Tema 2. Mantenimiento y evolución del software

Desarrollo de Software
Curso 2022-2023
3º - Grado Ingeniería Informática

Dpto. Lenguajes y Sistemas Informáticos
ETSIIT
Universidad de Granada

14 de marzo de 2023

Tema 2. Mantenimiento y evolución del software

Contenidos

Mantenimiento y evolución del software

Este capítulo ha sido extraído principalmente del primer capítulo del libro de Priyadarshi Tripathy y Kshirasagar Naik titulado *Software Evolution and Maintenance* (Tripathy and Naik, 2014). Para evitar repetidas referencias al mismo trabajo, sólo se citarán de forma explícita otros autores, entendiéndose que el texto no citado es una traducción literal o resumida del libro mencionado.

2.1. Evolución vs mantenimiento

El concepto de *mantenimiento software* se refiere a corregir los errores del software que le impiden que realice las funcionalidades previstas en la entrega. Así, todas las actividades de soporte realizadas después de la entrega del software se incluyen en la categoría de mantenimiento. El mantenimiento de los sistemas de software significa principalmente corregir errores pero preservar sus funcionalidades. Las tareas de mantenimiento están muy planificadas, como por ejemplo la que se refiere a la corrección de errores. Además de las actividades planificadas, también se realizan actividades no planificadas. Por ejemplo, puede surgir un nuevo uso del sistema. En general, el mantenimiento no implica realizar cambios importantes en la arquitectura del sistema. En otras palabras, el mantenimiento significa mantener un sistema instalado funcionando sin cambios en su diseño.

El concepto de *evolución software* se refiere a un cambio continuo del software desde un estado menor, más simple o peor, a un estado más alto o mejor. Así, todas las actividades realizadas para efectuar cambios en los requisitos se consideran dentro de la categoría de evolución. La evolución de un sistema software significa ampliar un diseño que ya existe. Algunos ejemplos son la adición de funcionalidad, la mejora del rendimiento del sistema o el hacerlo ejecutable en un sistema operativo diferente. Conforme pasa el tiempo, las partes interesadas comprenden mejor lo que el sistema debe hacer y ponen manos a la obra para que se acerque al nuevo concepto que tienen del mismo. Por lo tanto, el sistema evoluciona de varias maneras, pero siempre es primero la persona humana la que empieza a "soñarlo" de manera distinta.

Evolución software

A continuación se exponen las primeras tres leyes de la evolución del software (Belady and Lehman, 1976):

- Ley de cambio continuo.- A menos que un sistema se modifique continuamente para satisfacer las necesidades emergentes de los usuarios, el sistema se vuelve cada vez menos útil.
- Ley de entropía/complejidad creciente.- A menos que se realice un trabajo adicional para reducir explícitamente la complejidad de un sistema, el sistema se volverá cada vez más complejo y menos estructurado debido a los cambios relacionados con elmantenimiento.
- Ley de crecimiento estadísticamente suave o autorregulación.- El proceso de evolución puede parecer aleatorio en vistas puntuales de tiempo y de espacio pero a lo largo del tiempo, se aprecia a nivel estadístico una autorregulación (ley de conservación) cíclica, con tendencias bien definidas a largo plazo. Así, las medidas del nivel de mantenimiento requerido por los distintos productos y procesos, que se producen durante la evolución, siguen una distribución cercana a la normal.

Se han dado hasta ocho leyes de evolución, pero algunas son bastante controvertidas. En concreto se considera que pueden variar según se estudie la evolución en sistemas software

registrados o patentados y monolíticos, software académico, software industrial, software gratuito, software de código abierto, etc.

2.1.1. Mantenimiento software

E. Burton Swanson definió inicialmente tres categorías de actividades de mantenimiento software, a saber, *correctivo*, *adaptativo* y *perfectivo*. Esas definiciones se incorporaron posteriormente a la ingeniería del software estándar (procesos del ciclo de vida del software). Posteriormente se introdujo una cuarta categoría llamada mantenimiento *preventivo*, aunque algunos investigadores y desarrolladores la consideran como un subconjunto del mantenimiento perfectivo.

La clasificación de Swanson de las actividades de mantenimiento se basa en el criterio de la intención del ingeniero de mantenimiento, pues reflejan las distintas intenciones de realizar tareas de mantenimiento específicas en el sistema. Así, la intención de una actividad depende de las motivaciones para el cambio.

Otro criterio de clasificación de las modificaciones al software se basa en las actividades que son realizadas:

- Actividades para hacer correcciones (similar al mantenimiento correctivo de Swanson).- Si hay discrepancias entre el comportamiento esperado de un sistema y el comportamiento real, entonces se realizan algunas actividades para eliminar o reducir las discrepancias.
- Actividades para realizar mejoras.- Se realizan una serie de actividades para producir un cambio en el sistema, cambiando así el comportamiento o la implementación del sistema. Esta categoría de actividades se refina aún más en tres subcategorías:
  - mejoras que modifican los requisitos existentes;
  - mejoras que crean nuevos requisitos; y
  - mejoras que modifican la implementación sin cambiar los requisitos.

PREGUNTA 2.1.1. Evolución de software y tipos de mantenimiento

¿Qué relación encuentras entre el concepto de evolución de software del apartado anterior y los tipos de mantenimiento vistos ahora?

Especificidades de mantenimiento del software comercial listo para usar

No es lo mismo enfrentarse a la tarea de mantener un sistema con código desarrollado por completo a medida, *software a medida*, que enfrentarse a las tareas, más habituales en la actualidad, de desarrollar *sofware basado en componentes* (CBS, del inglés *Component Based Software system*) de fuentes heterogéneas, en el que los distintos componentes provienen de distintas fuentes de *software comercial listo para usar* (COTS, de inglés *Commercial Off-the Shelf*) que se proporciona como componentes reutilizables, además de fuentes de desarrollo

propias y de posiblemente código abierto. Las características específicas más importante del mantenimiento de CBSs de fuentes heterogéneas, incluyendo diversos componentes COTS son:

- Mantenimientos divididos.- Cada producto COTS lo mantiene el equipo de mantenimiento de la empresa que lo proporciona, que es distinto al equipo de la empresa que lo aplica en su propio software. Aunque los primeros pueden dar soporte a los segundos, el tener objetivos comerciales distintos puede añadir ciertas complicaciones.

- Habilidades específicas.- El que trabaja en un equipo de mantenimiento de un software basado en componentes, ve cada componente como una caja negra y se especializa más en la integración de los mismos.

- Comunidad de usuarios mayor: confianza vs control.- Siempre tiene ventajas que sean más los usuarios que utilizan un producto para garantizar un mejor mantenimiento. Aunque no podemos controlar los cambios concretos en un producto desarrollado por terceras partes, hay que confiar en que el mayor número de usuarios a la larga dirigirá la evolución del componente hacia un lugar más provechoso.

- Desplazamiento de los costes.- Generalmente es más aconsejable usar software que ya está preparado que desarrollar el nuestro propio, si se ajusta bien a nuestras necesidades. Sin embargo, al ahorro en el desarrollo y mantenimiento de software propio hay que descontarle los gastos de adquisición (licencia) y de mantenimiento externos y los que se deriven de cambios importantes futuros que pudieran afectar al resto de los componentes de nuestro propio sistema.

- Planificación más difícil.- No se puede predecir cuándo se harán los cambios por las distintas empresas que desarrollan COTS usados por nuestro sistema.

2.1.2. Modelos y procesos de mantenimiento y evolución software

Se han propuesto muchos modelos que explican la labor de mantenimiento software. En el clásico modelo en cascada del ciclo de vida software, consiste en una única fase final, que empieza una vez que el software está en explotación.

Uno de estos modelos, explica el mantenimiento a su vez como un proceso en fases, con las siguientes etapas:

- Desarrollo inicial.- Aún no ha empezado el mantenimiento, y el sistema está constantemente siendo modificado hasta que alcanza el estado de estabilidad suficiente para lanzarlo y empezar a usarlo.

- Evolución.- Nada más lanzado es fácil hacer cambios simples. Los más complejos llevan más costos y riesgos, pero son asumibles en su gran mayoría. Los desarrolladores pueden encargarse del mantenimiento aunque se dediquen a otras tareas, pues tienen el sistema fresco y les cuesta poco mantenerlo.

- Servicio de mantenimiento.- Poco a poco sus desarrolladores ya no están disponibles y el nuevo equipo de desarrollo se dedica a hacer sobre todo mantenimiento correctivo (arreglar errores). Ya no se invierte en cambios mayores y cualquier cambio es mucho más costoso.

- Retirada gradual ("phaseout").- En algún momento el servicio de mantenimiento mínimo se hace demasiado caro o bien aparecen soluciones mejores. Se tiene que planificar el cambio al nuevo sistema, incluyendo migración de datos y uso de patrones fachada o envoltorios. Cuando el nuevo sistema está completamente probado, el antiguo termina su servicio (a veces pueden funcionar durante cierto tiempo en paralelo). En esta fase, en la que el software antiguo aún se está usando pero ya se está desarrollando uno nuevo, a la parte del antiguo que se pretende "reutilizar" y formará parte del nuevo de alguna manera, se le suele denominar *software heredado* (en inglés, *legacy software*).

2.1.3. Estudio de impacto software

El *análisis o estudio de impacto* es una tarea que pretende examinar los efectos potenciales que puede tener en las distintas partes del sistema la realización de una tarea concreta de mantenimiento. Se suelen usar dos tecnología distintas para describir dicho impacto:

- *Análisis de trazabilidad*.- Pretende hacer un rastreo de todas las partes posibles afectadas o *artefactos de alto nivel* (diseño, código, casos de prueba, etc.) usando un modelo de asociación entre los artefactos a modificar y los que pueden verse afectados.

- Análisis de dependencias.- Se describen los cambios a nivel de dependencias semánticas entre las distintas entidades del producto software mediante su identificación con las dependencias sintácticas que se extraen del código.

Un buen estudio de impacto debe incluir también un *análisis de los efectos de propagación en cadena* (en inglés, *ripple effect analysis*) que puede llevar un cambio. Se trata de ver el producto software como un todo evolutivo, un sistema, de forma que para cada nuevo cambio se mide:

- El cambio (aumento o disminución) de la complejidad del sistema con respecto a la versión anterior.

- Las diferencias en niveles de complejidad de las distintas partes del sistema.

- El efecto que tiene un nuevo módulo para la complejidad total del sistema.

2.1.4. Reingeniería

La *reingeniería* se define como la transformación de un sistema simple en otro mejor. Una forma alternativa de ver la evolución software y, en general, el ciclo de vida del software, es considerar un modelo en un nivel de abstracción más elevado en el que no se describa el proceso que sufre cada producto software sino el de un sistema completo que va evolucionando

a través de distintos productos software que se suceden en el tiempo, de forma que cada producto parte del anterior, y no hay límite en la evolución. En este sentido, la evolución software puede verse como un proceso iterativo de reingeniería. En el análisis de efectos de propagación en cadena visto en el apartado anterior, se pone el énfasis en considerar el producto software como sistema evolutivo. Aún así, es un sistema que nace y muere. La reingeniería no pone la vista en el producto software concreto con su ciclo de vida, sino en un proceso a un nivel más alto de abstracción, formado por una cadena de productos software, en la que cada producto antes de morir ha permitido el nacimiento de otro nuevo, tratándose así de un proceso en mejora continua.

Una definición más detallada es considerar la reingeniería como el examen y análisis de un software existente (especificación, diseño, implementación y documentación), reestructurándolo y concibiéndolo de otra forma a partir de la cual se vuelva a implementar, mejorando la funcionalidad y los atributos de calidad del sistema, tales como capacidad de evolución, rendimiento, reusabilidad, eficiencia, portabilidad, etc. Se trata de pasar de un "mal" sistema a uno "bueno", aunque puede haber algunos riesgos: (1) que la funcionalidad anterior no se mantega; (2) que la calidad sea inferior; y (3) que los beneficios no se consigan en el tiempo esperados.

Razones o necesidades por las que se decide hacer reingeniería

Para llevarla a cabo, al menos debe surgir una de las siguientes necesidades:

- Mejorar la mantenibilidad.- Por la segunda ley de Lehman (*complejidad incremental*), el mantenimiento de un sistema aumenta con el tiempo hasta hacerse demasiado difícil y costoso. En algún momento habrá que hacer uno nuevo con interfaces más explícitas y módulos funcionales más relevantes, actualizando además toda la documentación interna y externa del sistema.

- Migrar a una nueva tecnología.- Por la primera ley de Lehman (*cambio continuo*), los sistemas están en continua adapatación a su entorno siendo cada vez menos satisfactorios. La rapidez con la que avanza la tecnología hace que el software caduque más rápidamente. Además el software antiguo es más difícil de ser mantenido por las empresas, siendo cada vez más incompatible y caro de mantener. También los empleados van cambiando a las nuevas tecnologías, habiendo cada vez menos empleados que sean capaces de mantener sistemas antiguos. Así, muchas empresas con software operativo y útil se ven forzadas a migrarlos a plataformas de ejecución más modernas que incluyen nuevo hardware, sistema operativo y/o lenguaje.

- Mejorar la calidad.- Según la séptima ley de Lehman (*descenso de calidad*), las partes interesadas en un software perciben una bajada de calidad en los sistemas que no se mantienen de forma rigurosa ni se adaptan al entorno. Cada cambio produce efectos en cadena, a veces causando más problemas que mejoras hasta que la fiabilidad del software se hace insostenible. En ese momento se hace necesaria la reingeniería para conseguir aumentar la fiabilidad del mismo.

- Prepararse para una mejora de la funcionalidad.- La sexta ley de Lehman (*crecimiento continuo*) afirma que la funcionalidad del software debe estar constantemente ampliándose para mantender la satisfacción del usuario con ese software a lo largo de su tiempo de vida. Esta ley refleja el conjunto ilimitado de posibles mejoras en su funcionalidad. A menudo la reingeniería más que buscar añadir funcionalidad busca mejorar la facilidad de añadir funcionalidad en el futuro, por ejemplo cambiando el estilo arquitectónico o el paradigma de programación.

Conceptos de reingeniería

DEFINICIÓN 2.1.1: Ingeniería directa

Se trata de pasar del nivel más alto de abstracción en la representación de un sistema, donde solo se muestran las características más relevantes de un sistema escondiendo los detalles (principio de abstracción), al nivel más bajo del mismo, con el máximo detalle de sus distintos aspectos (principio de refinamiento).

Los pasos o actividades recorridas en la ingeniería directa empiezan por la formulación conceptual del sistema para indentificar los requisitos y terminan con la implementación del diseño.

DEFINICIÓN 2.1.2: Ingeniería inversa

Se trata del movimiento contrario por el que se pasa de los niveles más altos de detalle a los niveles más altos de abstracción.

Lo pasos o actividades de la ingeniería inversa empiezan por el análisis del software para entender sus componentes y la relación entre los mismos y terminan por la representación del sistema en un nivel más alto de abstracción o de una forma distinta.

Algunos ejemplos de ingeniería inversa son la decompilación (el código se traduce en un programa de alto nivel), la extracción arquitectónica (se deriva el diseño del programa a partir de su código), la generación de documentación (se produce información a partir del código fuente para comprender mejor el programa) y la visualización software (se dibujan algunos aspectos del programa con alguna forma de abstracción).

La figura 2.1 muestra la relación entre abstracción, refinamiento y cuantro niveles distintos de abstracción/refinamiento de un sistema software: (1) conceptual (**¿por qué** ese software?), (2) de requerimientos (**¿qué** hace el software?), (3) de diseño (**¿cómo** hace para conseguir esa funcionalidad?), y (4) de implementación (**¿cómo exactamente** es implementado el sistema?).

Junto a los principios de abstracción y refinamiento, en la reingeniería también interviene el principio de *alteración*, por el cual se hacen cambios en la representación de un sistema, sin cambiar el nivel de abstracción.

Un caso específico de alteración que no cambia el comportamiento externo del sistema es la *reestructuración*.

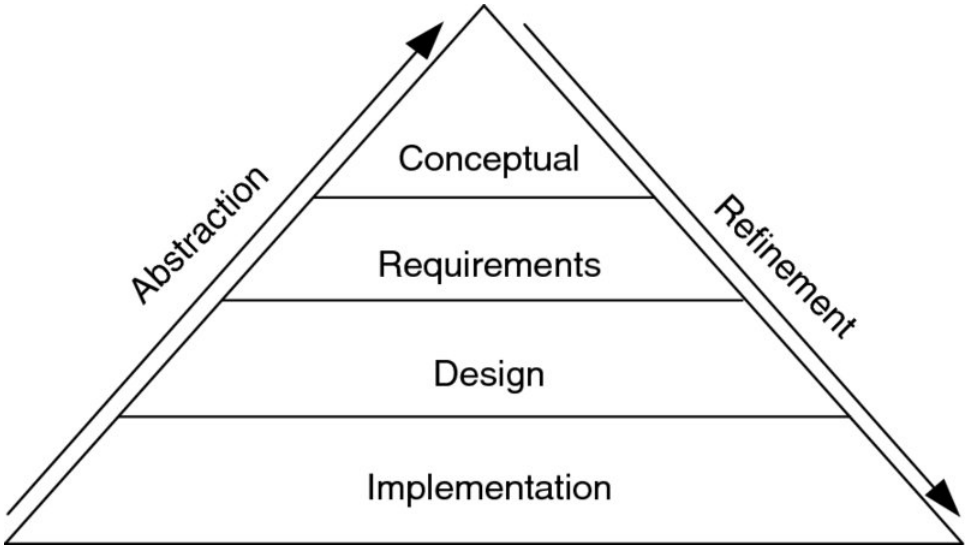

Figura 2.1: Niveles de abstracción y refinamiento. [Fuente: (Tripathy and Naik, 2014, cap. 4)].

En el proceso de reingeniería se da la secuencia de abstracción->alteración->refinamiento. La figura 2.2 muestra esta secuencia.

La reingeniería ha sido definida de forma clásica con la siguiente ecuación:

$$\text{Reingeniería} = \text{ingeniería inversa} + \Delta + \text{ingeniería directa}.$$

Veamos los tres componentes de la parte derecha de la ecuación:

- La *ingeniería inversa* consiste en la re-elaboración del modelo que representa al sistema actual de una forma más abstracta y fácil de entender. El punto de partida es el código actual y el resultado es una nueva DA o diseño de alto nivel del producto actual. No se trata todavía en esta fase de cambiar nada, sino de ver en profundidad, de examinar, el sistema actual.
- El segundo componente de la parte derecha de la ecuación, la *alteración* representa a la actividad de decidir los cambios que se harán para producir un nuevo producto software a partir del examen del producto actual. Se trata del proceso de especificación de requisitos y elaboración de una DA, pero partiendo del producto anterior.
- La *ingeniería directa* consiste en el desarrollo del nuevo producto software a partir de la nueva DA.

La Figura 2.3 muestra los distintos niveles de alteración que se corresponden con los distintos niveles de abstracción/refinamiento del código: repensar (cuando se cambia el sistema a nivel conceptual), re-especificar (cuando se cambia a nivel de requerimientos), rediseñar (cuando se cambia a nivel de diseño) y recodificación (cuando se cambia a nivel de implementación).

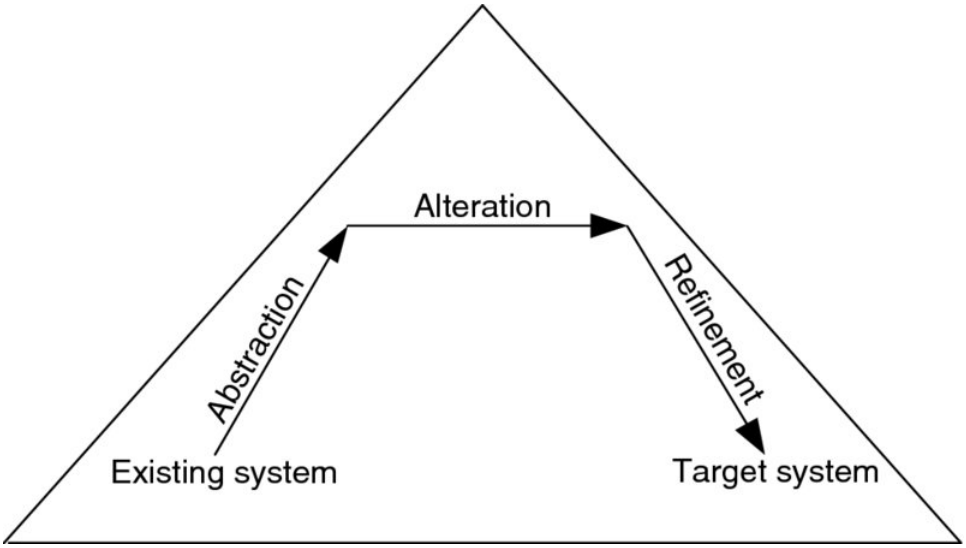

Figura 2.2: Bases conceptuales en el proceso de reingeniería. [Fuente: (Tripathy and Naik, 2014, cap. 4)].

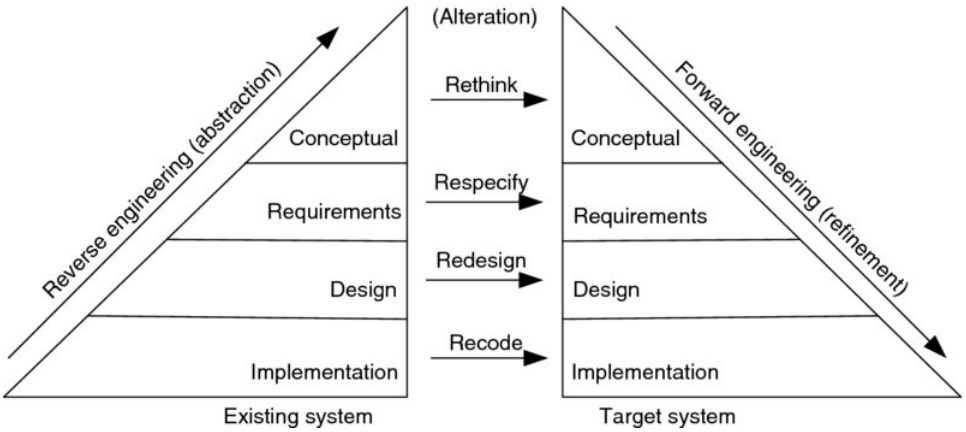

Figura 2.3: Bases conceptuales en el proceso de reingeniería. [Fuente: (Tripathy and Naik, 2014, cap. 4)].

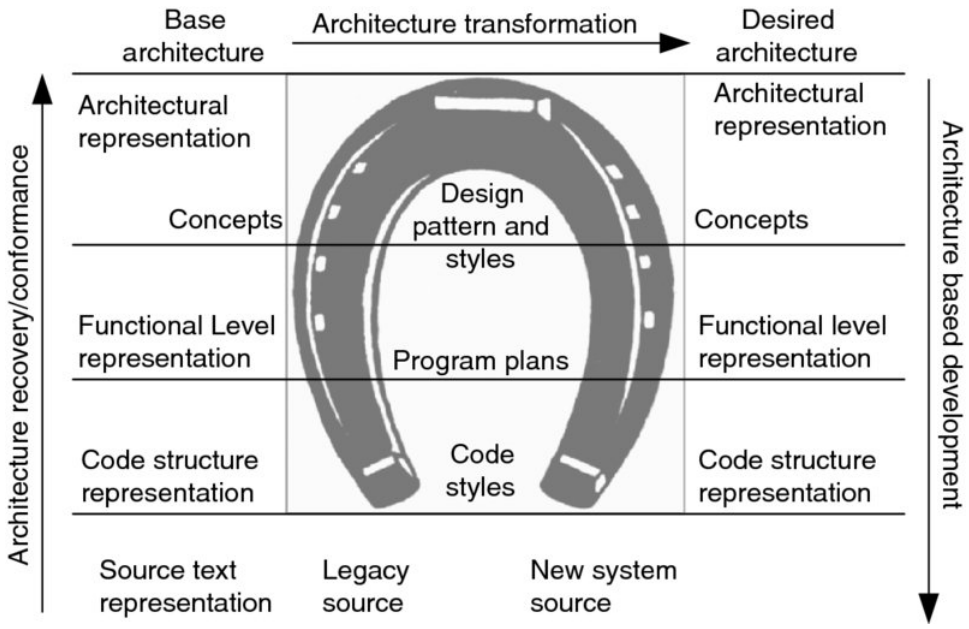

Figura 2.4: Bases conceptuales en el proceso de reingeniería. [Fuente: (Tripathy and Naik, 2014, cap. 4)].

En la figura 2.4 puede verse un modelo similar, llamado modelo de herradura, en el proceso de reingeniería. Obsérvese que este modelo reduce a tres los niveles de abstracción (arquitectónica, funcional y de código, de mayor a menor nivel). Bajo este modelo, los patrones y estilos de diseño afectan al nivel superior, el arquitectónico, que trabaja con los conceptos del sistema.

2.1.5. Software heredado (legacy software)

A menudo las empresas tienen sistemas software que realizan funciones vitales pero que ya no se pueden mantener con facilidad por ser muy antiguos, escritos en lenguajes de programación obsoletos, mal documentados, con una mala gestión de los datos, con una estructura degradada después de sufrir muchas modificaciones, muy pocas personas tienen experiencia para hacer mínimos cambios, etc.

DEFINICIÓN 2.1.3: Software heredado

Sistema software casi imposible de mantener pero cuya funcionalidad es necesario mantener.

Si la empresa no puede prescindir de él, descarta seguir su mantenimiento y también descarta empezar desde cero, se presentan solo dos operaciones posibles:

- Envoltura (wrap).- Es una técnica de caja negra: se construye una capa exterior con una apariencia moderna y mejorada y se diseñan las interfaces para interaccionar

con él, manteniendo ocultas la complejidad de sus interfaces, datos, módulos, etc. Se trata de una alternativa al mantenimiento. La figura 2.5 muestra un diagrama de un sistema envoltura. El *envoltorio* (*wrapper*) es un programa que recibe los mensajes desde el programa cliente y transforma la entrada en una representación que puede enviar al objetivo –el sistema heredado–. El *envoltorio* también intercepta la salida del objetivo, la transforma para hacerla inteligible al cliente y se la envía. Tiene para ellos los siguientes elementos (ver figura 2.5):

- Interfaz externa.- Generalmente se usa paso de mensajes y los datos del mensaje son cadenas ASCII.
- Interfaz interna.- Debe especificarse en el lenguaje del objetivo (el software heredado).
- Gestor de mensajes.- Usa buffers de entrada y salida de mensajes para almacenar los datos que necesitan esperar dadas las distintas velocidades de procesamiento del cliente y el objetivo.
- Convertidor de interfaces.- Se encarga de cambiar los parámetros necesarios para adaptarlos a cada una de las interfaces.
- Emulador E/S.- Es el módulo que intercepta las entradas al objetivo y las salidas del mismo y pone la información en el buffer correspondiente.

- Migración.- Se trata de realizar reingeniería, de forma que el sistema sea exportado a una plataforma moderna, manteniendo la mayor parte de su funcionalidad y provocando los mínimos cambios en el modelo de negocio. Es más barato y rápido que empezar desde cero, pues se reutilizan partes importantes del mismo.

Métodos de migración

Existen diversos enfoques para llevar a cabo la migración. La elección del enfoque a aplicar dependerá del tipo concreto de software heredado, pues los distintos métodos presentan distintos niveles de complejidad, tamaño y riesgo de fallo al realizar la migración. Uno de los factores más decisivos que distinguen a los distintos métodos son las distintas formas de llevar a cabo la migración de los datos. Veremos tres de ellos:

***Migración dirigida* (*forward migration*) o *Base de datos primero* (*database first*)**
Primero se migran los datos a un Sistema de Gestión de Bases de Datos (SGBD) más moderno y después de forma gradual el software y sus interfaces. Mientras concluye el segundo paso, conviven ambos sistemas mediante una *pasarela dirigida* (*forward gateway*) que media entre los distintos componentes software (ver figura 2.6).

***Migración inversa* (*reverse migration*) o *Base de datos después* (*database last*)**
Los programas se van migrando de forma incremental pero permanece la base de datos del sistema heredado en su plataforma original. La interacción entre ambos sistemas de

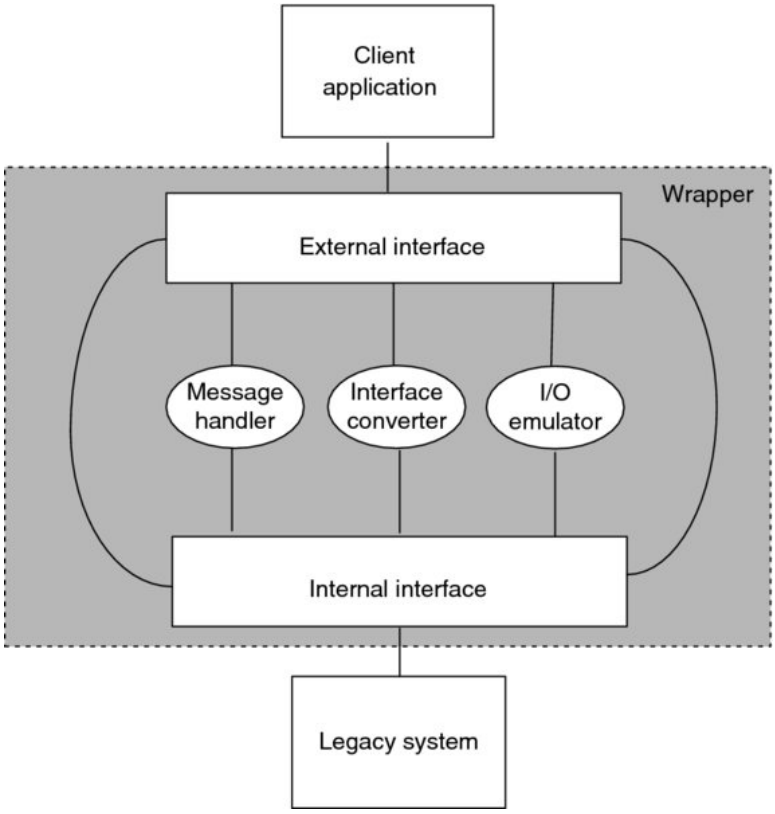

Figura 2.5: Módulos de un entorno envoltorio. [Fuente: (Tripathy and Naik, 2014, cap. 5)].

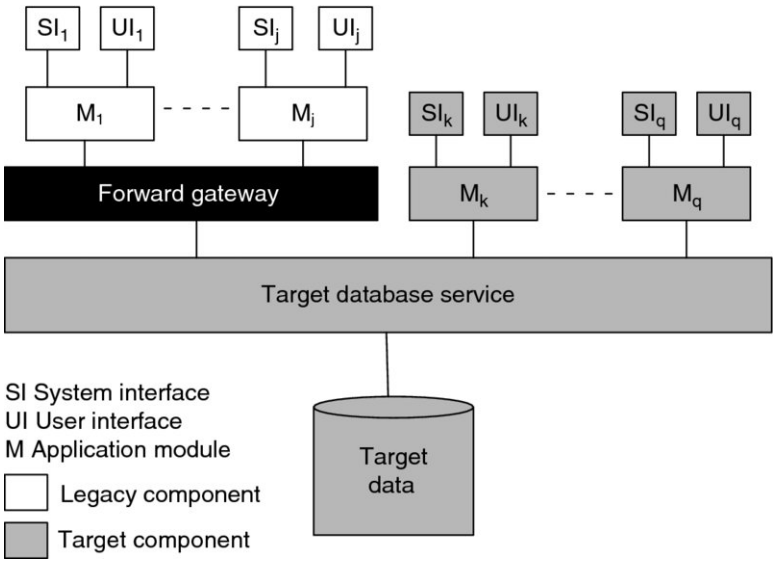

Figura 2.6: Enfoque *Base de datos primero*. [Fuente: (Tripathy and Naik, 2014, cap. 5)].

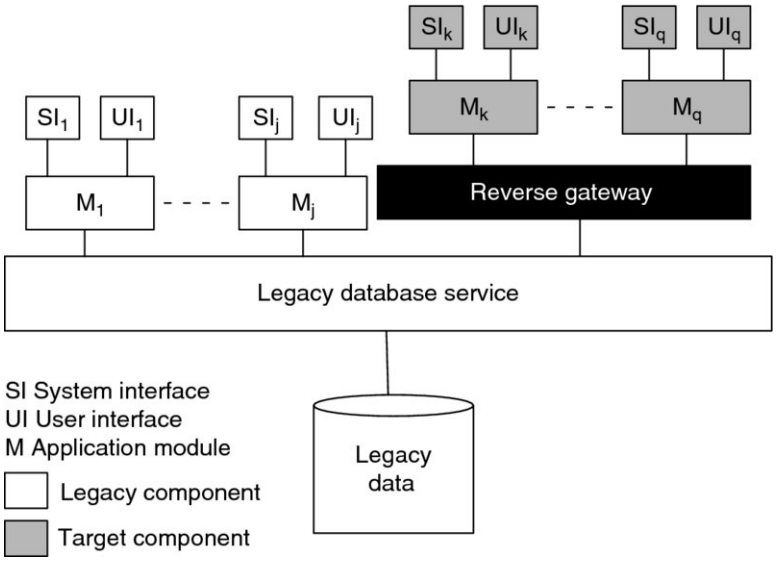

Figura 2.7: Enfoque *Base de datos después*. [Fuente: (Tripathy and Naik, 2014, cap. 5)].

información durante el tiempo de la migración se hace mediante una *pasarela inversa* (ver figura 2.7).

Base de datos compuesta Combina los dos enfoques anteriores (ver figura 2.8): Los programas van migrando de forma incremental y en ese período en el que se usan los dos sistemas conviven también las dos bases de datos, mediante un sistema compuesto que usa una pasarela dirigida y otra inversa. Los datos pueden estar duplicados en las dos bases de datos y se usan *coordinadores de las transacciones* para asegurar su integridad. Estos coordinadores interceptan las peticiones de actualización de las bases de datos por parte del software heredado y nuevo, asegurándose de que los datos duplicados se actualicen en ambas bases de datos.

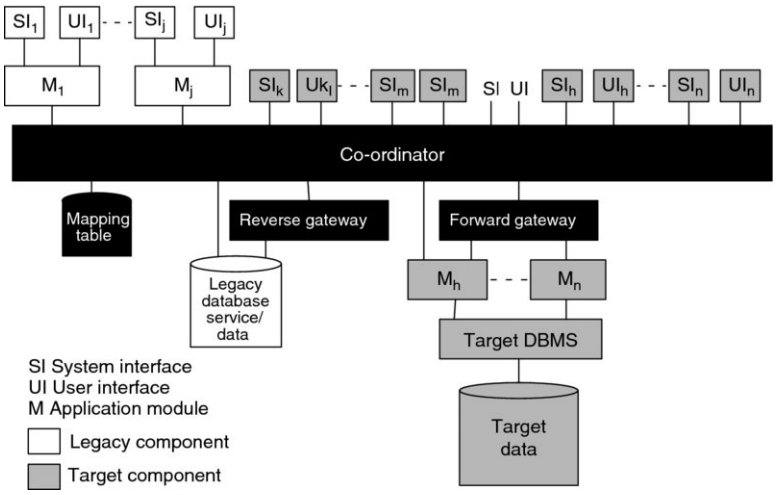

Figura 2.8: Enfoque *Base de datos compuesta*. [Fuente: (Tripathy and Naik, 2014, cap. 5)].

2.1.6. Refactorización y reestructuración

Reestructurar significa realizar cambios en la estructura del software para mejorar su calidad interna, es decir, para hacerlo más fácil de comprender y mantener, sin cambiar el comportamiento observable del sistema. En lenguajes OO, la reestructuración se limita al nivel de una función o un bloque de código. En sistemas OO, con lenguajes mucho más ricos que usan interfaces, ligadura dinámica, herencia, sobreescritura, polimorfismo, etc. la reestructuración es más compleja y se la conoce con el nombre de *refactorización*.

La refactorización se logra mediante la eliminación del código duplicado, la simplificación del código y el movimiento del código a una clase diferente, entre otros. Sin una refactorización continua, la estructura interna del software se hará cada vez más difícil de comprender, debido al mantenimiento periódico. Por lo tanto, la refactorización regular permite mantener una buena estructura. En una metodología de software ágil, como eXtreme Programming (XP), la refactorización se aplica continuamente con los siguientes objetivos:

- proporcionar estabilidad en la arquitectura,
- proporcionar legibilidad al código, y
- facilitar el mantenimiento perfectivo y la evolución, flexibilizando las tareas de integración de nuevas funcionalidades en el sistema.

La refactorización no añade nuevos requisitos al sistema existente. Otro aspecto de la refactorización es mejorar la estructura interna del sistema. También se puede aplicar reestructuración para transformar código heredado y migrarlo a otro lenguaje de programación. Así, la reestructuración y refactorización se pueden utilizar para rediseñar sistemas software.

2.1.7. Identificar qué refactorizar

DEFINICIÓN 2.1.4: Hediondez del código (code smell)

Cualquier síntoma en el código fuente del software que puede indicar la presencia de un problema más serio.

Aunque un código que "huele" no implica errores, sí que tiene más posibilidades de errores en cambios futuros o mayor dificultad para hacer cambios. Algunos ejemplos de hediondez son:

- Código duplicado.- Fuente de errores futuros porque implica doble mantenimiento.
- Larga lista de parámetros.- Los posibles errores vienen de la facilidad para cambiar el orden en las llamadas sin darnos cuenta.
- Métodos grandes.- Difícil de entender, mantener y validar.
- Clases grandes.- Por ejemplo, más de 8 métodos o más de 15 variables son difíciles de mantener.
- Cadena de mensajes.- Por ejemplo: `student.getID().getRecord().getGrade(course)`. Significa que falta algún método o función que permita hacer ese encadenamiento.

Cómo verificar la preservación del comportamiento (observable) del sistema

Se verifica asegurando que todas las pruebas pasadas antes de refactorizar se puedan pasar después de refactorizar. Se trata de utilizar las llamadas *pruebas de regresión*.

Refactorización básica y compleja

La refactorización incluye una serie concreta de actividades básicas, las cuales se pueden combinar para formar refactorizaciones complejas. Las transformaciones básicas (código OO) son:

1. agregar una clase, método o atributo;
2. renombrar una clase, método o atributo;
3. mover un atributo o método hacia arriba o hacia abajo en la jerarquía;
4. eliminar una clase, método o atributo; y
5. extraer fragmentos de código en métodos separados.

2.2. Comprensión de un programa

Para poder formar parte de un equipo de mantenimiento en un software desconocido para nosotros, debemos dedicar todo el tiempo necesario a la comprensión del programa. Se trata de una tarea clave, que puede representar alrededor del 50 % del esfuerzo total dedicado al mantenimiento de un producto software y que repercute directamente en los costes de mantenimiento. En términos de actividades concretas, la comprensión del programa implica la construcción de modelos mentales de un sistema subyacente en diferentes niveles de abstracción, que varían desde modelos de bajo nivel del código hasta modelos de muy alto nivel del dominio de la aplicación subyacente. Los científicos cognitivos han estudiado modelos mentales para comprender cómo los seres humanos conocen, perciben, toman decisiones y construyen comportamientos en un mundo real. En el dominio de la comprensión de software, un modelo mental describe la representación mental que tiene un programador de un programa concreto.

Un paso clave en el desarrollo de modelos mentales es generar hipótesis o conjeturas e investigar su validez. Es algo que hacemos generalmente de forma implícita cuando queremos entender algo en profundidad. En el caso de un software que debemos mantener, el nivel de conocimiento debe ser muy alto pues las consecuencias de entender algo mal y hacer cambios pueden ser muy dañinas.

De esta manera, las hipótesis se deben formular de forma explícita, como parte de la importantísima tarea concreta de comprensión del programa. Las hipótesis son una forma de que un programador entienda el código de manera incremental. Después de comprender el código, el programador formula una hipótesis y la verifica leyendo el código. La verificación de la hipótesis da como resultado aceptar la hipótesis o rechazarla. A veces, una hipótesis puede no ser completamente correcta debido a la comprensión incompleta del código por parte del programador. Al formular continuamente nuevas hipótesis y verificarlas, el programador comprende cada vez más código y cada vez más detalles.

Se pueden aplicar varias estrategias para llegar a hipótesis significativas, como las estrategias de abajo hacia arriba (*bottom-up*), de arriba hacia abajo (*top-down*) y combinaciones entre ellas. Una estrategia *bottom-up* funciona comenzando por el código, mientras que una estrategia *top-down* comienza por un objetivo de alto nivel. Cada estrategia se formula mediante la identificación de acciones para lograr un objetivo. Por ejemplo, en el caso de aplicar una estrategia hacia arriba, a partir del código vamos aplicando los mecanismos de agrupación y referencias cruzadas para producir estructuras de abstracción de nivel superior:

- La agrupación crea nuevas estructuras de abstracción de nivel superior a partir de estructuras de nivel inferior.
- Las referencias cruzadas enlazan elementos de diferentes niveles de abstracción.

Esto ayuda a construir un modelo mental del programa en estudio en diferentes niveles de abstracción.

Acrónimos

CBS Component Based Software system

COTS Commercial Off-the Shelf

SGBD Sistema de Gestión de Bases de Datos

Bibliografía

L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 1(15):225–252, 1976.

Priyadarshi Tripathy and Kshirasagar Naik. *Software Evolution and Maintenance*. Johh Wiley, EE.UU., 2014.