# MASTER
# EN INGENIERÍA DEL SOFTWARE

## UPM

*2007*

*Francisco Ruiz, Macario Polo*

# MANTENIMIENTO
# DEL SOFTWARE

# Índice de Contenidos

Índice de Contenidos .......... I
Índice de Figuras .......... V
Índice de Tablas .......... VII
Presentación .......... IX

**1. Introducción .......... 1**

1.1. Definición de mantenimiento .......... 1
1.1.1. Mantenimiento correctivo .......... 2
1.1.2. Mantenimiento adaptativo .......... 3
1.1.3. Mantenimiento perfectivo .......... 4
1.1.4. Mantenimiento preventivo .......... 4
1.2. Actividades de mantenimiento .......... 4
1.3. Costes del mantenimiento .......... 7
1.4. Dificultades del mantenimiento .......... 10
1.4.1. Código heredado .......... 10
1.4.2. Problemas del mantenimiento .......... 11
1.4.3. Efectos secundarios del mantenimiento .......... 12
1.4.3.1. Efectos secundarios sobre el código .......... 12
1.4.3.2. Efectos secundarios sobre los datos .......... 13
1.4.3.3. Efectos secundarios sobre la documentación .......... 13
1.5. Soluciones al problema del mantenimiento .......... 14
1.5.1. Soluciones de gestión .......... 14
1.5.1.1. Recursos dedicados al mantenimiento .......... 14
1.5.1.2. Gestión de la calidad .......... 15
1.5.1.3. Gestión estructurada del mantenimiento .......... 15
1.5.1.4. Organización del equipo humano .......... 16
1.5.1.5. Documentación de los cambios .......... 17
1.5.2. Soluciones técnicas .......... 18
1.6. Mantenibilidad .......... 20
1.7. Estándares .......... 21

**2. El Proceso de Mantenimiento .......... 23**

2.1. Procesos del ciclo de vida del software .......... 23
2.1.1. Procesos principales .......... 23
2.1.2. Procesos de soporte .......... 26
2.1.3. Procesos organizacionales .......... 27
2.2. Actividades y tareas del proceso de mantenimiento .......... 29

2.2.1. Implementación del proceso ........ 30
2.2.2. Análisis de problemas y modificaciones ........ 30
2.2.3. Implementación de las modificaciones ........ 30
2.2.4. Revisión y aceptación del mantenimiento ........ 30
2.2.5. Migración ........ 31
2.2.6. Retirada de software ........ 31
2.3. El mantenimiento en el borrador del estándar ISO/IEC 14764 ........ 32
2.4. Otras consideraciones sobre el proceso de mantenimiento ........ 35
**3. Mantenibilidad del Software ........ 37**
3.1. Concepto de mantenibilidad del software ........ 38
3.1.1. Aspectos que influyen en la mantenibilidad ........ 38
3.1.2. Atributos de mantenibilidad del código fuente ........ 39
3.1.3. Propiedades de la mantenibilidad ........ 41
3.1.3.1. Reparabilidad ........ 41
3.1.3.2. Flexibilidad ........ 41
3.2. Efectos de los cambios en el software ........ 42
3.2.1. Efectos sobre la complejidad ........ 43
3.2.2. Efectos sobre la mantenibilidad ........ 43
3.3. Estándar ISO/IEC 9126 ........ 44
3.4. Medida de la mantenibilidad ........ 45
3.4.1. Medidas externas de la mantenibilidad ........ 47
3.4.2. Medidas internas de la mantenibilidad ........ 48
**4. El estandar ISO 14764 ........ 49**
4.1. Actividades y Tareas ........ 50
**5. Técnicas para el Mantenimiento ........ 55**
5.1. Introducción y conceptos básicos ........ 55
5.2. Ingeniería inversa de programas ........ 57
5.2.1. Identificación y recopilación de componentes funcionales ........ 59
5.2.2. Asignación de valor semántico a los componentes funcionales ........ 60
5.3. Reconstrucción de programas ........ 62
5.3.1. Reestructuración ........ 62
5.3.1.1. Un ejemplo: reestructuración de C++ a C++ ........ 63
5.4. Ingeniería inversa y reingeniería de bases de datos relacionales ........ 65
5.4.1. Metodología de diseño "hacia delante" ........ 65
5.4.2. Recuperación del diseño de bases de datos ........ 66
5.5. Ingeniería inversa y reingeniería de interfaces de usuario ........ 70
5.6. Costes y beneficios de la reingeniería ........ 71
5.6.1. Justificación del proyecto de Reingeniería ........ 71
5.6.2. Análisis de la cartera de aplicaciones ........ 72
5.6.3. Estimación de costes ........ 74
5.6.4. Análisis de costes/beneficios ........ 74
5.7. Otras técnicas: detección de clones ........ 75

**6. Metodologías y Gestión del Mantenimiento ........ 79**
6.1. Planteamiento de la metodología MANTEMA ........ 79
6.2. Descripción de las tareas ........ 80
6.3. Estructura detallada de MANTEMA ........ 82
6.3.1. Actividades y tareas iniciales comunes ........ 82
6.3.2. Actividades y tareas del mantenimiento no planificable ........ 83
6.3.3. Actividades y tareas del mantenimiento planificable ........ 84
6.3.4. Actividades y tareas finales comunes ........ 84
6.3.5. Documentación ........ 86
6.3.6. Métricas ........ 86
**7. Referencias Bibliográficas ........ 87**

# Índice de Figuras

Figura 1. Las fuentes del mantenimiento del software. .................................................... 2
Figura 2. Origen de los defectos del software. ................................................................. 3
Figura 3. Costes relativos de cada tipo de mantenimiento. ............................................. 5
Figura 4. Coste relativo aproximado de detectar y corregir defectos. .............................. 9
Figura 5. Procesos del ciclo de vida software según la norma ISO 12207..................... 24
Figura 6. Procesos generales y procesos de soporte según la norma ISO 12207. .......... 29
Figura 7. Proceso de mantenimiento de ISO/IEC (1999). .............................................. 34
Figura 8. Jerarquía en la Mantenibilidad. ....................................................................... 40
Figura 9. Calidad de un producto software (modelo ISO 9126). .................................. 45
Figura 10. Peticiones de modificación y tipos de mantenimiento. ................................. 49
Figura 11. El proceso de mantenimiento según ISO. ..................................................... 50
Figura 12. Ingeniería directa, Ingeniería inversa, Reingeniería y Redocumentación..... 57
Figura 13. Código funcionalmente correcto, pero poco legible. .................................... 57
Figura 14. Código equivalente al de la figura anterior, pero legible y bien documentado. ..................................................................................................................... 58
Figura 15. Ejemplo de comentario no actualizado. ........................................................ 61
Figura 16. Inserción de una función dentro de una clase. .............................................. 63
Figura 17. Encapsulación de un bloque de código en una función. ............................... 64
Figura 18. Fases en la Ingeniería inversa de una BD relacional.................................... 66
Figura 19. Selección de grupos de tablas según sus claves primarias. ........................... 68
Figura 20. Diagrama de elementos abstractos. ............................................................... 69
Figura 21. Vista general del proceso de ingeniería inversa de BBDD relacionales. ...... 70
Figura 22. Diagrama para el análisis de la cartera de aplicaciones. ............................... 73
Figura 23. Medidas de calidad obtenidas por una de las herramientas de Sneed. .......... 73
Figura 24. Dos fragmentos de código semánticamente similares, pero sintáctica y léxicamente diferentes. .............................................................................................. 76
Figura 25. Un fragmento de programa y su posible representación como un árbol de sintaxis abstracta. ........................................................................................................ 77
Figura 26. Algoritmo básico para la detección de clones. .............................................. 77
Figura 27. Vista general de la metodología MANTEMA. ............................................. 80
Figura 28. Procesos con los que se establece interfaz. ................................................... 81
Figura 29. Estructura del mantenimiento planificable.................................................... 85

# Índice de Tablas

Tabla 1. Distribución del esfuerzo en las actividades de mantenimiento......................... 6
Tabla 2. Evolución de los costes del mantenimiento........................................................ 7
Tabla 3. Principales actividades y tareas del mantenimiento según ISO 12207. ........... 32
Tabla 4. Niveles de capacidad y atributos en SPICE. .................................................... 36
Tabla 5. Atributos de mantenibilidad del código fuente. ............................................... 40
Tabla 6. Actividades y tareas del proceso de mantenimiento según ISO....................... 54
Tabla 7. Diferentes métodos para la ingeniería inversa de BB.DD. relacionales........... 67
Tabla 8. Métodos propuestos en ingeniería inversa de tablas relacionales. ................... 69
Tabla 9. Parámetros a considerar en proyectos de Reingeniería. ................................... 75
Tabla 10. Estructura del mantenimiento no planificable (sigue).................................... 83
Tabla 11. Estructura del mantenimiento no planificable (continuación)........................ 84
Tabla 12. Algunos de los documentos generables durante el mantenimiento y descritos en MANTEMA. ............................................................................................ 86

# Presentación

Este documento está basado en gran parte en el libro sobre el mismo tema publicado por los autores en la editorial Ra-Ma, cuyos datos son los siguientes:

Piattini, M. y otros.

Mantenimiento del Software: Modelos, técnicas y métodos para la gestión del cambio.

Diciembre 2000. Rústica, 336 Págs. ISBN: 8478974482.

http://www.ra-ma.es/libros/0001532.htm

# 1. Introducción

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). A finales de los sesenta ya se había popularizado el término Crisis del Software para referir esta situación. Los síntomas de esta crisis han estado repercutiendo desde entonces en la industria de desarrollo de software y todavía se sienten sus efectos. Para resolver el problema ha surgido un área de la informática que recibe el nombre de Ingeniería del Software [Sommerville, 1992].

Una de las principales causas de esta situación ha sido la poca importancia que se le ha dado al proceso de Mantenimiento 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 al Mantenimiento del Software, sus características, importancia (especialmente en términos económicos), problemática y soluciones.

## 1.1. Definición de mantenimiento

El estándar IEEE 1219 [IEEE, 1993] 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, 1995] se establece que "el Proceso de Mantenimiento contiene las actividades y tareas realizadas por el mantenedor. Este proceso se activa cuando el producto software sufre modificaciones en el código y la documentación asociada, debido a un problema o a la necesidad de mejora o adaptación. El objetivo es modificar el producto software existente preservando su integridad. Este proceso incluye la migración y retirada del producto software. El proceso termina con la retirada del producto software". El mantenedor es la organización que proporciona el servicio de mantenimiento.

En otras fuentes bibliográficas clásicas aparecen definiciones similares a las anteriores. Por ejemplo, Pressman [1998] dice que "la fase 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. 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], produciendo demandas de mantenimiento adaptativo y perfectivo respectivamente.

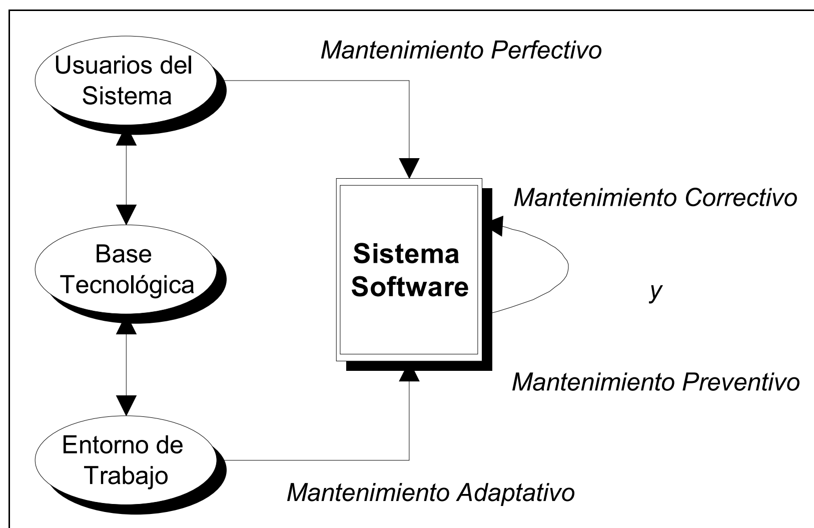



Figura 1. Las fuentes del mantenimiento del software.

### 1.1.1. 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 en un sistema 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.

En la Figura 2 se muestra una distribución de las causas de los defectos según un estudio estadístico realizado por Grady [1994]. Según dicho estudio 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.

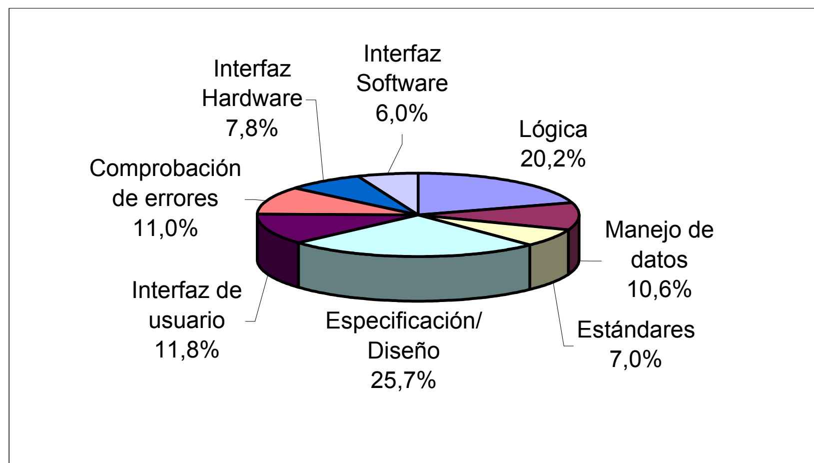



Figura 2. Origen de los defectos del software.

### 1.1.2. 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 al entorno de desarrollo del software (incorporación de nuevos elementos o herramientas como ODBC). 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 ficheros clásico y sustituirlo por un sistema de gestión de bases de datos relacionales.
- 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 cada dos años, nuevos sistemas operativos -ó versiones de los antiguos- que se anuncian regularmente, 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 diez años [Pressman, 1993].

### 1.1.3. 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.
- 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.

### 1.1.4. 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 pueden incluir sentencias que comprueben la validez de los datos de entrada, reestructurar los programas para mejorar su legibilidad, o 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.

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 mas 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 mas generalizada entre los gestores, usuarios, e incluso entre los propios 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 3, tomada de Frazer [1992]).

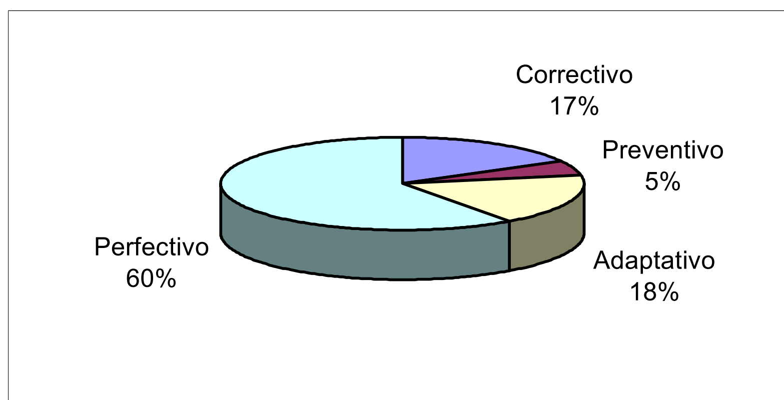



Figura 3. Costes relativos de cada tipo de mantenimiento.

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 reproducimos la distribución media de esfuerzo de los cinco proyectos:

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 perfectivo.
- La proporción de esfuerzo empleado en inspección, certificación y consultoría es mucho mayor en el caso de mantenimiento perfectivo que en 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.<br>Comprensión del cambio. | 13% |
| Diseño | Diseño del cambio.<br>50% de inspección, certificación y consultoría. | 16% |
| Implementación | Codificación y pruebas unitarias.<br>50% de inspección, certificación y consultoría. | 29% |
| Pruebas | Pruebas de integración.<br>Pruebas de aceptación.<br>Pruebas de regresión. | 24% |
| Otras | Documentación del sistema.<br>Otra documentación.<br>Otras actividades. | 18% |

Tabla 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]. La tendencia es creciente con el paso del tiempo, tal como puede observarse en la Tabla 2, en la cual se indica el porcentaje que supone el mantenimiento respecto del coste total.

| Referencia | Fechas | % Mantenimiento |
|---|---|---|
| [Pressman, 1993] | años 70 | 35%-40% |
| [Lientz y Swanson, 1980] | 1976 | 60% |
| [Pigoski, 1997] | 1980-1984 | 55% |
| [Pressman, 1993] | Años 80 | 60% |
| [Rock-Evans y Hales, 1990] | 1987 | 67% |
| [Schach, 1990] | 1987 | 67% |
| [Pigoski, 1997] | 1985-1989 | 75% |
| [Frazer, 1992] | 1990 | 80% |
| [Pressman, 1993] | Años 90 (prev.) | 90% |

Tabla 2. Evolución de los costes del mantenimiento.

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.

Los estudios sobre la situación del mercado comercial del mantenimiento del software son relativamente escasos:

- En Francia [Bardou, 1997], el “Gabinete Pierre Audouin Conseil” ha señalado unos gastos en software de 48 miles de millones de francos en 1991, de los cuales, el mantenimiento supuso 34 miles de millones, es decir, mas del 70%.
- En España, el estudio sobre las Tecnologías de la Información en España, elaborado por el Ministerio de Industria y Energía y la Asociación Española de Empresas de Tecnologías de la Información calcula que en 1998 el mantenimiento del software supuso 41.793 millones de pesetas, a los que habría que añadir una parte importante de los 53.708 millones reseñados en el epígrafe de externalización (outsourcing) [MINER-SEDISI, 1999].

Hace pocos años, el famoso "Efecto 2000" ha confirmado 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 han realizado en este tema. En el caso de la Unión Europea, se une también otra segunda situación de necesidad de mantenimiento de software a gran escala: la adaptación a la implantación de la nueva moneda, el Euro. Ambos casos son situaciones caracterizadas porque debe realizarse el mantenimiento de miles de aplicaciones software, con decenas de millones de líneas de código.

El Ministerio de Administraciones Públicas de España ha estimado unas inversiones, sólo en el ámbito de la administración general (central) del estado, de 178 millones de euros (29.586 millones de pesetas) [MAP, 1999]. A nivel mundial, el informe de Gartner Group sobre el Efecto 2000, publicado en el segundo cuatrimestre de 1999, estimó unos costes totales de 600.000 millones de dólares [Gartner, 1999].

Son varias las causas de que en la mayoría de las organizaciones actuales 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 10 ó 20 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 (ver Figura 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.

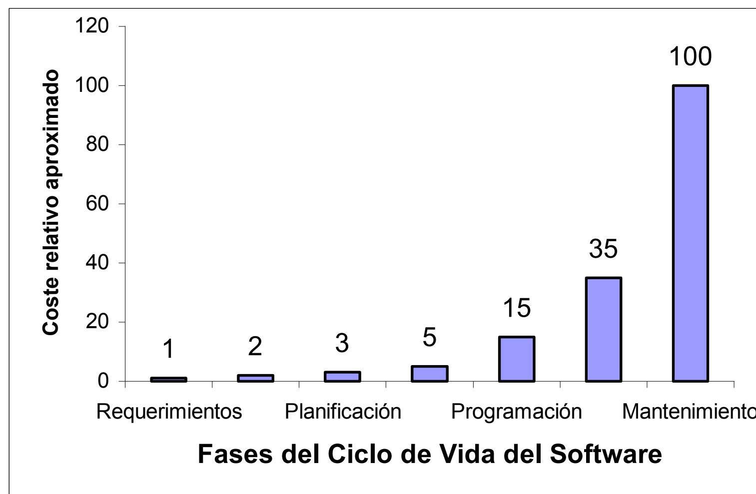



Figura 4. Coste relativo aproximado de detectar y corregir defectos.

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 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

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. La pregunta a formular es la siguiente: ¿cómo debe mantenerse el software para preservar su fiabilidad? A continuación veremos las circunstancias que hacen que la respuesta a esta pregunta no sea fácil y esté muy 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 éste 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 el 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 O*rganizacional – 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 constituyen 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 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; y
- Que 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.

#### 1.5.1.1. 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. 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.

### 1.5.1.2. 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.

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

### 1.5.1.3. 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, dibujar 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.
- Señalar en el código las instrucciones cambiadas.
- Asegurarse antes de eliminar código (guardando una copia aparte por si acaso).
- 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.

### 1.5.1.4. 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:

a) *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.

b) *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.

c) *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.

### 1.5.1.5. 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 informaciones que deben recogerse para cada cambio ([Swanson, 1976], [Jorgensen, 1995], [Basili et al., 1996], [Briand et al., 1998]), aunque puedan diferir en algunas de ellas. Briand et al. [1998] proponen recoger las siguientes 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.
        - 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 programa 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 (test) 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 y la reestructuración:

- Reingeniería consiste en el examen y modificación de un sistema para reconstruirlo en una nueva forma [Bennett et al., 1990].
- 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 y cambiar o menos susceptible de incluir errores en cambios posteriores [Arnold, 1986]. Se diferencia de la ingeniería inversa en que el software reestructurado tiene el mismo nivel de abstracción que el original.

Estas tres técnicas tienen relación entre ellas y a menudo se utilizan varias conjuntamente. Así, por ejemplo, la reingeniería y la ingeniería inversa aplicadas al software suelen implicar la reestructuración del código de un programa existente (por ejemplo, troceándolo en módulos y procedimientos). Los programas pueden ser reestructurados sin necesidad de afectar al nivel del diseño lógico. Aunque los programas pueden ser transformados a un nivel de abstracción semántica mayor sin necesidad de ser reestructurados, es recomendable realizar dicha reestructuración porque se simplifica el proceso de transformación. Diversos autores [Bennett et al., 1990] han propuesto más de 20 utilidades de la ingeniería inversa de programas agrupadas en tres categorías: depuración de programas, cambio de programas y comprensión de programas. Todas estas técnicas pueden ser automatizadas, es decir, soportadas por computador. En el capítulo 5 se realiza un estudio detallado del uso de la ingeniería inversa y la reingeniería en el mantenimiento del software.

También han sido propuestos otros métodos diferentes de los tres citados. Por ejemplo, [Miller, 1979] propuso el método de Retroajuste Estructurado, definido por el propio autor como “la aplicación de las técnicas de programación estructurada de hoy a los sistemas de ayer para atender las demandas de mañana”.

La Transformación de Programas es un método formal para la construcción o modificación de programas que también ha sido propuesto como solución parcial para el problema del mantenimiento del software [Bull, 1994]. Este método parte de una especificación o programa ya existente para obtener otro programa equivalente por medio de una serie de transformaciones sucesivas. Estas transformaciones consisten en un cambio realizado en el texto fuente (código) del programa de forma que la semántica del programa no cambia. También se caracterizan por preservar la exactitud de cualquier programa al que se apliquen (lo cual es demostrable formalmente). Un sistema de transformación de programas es similar a un compilador en el sentido de que ambos pueden traducir un programa de más alto nivel (o especificación) a uno de más bajo nivel semánticamente equivalente al primero.

# 1.6. Mantenibilidad

Todas las características del mantenimiento del software -anteriormente reseñadas- están afectadas por la mantenibilidad o facilidad de mantenimiento del software. Esta propiedad del software se puede definir como la medida cualitativa de la facilidad de comprender, corregir, adaptar y/o mejorar el software [Pressman, 1993].

Los factores que influyen en la mantenibilidad son los siguientes [Kopetz, 1979]:

- Falta de cuidado en las fases de diseño, codificación o prueba.
- Pobre configuración del producto software.
- Adecuada cualificación del equipo de desarrolladores del software.
- Estructura del software fácil de comprender.
- Facilidad de uso del sistema.
- Empleo de lenguajes de programación y sistemas operativos estandarizados.
- Estructura estandarizada de la documentación.
- Documentación disponible de los casos de prueba.
- Incorporación en el sistema de facilidades de depuración.
- Disponibilidad del equipo (computador y periféricos) adecuado para realizar el mantenimiento.
- Disponibilidad de la persona o grupo que desarrolló originalmente el software.
- Planificación del mantenimiento.

El último factor señalado es, seguramente, el que más influye positivamente en la mantenibilidad del software. Si vemos el software como un producto que estará sujeto a cambios casi con total seguridad, las etapas previas del ciclo de vida se podrán realizar de forma que aumenten las posibilidades de producir software más fácilmente mantenible.

Aunque la mantenibilidad del software es una propiedad difícil de cuantificar, puede medirse indirectamente realizando una evaluación cuantitativa de los atributos que la caracterizan. Para ello, algunos autores han propuesto diversas clases de métricas de mantenibilidad:

- De *esfuerzo*: En Gilb [1979] se propone evaluar el esfuerzo requerido durante la fase de mantenimiento mediante métricas que indican el tiempo dedicado a las diversas tareas realizadas;
- De *complejidad*: Kafura y Reddy [1987] sugieren utilizar métricas de complejidad para evaluar la mantenibilidad; y
- De *estructura*: Rombach [1987] analiza la correlación entre la estructura de un programa y su facilidad de mantenimiento.

Dada la importancia de la mantenibilidad para evaluar los costes de mantenimiento y para planificar el proceso de mantenimiento, se dedica todo el capítulo 3 para estudiarla con detalle.

## 1.7. Estándares

En los últimos años se ha realizado un considerable esfuerzo en la comunidad internacional para elaborar estándares, tanto para el ciclo de vida completo del software como para la fase de mantenimiento.

Diversos organismos e instituciones han elaborado propuestas para contemplar el objetivo de la mantenibilidad en el proceso de desarrollo del software. En González [1995] se puede encontrar una presentación de los principales aspectos sobre la normalización del software y en particular sobre las normas para aseguramiento de su calidad (incluyendo la mantenibilidad). También se analizan las actividades que actualmente desarrollan en esta área diversas instituciones internacionales (ISO, CEN, OTAN y ANSI).

Algunas de las propuestas formuladas son las siguientes:

- U.S. Gobierno Federal [FIPS, 1984].
- U.S. Air Force [AFOTEC,1989].
- National Institute of Science and Technology [Osborne, 1989].
- IEEE Computer Society [IEEE, 1992], [IEEE, 1993].
- ISO/IEC [ISO/IEC, 1998a], [ISO/IEC, 1995].
- Agencia Espacial Europea [Mazza et al., 1994].

Fruto de estos trabajos ha sido la publicación de diversos estándares por parte de IEEE y de ISO que tienen relación directa o indirecta con el problema del mantenimiento del software:

- Para los procesos del ciclo de vida del software: ISO 12207, IEEE 1074.
- Para la calidad del software y sus métricas: ISO 9126, IEEE 1061.
- Para el mantenimiento del software: IEEE 1219 y el nuevo estándar ISO 14764 [ISO/IEC, 1999].

Los estándares para los procesos del ciclo de vida del software nos permiten integrar y asociar el proceso de mantenimiento con los demás procesos existentes para el software. Los estándares de calidad del software interesan en mantenimiento del software porque los factores de calidad del software (especialmente la complejidad y la mantenibilidad) inciden directamente sobre el esfuerzo de mantenimiento necesario.

# 2. El Proceso de Mantenimiento

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.

## 2.1. Procesos del ciclo de vida del software

Como señalamos en el capítulo 1, recientemente se han publicado dos estándares sobre los procesos del ciclo de vida del software, ISO/IEC 12207 e IEEE 1074. El primero de ellos "establece un marco de referencia común para los procesos del ciclo de vida software, con una terminología bien definida, que puede ser referenciada por la industria software" [ISO/IEC 1995]. En este marco se definen los procesos, actividades (que forman cada proceso) y tareas (que constituyen cada actividad) presentes en la adquisición, suministro, desarrollo, operación y mantenimiento del software

A continuación presentamos de forma resumida la norma ISO, en la que nos basaremos para exponer las actividades y tareas del proceso de mantenimiento. Esta norma agrupa, como puede verse en la Figura 5, las actividades que pueden ser realizadas durante el ciclo de vida del software en cinco procesos principales, ocho procesos de soporte, y cuatro procesos organizativos, así como el proceso de adaptación.

### 2.1.1. Procesos principales

Como se muestra en la Figura 5, ISO/IEC 12207 define cinco procesos principales, de cuya ejecución se encargan las denominadas partes principales. En la norma se considera "parte principal" la que inicia o realiza uno de los cinco procesos principales, por lo cual las partes principales son: el adquiriente, el suministrador, el desarrollador, el operador y el personal de mantenimiento del software. Los procesos principales son los comentados a continuación.

Figura 5. Procesos del ciclo de vida software según la norma ISO 12207.

Proceso de adquisición

Este proceso comienza definiendo la necesidad de adquirir un sistema o un producto software y continúa con la preparación y publicación de la solicitud de propuestas, la selección de un suministrador y la gestión de los procesos de adquisición hasta la aceptación del producto.

Proceso de suministro

Este proceso puede iniciarse bien por una decisión de preparar una propuesta para responder a una petición de un adquiriente, bien por la firma de un contrato con el adquiriente para proporcionar el producto software. El proceso continúa con la identificación de los procedimientos y recursos necesarios para gestionar y asegurar el proyecto, incluyendo el desarrollo de los planes del proyecto y la ejecución de los planes hasta la entrega del producto software.

Proceso de desarrollo

Este proceso contiene las actividades para el análisis de requisitos, diseño, codificación, integración, pruebas, e instalación y aceptación relativas al software. El desarrollador selecciona y realiza, o presta apoyo, a las siguientes actividades de acuerdo con el contrato. Como en el mantenimiento se vuelven a aplicar las actividades del desarrollo, vamos a analizarlas de forma más detallada.

Según la norma ISO 12207, el proceso de desarrollo consta, entre otras, de las siguientes actividades:

- *Análisis de los requisitos* del software. Se establecen, documentan y evalúan los requisitos del software, incluyendo la especificación de las características de calidad tales como: especificaciones funcionales y de aptitud, interfaces externas, requisitos de cualificación, especificaciones de prevención y seguridad, interacción hombre-máquina, restricciones de personal, documentación de usuario. El desarrollador evaluará los requisitos del software. También se llevan a cabo reuniones de revisión y se establece una línea base para los requisitos de los elementos de la configuración una vez que se pasen las revisiones con éxito.
- *Diseño de la arquitectura* del software. Se transforman los requisitos en una arquitectura que describa su estructura de alto nivel e identifique los componentes principales. Incluye el diseño de alto nivel para la interfaz externa con otros elementos y para la base de datos, así como la elaboración de la versión preliminar de los manuales de usuario y de los requisitos de las pruebas preliminares y la planificación para la integración del software.
- *Diseño detallado* del software. Se realiza un diseño detallado para cada componente software (incluyendo bases de datos e interfaces), que será refinado en niveles inferiores conteniendo unidades de software que puedan ser codificadas, compiladas y probadas. También se documentan los requisitos de prueba, incluidas las de sobrecarga (stress).
- *Codificación* del software *y pruebas*. Se desarrollan y documentan cada unidad de software y la base de datos, así como los procedimientos de prueba y los datos para probarlas. Posteriormente se llevan a cabo las pruebas y se evalúan sus resultados.
- *Integración* del software. Se integran las unidades y los componentes software junto con las pruebas, según se vayan desarrollando los conjuntos de acuerdo con el plan de integración. También se actualizan los manuales de usuario.
- *Prueba de cualificación* del software. Se lleva a cabo la prueba de cualificación de acuerdo con los requisitos especificados.
- *Instalación* del software, se instala el software de acuerdo con el plan de instalación.
- *Soporte a la aceptación* del software. El desarrollador presta ayuda a la revisión de aceptación y prueba que realice el adquiriente del software, completando la documentación y el código del software.

Proceso de operación

Este proceso abarca la operación del software y el soporte a usuarios. Debido a que la operación del software se integra en la operación del sistema, las actividades y tareas del proceso de operación se refieren al sistema.

Proceso de mantenimiento

Este proceso se activa cuando el software sufre modificaciones de código o de documentación asociada debido a un error, una deficiencia, un problema o la necesidad de mejora/adaptación. Se trata en profundidad en este mismo capítulo.

### 2.1.2. Procesos de soporte

Como ya hemos señalado, la norma contiene ocho procesos, que pueden ser empleados por los procesos de adquisición, suministro, desarrollo, operación, mantenimiento o cualquier otro proceso de soporte. Estos procesos se emplean en varios puntos del ciclo de vida y pueden ser realizados por la organización que los emplea, por una organización independiente (como un servicio), o por un cliente como elemento planificado o acordado del proyecto.

Proceso de documentación

El proceso de documentación registra la información producida por un proceso o actividad del ciclo de vida. Este proceso contiene el conjunto de actividades que planifican, diseñan, desarrollan, producen, editan, distribuyen y mantienen los documentos necesarios para todos los interesados tales como directores, ingenieros y usuarios del sistema.

Proceso de gestión de configuración

Este proceso es bastante conocido en general en el ámbito de la Ingeniería de Software y, según ISO, consta, además de la implementación del propio proceso (en la que se incluye la elaboración del plan de gestión de configuración), de las actividades de:

- Identificación de la configuración.
- Control de la configuración.
- Contabilidad del estado de la configuración.
- Evaluación de la configuración.
- Gestión y entregas de liberaciones (releases).

Proceso de aseguramiento de la calidad

Este proceso proporciona el aseguramiento adecuado de que los procesos y productos de soporte del ciclo de vida del proyecto cumplen con los requisitos especificados y que se ajustan a los planes establecidos. La norma destaca que el aseguramiento de la calidad puede ser interno o externo dependiendo de que se demuestre la calidad del producto o procesos a la dirección del proveedor o del adquiriente.

Proceso de verificación

El proceso de verificación determina si los requisitos de un sistema o software son completos y correctos y si los productos de software en cada fase de desarrollo cumplen los requisitos o condiciones impuestos sobre ellos en las fases previas. La verificación puede integrarse en el proceso que la emplee.

Proceso de validación

Este proceso sirve para determinar si los requisitos y el software construido cumplen con su uso proyectado. Al igual que el anterior, este proceso puede ejecutarlo una organización independiente del proveedor, desarrollador, operador o personal de mantenimiento.

Proceso de revisión conjunta

El proceso de revisión conjunta define las actividades, que emplea cualquiera de las partes, para evaluar el estado y los productos de las actividades.

Proceso de auditoría

El proceso de auditoría permite determinar el cumplimiento de los requisitos, planes y contrato según sea apropiado.

Proceso de resolución de problemas

Este proceso sirve para analizar y resolver los que se descubren durante el ciclo de vida del software. El objetivo de este proceso según la norma ISO es "proporcionar los medios oportunos, responsables y documentados para asegurar que todos los problemas descubiertos se analizan y resuelven y que se reconocen las tendencias".

Las actividades de las que consta el proceso son: la implementación del proceso y la resolución de problemas. El estándar de ISO destaca que este proceso debe cumplir una serie de requisitos:

- Ser un bucle cerrado que asegure que los problemas se detectan y se solucionan.
- Contener un esquema para categorizar y priorizar problemas.
- Debe llevar a cabo análisis para detectar tendencias en los problemas.
- Debe evaluar las resoluciones y disposiciones.

### 2.1.3. Procesos organizacionales

Estos procesos los emplea una organización para llevar a cabo funciones tales como gestión, formación del personal o mejora del proceso. Estos procesos ayudan a establecer, implementar y mejorar, consiguiendo una organización más eficiente. Se

lleva a cabo normalmente a nivel organizacional (corporativo), fuera del ámbito de proyectos y contratos específicos.

Proceso de gestión

Este proceso contiene las actividades y tareas genéricas que puede emplear cualquier parte en la dirección de sus respectivos procesos; comprende la iniciación y definición del alcance, planificación, ejecución y control, revisión y evaluación, y cierre.

Proceso de infraestructura

Este proceso establece la infraestructura necesaria para cualquier otro proceso, que puede incluir hardware, software, herramientas, técnicas, normas e instalaciones para desarrollo, operación o mantenimiento.

Proceso de mejora

El proceso de mejora sirve para establecer, valorar, medir, controlar y mejorar los procesos del ciclo de vida del software.

Proceso de formación

El proceso de formación sirve para proporcionar y mantener un personal formado.

Proceso de adaptación

Este proceso permite realizar la adaptación básica de la norma ISO a distintos proyectos de software, teniendo en cuenta las variaciones en las políticas y procedimientos organizacionales, los métodos y estrategias de adquisición, el tamaño y complejidad de los proyectos, los requisitos de sistema y métodos de desarrollo, etc. Este proceso consta de las siguientes actividades:

- Identificar el entorno del proyecto.
- Solicitar entradas (inputs).
- Seleccionar procesos, actividades y tareas.
- Documentar decisiones de adaptación y sus causas.

En la Figura 6 se resumen los procesos del ciclo de vida según ISO, mostrándolos bajo diferentes "visiones" (puntos de vista) de la utilización de la norma: contrato, gestión, operación, ingeniería y soporte.

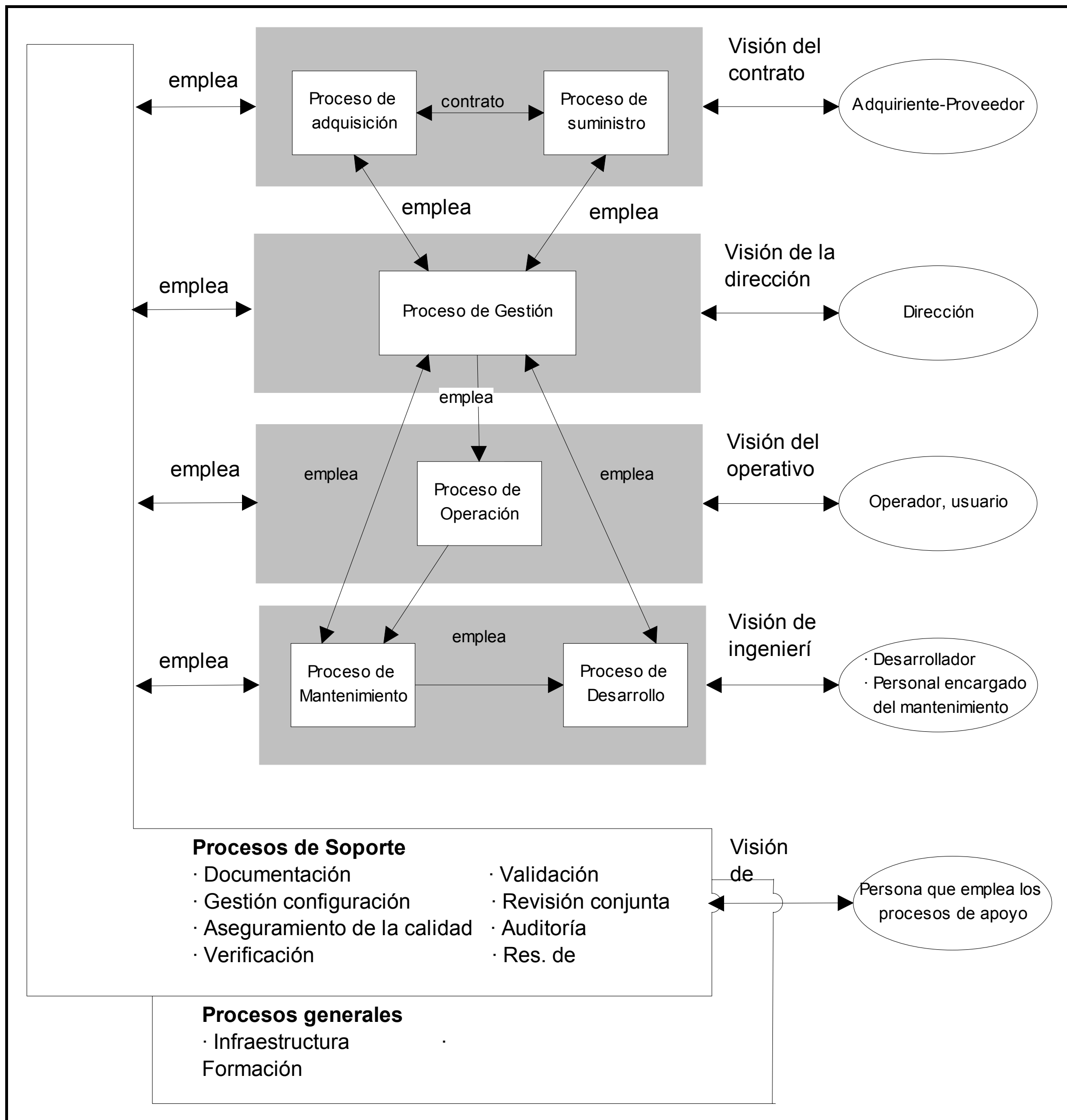



Figura 6. Procesos generales y procesos de soporte según la norma ISO 12207.

## 2.2. Actividades y tareas del proceso de mantenimiento

Como hemos visto en el apartado anterior, la norma de ISO considera el mantenimiento como uno de los procesos principales del ciclo de vida del software, ya que “define las actividades de la organización (maintainer) 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”. En la definición de ISO el mantenimiento incluye la migración y la retirada del software y consta de las siguientes actividades:

### 2.2.1. Implementación del proceso

En esta actividad se desarrollan los planes correspondientes para llevar a cabo las actividades y tareas del mantenimiento. También debe definir los procedimientos necesarios para la gestión de problemas y petición de modificaciones (empleando el proceso de resolución de problemas), e implementar el proceso de gestión de configuración para gestionar las modificaciones al sistema existente.

### 2.2.2. Análisis de problemas y modificaciones

Esta actividad consiste en analizar los problemas o peticiones de modificación con el fin de evaluar su impacto en el sistema y la organización existentes, determinando el tipo de modificación (preventiva, correctiva, etc.), su alcance (tamaño, coste, tiempo, etc.) y su criticidad (rendimiento, seguridad, etc.).

La organización encargada del mantenimiento debe también verificar el problema, elaborar distintas opciones para implementar las modificaciones, y documentar el problema o la petición de modificación, así como los resultados del análisis y las opciones de implementación. Por último, deberá obtener la aprobación para la opción seleccionada.

### 2.2.3. Implementación de las modificaciones

En esta actividad se incluyen todas las tareas relativas a determinar qué documentación, unidades de software y versiones deben modificarse, y se utiliza el proceso de desarrollo para implementar las modificaciones.

Los requisitos del proceso de desarrollo deberán complementarse, según el estándar, de la siguiente manera:

- Se deberán definir y documentar los criterios de evaluación y prueba para probar y evaluar las partes del sistema (unidades, componentes y elementos de la configuración) modificadas y no modificadas.
- Se deberá asegurar la completa y correcta implementación de los requisitos nuevos y modificados que no se vean afectados los requisitos originales no modificados. También se deberán documentar los resultados de las pruebas.

### 2.2.4. Revisión y aceptación del mantenimiento

Esta actividad consiste en la revisión de la integridad del sistema modificado, que llevará a cabo la organización encargada del mantenimiento junto con la organización que autorizó la modificación.

La organización encargada del mantenimiento deberá obtener también la aprobación de terminación satisfactoria de la modificación.

### 2.2.5. Migración

ISO especifica que se deberá asegurar que cualquier software o dato producido o modificado durante la migración se ajuste a la norma 12207.

Si se trata de una migración, el estándar aconseja desarrollar un plan de migración en el que se especifiquen al menos las siguientes cuestiones:

- Análisis de requisitos y definición de la migración.
- Desarrollo de herramientas de migración.
- Conversión del software y de los datos.
- Ejecución de la migración.
- Verificación de la migración.
- Soporte del entorno antiguo en el futuro.

También se insiste en la necesidad de notificar a los usuarios la intención de llevar a cabo la migración (describir el nuevo entorno, la fecha en que estará operativo, etc.), así como de ejecutar de forma paralela los dos entornos y de informar a los usuarios cuando se realice la migración prevista.

En esta actividad también se incluye una tarea de revisión postoperación con el fin de evaluar el impacto que suponga el cambio al nuevo entorno.

Por último, en el estándar se insiste en que se deberá poder acceder a los datos utilizados o asociados al antiguo entorno de acuerdo con los requisitos organizacionales para la protección y auditoría aplicables a los datos.

### 2.2.6. Retirada de software

De acuerdo al estándar ISO es necesario desarrollar y documentar un “plan de retirada” que aborde cuestiones como las siguientes:

- Cese de soporte total o parcial después de un cierto tiempo.
- Archivo del producto software y su documentación asociada.
- Responsabilidad sobre cuestiones de soporte residual futuro.
- Transición al nuevo producto.
- Accesibilidad de copias de datos.

Es importante que se tenga en cuenta a los usuarios a la hora de planificar la retirada del software y que se les notifique el plan. Las notificaciones deberán incluir lo siguiente:

1. Descripción de la sustitución o actualización con su fecha de disponibilidad.
2. Informe de por qué no se soportará más el software.

3. Descripción de otras opciones de soporte disponibles una vez que se haya eliminado el soporte.

También se recomienda llevar a cabo operaciones paralelas entre el software nuevo y el retirado y proporcionar formación a los usuarios.

Cuando tenga lugar la retirada planeada, se deberá notificar a todos los involucrados. Se debe archivar, según sea apropiado, toda la documentación, ficheros y código.

Por último, al igual que en el caso de la migración, se deberá poder acceder a los datos utilizados por, o asociados con, el software retirado de acuerdo a los requisitos organizacionales de protección y auditoría aplicables a los datos.

| ACTIVIDADES | TAREAS |
|---|---|
| IMPLEMENTACIÓN DEL PROCESO | Desarrollar planes de mantenimiento |
| | Definir procedimientos de petición de modificación |
| | Implementar la gestión de configuración |
| ANÁLISIS DE PROBLEMAS Y MODIFICACIONES | Evaluar impacto |
| | Verificar problema |
| | Elaborar alternativas |
| | Documentar el problema |
| | Obtener aprobación |
| IMPLEMENTACIÓN DE LAS MODIFICACIONES | Determinar objetos a modificar |
| | Desarrollar modificaciones |
| REVISIÓN Y ACEPTACIÓN DEL MANTENIMIENTO | Revisar integridad |
| | Obtener aprobación |
| | Asegurar ajuste a la norma |
| MIGRACIÓN | Desarrollar plan |
| | Notificar la futura migración |
| | Ejecutar paralelo |
| | Notificar migración |
| | Realizar revisión post-operación |
| | Archivar datos entorno antiguo |
| RETIRADA | Desarrollar plan |
| | Notificar futura retirada |
| | Ejecutar paralelo |
| | Notificar retirada |
| | Archivar datos entorno antiguo |

Tabla 3. Principales actividades y tareas del mantenimiento según ISO 12207.

## 2.3. El mantenimiento en el borrador del estándar ISO/IEC 14764

Este borrador [ISO/IEC, 1999] es parte de la familia de documentos del estándar internacional ISO/IEC 12207 [ISO/IEC, 1995], desarrollando el Proceso de

Mantenimiento de este último. El borrador no especifica cómo realizar las actividades y tareas del Proceso de Mantenimiento, sino que únicamente especifica una lista de las que deberían (o podrían, ya que puede aplicarse el Proceso de Adaptación a este borrador) realizarse.

Tipos de mantenimiento

En el borrador se definen los siguientes tipos de mantenimiento:

a) Mantenimiento correctivo: modificación realizada sobre un producto software después de su entrega para corregir problemas.

b) Mantenimiento de mejora: cambio realizado en el software que no es una corrección.

c) Mantenimiento adaptativo: modificación de un producto software, después de su entrega, para conservarlo utilizable en un entorno cambiado o cambiante.

d) Mantenimiento preventivo: modificación de un producto software después de su entrega para detectar y corregir errores latentes del software, antes de que lleguen a ser errores efectivos.

Consideración de la mantenibilidad

Los requisitos de mantenibilidad deberían ser incluidos en el Proceso de Adquisición de ISO/IEC 12207, de manera que sus niveles puedan ser evaluados durante el Desarrollo. Deben considerarse diferentes aspectos de mantenibilidad, relativos a Lenguaje de programación utilizado, Análisis de requisitos software, Diseño de la arquitectura software, Diseño detallado del software, Codificación y Pruebas.

Descripción del proceso de mantenimiento

Las actividades y tareas del Proceso de Mantenimiento son responsabilidad de la Organización de mantenimiento, definida en ISO/IEC (1995). Las actividades que componen este proceso son las ya descritas en éste (Implementación del proceso, Análisis de problemas y modificaciones, implementación de la modificación, Revisión y aceptación del mantenimiento, Migración y Retirada); sin embargo, el borrador precisa aún más la estructura del proceso, como se muestra en la Figura 7.

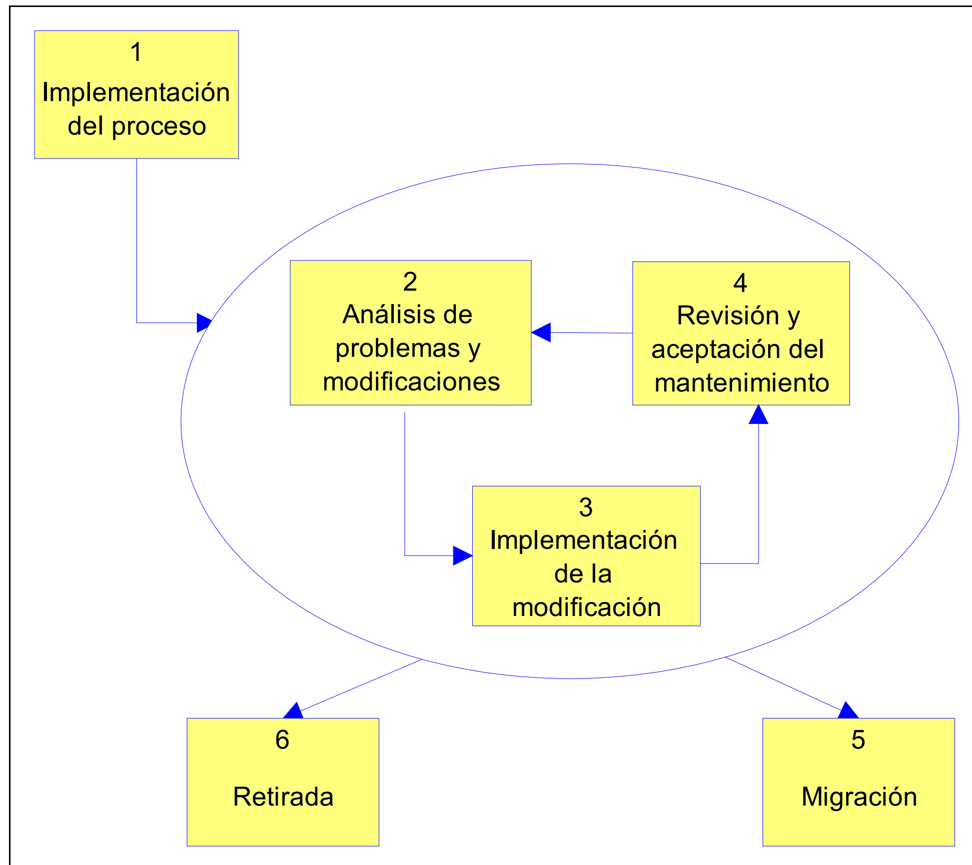



Figura 7. Proceso de mantenimiento de ISO/IEC (1999).

**Actividad 1. Implementación del proceso**

La Organización de mantenimiento establece los planes y procedimientos que serán ejecutados durante el proceso. 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.

**Actividad 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, que será uno de los cuatro indicados en este borrador. Igualmente, se debe indicar la magnitud del cambio (por ejemplo: tamaño de la modificación, coste, tiempo necesario).

**Actividad 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:

1. Dirigir y determinar análisis para determinar qué documentación, unidades de software y versiones necesitan ser modificadas.
2. Entrar en el Proceso de Desarrollo de ISO/IEC 12207 (IESO/IEC, 1995) para implementar las modificaciones.
3. Realizar una revisión conjunta de la modificación realizada.
4. Documentar y asegurar la calidad de la modificación.

**Actividad 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.

**Actividad 5: Migración**

Esta actividad se realiza cuando el sistema tiene que ser modificado para ser ejecutado en un nuevo entorno.

**Actividad 6: Retirada**

Se realiza cuando el producto software ha alcanzado el final de su vida útil.

## 2.4. Otras consideraciones sobre el proceso de mantenimiento

Además de los estándares ISO e IEEE ya mencionados, existen otros que también se ocupan del mantenimiento de software, como por ejemplo SPICE (Software Process Improvement and Capability dEtermination), que está siendo desarrollado por ISO, tomando como fuente de inspiración el “modelo de madurez de capacidad” (CMM) desarrollado por el SEI.

Tanto SPICE como CMM definen una serie de niveles de madurez que pueden alcanzar las organizaciones o los procesos relacionados con el software. En concreto en SPICE

se describen los resultados de una evaluación de procesos teniendo en cuenta dos dimensiones:

- Dimensión del *proceso*: los procesos se corresponden con los de la norma ISO 12207.
- Dimensión de la *capacidad*: formada por seis niveles de capacidad y nueve atributos de proceso. Cada nivel de capacidad está formado por uno o varios atributos que proporcionan una mejora importante en la capacidad de realizar un proceso. Los niveles de capacidad y sus atributos asociados se resumen en la Tabla 4.

La evaluación de SPICE permite obtener un perfil de cada uno de los procesos, examinando un número de instancias de cada proceso a cuyos atributos se le asignan valores. El perfil de cada instancia de un proceso se obtiene asignando cuatro valores (no conseguido, parcialmente conseguido, bastante conseguido y completamente conseguido) a cada uno de los nueve atributos de la instancia. Para alcanzar un nivel de capacidad una instancia del proceso debe ser evaluada como completamente conseguida para todos los atributos asociados con los niveles inferiores, y conseguida en su mayoría por los atributos de ese nivel.

Se pueden aplicar, por tanto, estas indicaciones para valorar el proceso de mantenimiento de una organización, cuyo propósito según SPICE, es "gestionar la modificación, migración y retirada de componentes del sistema (como hardware, software, operaciones manuales y redes) en respuesta a peticiones del usuario" y que se incluye dentro de los "procesos de ingeniería".

| **PROCESO** | **ATRIBUTO** |
|---|---|
| NIVEL 0.- PROCESO INCOMPLETO | |
| NIVEL 1.- PROCESO REALIZADO | RENDIMIENTO DEL PROCESO |
| NIVEL 2.- PROCESO GESTIONADO | GESTIÓN DEL RENDIMIENTO<br>GESTIÓN DEL PRODUCTO |
| NIVEL 3.- PROCESO ESTABLECIDO | DEFINICIÓN DEL PROCESO<br>RECURSOS DEL PROCESO |
| NIVEL 4.- PROCESO PREVISIBLE | MEDICIÓN DEL PROCESO<br>CONTROL DEL PROCESO |
| NIVEL 5.- PROCESO OPTIMIZADO | CAMBIO DE PROCESO<br>MEJORA CONTINUA |

Tabla 4. Niveles de capacidad y atributos en SPICE.

# 3. Mantenibilidad del Software

El término software puede hacer referencia a dos aspectos diferentes según se quiera señalar un producto software o bien un proceso utilizado para producir dicho producto software. Estos productos y procesos software tienen diversas propiedades que afectan a la calidad. Las más importantes son [Ghezzi et al., 1991]:

- Corrección (funcionamiento correcto de acuerdo a las especificaciones),
- Fiabilidad (probabilidad de que el software no falle),
- Robustez (actuación razonable ante circunstancias imprevistas),
- Eficiencia (rendimiento eficiente de los recursos utilizados)
- Amigabilidad (facilidad de utilización por los usuarios),
- Verificabilidad (facilidad de verificar las propiedades del software),
- Mantenibilidad (facilidad de mantenimiento),
- Reusabilidad (capacidad de reutilización),
- Transportabilidad (capacidad de ejecutar el software en entornos diferentes),
- Comprensibilidad (facilidad de comprender y/o entender el software),
- Interoperabilidad (capacidad de cooperar con otros sistemas),
- Productividad (eficiencia de los procesos),
- Oportunidad (capacidad de desarrollar el software a tiempo), y
- Visibilidad (transparencia de los procesos para examen externo).

De todas ellas, 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], es decir, existen situaciones en que la relación costes/beneficios es negativa:

$$\frac{Esfuerzo\ actual\ incrementar\ mantenibilidad}{\mathrm{Re}ducción\ \ futura\ esfuerzo\ mantenimiento} > 1$$

# 3.1. Concepto de mantenibilidad del software

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

- El Gobierno Federal de Estados Unidos 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 un conjunto de atributos referidos al esfuerzo necesario para realizar las modificaciones especificadas o implicadas [ISO/IEC, 1994].

Actualmente, no se incentiva a los desarrolladores para que construyan sistemas software mantenibles. 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 si se quieren 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.

## 3.1.1. Aspectos que influyen en la mantenibilidad

Existen unos pocos 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 de 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 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 u 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. Estos autores señalan que la utilización de algunas prácticas de desarrollo (técnicas de programación estructurada, paquetes software estándares y generadores de listados) suponen la reducción del esfuerzo de mantenimiento, mientras que otras prácticas (uso de generadores de código) implican un aumento de la necesidad de mantenimiento.

### 3.1.2. 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 8 se 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. A continuación se presenta una relación de todos ellos:

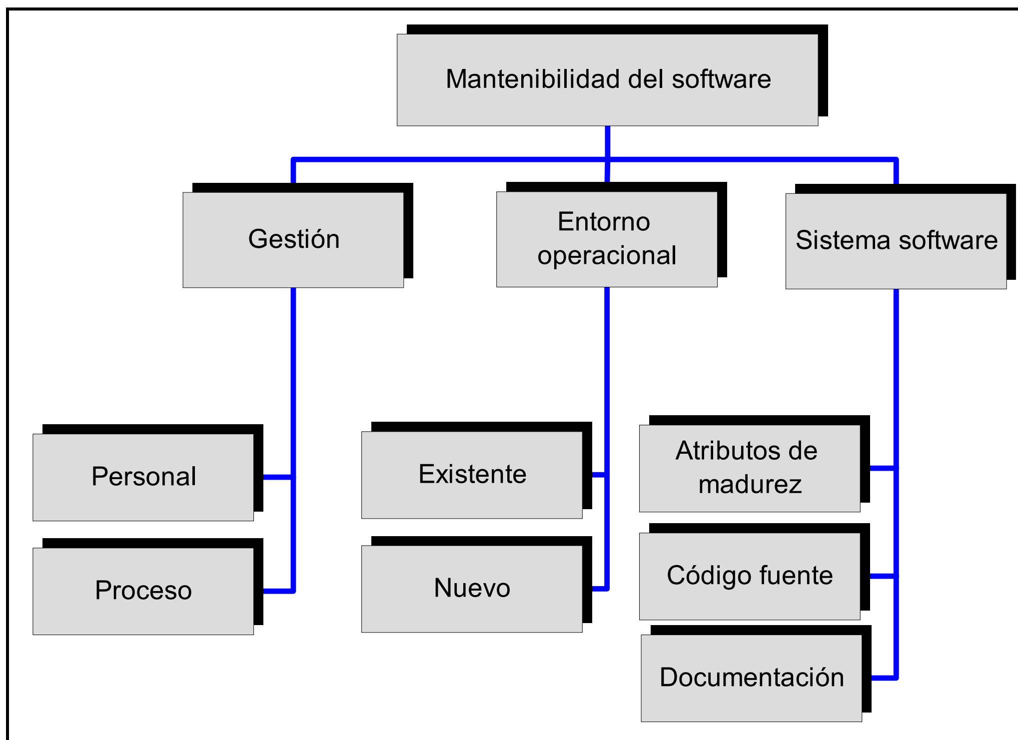



Figura 8. Jerarquía en la Mantenibilidad.

| **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. 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 objetos (POO) como el polimorfismo o la herencia pueden hacer menos comprensibles los programas y, por tanto, reducir su mantenibilidad [Daly et al., 1996].

## 3.1.3. 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; y es flexible (evolucionable) si permite cambios para que satisfagan nuevos requerimientos.

### 3.1.3.1. Reparabilidad

Un sistema software es reparable si permite la corrección de sus defectos con una cantidad de trabajo limitada y razonable. 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. Pero los componentes software no se deterioran, por lo que, aunque el uso de piezas estándar reduce los costes de producción del software, el concepto de pieza software sustituible no tiene sentido.

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.

### 3.1.3.2. 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 es mejorada con la modularización, los cambios sucesivos tienden a reducir la modularidad del sistema original. La situación todavía puede empeorar si las modificaciones son aplicadas 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.

## 3.2. 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) 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. Los ingenieros de software deberán evaluar los efectos de una modificación sobre las siguientes características del software [Mazza et al., 1994]:

- Eficiencia.
- Consumo de recursos.

- Cohesión.
- Acoplamiento.
- Complejidad.
- Consistencia.
- Transportabilidad.
- Fiabilidad.
- Mantenibilidad.
- Seguridad .

Estos efectos pueden evaluarse con la ayuda de herramientas de ingeniería inversa, comentadas en el capítulo 5, 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).

### 3.2.1. 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.

### 3.2.2. 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 acabo 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.

## 3.3. Estándar ISO/IEC 9126

Durante muchos años la comunidad de usuarios ha estado esperando la aprobación de un estándar de modelo de calidad del software. En 1992 se aprobó el estándar ISO/IEC llamado Software Product Evaluation: Quality Characteristics and Guidelines for their Use, ISO 9126 [ISO/IEC, 1991].

Como las características de calidad, subcaracterísticas y métricas asociadas pueden utilizarse no sólo para evaluar un producto software, sino también para definir requerimientos de calidad y otros usos adicionales, el documento publicado en 1991 está siendo reelaborado en dos estándares separados. Por un lado, el nuevo ISO/IEC 9126 [ISO/IEC, 2000] llamado Software Quality Characteristics and Metrics; y por otro lado, el ISO/IEC 14598 [ISO/IEC, 1998b] llamado Software Product Evaluation.

En [Genero y otros, 2000] se presenta un resumen del nuevo ISO 9126. Este estándar reformado define un modelo de calidad del software en el que la calidad se define como la totalidad de características relacionadas con su habilidad para satisfacer necesidades establecidas o implicadas. Los atributos de calidad se clasifican según seis características, las cuales a su vez se subdividen en subcaracterísticas (ver Figura 9). También se describen métricas de calidad del software basadas en atributos internos y en el comportamiento externo del sistema.

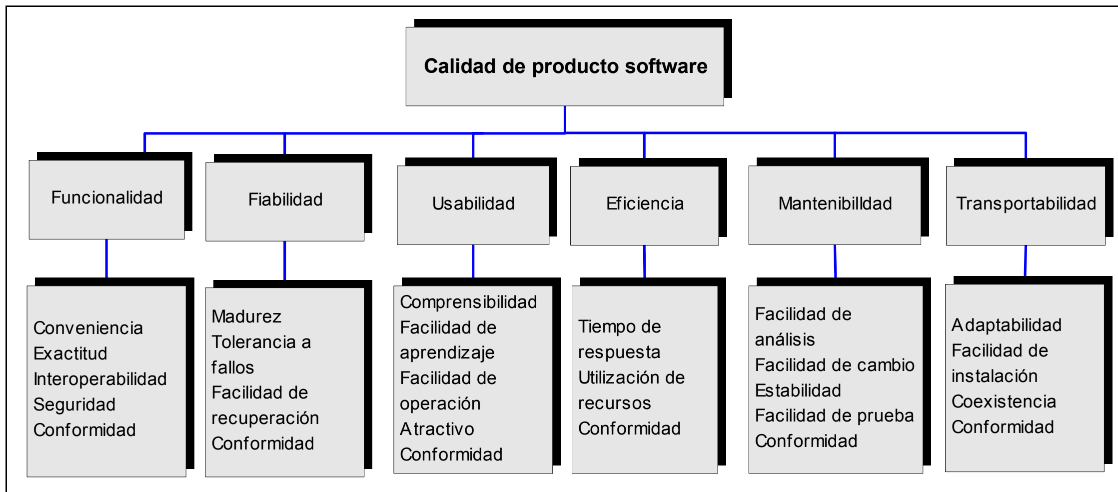



Figura 9. Calidad de un producto software (modelo ISO 9126).

En este estándar se establece que cualquier componente de la calidad del software puede ser descrito en términos de algunos aspectos de una o más de estas seis características.

La mantenibilidad se define como la capacidad de un producto software para ser modificado. Las modificaciones pueden incluir correcciones, mejoras o adaptación del software a cambios en el entorno, en los requerimientos o en las especificaciones funcionales. La mantenibilidad se subdivide en seis subcaracterísticas:

- *Analizabilidad*: Capacidad del producto software de diagnosticar sus deficiencias o causas de fallos, o de identificar las partes que deben ser modificadas.
- *Cambiabilidad*: Capacidad del producto software de permitir implementar una modificación especificada previamente. La implementación incluye los cambios en el diseño, el código y la documentación. Si el software es modificado por el usuario final, entonces, la cambiabilidad puede afectar a la operabilidad.
- *Estabilidad*: Capacidad del producto software de minimizar los efectos inesperados de las modificaciones.
- *Facilidad de prueba*: Capacidad del producto software de permitir evaluar las partes modificadas.
- *Conformidad*: Capacidad del producto software de satisfacer los estándares o convenciones relativas con la mantenibilidad.

# 3.4. Medida 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 ayudaría 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 medidas 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 medidas 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. Esta idea fue sugerida varios años antes por la comunidad científica [Lewis y Henry, 1989].

En los últimos años 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 (a mayor complejidad, menor mantenibilidad) y entre la calidad y la mantenibilidad (a mayor calidad, mayor 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 consideren 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.

En el estándar ISO 9126 [ISO/IEC, 2000] se incluye un análisis de las métricas de calidad (y entre ellas, las de mantenibilidad). Se definen los conceptos de atributos y características, métricas internas, métricas externas, y las interrelaciones entre unas y otras. También se expone el problema de la elección de métricas y los criterios de medida, así como los requerimientos que deben cumplir las mediciones utilizadas para comparar.

Dada la novedad de este estándar, en este momento todavía se están discutiendo las extensiones del documento original que proponen métricas internas y externas concretas para evaluar las características de la calidad, incluida la mantenibilidad.

### 3.4.1. Medidas externas de la mantenibilidad

Desde este punto de vista externa, 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 valido 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 con esmero 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 revisones).
- 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. Por ejemplo, en GILB [1988] se propone la evaluación externa de la mantenibilidad midiendo siete aspectos diferentes y comparándolos con un estudio realizado sobre mas de 5.000 programas, mantenidos por 250 programadores, con una media de 5.000 líneas de código cambiadas (NCLOC) por programa.

### 3.4.2. Medidas internas de la mantenibilidad

Se han propuesto numerosas medidas de atributos internos como indicadores de la mantenibilidad. En concreto, un determinado conjunto de medidas de complejidad han sido correlacionadas significativamente con los niveles de esfuerzo para mantenimiento. Pero la correlación con una característica no significa una medida de esa característica. No obstante, algunas de las medidas estructurales pueden servir para evitar riesgos con respecto a la mantenibilidad, ya que existe una clara conexión intuitiva entre productos pobremente estructurados y pobremente documentados y la facilidad de mantenimiento futura de dichos productos una vez se han implementado.

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 las 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 muy novedosa para definir modelos de esfuerzo de mantenimiento utilizando análisis de regresión sobre medidas de atributos internos.

# 4. El estandar ISO 14764

Como hemos visto anteriormente, la norma ISO 12207 considera el mantenimiento como uno de los procesos principales del ciclo de vida del software, que define las actividades y tareas del mantenedor (la organización que proporciona el servicio de mantener el producto software). A continuación se presenta el modelo de PMS establecido a grandes rasgos en ISO 12207 y detallado en el documento borrador del nuevo estándar ISO 14764 (ISO/IEC, 1998e). Al igual que pasa con los demás estándares, en estos documentos únicamente se especifica cuáles deben ser las actividades y tareas del proceso de mantenimiento pero no se indica cómo realizarlas.

En ISO 14764 se considera que el PMS se activa cuando se modifica un producto software después de su entrega. Se distinguen cuatro tipos de mantenimiento diferentes según los objetivos de dicha modificación sean:

- Corregir problemas detectados (*Correctivo*).
- Mejorar la funcionalidad, o el rendimiento, mantenibilidad u otros atributos del software (*Perfectivo*).
- Hacer que siga siendo utilizable en un entorno nuevo o cambiado (*Adaptativo*).
- Detectar y corregir errores latentes en el software antes de que se conviertan en fallos reales (*Preventivo*).

Cuando el mantenedor recibe una petición de modificación (propuesta de cambio de un producto software que está siendo mantenido), según su naturaleza, implicará una corrección o una mejora en el software. Una corrección supone un mantenimiento de tipo correctivo, mientras que una mejora puede suponer un mantenimiento preventivo, adaptativo o perfectivo (Figura 10).

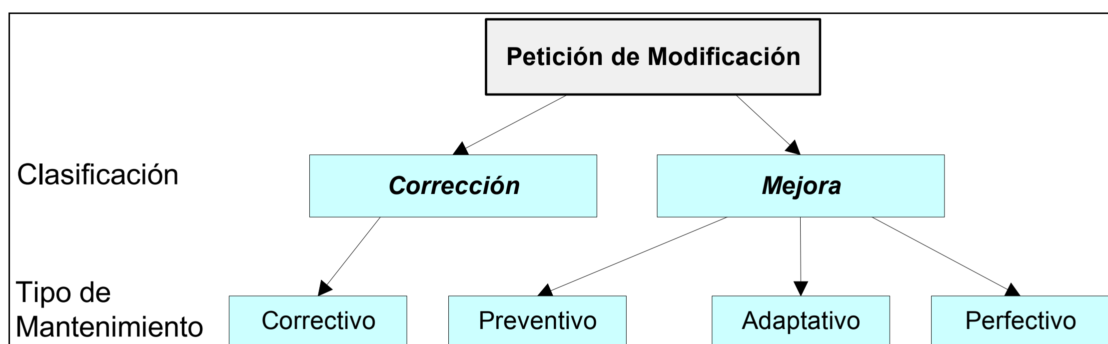



Figura 10. Peticiones de modificación y tipos de mantenimiento.

# 4.1. Actividades y Tareas

En la definición de ISO (ISO/IEC, 1995 y 1998e), el PMS consta de las seis actividades siguientes (Figura 11):

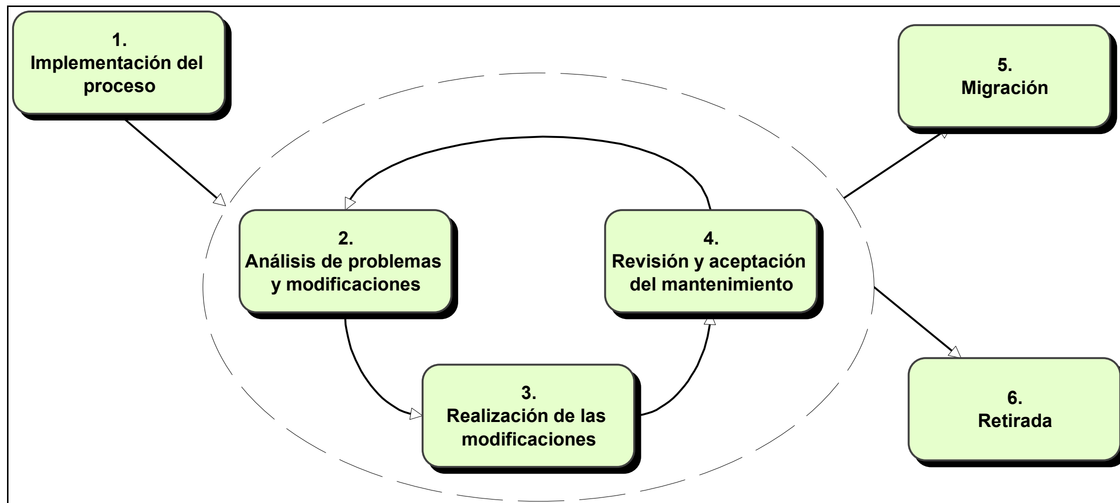



Figura 11. El proceso de mantenimiento según ISO.

**A1: Implementación del proceso.**

En esta actividad se desarrollan los planes (por ejemplo, el plan de mantenimiento) y procedimientos correspondientes para llevar a cabo las actividades y tareas del PMS. También se deben definir los procedimientos necesarios para la gestión de los problemas (empleando el proceso de resolución de problemas) y de las peticiones de modificación, e implementar el proceso de gestión de la configuración para gestionar los cambios en los elementos software existentes. Consta de tres tareas:

1. Elaborar, documentar y ejecutar planes y procedimientos para conducir las actividades y tareas del proceso.
2. Establecer procedimientos para recibir, almacenar y controlar los informes de problemas y solicitudes de modificación de los usuarios, y proporcionar a éstos realimentación.
3. Implementar (o establecer una interfaz organizacional con) el proceso de Gestión de la Configuración, para gestionar las modificaciones del sistema existente.

**A2: Análisis de problemas y modificaciones.**

Antes de modificar el software mantenido, el mantenedor deberá analizar los problemas o peticiones de modificación, elaborar y documentar distintas soluciones potenciales, y obtener la aprobación para implementar la opción elegida. En consecuencia, durante esta actividad, el mantenedor realiza las siguientes tareas:

4. Analizar el impacto de la petición o informe en la organización, el sistema existente y los interfaces con otros sistemas, determinando el tipo de mantenimiento (uno de los cuatro anteriores), el alcance (tamaño, coste y tiempo para hacer la modificación) y su criticidad (impacto en el rendimiento, la seguridad, etc.).
5. Replicar o verificar el problema.
6. Elaborar opciones para implementar la modificación.
7. Documentar el informe o petición, los resultados del análisis, y las opciones de implementación.
8. Obtener aprobación para la opción de modificación elegida.

**A3: Realización de las modificaciones.**

En esta actividad se incluyen todas las tareas relativas a determinar qué elementos del sistema software (documentación, unidades o módulos y versiones) deben modificarse, y se utiliza el proceso de desarrollo para implementar las modificaciones. Las tareas que lleva a cabo el mantenedor durante esta actividad son las siguientes:

9. Realizar un análisis para determinar qué documentación, unidades de software y versiones necesitan ser modificadas. Documentar esta información.
10. Entrar en el proceso de Desarrollo para implementar las modificaciones. Este proceso, según es definido en ISO 12207 (ISO/IEC, 1995), debe complementarse con lo siguiente:
    - definir y documentar criterios para probar y evaluar las partes del sistema modificadas y no modificadas;
    - comprobar la implementación completa y correcta de los requisitos nuevos o modificados;
    - asegurarse de que los requisitos originales no modificados no han sido afectados; y
    - documentar los resultados de estas pruebas.

**A4: Revisión y aceptación del mantenimiento.**

En esta actividad se asegura que las modificaciones al sistema son correctas y que cumplen los estándares aprobados utilizando una metodología correcta. Las dos tareas que lleva a cabo el mantenedor durante esta actividad son:

11. Realizar revisiones conjuntas entre el mantenedor y la organización que debe aprobar la modificación para determinar la integridad del sistema.
12. Obtener la aprobación, según los términos indicados en el contrato o acuerdo de mantenimiento, de que la modificación se ha completado satisfactoriamente.

**A5: Migración.**

Esta actividad se realiza cuando el sistema tiene que ser modificado para ser ejecutado en un entorno diferente. En ese caso, el mantenedor necesita determinar las acciones necesarias para lograr la migración, llevarlas a cabo, y documentar los efectos. Las tareas incluidas en este actividad son las siguientes:

13. Asegurar que cualquier producto software o dato producido o modificado durante la migración lo ha sido de conformidad a lo establecido en la norma ISO 12207.
14. Elaborar y documentar un "plan de migración" en el que se incluirán, al menos, las siguientes cuestiones:
    - Análisis y definición de los requisitos de la migración.
    - Herramientas para la migración.
    - Conversiones del software y de los datos.
    - Ejecución de la migración.
    - Verificación de la migración.
    - Soporte futuro del entorno antiguo.
15. Notificar a los usuarios las intenciones de migración previstas, indicando lo siguiente:
    - explicación de porqué no se puede seguir soportando el entorno antiguo;
    - descripción del nuevo entorno, indicando la fecha de disponibilidad prevista; y
    - descripción de otras opciones de soporte cuando el entorno antiguo haya sido eliminado.
16. Realizar operaciones en paralelo en ambos entornos, el viejo y el nuevo, para que la transición sea suave. A la vez, proveer a los usuarios de la formación necesaria.
17. Notificar a todos los afectados de que se ha completado la migración. Toda la documentación, elementos del sistema y datos del entorno antiguo deberá ser recopilada y archivada.
18. Realizar una revisión post-operación para evaluar el impacto del cambio al entorno nuevo. Los resultados serán enviados a las autoridades adecuadas.
19. Establecer la accesibilidad a los datos utilizados o asociados con el entorno antiguo de acuerdo con los requisitos del contrato en cuanto a protección de datos y auditoría.

**A6: Retirada.**

Esta actividad se realiza cuando un producto software ha alcanzado el final de su vida útil. Para ello, se deben llevar a cabo las siguientes tareas:

20. Elaborar y documentar un "plan de retirada", incluyendo los siguientes apartados:
    - Cese total o parcial del soporte después de un cierto tiempo.

- Archivo del producto software y su documentación asociada.
- Responsabilidad sobre cuestiones de soporte residual futuro.
- Transición al nuevo producto, si lo hubiera.
- Verificación de la migración.
- Accesibilidad a las copias de los datos archivados.

21. Notificar a los usuarios las intenciones de retirada previstas, indicando lo siguiente:
    - descripción de la sustitución o actualización previstas, indicando la fecha de disponibilidad;
    - explicación de porqué no se puede seguir soportando el producto retirado; y
    - descripción de otras opciones de soporte disponibles una vez el producto se haya retirado.
22. Realizar operaciones para una transición suave a la nueva situación, incluyendo la formación necesaria a los usuarios.
23. Notificar a todos los afectados de que se ha completado la retirada. Toda la documentación, elementos del sistema y datos deberá ser recopilada y archivada.
24. Establecer la accesibilidad a los datos utilizados o asociados con el producto retirado de acuerdo con los requisitos del contrato en cuanto a protección de datos y auditoría.

Las actividades de implementación del proceso y de retirada se deben realizar obligatoriamente una sola vez por parte del mantenedor al inicio y final del servicio o proceso de mantenimiento. La actividad de migración podrá ocurrir varias veces o ninguna en momentos aleatorios, según se produzcan cambios en el entorno o no (sistema operativo, SGBD, red, hardware, etc.). Las tres actividades intermedias ocurren una vez por cada petición de modificación o informe de problemas recibido por el mantenedor. Estas tres actividades forman un ciclo en el sentido de que el resultado de una revisión y aceptación del mantenimiento puede conducir a nuevas peticiones de modificación o informes de problemas.

En la Tabla 6 se muestra un resumen de todas las actividades y tareas relatadas anteriormente.

| **Proceso de Mantenimiento del Software** | | |
|---|---|---|
| **Actividades** | **Tareas** | |
| Implementación del proceso | 1 | Desarrollar planes y procedimientos de mantenimiento |
| | 2 | Establecer procedimientos para peticiones de modificación |
| | 3 | Implementar la gestión de la configuración |

| **Proceso de Mantenimiento del Software** | | |
|---|---|---|
| **Actividades** | **Tareas** | |
| Análisis de problemas y modificaciones | 4 | Analizar el informe o petición |
| | 5 | Replicar o verificar el problema |
| | 6 | Elaborar opciones para implementar la modificación |
| | 7 | Documentar |
| | 8 | Obtener aprobación para la opción elegida |
| Realización de las modificaciones | 9 | Determinar los elementos a modificar |
| | 10 | Invocar al proceso de desarrollo |
| Revisión y aceptación del mantenimiento | 11 | Revisar la integridad del sistema modificado |
| | 12 | Obtener aprobación |
| Migración | 13 | Asegurar adaptación a las normas |
| | 14 | Elaborar plan de migración |
| | 15 | Notificar intenciones a los usuarios |
| | 16 | Ejecutar operaciones en paralelo y formar a los usuarios |
| | 17 | Notificar la compleción de la migración |
| | 18 | Realizar revisión post-operación |
| | 19 | Archivar datos del entorno antiguo |
| Retirada | 20 | Desarrollar el plan de retirada |
| | 21 | Notificar futura retirada |
| | 22 | Ejecutar en paralelo |
| | 23 | Notificar retirada |
| | 24 | Archivar datos del entorno antiguo |

Tabla 6. Actividades y tareas del proceso de mantenimiento según ISO.

# 5. Técnicas para el Mantenimiento

Los tipos de mantenimiento que conocemos (perfectivo, correctivo, adaptativo y preventivo) tienen en común el hecho de que reconstruyen alguna funcionalidad de la aplicación o, incluso, el sistema completo. Si para el desarrollo del producto usamos técnicas de Ingeniería de Software, que constan de unas etapas bien definidas y más o menos secuenciales, ¿por qué no utilizar, para la reconstrucción del sistema, técnicas igualmente rigurosas? En este capítulo introducimos los conceptos de Ingeniería Inversa, Reingeniería y Reestructuración, presentando algunos métodos y técnicas para utilizarlos. Además, se introducen y comentan otra serie de soluciones técnicas, destinadas a facilitar las labores del mantenimiento de software.

## 5.1. Introducción y conceptos básicos

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 puede 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 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.

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 12, vemos cómo realizamos redocumentación en cada nivel de abstracción del proceso de Reingeniería. Profundizando en esta idea, Pressman [1998] menciona tres opciones respecto a qué y cómo redocumentar:

a) Si el sistema funciona y la redocumentación consume muchísimos recursos, es posible que sea más correcto dejar el tema como está y no redocumentarlo.

b) 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.

c) 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.

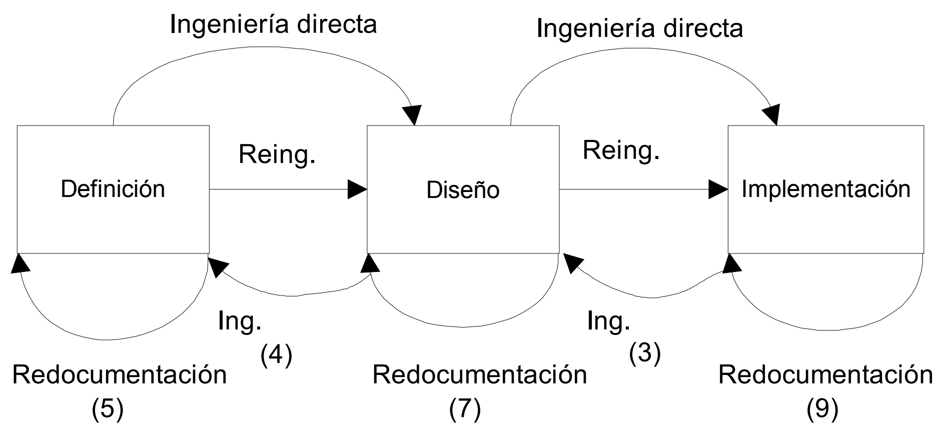



Figura 12. Ingeniería directa, Ingeniería inversa, Reingeniería y Redocumentación.

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.

# 5.2. 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 más abajo, 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 13. Código funcionalmente correcto, pero poco legible.

Deducimos, de un primer vistazo, que las líneas de código de la Figura 13 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 en soporte magnético u óptico 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 14.

```
#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 14. Código equivalente al de la figura anterior, 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 14, 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?".

### 5.2.1. 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 Pila 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 14, 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 puntillosas 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 15, donde ni el comentario de cabecera ni el nombre del método TotalConIVA son ya válidos, puesto que, como se observa, 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.

### 5.2.2. 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.

```
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 15. Ejemplo de comentario no actualizado.

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 y otro tipo de defectos de los cuales, aunque no sea nuestra labor corregirlos en esta fase, sí que deberemos tomar buena nota.

# 5.3. 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”.

## 5.3.1. 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.

En la actualidad, la mayor parte de las técnicas de reestructuración tienen como objetivo la transformación de aplicaciones construidas hace varias décadas mediante metodologías estructuradas al paradigma orientado a objeto que, como es bien sabido, ofrece niveles muy elevados de portabilidad, reutilización y mantenibilidad [Bucci et al., 1998]. Existen, sin embargo, tantas técnicas de reestructuración de sistemas como tipos de sistemas: así, por ejemplo, Sellink et al. [1999] proporcionan una estrategia para la reestructuración de sistemas transaccionales compuestos de código tanto en Cobol como en CICS, para facilitar su migración a otros entornos, como cliente/servidor. Penteado et al. [1999] presentan un método para reestructurar programas en C (generando así mismo código C), que además produce documentación orientada a objetos del sistema. Taschwer et al. [1999] explican un método y una herramienta para convertir código C a C++. Cremer [1998] presenta un método para reestructurar componentes de aplicaciones escritas en Cobol, de manera que sean utilizables en entornos distribuidos.

Además de los métodos mencionados en el párrafo anterior, que trabajan directamente sobre el código fuente, también pueden reestructurarse productos software de un nivel más alto de abstracción: Borne y Romanczuk [1998], por ejemplo, presentan un método para transformar diagramas MERISE a diagramas OMT.

#### 5.3.1.1. Un ejemplo: reestructuración de C++ a 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:

- Insertar funciones en clases.
- Encapsular código en funciones.
- Expulsar funciones de clases.

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 cabecera de la función (preceder con el identificador de clase; 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 16 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.

```
class A
{
  public:
      int i;
  protected:
      char c;
};
      ...

int f(A& a)
{
      a.i=4;
      ...
}
```

```
class A
{
  public:
      int i;
      int f();
  protected:
      char c;
};
      ...

int A :: f() {
      this->i=4;
      ...
};
```

Figura 16. Inserción de una función dentro de una clase.

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.

En la Figura 17 hemos enmarcado el bloque de código que el usuario desea encapsular en una función. Como se observa a la derecha, la herramienta ha considerado que el bloque es sintácticamente completo y lo sustituye por una llamada a la función newfun.

```
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 17. Encapsulación de un bloque de código en una función.

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.

# 5.4. Ingeniería inversa y reingeniería de bases de datos relacionales

La aplicación de un proceso de Ingeniería Inversa a una base de datos puede ser debida al deseo de migrar desde modelos de datos antiguos (jerárquico, red) al modelo relacional; al hecho de cambiar de un sistema gestor a otro, posiblemente los dos relacionales; a la detección de errores de integridad; a la migración de un sistema relacional a otro orientado al objeto, etc.

## 5.4.1. Metodología de diseño "hacia delante"

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.

## 5.4.2. Recuperación del diseño de bases de datos

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 18:

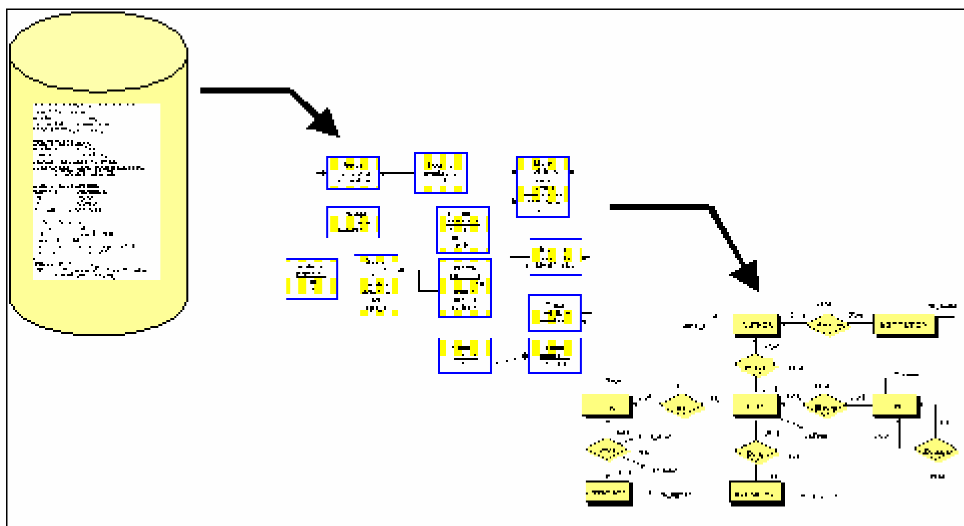


Figura 18. Fases en la Ingeniería inversa de una BD relacional.

Pedro de Jesus y Sousa [1999] presentan una interesante comparación entre diferentes métodos aplicables a la Ingeniería inversa de bases de datos relacionales. En la Tabla 7 mostramos resumidamente los diferentes métodos analizados, las condiciones necesarias para su aplicación, las entradas que toman y el tipo de resultado que producen.

| **Método** | **Entradas** | **Salidas** | **Precondiciones** |
|---|---|---|---|
| Chiang et al. | Datos<br>Relaciones<br>Claves primarias | EER | 3FN<br>Consistencia de nombres<br>Ausencia de errores en PK |
| Johannesson | Relaciones<br>Dependencias funcionales<br>Dependencias de inclusión | Par (L, IC) | 3FN |
| Markowitz y Makowsk | Relaciones<br>Dependencias clave<br>Restricciones de integridad referencial | EER | Relaciones en FN de Boyce-Codd |

| Método | Entradas | Salidas | Precondiciones |
| --- | --- | --- | --- |
| Navathe y Awong | Relaciones | EER | Relaciones en 3FN o FN de Boyce-Codd<br>Consistencia en los nombres de los atributos<br>Ausencia de ambigüedades u homónimos en FK<br>Especificación de todas las claves candidatas |
| Petit et al | Relaciones con restric. de unicidad no nulas<br>Datos<br>Código | EER | Ninguna |
| Premerlani y Blaha | Relaciones<br>Datos | Modelo de clases OMT | Ninguna |
| Signore et al | Relaciones<br>Código | ER | Ninguna |

Tabla 7. Diferentes métodos para la ingeniería inversa de BB.DD. relacionales.

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 19 (en donde Ri representa una tabla y Kj un atributo de su clave primaria), podríamos realizar uno de los dos agrupamientos de tablas mostrados a la derecha. Nótese que, por ejemplo, R5 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 R4.

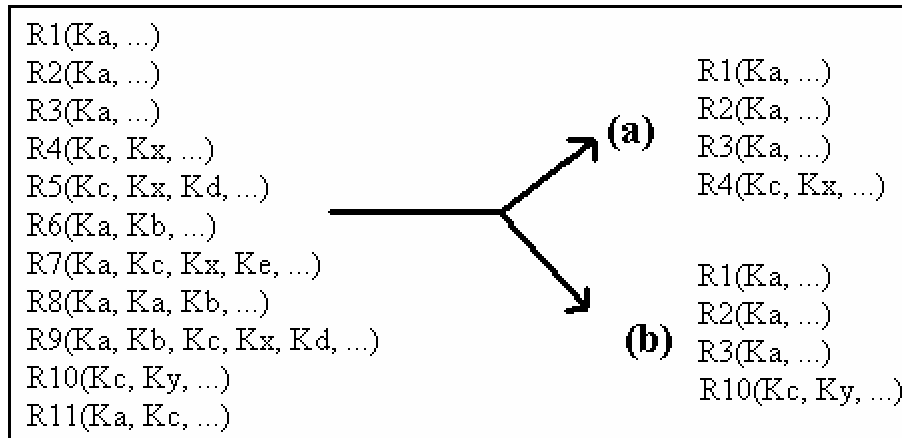



Figura 19. Selección de grupos de tablas según sus claves primarias.

- 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 anterior y asignarlo a una "entidad abstracta". Si seleccionamos el a, tendremos en principio dos entidades abstractas, cada una compuesta por tablas con las mismas PK; por tanto, habrá dos entidades abstractas:

  AE1={R1 , R2 , R3 } y AE2={R4 }.

- 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 R5: los dos atributos KC y KX 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 (Kd) no se encuentra incluido en la PK de la entidad abstracta AE1, por lo que podemos ubicar R5 en AE2. Repitiendo esta operación con las tablas restantes, tendremos que:

  AE1= { R1, R2, R3, R6, R8 } ; AE2= { R4, R5, R10 }

- Nótese que las tablas R7 , R9 y R11 han quedado sin asignar a ninguna de las dos entidades abstractas identificadas hasta el momento. Fijándonos en R7 , encontramos la justificación en que el atributo Ka forma parte de la clave primaria de AE1 y los atributos Kc y KX 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={R7 , R9, R11} 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 20.

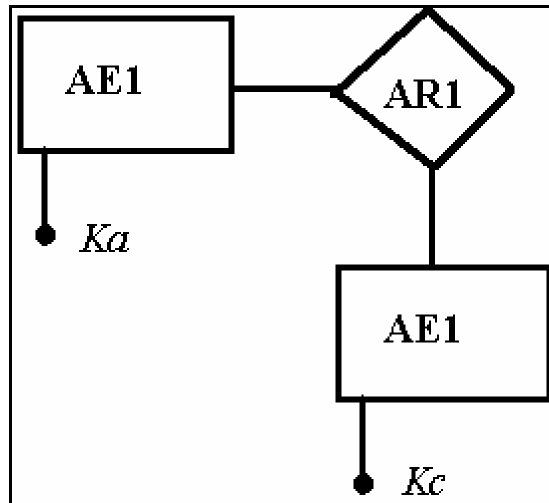



Figura 20. Diagrama de elementos abstractos.

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...* | *...se 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 8. Métodos propuestos en ingeniería inversa de tablas relacionales.

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 21: se crea el esquema de elementos abstractos de acuerdo al 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.

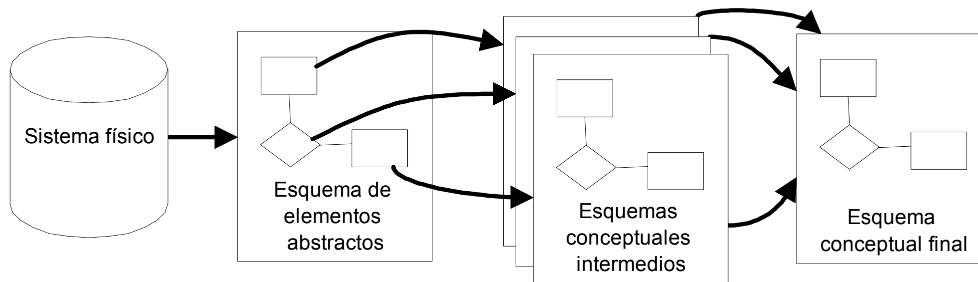



Figura 21. Vista general del proceso de ingeniería inversa de BBDD relacionales.

## 5.5. Ingeniería inversa y reingeniería de interfaces de usuario

En muchas ocasiones, los productos de 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:

1) 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.

2) Entrevistas.

Entrevistar a los diferentes grupos de usuarios y observación de sus métodos de trabajo, distinguiendo:

- Entrevistas a la dirección para identificar 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.

Por otra parte, y aunque es muy probable que los usuarios no nos transmitan detalles como "Quiero que el fondo de las ventanas sea de color azul", el ingeniero de software debe tener siempre presentes los requisitos no solicitados explícitamente por el cliente pero que son inherentes a todo software profesional (por ejemplo, que el programa no se interrumpa al escribir una letra en una entrada numérica).

3) 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, contadores 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).

# 5.6. Costes y beneficios de la reingeniería

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:

## 5.6.1. Justificación del proyecto de Reingeniería

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:

a) 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 Reingeniería.

b) 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. Estos valores son incorporados a una base de datos que es utilizada por otra herramienta para realizar comparaciones y obtener resultados.

c) 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.

d) Evaluación del valor de negocio del sistema actual, que es realizado por la dirección de la organización.

### 5.6.2. 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 22. 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.

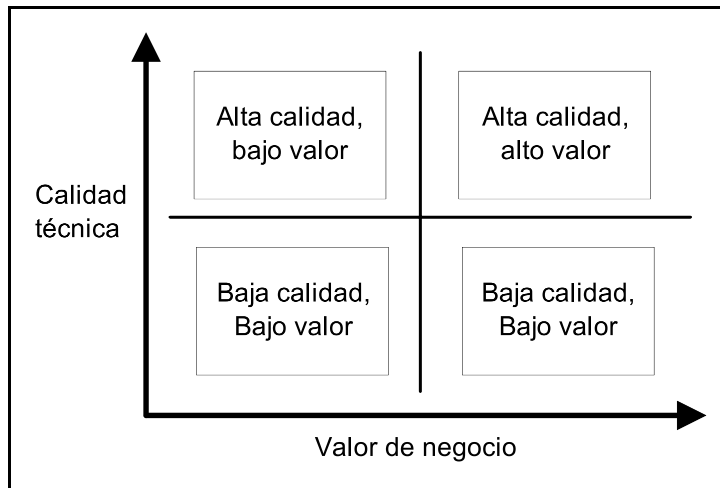



Figura 22. Diagrama para el análisis de la cartera de aplicaciones.

Este mismo autor ha construido docenas de herramientas diferentes que estiman automáticamente la calidad técnica de los programas en múltiples lenguajes de programación. La Figura 23 muestra un fragmento del informe producido por su analizador de código C y C++. Los diferentes valores de modularidad, portabilidad, etc. se calculan en función de las más de sesenta métricas calculadas por la herrmienta.

```
+-------------------------------------------------------------------+
|                                                                   |
|                   Q U A L I T Y    M E T R I C S                  |
|                                                                   |
+-------------------------------------------------------------------+
|       DEGREE OF MODULARITY                =======>  0.395         |
|       DEGREE OF PORTABILITY               =======>  0.164         |
|       DEGREE OF FLEXABILITY               =======>  0.080         |
|       DEGREE OF CONFORMITY                =======>  0.747         |
|       DEGREE OF TESTABILITY               =======>  0.882         |
|       DEGREE OF READABILITY               =======>  0.172         |
|       DEGREE OF REUSABILITY               =======>  0.260         |
|       DEGREE OF MAINTAINABILITY           =======>  0.372         |
|                                                                   |
|       AVERAGE  PROGRAM QUALITY            =======>  0.337         |
|                                                                   |
+-------------------------------------------------------------------+
```

Figura 23. Medidas de calidad obtenidas por una de las herramientas de Sneed.

### 5.6.3. 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.

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:

$$\textit{Tiempo de desarrollo} = 2.5 * (\textit{esfuerzo})^{0.19}$$

### 5.6.4. 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 16 parámetros que mostramos en la Tabla 9 para tomar la decisión de abandonar, redesarrollar o aplicar Reingeniería a aplicaciones heredadas.

| |
|---|
| $P_3$=Valor de negocio actual (anual). |
| $P_4$=Coste previsto de mantenimiento tras la Reingeniería (anual). |
| $P_5$=Coste previsto de operaciones tras la reingeniería (anual). |
| $P_6$= Valor de negocio previsto tras las reingeniería actual (anual). |
| $P_7$=Coste estimado de la Reingeniería. |
| $P_8$=Duración estimada de la Reingeniería. |
| $P_9$=Factor de riesgo de la Reingeniería. |
| $P_{10}$=Coste previsto de mantenimiento tras el redesarrollo (anual). |
| $P_{11}$=Coste previsto de operaciones tras el redesarrollo (anual). |
| $P_{12}$=Valor de negocio previsto del nuevo sistema (anual). |
| $P_{13}$=Coste estimado del redesarrollo. |
| $P_{14}$= Duración estimada del redesarrollo. |
| $P_{15}$=Factor de riesgo del redesarrollo. |
| $P_{16}$=Vida esperada del sistema. |

Tabla 9. Parámetros a considerar en proyectos de Reingeniería.

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:

$$B_R=[(P_6-(P_4+P_5))*(P_{16}-P_8)-(P_7*P_9)]-B_M$$

# 5.7. Otras técnicas: detección de clones

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"

Baxter et al. [1998] presentan un original método para detectar clones. 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 24, 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.

**Fragmento 1 en módulo A**

```
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;
}
```

**Fragmento 2 en módulo B**

```
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 24. Dos fragmentos de código semánticamente similares, pero sintáctica y léxicamente diferentes.

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.

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 25.

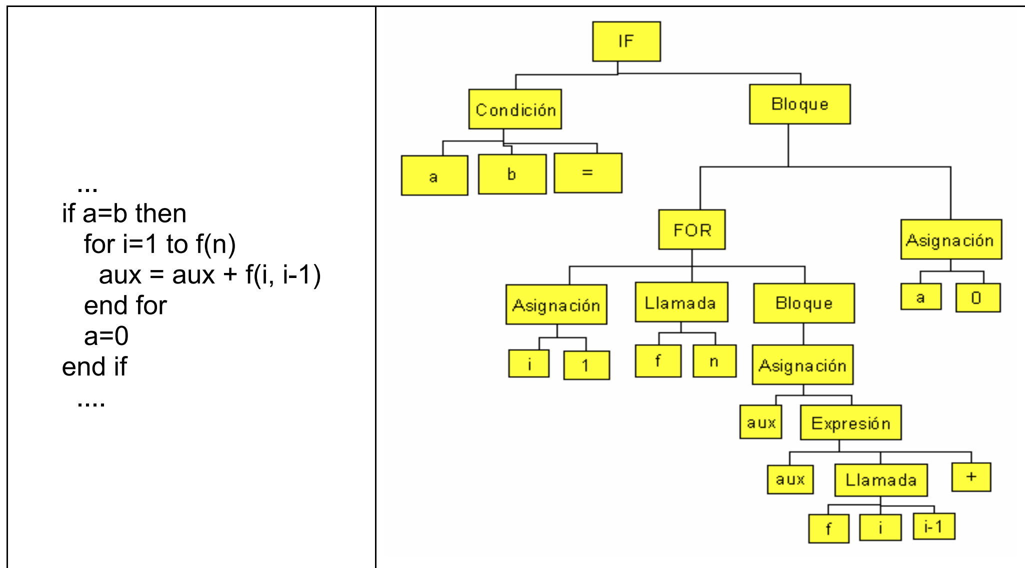
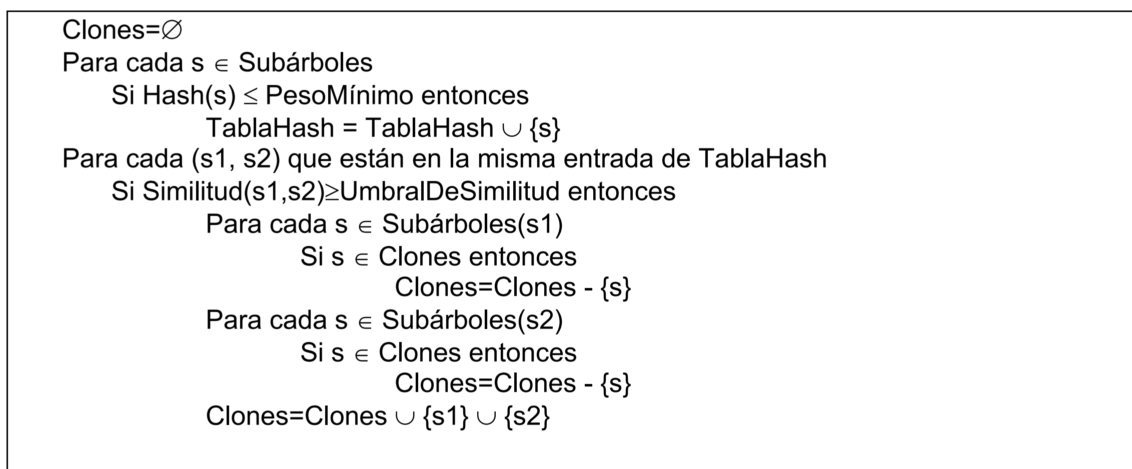



Figura 25. Un fragmento de programa y su posible representación como un árbol de sintaxis abstracta.

El algoritmo, cuya versión más básica se detalla en la Figura 26, 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.



```
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 ∪ {s1} ∪ {s2}
```

Figura 26. Algoritmo básico para la detección de clones.

El segundo "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 s1 y s2 que acaba de identificar como clones ya estaban clasificados como clones. Nuevamente en caso afirmativo, saca los subárboles s del conjunto de clones, pues s1 y s2 son clones más grandes, en los cuales ya estaban incluidos sus subárboles s.

# 6. Metodologías y Gestión del Mantenimiento

En este capítulo se presenta la visión del proceso de mantenimiento que ofrece MANTEMA, una metodología para mantenimiento de software que integra todas las actividades relacionadas con este proceso. El objetivo de MANTEMA 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 ha sido desarrollada entre el Grupo Alarcos del Departamento de Informática de la Universidad de Castilla La Mancha y Atos ODS.

## 6.1. Planteamiento de la metodología MANTEMA

MANTEMA aborda por entero el problema del mantenimiento, intentando disminuir los costes de todas sus actividades. Para conseguirlo, una de las primeras tareas que debe completarse es la definición clara del conjunto de actividades que se deben ejecutar a lo largo del proceso de mantenimiento.

En este sentido, es útil seguir las recomendaciones de ISO/IEC 12207 [ISO/IEC, 1995], IEEE 1219 [IEEE, 1993], o Pressman [1993], que abogan por ejecutar cada petición de modificación según el tipo de mantenimiento que le corresponda. Por ello, en MANTEMA se definen 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 desea mejorar las características inter-nas de un producto para hacerlo, por ejemplo, más fácilmente mantenible.

Sin embargo, durante la construcción y 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 27.

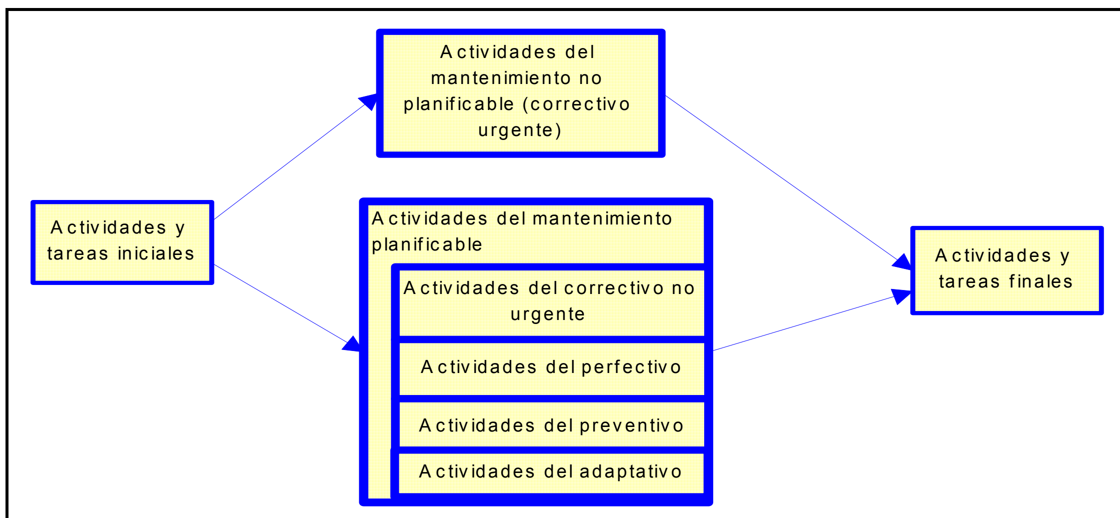



Figura 27. Vista general de la metodología MANTEMA.

## 6.2. Descripción de las tareas

Siguiendo la idea de ISO/IEC 12207, cada nodo del grafo mostrado en la Figura 27 está formado por un conjunto de actividades, estando cada actividad compuesta de un conjunto de 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 podrán 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 28, 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. Quedan, sin embargo, pendientes de integración y de establecimiento de interfaz los procesos de Auditoría y de Mejora de la 12207.

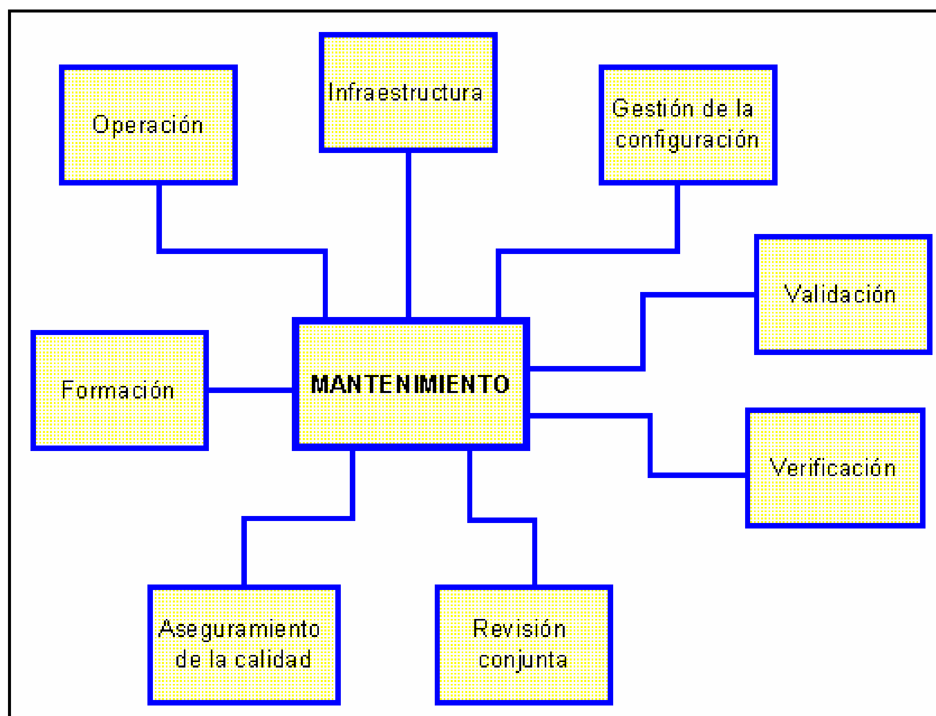



Figura 28. Procesos con los que se establece interfaz.

## 6.3. Estructura detallada de MANTEMA

En esta sección describimos más detalladamente los nodos que componen la metodología MANTEMA (Figura 27).

### 6.3.1. Actividades y tareas iniciales comunes

Este conjunto de actividades está representado por el nodo inicial del grafo mostrado en la Figura 27. 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 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).

### 6.3.2. Actividades y tareas del mantenimiento no planificable

Detallamos en las dos tablas siguientes 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<br>Investigar y analizar causas | NP2.1<br>Realizar acciones correctivas | NP2.2<br>Cumplimentar documentación |
| Entradas | Producto software en explotación con error bloqueante o crítico<br>Petición de modificación | Conjunto de elementos software a corregir | Elementos software antiguos (con errores visibles)<br>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<br>Usuario | Equipo de mantenimiento | Equipo de mantenimiento |
| Interfaces con otros procesos | | Aseguramiento de la calidad<br>Gestión de la configuración | |

Tabla 10. Estructura del mantenimiento no planificable (sigue).

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

Tabla 11. Estructura del mantenimiento no planificable (continuación).

### 6.3.3. 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 29 representa una tarea de este tipo de mantenimiento. Como se observa, 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 29 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.

### 6.3.4. Actividades y tareas finales comunes

MANTEMA también detalla este último nodo del grafo mediante tablas similares a las usadas en el epígrafe precio. Este nodo consta de las siguientes cuatro actividades:

1. *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.
2. *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.
3. *Retirada*, que será realizada conforme a la Retirada indicada en ISO/IEC 12207.

4. *Fin de la externalización*, que ocurrirá si ha existido relación de outsourcing entre la Organización de Mantenimiento y el Cliente.

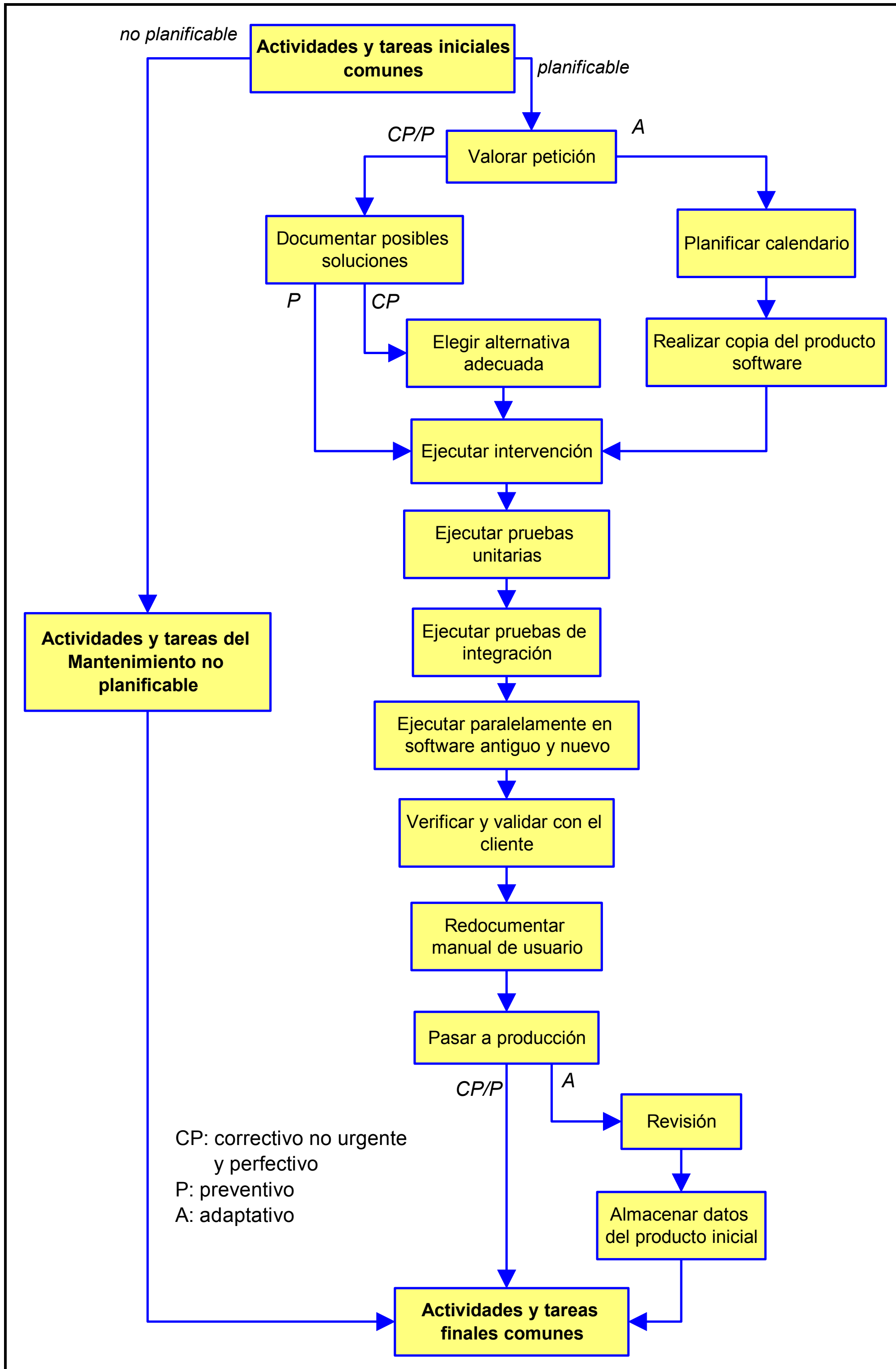



Figura 29. Estructura del mantenimiento planificable.

### 6.3.5. 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 12 se recogen algunos de ellos.

| | |
|---|---|
| 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 12. Algunos de los documentos generables durante el mantenimiento y descritos en MANTEMA.

### 6.3.6. 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, se propone también recoger otras diseñadas para bases de datos que se han elaborado durante la construcción de la metodología [Polo et al., 1998; Calero et al., 1999]. Es bien sabido que las bases de datos van teniendo cada vez mayor influencia sobre la complejidad de los sistemas de información de las organizaciones.

# 7. Referencias Bibliográficas

[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.

[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.

[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.

[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 (pp. 464-474). Los Alamitos, California: IEEE Computer Society.

[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 (pp. 250-259). Los Alamitos, California: IEEE Computer Society.

[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 (pp. 368-377). Los Alamitos, California: IEEE Computer Society.

[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.

[Berns, 1984]
Berns, G. "Assessing Software Maintainability". *Communications of the ACM*, enero 1984.

[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.

[Blank y Krijger, 1983]
Blank, J. y Krijger, M.J., *Software Engineering: Methods and Techniques*. Ed. John Wiley & Sons, Alemania 1983.

[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 (pp. 213-325). Los Alamitos, California: IEEE Computer Society.

[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*, 10(4). 249-278.

[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 (pp. 36-46). Los Alamitos, California: IEEE Computer Society.

[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.

[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 (pp. 150-156). Los Alamitos, California: 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.

[Chapin, 1987]
Chapin, N., “The Job of Sofware Maintenance”. En *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society. Estados Unidos, 1987.

[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.

[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 (pp. 142-148). Los Alamitos, California: IEEE Computer Society.

[Daly *et al.*, 1996]

Daly, J., Brooks, A., Miller, J., Roper, M. y Wood, M., "Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software". *Empirical Sofware Engineering*, nº 1, pp. 109-132. Holanda, 1996.

[Daly, 1979]

Daly, E.B., "Organizing for Successful Software Development". *Datamatio*n, vol. 25, diciembre 1979.

[De Young y Kampen, 1979]

De Young, G.E. y Kampen, G.R., "Program Factors as Predictors of Program Readability". *Proceedings of the Computer Software and Applications Conference* (COMPSAC). Ed. IEEE Computer Society Press, pp. 668-673, 1979.

[Dolado y Fernández, 1999]

Dolado, J. y Fernández, L. ¿Merece la pena usar los puntos de función? *Novática*, (140), 57-62.

[ECMA, 1990]

ECMA std. 149: *Portable Common Tool Enviroment (PCTE)*. European Computer Manufacturers Association, diciembre, 1990.

[Fanta y Rajlich, 1998]

Fanta, R. y Rajlich, V. *Reengineering Object-Oriented Code.* Proceedings of the International Conference on Software Maintenance (pp. 238-246). IEEE Computer Society Press.

[Fenton *et al.*, 1995]

Fenton, N., Whitty, R. y Iizuka, Y. (editores), *Software Quality Assurance and Measurement: A Worldwide Perspective*. Ed. International Thompson Computer Press. Reino Unido, 1995.

[Fenton y Pfleeger, 1997]

Fenton, N. E. y Pfleeger, S. L. *Software metrics. A rigorous & practical approach.* International Thomson Computer Press, 1997.

[Fioravanti et al., 1999]

Fioravanti, F., Nesi, P. y Polo, M. *Complexity/Size Metrics for Object-Oriented Systems*. Technical Report. (DSI-RT 17/99). Florencia, Italia: Department of Systems and Informatics, Università di Firenze.

[FIPS, 1984]

Federal Information Processing Standards Publication 106. *Guideline on Software Maintenance,* Estados Unidos, 1984.

[Frazer, 1992]

Frazer, A., "Reverse Engineering- hype, hope or here?" En *Software Reuse and Reverse Engineering in Practice*. Ed. Chapman & Hall, 1992.

[Freedman y Weinberg, 1990]

Freedman, D.P. y Weinberg, G.M., *Handbook of Workthroughs, Inspections and Technical Reviews*, 3ª edición. Ed. Dorset House, 1990.

[Garrigue, 1997]
Garrigue, E. “Using COBOL Defensive Traps”. *Software Maintenance: Research and Practice*, 9, pp. 329-342.

[Gartner, 1999]
Gartner Group: *Year 2000 World Status, 2Q99: The Final Countdown*. En http://www.gartnerweb.com/public/static/y2k/, consultado el 20/3/2000.

[Ghezzi *et al.*, 1991]
Ghezzi, C., Jazayeri, M. y Mandrioli, D., *Fundamentals of Software Engineering*. Ed. Prentice-Hall. Estados Unidos, 1991.

[Gilb, 1979]
Gilb, T.; A “Comment on the Definition of Reliability”. *ACM Software Engineering Notes,* vol. 4, nº 3, 1979.

[Gilb, 1988]
Gilb, T., *Principles of Software Engineering Management*. Ed. Addison-Welsey, Estados Unidos, 1988.

[Gill y Kemerer, 1991]
Gill, G. K. and Kemerer, C. F. “Cyclomatic complexity density and software maintenance productivity”. *IEEE Transactions on Software Engineering,* vol. 17, nº 12, dic. 1991.

[Grady, 1994]
Grady, R.B., “Succesfully Applying Software Metrics”. *Computer,* vol. 27, nº 9, pp. 18-25.

[Granja y Barranco, 1997]
Granja Álvarez, J. C. y Barranco García, M. “A mehod for estimating maintenance cost in a software project: a case study”. *Journal of Software Maintenance*, vol. 9, nº 3, mayo/junio, 1997.

[Griswold y Notkin, 1993]
Griswold, W.G. y Notkin, D. Automated Assistance for Program Restructuring. ACM Transactions on Software Engineering and Methodology, 2(3), 228-269.

[Hainaut *et al.*, 1994]
Hainaut, J-L., Henrard, J. , Hick, J-M., Roland, D. y Englebert, V. “Database design recovery”. *Proceedings of the Entity-RelationShip Approach*, ER’94. Loucopoulos (ed.), 1994.

[Halstead, 1977]
Halstead, M. *Elements of software science*. North-Holland, 1977.

[Hanna, 1993]
Hanna, M., Maintenance “Burden Begging for a Remedy”. *Datamation,* abril 1993, pp. 53-63.

[Harjani y Queille, 1992]
Harjani, D.R. y Queille, J.P. *A process model for the maintenance of large space systems software.* Proceedings of the Conference on Software Maintenance (pp. 127-136), Los Alamitos, California: IEEE Computer Society.

[Henderson-Sellers, 1996]
Henderson-Sellers, B. *Object-oriented metrics. Measures of complexity*, Prentice Hall, 1996.

[Hybertson *et al.*, 1997]
Hybertson, D. W., Ta, A. D., Thomas, W. M., "Maintenance of COTS-intensive Software Systems". *Journal of Software Maintenance: Research and Practice,* vol. 9, nº 4, 1997, pp. 203-216.

[IEEE, 1983]
IEEE, *Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Standard 729, 1983.

[IEEE, 1987]
ANSI/IEEE, std 1042: *Guide to Software Configuration Management,* Estados Unidos, 1987.

[IEEE, 1990]
ANSI/IEEE std. 610: *Standard Glossary of Software Engineering Terminology*. IEEE Computer Society. Estados Unidos, 1990.

[IEEE, 1990]
IEEE std 610: *Computer Dictionary,* Estados Unidos, 1990.

[IEEE, 1992]
ANSI/IEEE, std 1061: *Standard for a Software Quality Metrics Methodology.* IEEE Computer Society Press, Estados Unidos, 1992.

[IEEE, 1992]
IEEE, std 1209: *Recommend for Evaluation and Selection of CASE*. Estados Unidos, 1992.

[IEEE, 1993]
IEEE, std 1219: *Standard for Software Maintenance*. IEEE Computer Society Press. Estados Unidos, 1993.

[IEEE, 1995]
IEEE, std 1074: *Standard for Developing Software Life Cycle Processes*. IEEE Computer Society Press. Estados Unidos, 1995.

[ISO/IEC, 1991]
ISO 9126: *Software Product Evaluation: Quality Characteristics and Guidelines for their Use*. Suiza, 1991.

[ISO/IEC, 1994]
ISO 8402: *Quality Management and Quality Assurance Vocabulary*. Suiza, 1994.

[ISO/IEC, 1995]
International Standard Organization, ISO 12207: Information Technology-Software Life Cycle Processes. Suiza, 1995.

[ISO/IEC, 1998a]
ISO 9126: *Software Quality Characteristics and Metrics*. Canadá, 1998.

[ISO/IEC, 1998b]
ISO 14598: *Software Product Evaluation*. Canadá, 1998.

[ISO/IEC, 1999]
ISO/IEC 14764: *Software Engineering – Software Maintenance*. ISO/IEC JTC1/SC7 Secretariat. Canadá, 1999. (aprobado en noviembre de 1999).

[Jahnke, J.H. y Wadsack, J., 1999]
Jahnke, J.H. y Wadsack, J. Integration of Analysis and Redesign Activities in Information System Reengineering. Proceedings of the Third European Conference on Software Maintenance and Reengineering (pp. 160-168). Los Alamitos, Cali-fornia: IEEE Computer Society.

[Jones, 1991]
Jones, C. Applied Software Measurement. McGraw-Hill, 1996.

[Jorgensen, 1995]
Jorgensen, M. Experience with the Accuracy of Software Maintenance Task Effort Prediction Models. *IEEE Transactions on Software Engineering*, 21(8), 674-681.

[Kafura y Reddy, 1987]
Kafura, D., Reddy, G. R., "The Use of Software Complexity Metrics in Software Maintenance". *IEEE Transactions on Software engineering,* vol. Se-13, nº 3, 1987, pp. 335-343.

[Kilpi, 1997]
Kilpi, T., "New Challenges for Version Control and Configuration Management: a Framework and Evaluation", *Proceedings of the Euromicro 97: Conference on Software Maintenance and Reengineering*. Ed. IEEE Computer Science. Estados Unidos, 1997.

[Kopetz, 1979]
Kopetz, H., *Software Reliability*. Ed. Springer-Verlag, 1979.

[Lehman et al., 1998]
Lehman, M.M., Perry, D.E. y Ramil, J.F. (1998). Implications of Evolution Metrics on Soft-ware Maintenance. En Khoshgoftaar y Bennet (eds.), *Proceedings of the International Conference on Software Maintenance* (pp. 208-217). Maryland, USA: IEEE Computer Society. USA.

[Lehman y Belady, 1985]
Lehman, M.M., y Belady, L.; *Program Evolution: Processes of Software Change*. Ed. Academic Press, 1985.

[Lehman, 1980]
Lehman, M.M., "Programs, Life Cycles and Laws of Software Evolution". *Proceedings of the IEEE,* vol. 19, 1980, pp. 1060-1076.

[Letjer *et al.*, 1992]
Lejter, M.; Meyers, S. y Reiss, S.P., Support for Maintaining Object-Oriented Programs. *IEEE Transactions on Software Engineering,* vol. 18, nº 12. Ed. IEEE Computer Society, diciembre 1992.

[Lewis y Henry, 1989]
Lewis, H. y Henry, S., "A Methodology for Integrating Maintainability Using Software Metrics". *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society. Estados Unidos, 1989.

[Li y Cheng, 1987]
Li, H. F. y Cheng, W. K. "An empirical study of software metrics". *IEEE Transactions on Software Engineering* SE-12 (6), 679-708.

[Li y Henry, 1993]
Li, W. y Henry, S. "Object-oriented metrics that predict maintainability". *J. Systems Software*, 23, pp. 11-122.

[Lientz y Swanson, 1980]
Lientz, B.P. y Swanson, E.F., *Software Maintenance Management*. Ed. Addison-Wesley, 1980.

[Lorenz y Kidd, 1994]
Lorenz, M. y Kidd, J. *Object-oriented software metrics.* Prentice-Hall, Nueva Jersey.

[MAP, 1999]
Ministerio de Administraciones Públicas. *Situación de los sectores estratégicos de España en relación al Efecto 2000 (23-julio-1999)*. En www.map.es/a2000/pg7020_209.htm, consultado el 11/10/1999.

[Marciniak y Reifer, 1990]
Marciniak, J. J. y Reifer, D. J., *Software Acquisition Management: Managing the Acquisition of Custom Software Systems*. Ed. John Wiley & Sons. Estados Unidos, 1990.

[Mazza *et al.*, 1994]
Mazza, C., Fairclough, J., Melton, B., de Pablo, D., Scheffer, A. y Stevens, R., *Software Engineering Standards*. Ed. Prentice-Hall, 1994.

[Mazza *et al.*, 1996]:
Mazza, C. *et al.*, *Software Engineering Guides*. Ed. Prentice Hall. Reino Unido, 1996.

[McCabe, 1976]
McCabe, T., “A Software Complexity Measure”. *IEEE Transactions on Software Engineering,* vol. 2, nº 4, pp. 308-320, 1976.

[McCabe, 1982]
McCabe, T.J., “Structured Testing: A Software testing Methodology Using the Cyclomatic Complexity Metric”. *National Bureau of Standards Special Publications* pp. 500-99, 1982.

[McClure, 1992]
McClure, C., *The Three R's of Software Automation: Re-engineering, Repository, Reusability*. Ed. Prentice-Hall, USA 1992.

[McCracken, 1980]
McCracken, D., “Software in the 80's - Perils and Promises”. *Computerworld* (edición especial), vol. 14, nº 38, septiembre 1980.

[McDermid, 1991]
McDermid, J., *Software Engineering Reference Book*. Ed. Butterworth Heinemann, Alemania, 1991.

[McKee, 1984]
McKee, J.R. *Maintenance as a Function of Design.* Proceedings of the National Computer Conference, pp. 187-193.

[Melton, 1995]
Melton, A.(editor) *Software Measurement*. International Thompson Computer Press, 1995.

[Miller, 1979]
Miller, J.C., Software Re-Engineering: Getting it Done is Twice the Fun. En *Techniques of Program and System Maintenance*. QED Information Systems, 1979.

[MINER, 1996]
Ministerio de Industria y Energía, *Las Tecnologías de la Información en España*. Colección Informes y Estudios, 1996.

[MINER, 1998]
Ministerio de Industria y Energía. *Las Tecnologías de la Información en España 1998*. MINER – servicio de publicaciones, España.

[Nassi *et al.*, 1993]
Nassi, I. y Shneiderman, B. “Flowchart Techniques for Structured Programming”, *SIGPLAN Notices*, ACM, 1993.

[Natale, 1995]
Natale, D. *Qualità e quantità nei sistemi software. Teoria de esperienze*. Ed. FrancoAngeli, Milán.

[Nesi y Querci, 1998]
Nesi, P. y Querci, T. Effort estimation and prediction of object-oriented systems. *The Journal of Systems and Software*, (42), 89-102.

[Niessink y van Vliet, 1997]
Niessink, F. y van Vliet, H. *Predicting Maintenance Effort with Function Points*. Proceedings of the International Conference on Software Maintenance (pp. 32-39). Los Alamitos, California: IEEE Computer Society.

[Norman, 1992]
Norman, S., Dynamic Testing Tools, a Detailed Product Evaluation. Ed. Ovum, 1992.

[Oman *et al.*, 1991]
Oman P.W., Hagemeister, J. y Ash, D., *A Definition and Taxonomy for Software Maintainability*. Software Engineering Test Lab Technical Report–University of Idaho, 1991.

[Osborne y Chikofsky, 1990]
Osborne, W.M., Chikofsky, E.J., “Fitting Pieces to the Maintenance Puzzle”. *IEEE Software,* enero 1990, pp. 10-11.

[Osborne, 1989]
Osborne, W.M., “Building Maintainable Software”. *Handbook on Systems Management (Development and Support)*. Ed. Auerbach Pub, Estados Unidos, 1989.

[Pedro de Jesus y Sousa, 1998]
Pedro de Jesus, L. y Sousa, P. *Selection of Reverse Engineering Methods for Relational Databases*. Proceedings of the 3rd European Conference on Software Maintenance (pp. 194-197). Los Alamitos, California: IEEE Computer Society.

[Penteado et al., 1999]
Penteado, R., Masiero, P. y Cagnin, M. *An Experiment of Legacy Code Segmentation to Improve Maintainability*. Proceedings of the Third European Conference on Software Maintenance and Reengineering (pp. 111-119). Los Alamitos, California: IEEE Computer Society.

[Piattini *et al.*, 1996]
Piattini, M. G., Calvo-Manzano, J. A., Cervera, J. y Fernández, L., *Análisis y Diseño Detallado de Aplicaciones Informáticas de Gestión*. Ed. RA-MA, 1996.

[Piattini y Daryanani, 1995]
Piattini, M. G., Daryanani, S. N., *Elementos y Herramientas en el Desarrollo de Sistemas de Información*. Ed. RA-MA, 1995.

[Pigoski, 1996]
Pigoski, T. M., *Practical Software Maintenance. Best Practices for Managing Your Investment*. Ed. John Wiley & Sons. Estados Unidos, 1996.

[Plaisant *et al.*, 1997]
Plaisant, C., Rosse, A., Shneiderman, B. y Vanniamparampil, A.J. "Low-effort, high-payoff user interface reingeneering". *IEEE Software*, vol. 14, nº 4, julio-agosto, 1997.

[Polo et al., 1998]
Polo, M., Calero, C., Ruiz, F. y Piattini, M. *Métricas de calidad y complejidad para bases de datos.* Actas de las III Jornadas de Ingeniería del Software (pp. 79-90). Murcia: DM Librero-Editor.

[Polo et al., 1999]
Polo, M., Piattini, M., Ruiz, F. y Calero, C. Roles in the Maintenance Process. *ACM Software Engineering Notes*, 24(4), 84-86.

[Porter y Selby, 1990]
Porter, A. y Selby, R., "Empirically-guided Software Development using Metric-based Classification Trees". *IEEE Software,* vol. 7, nº 2, pp. 46-54, marzo, 1990.

[Poston y Sexton, 1992]
Poston, R. M. y Sexton, M.P., Evaluating and Selecting Testing Tools. *IEEE Software,* mayo, 1992, pp. 33-42.

[Premerlani *et al.*, 1994]
Premerlani, William J. y Blaha , Michael R., "An approach for Reverse Engineering of Relational Databases". *Communications of the ACM*, vol. 37, nº 5, mayo, 1994.

[Pressman, 1993]
Pressman, Roger S. *Ingeniería del Software, un enfoque práctico (3ª edición).* Editorial McGraw-Hill, 1993.

[Pressman, 1998]
Pressman, R.S., *Ingeniería del Software. Un Enfoque Práctico (4ª edición)*. Ed. McGraw-Hill Interamericana, España, 1998

[Putnam, 1978]
Putnam, L. "A general empirical solution to the macro software sizing and estimating problem". *IEEE Transactions on software engineering*, vol. 4, nº 4, 1978.

[Quang, 1993]
Quang, P. T., *Réussir la Mise en Place du Génie Logiciel et de l'Assurance Qualité*. Ed. Eyrolles. Francia, 1993.

[Rajlich y Adnapally, 1996]
Rajlich, V. y Adnapally, S. R., "VIFOR 2: A Tool for Browsing And Documentation." *Proceedings of International Conference on Software Maintenance,* Ed. IEEE Computer Society, Estados Unidos, 1996.

[Rock-Evans y Hales, 1990]
Rock-Evans, R. y Hales, K., *Reverse Engineering: Market, Methods and Tools,* 1990.

[Rombach, 1987]
Rombach, H.D., “A Controlled Experiment on the Impact of Sofware Structure on Maintainability”. *IEEE Transactions on Software Engineering,* vol. Se-13, nº 3, 1987, pp. 344-354.

[Rossi *et al.*, 1991]
Rossi, P., Antonini, P. y Lanza, T. “Experience of software maintainability in SIP”. *Proceedings of the International Conference on Data Engineering.* IEEE, 1996.

[Ruiz et al., 1999]
Ruiz, F., Piattini, M., Polo, M. y Calero, C. Maintenance types in the MANTEMA methodology. *International Conference on Enterprise Information Systems*. Portugal, 1999.

[Schach, 1990]
Schach S.R., *Sofware Engineering*. Ed. Irwin & Aksen. USA 1990.

[Schach, 1992]
Schach, S.R., *Practical Sofware Engineering*. Ed. Irwin & Aksen. USA 1992.

[Schamp, 1995]
Schamp, A., Configuration Management Tool Evaluation and Selection. *IEEE Software,* junio, 1995, pp. 114-118.

[Schneidewind, 1987]
Schneidewind, N.F. “The State of Software Maintenance”. *Conference on Software Maintenance-1985,* IEEE, noviembre 1985, pp. 114-119.

[Sellink et al., 1999]
Sellink, A., Sneed, H. y Verhoef, C. Reestructuring of COBOL/CICS Legacy Systems. Proceedings of the Third European Conference on Software Maintenance and Reengineering (pp. 72-82). Los Alamitos, California: IEEE Computer Society.

[Sharble y Cohen, 1993]
Sharble, R.C. y Cohen, S.S. “The object-oriented brewery: a comparison of two object-oriented development methods”. *ACM SIGSOFT Software Engineering Notes*, vol. 18, nº 2, pp. 60-73.

[Shepperd et al., 1996]
Shepperd, M., Schofield, C. and Kitchenham, B. *Effort Estimation Using Analogy*. Proceedings of 18th International Conference on Software Engineering. Los Alamitos, Cali-fornia: IEEE Computer Society Press.

[Singer, 1998]
Singer, J. *Practices of Software Maintenance*. Proceedings of the International Conference on Software Maintenance (pp. 139-145). Los Alamitos, California: IEEE Computer Society.

[Slaughter y Banker, 1996]
Slaughter, S.A. y Banker, R.D., “A Study of the Effects of Software Development Practices on Software Maintenance Effort”. *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society. Estados Unidos, 1996.

[Smith, 1999]
Smith, D. S., Designing Maintainable Software. Ed. Springer-Verlag, USA, 1999.

[Sneed, 1991]
Sneed, H.M., "Economics of Software Re-engineering". *Journal of Software Maintenance,* vol. 3, nº 3, 1991, pp. 163-182.

[Sneed, 1995]
Sneed, Harry M. "Planning the Reengineering of Legacy Systems". *IEEE Software,* vol. 12, nº 1, enero de 1995.

[Sneed, 1997]
Sneed, H. M. *Proceedings of the First Euromicro Conference on Software Maintenance and Reengineering,* mayo 1997, pp. 119-127.

[Sommerville, 1992]
Sommerville, I.., *Software Engineering* (4ª edición). Ed. Addison-Wesley, USA, 1992.

[Sorrentino y De Gregori, 1995]
Sorretino, M. y De Gregori, G., *La Manutenzione del Software Applicativo*. Ed. FrancoAgeli, Italia, 1995.

[Stoneman, 1980]
Stoneman, Requirements for Ada Programming Support Environments. Ed. Dep. of Defence. Estados Unidos, febrero, 1980.

[Swanson, 1976]
Swanson, E.B., "The Dimensions of Maintenance. Proceedings of 2nd International Conference on Software Engineering". *IEEE*, octubre 1976, pp. 492-497.

[Taschwer et al., 1999]
Taschwer, M., Rauner-Reithmayer, D. y Mittermeir, R. *Generating Objects from C Code. Features of the CORET Tool-Set*. Proceedings of the Third European Conference on Software Maintenance and Reengineering (pp. 91-100). Los Alami-tos, California: IEEE Computer Society.

[Waters *et al.*, 1994]
Waters, Richard C. y Chikofsky, E., "Reverse Engineering, Progress along many dimensions". *Communications of the ACM*, vol. 37, nº 5, mayo 1994.

[Weide, 1995]
Weide, Bruce W., Heym, Wayne D. y Hollingsworth, Joseph E. "Reverse Engineering of Legacy Code Exposed", *Proceedings of the 17th International Conference on Software Engineering*, abril 1995.

[Welker y Oman, 1995]
Welker, K. y Oman, P., "Software Maintainability Metrics Models in Practice". CROSSTALK: *The Journal of Defense Software Engineering*. 8 (11): 19-32.

[Weyuker, 1988]
Weyuker, E.J. "Evaluating software complexity measures". *IEEE Transaction on Software Engineering*, vol. 9, nº. 19, pp. 1357-1365.

[Wilde y Huitt, 1992]
Wilde, N. Y Huitt, R., "Maintenance Support for Object-Oriented Programs". *IEEE Transactions on Software Engineering,* vol. 18, nº 12, pp. 1038-1044, diciembre, 1992.

[Yourdon, 1975]
Yordon, E., *Techniques of Program Structure and Design*. Ed. Prentice-Hall, 1975.

[Zage et al., 1994]
Zage, W.M., Zage, D.M. y Wilburn, C. *Achieving Software Quality Through Design Metrics Analysis.* Proceedings of the Twelfth Annual Pacific Northwest Software Quality Conference, Portland, OR.

[Zelkowitz *et al.*, 1979]
Zelkowitz, M.V., Shaw, A.C. y Gannon, J.D., *Principles of Software Engineering and Design*. Ed. Prentice-Hall, USA, 1979.