FALSE\_FLOW(x,z) \Rightarrow EXCLUSIVE\_GATEWAY(x) \wedge TASK(y) \wedge TASK(z) \wedge SEQUENCE\_FLOW(x,y) \wedge SEQUENCE\_FLOW(x,z)$	
P4. Collaboration. Cada llamada a una unidad invocable externa (por ejemplo, librerías, componentes externos al sistema legado, etc.) es transformada en una tarea auxiliar con un flujo de secuencia de ida y vuelta.	$\forall x,\exists y\ CALLABLE\_UNIT(x) \wedge FOREIGN\_CALL(x,y) \Rightarrow TASK(x) \wedge TASK(y) \wedge SEQUENCE\_FLOW(x,y) \wedge SEQUENCE\_FLOW(y,x)$	
P5. Data Input. Este patrón construye un objeto de datos en el modelo BP por cada entrada de datos que es consumida por una unidad invocable en el modelo KDM, así como una asociación entre la tarea correspondiente y el objeto de datos. Este patrón no considera absolutamente todos los datos auxiliares y variables de programa usados por una unidad invocable.	$\forall x,\exists y\ CALLABLE\_UNIT(x) \wedge STORABLE\_UNIT(y) \wedge READ(x,y) \Rightarrow TASK(x) \wedge DATA\_OBJECT(y) \wedge ASSOCIATION(y,x)$	

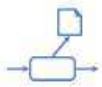
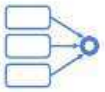
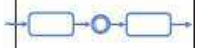
P6. Data Output. Similar al anterior, este patrón construye objetos de datos y asociaciones por cada dato de salida producido o generado por la respectiva unidad invocable en el modelo KDM.	$\forall x, \exists y\ CALLABLE\_UNIT(x) \wedge STORABLE\_UNIT(y)$ $\wedge WRITE(x,y) \Rightarrow TASK(x) \wedge DATA\_OBJECT(y)$ $\wedge ASSOCIATION(x,y)$	
P7. Start. Crea un punto de inicio en el modelo BP respecto a las unidades invocables iniciales en el modelo KDM construyendo flujos de secuencia desde el punto inicial a las tareas subyacentes.	$\forall x, y\ TOP\_CALLABLE\_UNIT(x)$ $\Rightarrow START\_EVENT(y) \wedge TASK(x)$ $\wedge SEQUENCE\_FLOW(y,x)$	
P8. Implicit Termination. Similar al patrón anterior, define el punto de finalización en el diagrama BPMN con flujos de secuencia desde las que son consideradas tareas finales.	$\forall x, \nexists y, \exists e\ TASK(x) \wedge SEQUENCE\_FLOW(x,y)$ $\wedge END\_EVENT(y) \Rightarrow SEQUENCE\_FLOW(x,e)$ $\wedge END\_EVENT(e)$	
P9. Conditional Sequence. Este patrón transforma cada llamada condicional en un flujo de secuencia condicional intermedio a lo largo de las tareas relacionadas con las unidades invocables.	$\forall x, \exists y, i, \nexists z\ CALLABLE\_UNIT(x)$ $\wedge CALLABLE\_UNIT(y) \wedge CALLABLE\_UNIT(z)$ $\wedge TRUE\_FLOW(x,y) \wedge FALSE\_FLOW(x,z)$ $\Rightarrow TASK(x) \wedge TASK(y)$ $\wedge CONDITIONAL\_EVENT(i)$ $\wedge SEQUENCE\_FLOW(x,i)$ $\wedge SEQUENCE\_FLOW(i,z)$	
P10. Exception. Cada llamada a una unidad invocable bajo cualquier condición excepcional se transforma en un flujo de llamada con evento de excepcionalidad de forma similar al patrón anterior.	$\forall x, \exists y, e\ CALLABLE\_UNIT(x)$ $\wedge CALLABLE\_UNIT(y)$ $\wedge EXCEPTION\_FLOW(x,y)$ $\Rightarrow TASK(x) \wedge TASK(y) \wedge ERROR\_EVENT(e)$ $\wedge SEQUENCE\_FLOW(x,e)$ $\wedge SEQUENCE\_FLOW(e,z)$	

Tabla 10.2. Patrones para obtención de elementos de procesos de negocio (N3) desde sistemas heredados (N2).

El conjunto de patrones se define en términos de KDM y BPMN, así la última transformación es independiente del lenguaje de programación y la plataforma del LIS en contraste con la primera y segunda transformación. Lo cual significa que el conjunto de patrones podría ser aplicado también a sistemas escritos en otros lenguajes de programación.

Los patrones son implementados por medio de *QVT relations*, ya que los patrones se implementan fácilmente de forma declarativa. Las transformaciones QVT consisten en una o más relaciones QVT por cada patrón. Cada relación QVT

define un dominio de entrada *checkonly* para describir los elementos que deberían ser reconocidos en el modelo KDM para disparar dicha regla. Además, cada relación QVT especifica un dominio de salida *enforce* que representa los elementos que han de ser creados cuando la regla es disparada.

La Figura 10.10 muestra como ejemplo cuatro relaciones QVT en notación gráfica. Al inicio, la relación *R1.CodeModel2BPD* es ejecutada y genera un diagrama de procesos de negocio por cada modelo KDM. La segunda relación *R2.Package2Pool* es invocada para cada paquete desde la cláusula de *R1*, la cual en QVT puede ser definida por medio de OCL (Object Constraint Language). La relación *R2* genera en el diagrama de procesos de negocio (el elemento raíz) un elemento *pool* que contiene un proceso de negocio, ambos con el mismo nombre que el paquete de acuerdo al patrón *P1. BPD skeleton*.

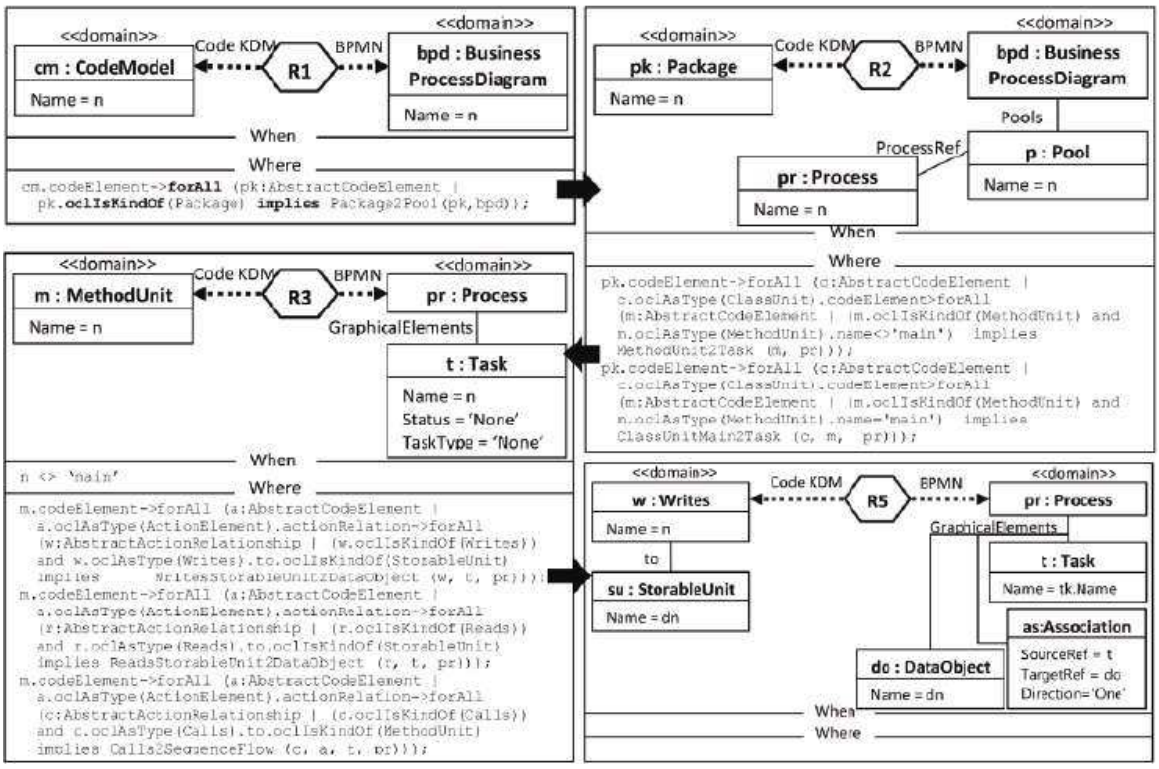


Figura 10.10. Ejemplo de cuatro relaciones QVT en representación gráfica de la transformación N2-a-N3.

Cada paquete del código fuente se considera la unidad mínima a ser transformada dentro de un modelo de proceso de negocio. Esta asunción puede conducir a procesos con bajo nivel de precisión. Sin embargo, la ventaja de esta solución es que esta no requiere de información inicial de correlación y provee automáticamente un esquema de los modelos de procesos de negocio subyacentes, los cuales pueden ser refinados por expertos posteriormente.

Al final de la ejecución de *R2*, la cláusula *where* invoca la relación *R3.MethodUnit2Task*, la cual crea la tarea relacionada con el método de acuerdo al patrón *P2. Sequence*. Esta relación básicamente transforma métodos en tareas e invocaciones entre métodos en flujos de secuencia de acuerdo con el principio *"a callable unit-a-task"* propuesto por [Zou et al., 2004]. Este enfoque es bueno para abordar problemas de granularidad ya que trata todos los métodos de forma homogénea. Finalmente, la relación *R3* invoca la relación *R5.WritesStorableUnit 2DataObject*. Esta relación implementa el patrón *P6. Data Output*, generando en el diagrama de procesos de negocio los objetos de datos asociados con las tareas obtenidas previamente.

Para finalizar con el ejemplo de esta sección, la Figura 10.11 muestra el modelo BPMN obtenido desde el modelo KDM del ejemplo después de ejecutar la transformación N2-a-N3. El modelo define una secuencia de tres tareas: Example, methodA y methodB, las cuales son obtenidas desde las tres instancias CallableUnit en el modelo KDM a través del patrón P2. Además, la invocación a la función 'println' desde el 'methodB' es transformada por medio de la aplicación del patrón P4 en una tarea con el mismo nombre y un flujo de ida y vuelta (tarea auxiliar).

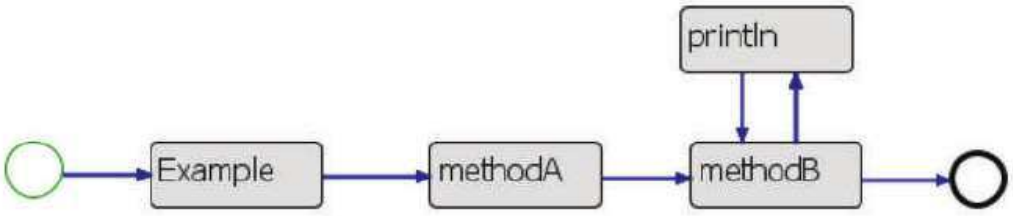


Figura 10.11. Un ejemplo de modelo BPMN obtenido después de la transformación N2-a-N3.

10.3 REFACTORIZACIÓN DE MODELOS DE PROCESOS DE NEGOCIO

Los modelos de procesos de negocio a refactorizar pueden proceder de la etapa anterior de ingeniería inversa a partir de código fuente, como es el caso de [Pérez-Castillo et al., 2011a]. En este caso, los modelos obtenidos son propensos a disponer de una menor calidad ya que cualquier mecanismo de ingeniería inversa se caracteriza por una perdida semántica cuando se aumenta el nivel de abstracción (en este caso desde sistemas de información existentes hasta procesos de negocio). En estos modelos es interesante realizar un proceso de refactorización con el fin de mejorar su calidad para hacerlos más entendibles al usuario, más fáciles de mantener y, por lo tanto, más fáciles de reutilizar.

Para realizar la refactorización de los modelos de procesos de negocio es necesario aplicar técnicas que detecten oportunidades de refactorización y apliquen dicha refactorización. En la actualidad existen diversas técnicas y enfoques de refactorización de modelos de procesos de negocio.

10.3.1 Desafíos para la calidad

La mejora de modelos de procesos de negocio entraña varios problemas que deben ser abordados. A continuación, se muestran los problemas más frecuentes que se identifican en los modelos de procesos de negocio obtenidos mediante ingeniería inversa.

10.3.1.1 COMPLETITUD

Al obtener los modelos de procesos de negocio mediante ingeniería inversa es posible que estos modelos estén incompletos. Esto es debido a que los datos pueden estar distribuidos en varias fuentes, no sólo en el propio código fuente, que no pueden ser obtenidos únicamente mediante un análisis estático.

El modelo de procesos de negocio puede presentar pérdida de nodos como tareas de negocio, *gateways*, eventos y objetos de datos, así como pérdida de conexiones como flujos de secuencia (entre tareas) y flujos de asociación (entre tareas y objetos de datos). Esta pérdida semántica afecta a la completitud del modelo [Overhage et al., 2012]. Todos estos elementos ausentes pueden no haber sido instanciados en tiempo de compilación y, por esa razón, pueden no estar presentes en el modelo de procesos de negocio. Por tanto, uno de los retos mayores es redescubrir aquellos elementos que no fueron recuperados en la fase de ingeniería inversa.

Otro reto importante relacionado con la completitud es establecer el orden entre las diferentes actividades de negocio. Desafortunadamente, la ingeniería inversa puede no proporcionar flujos de secuencia completos entre las actividades debido a que no toda la información puede ser derivada automáticamente a partir de código fuente. De forma similar, los puntos de inicio y de fin pueden no haber sido definidos en el modelo ya que no existe suficiente información para determinar qué actividades son el inicio o el final de un modelo o qué tarea es ejecutada antes que otra [Pérez-Castillo et al., 2011a].

10.3.1.2 GRANULARIDAD

Los diferentes tipos de granularidad presentes en las tareas de negocio es otro desafío importante. De acuerdo con el enfoque propuesto por [Zou et al., 2006],

cada *callable unit* (que puede ser traducida al español como unidad invocable) en un sistema de información es considerada una tarea de negocio candidata durante la ingeniería inversa. Sin embargo, los sistemas existentes normalmente contienen miles de *callable units* de diferente envergadura que por lo general son consideradas como tareas de negocio que pueden presentar diferentes niveles de granularidad [Pérez-Castillo et al., 2011c; Aalst et al., 2012]:

- Actividades pequeñas como los métodos *get* y *set* en programación orientada a objetos que sólo leen y escriben variables de programa, pero no realizan ninguna tarea de negocio real.
- Un conjunto de tareas pequeñas de negocio que tienen un comportamiento similar y desempeñan una tarea de negocio de forma conjunta.
- Un conjunto tareas pequeñas que dan soporte a otra tarea principal. Esta tarea principal es considerada la tarea *padre* y las tareas pequeñas son consideradas las *hijas*.

En los modelos de procesos de negocio obtenidos a partir de los sistemas de información es necesario reducir la granularidad de grado fino que hace que esos modelos sean aún cercanos a la perspectiva del código fuente.

[Polyvyanyy et al., 2010] propone abstraer esos modelos de procesos de negocio con el fin de que dicha generalización reduzca los detalles indeseados y que sea representada sólo la información que es significativa. En este trabajo se trata el tema de los diferentes tipos de granularidad, proponiendo dos técnicas para su solución: (1) Eliminación: eliminar aquellas tareas pequeñas que son consideradas como irrelevantes; y (2) Agregación: agrupar ciertas tareas con otras preservando la información.

10.3.1.3 INFORMACIÓN IRRELEVANTE

Cuando los modelos de procesos de negocio se obtienen a partir de los sistemas de información que los soportan y, en particular, desde código fuente, puede ser extraída cierta información que resulta irrelevante para el modelo de procesos de negocio. Esta información considerada como *ruido* producido durante el proceso de ingeniería inversa debería ser eliminada sin provocar una pérdida semántica, es decir, preservando la información relevante. Los elementos no relevantes, por ejemplo, tareas, eventos, *gateways*, etc., no llevan a cabo o soporta ninguna actividad de negocio de la organización.

La relevancia de los elementos presentes en un modelo de procesos de negocio es un aspecto importante ya que garantiza que el modelo contiene los elementos suficientes para transmitir su información [Overhage et al., 2012]. El reto, por tanto, es primero identificar partes relevantes y no relevantes y después descartar toda aquella información irrelevante presente en el modelo de procesos de negocio.

10.3.1.4 IDENTIFICADORES HEREDADOS

Como se ha dicho anteriormente, la mejora de la entendibilidad de un modelo de procesos de negocio es un aspecto importante a tener en cuenta ya que una pobre entendibilidad del modelo puede llevar a confusión. La entendibilidad normalmente suele empeorar en aquellos que han sido obtenidos mediante ingeniería inversa desde sistemas de información ya que los identificadores y nombres de los elementos recuperados pueden ser poco descriptivos. Esto se debe a que muchos identificadores son heredados desde los elementos de código fuente donde fueron inferidos.

Este aspecto está enfocado en la interpretabilidad desde una perspectiva del lenguaje, es decir, como de intuitivo es el lenguaje usado para definir los elementos del modelo. Por ese motivo, el nombrado de los elementos puede afectar negativamente a la interpretación que se realiza de un modelo cuando este nombrado no sigue convenciones apropiadas [Overhage et al., 2012]. Este problema debería ser solucionado mediante el renombrado de aquellos elementos de modelos de procesos de negocio que representan fielmente la semántica que en realidad desempeñan.

10.3.1.5 AMBIGÜEDAD

Otro reto a tener en cuenta en este tipo de modelos es la ambigüedad que puede presentarse. Este aspecto es importante para determinar la calidad de un modelo de procesos de negocio. La ambigüedad afecta a la entendibilidad y modificabilidad del modelo, es decir, cómo los elementos del modelo son intuitivamente formulados con respecto al contenido [Overhage et al., 2012]. La ambigüedad afecta negativamente a la habilidad de comunicar de forma eficiente el comportamiento del proceso de negocio.

Un modelo es considerado no ambiguo cuando está libre de redundancias y no contiene elementos que contradigan la lógica de negocio de otros elementos. Esto puede ocurrir debido a problemas de extracción de información irrelevante que a su vez es redundante respecto a otros elementos del modelo de proceso de negocio extraído. La ambigüedad debe ser abordada mediante la detección y eliminación de redundancias e inconsistencias en un modelo de procesos de negocio.

10.3.1.6 MARCO PARA LA MEJORA DE PROCESOS DE NEGOCIO

Tras los posibles defectos encontrados en los modelos de procesos de negocio, esta sección presenta como solución un marco de trabajo para la mejora de procesos de negocio obtenidos a partir de sistemas de información mediante ingeniería inversa denominado IBUPROFEN (***I**mprovement and **BU**siness **P**rocess **R**efactoring **OF E**mbedded **N**oise*). El objetivo de este marco de trabajo es mejorar los modelos de procesos de negocio obtenidos con el fin de reflejen, lo más fielmente posible, la realidad con unos niveles óptimos de calidad.

IBURPROFEN define una serie de medidas para realizar la evaluación de la calidad de un modelo, además define una serie de operadores de refactorización para abordar los retos descritos anteriormente.

10.3.1.7 MEDIDAS PARA EVALUAR LA CALIDAD DE LOS MODELOS DE PROCESO DE NEGOCIO

En esta sección se muestran una serie de medidas definidas para medir la entendibilidad y la modificabilidad de un modelo de procesos de negocio [Fernández-Ropero et al., 2012]. A continuación, se muestra su definición, así como sus abreviaturas que son utilizadas a lo largo del documento, divididas dependiendo la característica que evalúa.

Medidas para evaluar la entendibilidad

- **Número total de eventos** (TNE - *Total Number of events*): Esta variable es relativa al número total de eventos en un modelo de procesos de negocio, es decir, a la suma de eventos de inicio, eventos intermedios y eventos finales.
  - **Número total de eventos de inicio** (TNSE – *Total Number of Start Event*): Relativa únicamente a los eventos de inicio.
  - **Número total de eventos intermedios** (TNIE – *Total Number of Intermediate Event*): Relativa únicamente a los eventos intermedios.
  - **Número total de eventos de fin** (TNEE – *Total Number of End Event*): Relativa únicamente a los eventos de fin.
- **Número de objetos de datos de salida de actividades** (NDOOut - *Number of data objects which are outputs of activities*): Es el número de objetos de datos que son salida de actividades, es decir, que son el destino de un flujo de asociación en el que el origen es una actividad.
- **Número de objetos de datos de entrada de actividades** (NDOIn - *Number of data objects which are inputs of activities*): Es el número

de objetos de datos que son entrada de actividades, es decir, que son el origen de un flujo de asociación en el que el destino es una actividad.

- **Número de nodos** (NN - *Number of Nodes*): Esta variable es relativa al número de actividades y elementos de flujo en un modelo de procesos de negocio.
- **Grado promedio de puertas de enlace (Gateway)** (AGD - *Average Gateway Degree*): Expresa el promedio del número de arcos de entrada y salida que tiene una puerta de enlace.
- **Profundidad** (Dep - *Depth*): Es el máximo anidamiento de bloques estructurados en un modelo de procesos de negocio.

Medidas para evaluar la modificabilidad

- **Nivel de conectividad entre actividades** (CLA - *Connectivity level between activities*): Es el cociente entre el número total de actividades y el número de flujos de secuencia entre actividades.
- **Separabilidad** (Sep - *Separability*): Es el cociente entre el número de vértices de corte (como las puertas de enlace o los eventos intermedios) entre el número total de nodos del modelo de procesos de negocio.

Medidas para evaluar tanto entendibilidad como modificabilidad

- **Número total de puertas de enlace (Gateway)** (TNG - *Total Number of gateways*): Esta variable es relativa al número total de puerta de enlace en un modelo de procesos de negocio, sea cual sea su tipo.
- **Número de flujos de secuencia procedente de un evento** (NSFE - *Number of sequence flows from event*): Esta variable es relativa al número de flujos de secuencia que tiene como origen un evento.
- **Número de flujos de asociación** (NAF - *Number of association flows*): Es el número de flujos de asociación existentes en un modelo de procesos de negocio.
- **Número de flujos de secuencia procedentes de una puerta de enlace** (NSFG - *Number of sequence flows from gateways*): Esta variable es relativa al número de flujos de secuencia que tienen como origen una puerta de enlace.
- **Densidad** (Den - Density): Es el cociente del número total de arcos en un modelo de procesos de negocio entre el número máximo de arcos posibles.

- **Coeficiente de Conectividad** (CC - *Coefficient of Connectivity*): Es el cociente entre el número total de arcos en un modelo de procesos de negocio entre el número total de nodos.
- **Secuencialidad** (Seq - Sequentiality): Es el grado en que se construye el modelo mediante secuencias puras de tareas.

10.3.1.8 OPERADORES DE REFACTORIZACIÓN

IBUPROFEN considera los siguientes escenarios como los más relevantes en los que se hace necesario aplicar un operador de refactorización, que han sido recogidos de la literatura y adaptados al caso concreto de procesos de negocio obtenidos mediante ingeniería inversa. La figura adjunta muestra las refactorizaciones necesarias para cada uno de los escenarios y su resultado tras aplicar el operador de refactorización correspondiente agrupados en tres grupos. Los siguientes apartados detallan cada uno de los grupos y los operadores de refactorización contenidos en ellos.

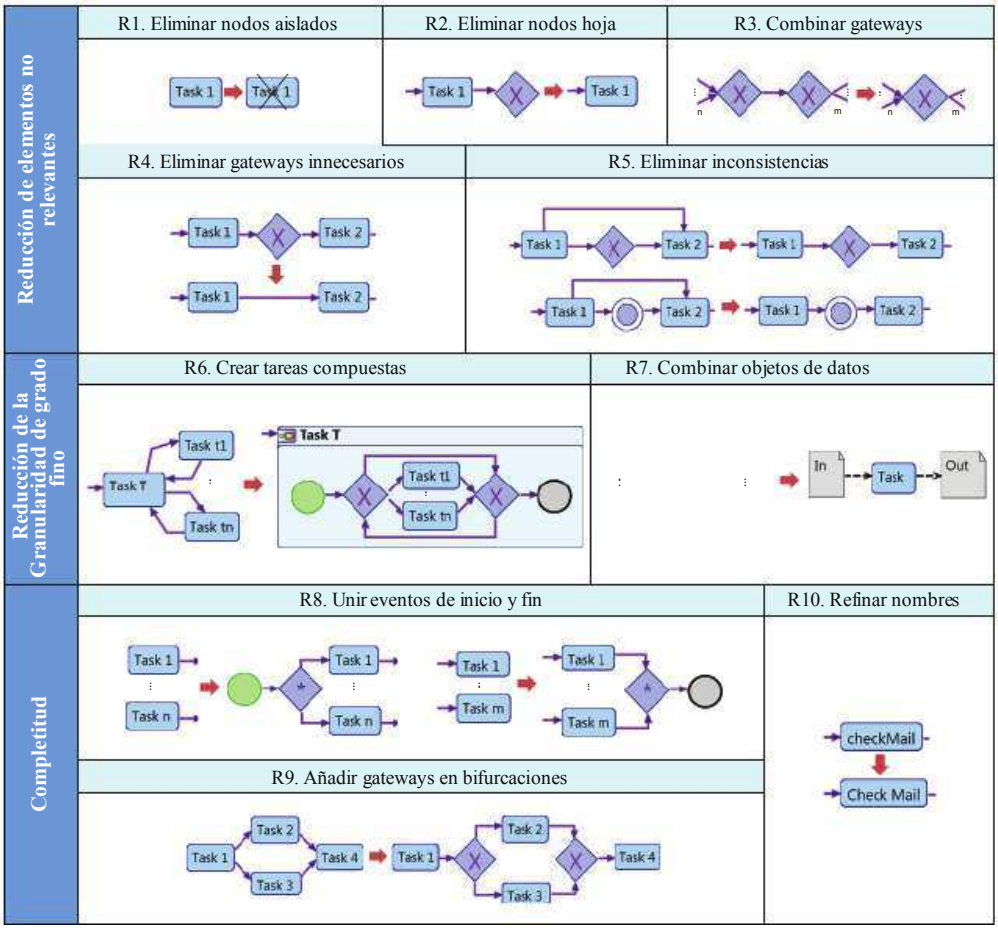


Tabla 10.3. Operadores de refactorización

10.3.2 Reducción de elementos no relevantes

Esta categoría agrupa aquellos operadores de refactorización encargados de eliminar las redundancias encontradas en los modelos de procesos de negocio tales como tareas aisladas, nodos hoja, anidamientos innecesarios de gateways o flujos inconsistentes.

Los operadores de refactorización presentes en esta categoría abordan el reto mencionado sobre la existencia de elementos no relevantes en el modelo dificulta su entendibilidad y modificabilidad.

En concreto esta categoría agrupa cinco operadores de refactorización que son mostrados en la Tabla 10.3 y que serán destallados a continuación. Estos operadores de refactorización abordan los desafíos de la presencia de información irrelevante así como de la presencia de ambigüedades.

R1. Eliminar nodos aislados

Un modelo de procesos de negocio debe representar la secuencia de pasos necesarios para conseguir los objetivos de una organización y debe representar únicamente los elementos que intervienen en dicha secuencia.

Al obtener los modelos de procesos de negocio mediante ingeniería inversa pueden presentarse nodos que no están conectados con ningún otro elemento. Estos elementos no forman parte de la lógica de negocio y es necesario proceder a su eliminación.

Esta refactorización se encarga de la labor de eliminar aquellos nodos (como tareas, gateways o eventos) en el modelo de procesos de negocio que no están conectados con ningún otro elemento.

R2. Eliminar nodos hoja

Al igual que en el caso anterior, los modelos de procesos de negocio obtenidos pueden presentar nodos cuya eliminación no provoca pérdida de información. Estos elementos pueden ser gateways o eventos intermedios que son representados como nodos hojas (sin flujos de salida) y que representan el final de un camino. Estos nodos no aportan lógica de negocio ya que no representan el cometido que deberían representar (servir de conexión entre varios nodos).

Esta refactorización se encarga de eliminar aquellos gateways o eventos intermedios que son representados como nodos hojas en el modelo de procesos de negocio, es decir, no disponen de flujos de salida.

R3. Combinar gateways

La presencia de varios gateways anidados provoca un aumento en la complejidad en el modelo que deriva en falta de entendibilidad del mismo.

Los modelos procedentes de los sistemas de información son más propensos a presentar esta anidación que aumenten la complejidad. En estos casos, se deben sustituir por alternativas equivalentes más fáciles de comprender por los usuarios, de forma que se aumente la entendibilidad del modelo y se haga su mantenimiento menos costoso.

Este operador de refactorización es el encargado de combinar gateways consecutivos del mismo tipo cuando el primero de ellos tiene sólo una salida y el segundo tiene sólo una entrada. Gráficamente puede verse en la Tabla 10.3.

R4. Eliminar gateways innecesarios

Los gateways son utilizados en la notación BPMN para controlar la divergencia y la convergencia de un flujo de secuencia, determinando la ramificación, la bifurcación, la fusión y la unión de caminos.

Los modelos de proceso de negocio pueden presentar gateways innecesarios que no aportan significado al modelo. Cuando un gateway conecta únicamente dos elementos es necesario eliminarlo ya que no representa ninguna convergencia o divergencia de caminos. Dos nodos conectados por un gateway representan el mismo comportamiento que un flujo de secuencia directo. En este caso, es conveniente eliminar dicho gateway ya que dificulta la entendibilidad del modelo.

Este operador de refactorización elimina gateways que conectan únicamente dos elementos. Además, crea un flujo de secuencia directo entre dichos elementos de forma que se mantenga la semántica pero evitando un elemento (gateway) innecesario.

R5. Eliminar inconsistencias

Un modelo de procesos de negocio debe estar libre de redundancias y contradicciones que dificulten su entendibilidad.

En estos modelos se pueden presentar elementos conectados mediante dos caminos, cada uno con un significado, que provoquen una inconsistencia en el modelo.

Este operador de refactorización implementa una heurística de manera que los caminos inconsistentes son solucionados. Cuando dos tareas están conectadas mediante un evento intermedio y mediante un flujo de secuencia directo se opta por eliminar el flujo de secuencia directo entre ellas ya que aporta menos restricciones que el primero. En el evento intermedio se añade la información de que en algunos casos puede no ser necesario esperar a ningún evento. Esta información se añade en la parte de documentación del elemento BPMN. Por otra parte, cuando dos tareas están conectadas mediante un gateway y mediante un flujo de secuencia directo se opta por eliminar el flujo directo ya que contradice el anterior. Nótese que este caso es distinto del caso anterior, en este caso el gateway no tiene por qué tener una única entrada y una única salida.

10.3.3 Reducción de la granularidad de grado fino

Esta categoría agrupa aquellos operadores de refactorización encargados de reducir la granularidad de grado fino presente en los modelos de procesos de negocio.

Los operadores de refactorización presentes en esta categoría abordan el reto mencionado anteriormente, en el que se presentaron los distintos tipos de granularidad que pueden estar presentes en un modelo, y cómo puede afectar a la entendibilidad y modificabilidad del modelo. A continuación, se presentan los dos operadores de refactorización pertenecientes a esta categoría que son mostrados en la Tabla 10.3.

R6. Crear tareas compuestas

Como ya se ha comentado, un modelo de procesos de negocio debe evitar disponer de tareas con distintos niveles de granularidad; el objetivo debe ser disponer de actividades de grano grueso que nos abstraigan de lo que es realizado por el sistema de información.

En este tipo de modelos se pueden presentar un conjunto de tareas pequeñas que dan soporte a una tarea principal. En este caso, la tarea principal es denominada *padre* y las demás son denominadas tareas *hijas*. Este escenario se puede ver fácilmente en aquellas actividades que tienen varias salidas y entradas de tareas.

Este operador de refactorización implementa una heurística para la detección de las tareas *padre* e *hijas*. Cuando una tarea *T* dispone de varias conexiones de ida y vuelta a varias subtareas *t1*... *tn* y éstas a su vez no tienen conexión con ninguna otra tarea más, el operador de refactorización crea una tarea compuesta *T*. Este escenario

refleja que en el código fuente un método llama a otros submétodos que devuelven un valor al primero. La tarea compuesta *T* representa un subproceso con un evento de inicio y de fin en el que las subtareas *t1... tn* son realizadas de forma exclusiva mediante la incorporación de gateways de división y unión.

R7. Combinar objetos de datos

Al igual que en el caso anterior, un modelo puede presentar elementos que no representen gran cantidad de información. Una tarea puede disponer de varios objetos de datos para su lectura y escritura que no representen gran cantidad de información. En ese caso, el operador de refactorización implementa una heurística para la combinación de los objetos de datos. Cuando *n* o más objetos de datos, siendo n el valor límite, son leídos, escritos o leídos y escritos por una única tarea, el operador de refactorización combina estos objetos de datos en uno solo. Se crea un objeto de dato de lectura, otro de escritura y otro de lectura y escritura, dependiendo de los objetos de datos que presente. En este caso, el límite propuesto es 2. Un ejemplo puede verse en la Tabla 10.3.

Con el fin de mitigar la pérdida semántica que conlleva combinar estos objetos de datos, todos los nombres de los objetos de datos agrupados son almacenados en el atributo *documentación* que proporciona el estándar BPMN.

10.3.4 Completitud

Esta última categoría agrupa aquellos operadores de refactorización encargados de completar los modelos de procesos de negocio, es decir, crear aquellos elementos que no han sido creados en el proceso de ingeniería inversa y son necesarios para su comprensión.

Los tres operadores de refactorización presentes en esta categoría abordan el reto de la completitud y los identificadores heredados.

R8. Unir eventos de inicio y fin

Los modelos de proceso de negocio especificados mediante el estándar BPMN deben comenzar con un evento de inicio y finalizar con un evento de fin con la finalidad de representar el flujo de trabajo completo realizado.

En el proceso de ingeniería inversa a partir de los sistemas de información pueden haberse perdido las conexiones entre las tareas iniciales y los eventos de inicio, así como las tareas finales y los eventos de fin, o incluso los eventos de inicio

y fin. Por ese motive, este operador de refactorización propone una heurística para añadir estas conexiones perdidas, así como crear dichos eventos en el caso de que no hubieran sido creados. El operador considera como tareas iniciales aquellas que no disponen de flujos de entrada, pero sí de salida. Estas tareas iniciales son conectadas al evento de inicio mediante un gateway complejo. De la misma forma, el operador considera como tareas finales aquellas que no disponen de flujos de salida, pero sí de entrada. Estas tareas finales son conectadas al evento de fin de la misma forma mediante un gateway complejo. Un ejemplo puede verse en la Tabla 10.3.

R9. Añadir gateways en bifurcaciones

En numerosos artículos recomiendan como buena práctica a la hora de modelar el usar siempre gateways divisores y unificadores [Weske, 2007]. La aplicación de estos gateways debe coincidir.

Al obtener los procesos de negocio a partir de ingeniería inversa existe la posibilidad de que los modelos no sigan las buenas prácticas recomendadas en el modelado de BPMN, es decir, una tarea tenga conexión directa con dos o más tareas sin la presencia de un gateway de división o, de forma inversa, varias tareas tengan conexión directa con otra sin presencia de un gateway de unión.

Este operador de refactorización detecta este escenario y añade un gateway exclusivo, tanto de unión como de división, entre estas tareas de la forma mostrada en la Tabla 10.3.

R10. Refinar nombres

Las actividades y procesos deben tener un nombre característico que revele su propósito, siendo la nomenclatura basada en un formato de verbo-objeto la más ampliamente conocida [Leopold et al., 2010].

Al obtener los procesos de negocio a partir de ingeniería inversa y teniendo en cuenta que el código fuente fue desarrollado siguiendo la nomenclatura oportuna (clases que comienzan en mayúscula, métodos que empiezan en minúscula, no espacios entre nombres, etc.) las actividades y procesos tendrán un nombre sin espacios basado en la nomenclatura *CamelCase* [Binkley et al., 2009] característica de los lenguajes de programación. Sin embargo, es necesario aplicar un algoritmo que separe las palabras y coloque el primer carácter en mayúscula con el fin de adecuarlas al lenguaje natural.

Este operador de refactorización, por tanto, implementa una heurística para mejorar los nombres de tal forma que separa las palabras teniendo en cuenta que una letra mayúscula indica el inicio de una nueva palabra y varias letras mayúsculas consecutivas indican una única palabra, normalmente siglas.

10.4 HERRAMIENTAS PARA LA ARQUEOLOGÍA DE PROCESOS DE NEGOCIO

Existen ciertas herramientas que de una u otra forma soportan los procesos de arqueología de procesos de negocio que pueden ayudar durante el proceso de mantenimiento Software. [Pérez-Castillo et al., 2011b] presenta MARBLE como una herramienta (que soporta la técnica con el mismo nombre) para la extracción de modelos de procesos de negocio.

MARBLE se ha desarrollado como un plug-in para Eclipse, lo que garantiza su aplicabilidad en la industria del software, así como su futura extensión. MARBLE está orientada a proyectos, por lo que todos los artefactos generados en cada una de las transformaciones se manejan dentro de un proyecto MARBLE. La Figura 10.12 muestra el aspecto de MARBLE, donde la parte A contiene los proyectos MARBLE creados con una jerarquía de carpetas que simbolizan cada uno de los niveles de abstracción; la parte B es el editor donde se muestran los modelos; la parte C corresponde al árbol sintáctico abstracto del código fuente mostrado en la parte B; y las partes D y E aportan detalles de los modelos y del resultado de las transformaciones.

Las transformaciones entre modelos se realizan de forma sucesiva, obteniendo los modelos de cada nivel de abstracción. La Figura 10.12 muestra los modelos generados por MARBLE al realizar las transformaciones. En primer lugar, se toma un fichero de código fuente (Figura 10.12 (A)). En segundo lugar, se obtiene el modelo de código fuente (Figura 10.12 (B)). En tercer lugar, esos modelos se integran en un único modelo KDM (Figura 10.12 (C)). Por último, los procesos de negocio son generados mediante la transformación del modelo KDM (Figura 10.12 (D)). Adicionalmente, MARBLE implementa mediante EMF/GMF (Eclipse/ Graphical Modeling Framework) un editor gráfico para visualizar los procesos de negocio (véase Figura 10.12 (E)). Esto facilita que los expertos de negocio puedan editar fácilmente los procesos de negocio para refinarlos o adaptarlos en un futuro y que puedan ser usados en procesos de mantenimiento software basados en la reingeniería.

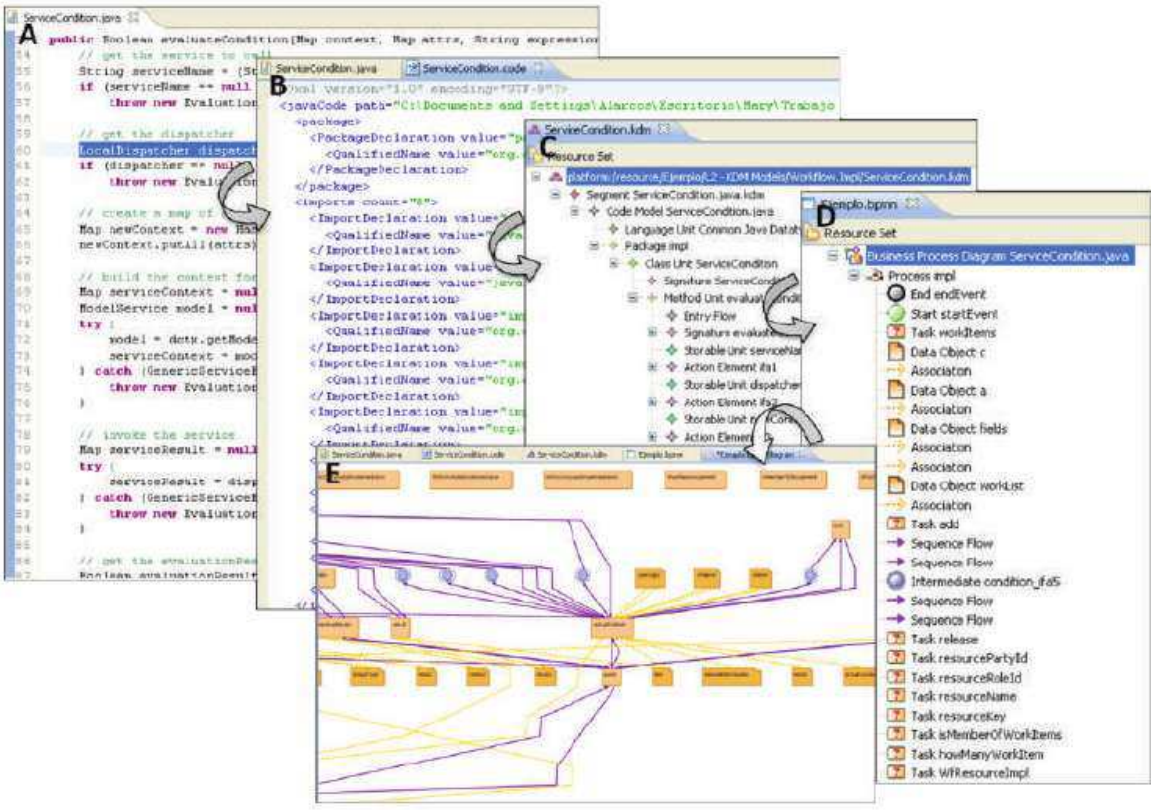


Figura 10.12. Artefactos generados por la herramienta MARBLE

10.5 LECTURAS RECOMENDADAS

- ✓ Pérez-Castillo, Ricardo, Ignacio García Rodriguez de Guzmán, and Mario Piattini. "Fundaments of Business Process Archeology." *Uncovering Essential Software Artifacts through Business Process Archeology*. IGI Global, 2014. 1-18.

- ✓ Arévalo Maldonado, Carlos, Isabel Ramos Román, and María José Escalona Cuaresma. "Discovering Business Models for Software Process Management-An Approach for Integrating Time and Resource Perspectives from Legacy Information Systems." *ICEIS 2015: 17th International Conference on Enterprise Information Systems (2015), p 353-359*. ScitePress Digital Library, 2015.

10.6 SITIO WEB

- ✓ Plug-In MARBLE para Eclipse: *https://marketplace.eclipse.org/content/marble*. Este sitio web presenta el plug-in que da soporte al proceso completo de recuperación de procesos de negocio a partir de sistemas heredados.

- ✓ Plug-In IBUPROFEN para Eclipse: *https://marketplace.eclipse.org/content/ibuprofen*. En este sitio web puede encontrarse el plug-in que implementa las transformaciones de refactorización para la mejora de la calidad de los procesos de negocio recuperados mediante técnicas de Business Process Archeology.

10.7 EJERCICIOS

Ejercicio 1

Explique cómo la arqueología de procesos de negocio influye o puede beneficiar el mantenimiento de software de sistemas de información heredados.

Ejercicio 2

¿Qué niveles de abstracción considera MARBLE para la arqueología de procesos de negocio?

Ejercicio 3

¿Qué rol juega el estándar del metamodelo KDM en la arqueología de procesos de negocio?

Ejercicio 4

¿Cuáles son los principales problemas de calidad de los modelos de procesos de negocio extraidos en procesos de arqueología (ingeniería inversa) y cómo pueden solucionarse?

Ejercicio 5

¿En qué consiste la refactorización de modelos de procesos de negocio?

ACRÓNIMOS

- **4GL.** Fourth Generation Language, Lenguaje de Cuarta Generación.
- **ADM.** Architecture-Driven Modernization, Modernización Dirigida por la Arquitectura
- **AENOR.** Asociación Española de Normalización y Certificación.
- **ANSI.** American National Standard Institute, Instituto Nacional Americano de Estándares.
- **BPMN.** Bueinss Process Modelling & Notation, Notación y Modelado de Procesos de Negocio
- **C/S.** Cliente/Servidor.
- **CASE.** Computer Aided Software Engineering, Ingeniería del Software Asistida por Computador.
- **COCOMO.** Cost Constructive Model, Modelo constructivo de costos.
- **E/R.** Entity/Relationship, Entidad/Interrelación.
- **E/S.** Entrada/Salida, Input/Output (I/O).
- **FIPS.** Federal Information Processing Standards, Estándares Federales para Procesamiento de la Información.
- **GCS.** Gestión de la Configuración del Software, Software Configuration Management (SCM).
- **HTML.** Hyper Text Markup Language, Lenguaje de Marcado Hiper Textual
- **IEC.** International Electrotechnical Commision, Comisión Electrotécnica Internacional.

- **IEEE.** Institute of Electrical and Electronics Engineers, Instituto de Ingenieros Eléctricos y Electrónicos.
- **IEEE-CS.** IEEE Computer Society, IEEE Sociedad Informática.
- **KDM.** Knowledge Discovery Metamodel, Metamodelo de Descubrimiento de Conocimiento
- **ISO.** International Organization for Standardization, Organización Internacional de Estándares.
- **LDC.** Líneas de Código.
- **MDA.** Model-Driven Architecture, Arquitectura Dirigida por Modelos
- **MDE.** Model-Driven Engineering, Ingeniería Dirigida por Modelos
- **MTTR.** Mean Time To Repair, Tiempo Medio para Reparar.
- **ODBC.** Open Database Conectivity, Conectividad Abierta a Bases de Datos.
- **OO.** Orientación a Objetos.
- **PCTE.** Portable Common Tools Environment, Entorno Portable Común de Herramientas.
- **POO.** Programación Orientada a Objetos.
- **PRO/SIM.** Prototipación/Simulación.
- **QVT.** Query/View/Transformation, Consulta/Vista/Transformación
- **SGBD.** Sistema de Gestión de Bases de Datos, DataBase Management System (DBMS).
- **SMR.** Sofware Modification Report, Informe de Modificación del Software.
- **SO.** Sistema Operativo.
- **SPICE.** Software Process Improvement and Capability dEtermination.
- **UML.** Unified Modeling Language, Lenguaje Unificado de Modelado
- **SQL.** Structured Query Language, Lenguaje de Consulta Estructurado.
- **XML.** eXtensible Markup Language, Lenguage de Marcado Extensible

BIBLIOGRAFÍA

[Aalst et al., 2003a]
Aalst, W. M. P. v. d., Hofstede, A. H. M. t., Kiepuszewski, B. y Barros., A. P. (2003). "Workflow Patterns." Distributed and Parallel Databases 14(3): 5-51.

[Aalst et al., 2003b]
Aalst, W. M. P. V. D., Hofstede, A. H. M. T. y Weske, M. (2003). Business process management: a survey. Proceedings of the 2003 international conference on Business process management. Eindhoven, The Netherlands, Springer-Verlag: 1-12.

[Aalst et al., 2012]
van der Aalst, W. (2012). "Process Mining: Overview and Opportunities." ACM Transactions on Management Information Systems (TMIS) 3(2): 7.

[Abran y Nguyenkim, 1991]
Abran, A. y Nguyenkim, H., «Analysis of Maintenance Work Categories Through Measurement». *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 1991, pp. 104-113.

[AFOTEC, 1989]
Air Force Operational Test and Evaluation Center Publication 800-2, Vol. 3: Software Maintainability–Evaluation Guide. Estados Unidos, 1989.

[Alarcos, 2009]
Alarcos Research Group. (2009). "PRECISO. A Reverse Engineering Tool to Discover Web Services from Relational Databases." Retrieved 5/30/2011, 2011, from http://alarcos.esi.uclm.es/per/rpdelcastillo/PRECISO/PRECISO.html.

[Albretch, 1979]
Albretch, A. J., «Measuring application development productivity». *Proceedings of the IBM application development symposium*. Monterrey, Canadá, oct. 1979

[Albretch, 1983]
Albretch, A. J., «Software function, source lines of code and development effor prediction: a software science validation». *IEEE Transactions on Software Engineering,* nov. 1983.

[Alexander, 2001]
Alexander, I. (2001). from http://www.scenarioplus.org.uk/papers/reqts_in_uml/reqts_in_uml.htm

[Aniche et al., 2016]
Aniche, M., Bavota, G., Treude, C., Van Deursen, A., & Gerosa, M. A. (2016, October). A Validated Set of Smells in Model-View-Controller Architectures. In Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on (pp. 233-243). IEEE.

[Arnold, 1986]
Arnold, R. S., «An Introduction to Software Restructuring». *Tutorial on Software Restructuring*. IEEE Computer Society, 1986, pp. 1-11.

[Arnold, 1989]
Arnold, R. S., Software Restructuring. *Proceedings of IEEE,* vol 77, nº 4, abril 1989, pp. 607-617.

[Arnold, 1992]
Arnold, R., *Software Reengineering*, IEEE Press, 1992.

[Arnold, 1993]
Arnold, R. S., *Software Reengineering*. Ed. IEEE Computer Society Press. Estados Unidos, 1993.

[Arvanitou et al., 2017]
Arvanitou, E. M., Ampatzoglou, A., Chatzigeorgiou, A., Galster, M. y Avgeriou, P. (2017). "A mapping study on design-time quality attributes and metrics." J. Syst. Softw. 127(C): 52-77.

[Banker *et al.*, 1993]
Banker, R. D., Datar, S.M., Kemerer, C. F. y Zweig, D., «Software complexity and manintenance costs». *Communications of the ACM*, vol. 36, nº 11, nov. 1993.

[Bardou, 1997]
Bardou, L., *Mantenimiento y Soporte Logístico de los Sistemas Informáticos*. Ed. RA-MA. España, 1997.

[Barranco y Granja, 1996]
Barranco García, M. y Granja Álvarez, J. C. «Maintainability as a factor key in maintenance productivity: a case study». *International Conference on Software Maintenance,* nov. 1996.

[Barros *et al.*, 1995]
Barros S., Bodhuin, T., Escudié, A., Queille, J. P. y Vidrot, J. F., Supporting Impact Analysis: A Semi-Automated Technique and Associated Tool. En *Proceedings of International Conference on Software Maintenance*. Ed. IEEE Computer Society. Estados Unidos, 1995.

[Basili *et al.*, 1996]
Basili, V., Briand, L., Condon, S., Kim, Y., Melo, W. Y Valett, J. D. (1996). «Understanding and Predicting the Process of Software Maintenance Releases». En *Proceedings of the Interna-tional Conference on Software Engineering*. Los Alamitos, California: IEEE Computer Society, pp. 464-474.

[Baxter y Pigdeon, 1997]
Baxter, I. D. y Pidgeon, W.D. «Software Change Through Design Maintenance». En *Proceedings of the International Conference on Software Engineering*. Los Alamitos, California: IEEE Computer Society, pp. 250-259.

[Baxter *et al.*, 1998]
Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M. y Bier, L. «Clone Detection Using Abstract Syntax Trees». *Proceedings of the International Conference on Software Maintenance*. Los Alamitos, California: IEEE Computer Society, pp. 368-377.

[Bennet *et al.*, 1991]
Bennet, K. H., Cornelius, B., Munro, M. Y Robson, D. «A change analysis process to characterize software maintenance projects». *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, pp. 38-49.

[Bennett *et al.*, 1990]
Bennett, K. H.; Martil, R. y Zuylen H. V., «A Model of Software Reconstruction». *Centre of Software Maintenance*. Durham, Reino Unido, 1990.

[Bennett, 1991]
Bennett, K. H., «Automated Support of Software Maintenance». *Information and Software Technology,* vol. 33, nº 1, enero 1991, pp. 74-85.

[Bennett y Rajlich]
Bennett, K. H. y V. T. Rajlich (2000). Software maintenance and evolution: a roadmap. Proceedings of the Conference on The Future of Software Engineering. Limerick, Ireland, ACM.

[Bermudez et al., 2017]
Bermudez Ruiz, F.J., García Molina, J. y Díaz García, O. (2017). On the application of model-driven engineering in data reengineering. Inf. Syst. 72:136-160

[Berns, 1984]
Berns, G. «Assessing Software Maintainability». *Communications of the ACM*, enero 1984.

[Bertrand y Bezivin, 2000]
Bertrand, T. y Bézivin, J. (2000): Ontological Support for Business Process Improvement. En Bustard, D., Kawalek, P. y Norris, M. (editores): Systems Modeling for Business Process Improvement. Artech House Publishers, capítulo 20, pp. 313-331.

[Bianchi et al, 2003]
Bianchi, A., Caivano, D., Marengo, V. y Visaggio, G. (2003). "Iterative Reengineering of Legacy Systems." IEEE Trans. Softw. Eng. 29(3): 225-241.

[Biggerstaff *et al.*, 1994]
Biggerstaff, Ted. J., Mitbander, Bharat G. y Webster, Dallas E., «Program Understanding and the Conccept Assignment Problem». *Communications of the ACM*, vol. 37, nº 5, mayo 1994.

[Binkley et al., 2009]
Binkley, D., Davis, M., Lawrie, D. y Morrell, C. (2009). To camelcase or under_ score, IEEE.

[Boehm, 1979]
Boehm, B.W., «Sofware Engineering - R&D Trends and Defense Needs». *Research Directions in Software Technology*. Ed. P. Wegner, MIT Press, 1979, pp. 44-86.

[Boehm, 1981]
Boehm, B. W., *Software Engineering Economics*. Ed. Prentice-Hall, USA 1981.

[Borne y Romanczuk, 1998]
Borne, I. y Romanczuk, A., «Towards a Systematic Object-Oriented Transformation of a Merise Analysis». En Nesi y Lehner (eds.), *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*. Los Alamitos, California: IEEE Computer Society, pp. 213-325.

[Briand *et al.*, 1996]
Briand, L., Morasca, S. y Basili, V., «Property-based software engineering measurement». *IEEE Transactions on Software Engineering*, vol. 22, nº 1, ene. 1996.

[Briand *et al.*, 1998]
Q-MOPP: «Qualitative Evaluation of Maintenance Organizations, Processes and Products». *Software Maintenance: Research and Practice*, vol 10(4), pp. 249-278.

[Brito e Abreu et al., 1994]
Brito e Abreu, F., y Carapuça, R. (1994). Object-Oriented Software Engineering: Measuring and controlling the development process. Paper presented at the Proceedings of the 4th International Conference on Software Quality, McLean (USA).

[Brito e Abreu et al., 1996]
Brito e Abreu, F., y Melo, W. (1996). Evaluating the Impact of Object-Oriented Design on Software Quality. Paper presented at the Proceedings of 3rd International Metric Symposium.

[Brown et al, 2005]
Brown, P., P. Haas, J. Myllymaki, H. Pirahesh, B. Reinwald y Y. Sismanis. (2005). Toward Automated Large-Scale Information Integration and Discovery. Data Management in a Connected World (Lecture Notes in Computer Science) 3551. 2005, pp. 161-180.

[Bryan y Siegel, 1984]
Bryan, W. L. y Siegel S. G., *Software Product Assurance, Techniques for Reducing Software Risk*. Ed. Elsevier, 1984.

[Bucci *et al.*, 1998]
Bucci, G., Fioravanti, F., Nesi, P. y Perlini, S. «Metrics and Tool for System Assessment». *Proceedings of the International Conference on Software Engineering*. Los Alamitos, California: IEEE Computer Society, pp. 36-46.

[Bull, 1994]
Bull, T., *Software Maintenance by Program Transformation in a Wide Spectrum Language*. Tesis doctoral. Universidad de Durham, Reino Unido 1994.

[Calero *et al.*, 1999]
Calero, C. y Piattini, M. *Caracterización formal de métricas para bases de datos relacionales.* Actas de las IV Jornadas de Ingeniería del Software y Bases de Datos. Cáceres.

[Calero et al., 2001]
Calero, C., Piattini, M., y Genero, M. (2001). Empirical Validation of Referential Integrity. Information and Software Technology, 43, 949-957

[Calero et al., 2006]
Calero, C., F. Ruiz, A. Baroni, F. Brito e Abreu, M.Piattini (2006). "An Ontological Approach To Describe the SQL:2003 Object-Relational Features." Journal of Computer Standards and Interfaces 28(6): 695-713.

[Calero y Piattini, 2015]
Calero, C., Piattini, M. (Eds.). Green In Software Engineering. 2015. Springer, London

[Calzolari *et al.*, 1998]
Calzolari, F., Tonella, P. y Antoniol, G. «Modelling Maintenance Effort by Means of Dynamic Systems». *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*. Los Alamitos, California: IEEE Computer Society, pp. 150-156.

[Canfora y Di Penta, 2007]
Canfora, G. y Penta, M. D. (2007). New Frontiers of Reverse Engineering. 2007 Future of Software Engineering, IEEE Computer Society.

[Canning, 1972]
Canning, R., «The Maintenance Iceberg». *EDP Analyzer,* vol. 10, nº 10, octubre 1972.

[Card y Glass, 1990]
Card, D. N. y Glass, R. L., *Measuring Software Design Quality*. Englewood Cliffs. Estados Unidos, 1990.

[Cardoso de Mello, 2012]
Cardoso de Mello, M. Agile processes for the maintenance cycle. A smarter work cycle for a Smarter Planet. IBM DeveloperWorks. March 06, 2012.

[Chapin, 1987]
Chapin, N., «The Job of Sofware Maintenance». En *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society. Estados Unidos, 1987.

[Chen et al, 2018]
Chen, Z., L. Chen, W. Ma, X. Zhou, Y. Zhou and B. Xu (2018). "Understanding metric-based detectable smells in Python software: A comparative study." Information and Software Technology 94: 14-29.

[Chen et al., 2017]
Chen, Z., Mohanavilasam, M., Kwon, Y. W. y Song, M. (2017). Tool Support for Managing Clone Refactorings to Facilitate Code Review in Evolving Software. 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC).

[Chidamber y Kemerer, 1994]
Chidamber, R. S. y Kemerer, C. F. «A metrics suite for object oriented design». *IEEE Transactions on Software Engineering*, vol. 20, nº 6, jun. 1994.

[Chikofsky y Cross, 1990]
Chikofsky, E. J. y Cross, J. H., «Reverse Engineering and Design Recovery: A Taxonomy». *IEEE Software,* vol. 7, nº 1, 1990, pp. 13-17.

[Choudhari y Suman, 2005]
Choudhari, J. y U. Suman. An Empirical Evaluation of Iterative Maintenance Life Cycle Using XP. SIGSOFT Softw. Eng. Notes 40(2): pp 1-14. 2005.

[Coleman *et al.*, 1994]
Coleman, D., Ash, D., Lowther, B. y Oman, P., «Using metrics to evaluate software system maintainability». *IEEE Computer*, agosto 1994, pp. 44-9.

[Cremer, 1998]
Cremer, K. «A Tool Supporting the Re-Design of Legacy Applications». *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*. Los Alamitos, California: IEEE Computer Society, pp. 142-148.

[Cruz-Lemus et al., 2005]
Cruz-Lemus, J.A., Genero, M., y Piattini, M. (2005). Metrics for UML Statechart Diagrams Metrics for Software Conceptual Models (pp. 237-272): Imperial College Press.

[Cruz-Lemus et al., 2007]
Cruz-Lemus, J.A. (2007). A Measurement-Based Approach for Assessing UML Statechart Diagrams Understandability: Tesis Doctoral, Universidad de Castilla-La Mancha.

[Cunha et al., 2016]
Cunha, J., Fernandes, J. P., Martins, P., Mendes, J., Pereira, R., Saraiva, J,. (2016). Evaluating refactorings for spreadsheet models. Journal of Systems and Software 118: 234-250.