Tema 2. Mantenimiento y evolución del software

Desarrollo de Software

Curso -

3º - Grado Ingeniería Informática

Dpto. Lenguajes y Sistemas Informáticos
ETSIIT
Universidad de Granada

31 de marzo de 2023

- *Mantenimiento software* se refiere a corregir los errores del software que le impiden que realice las funcionalidades previstas en la entrega
- *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.

Primeras tres leyes de la evolución software

- 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.

2.1.2. Mantenimiento software

Dos clasificaciones

- 1
  - Correctivo
  - Adaptativo
  - Perfectivo (y preventivo)
- 2
  - 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.

Mantenimiento del software basado en componentes (CBS: Component Based Systems)

- Definición: El CBS es un software 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
- Características (frente al *software a medida*):
  - Mantenimientos divididos
  - Habilidades específicas
  - Comunidad de usuarios mayor: confianza vs control
  - Desplazamiento de los costes
  - Planificación más difícil

- Desarrollo inicial.- Aún no ha empezado el mantenimiento
- Evolución.- Nada más lanzado es fácil hacer cambios simples
- 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)
- Retirada gradual ("phaseout").- En algún momento el servicio de mantenimiento mínimo se hace demasiado caro o bien aparecen soluciones mejores. 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*).

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

2.1.5. Reingeniería

Examen y análisis de un software existente (especificación, diseño, implementación y documentación), resestructurándolo 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".
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.

2.1.5. Reingeniería

Razones o necesidades por las que se decide hacer reingeniería

- Mejorar la mantenibilidad.- El mantenimiento de un sistema aumenta con el tiempo hasta hacerse demasiado difícil y costoso (2L). 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.- Los sistemas están en continua adapatación a su entorno siendo cada vez menos satisfactorios (1L). 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.- Las partes interesadas en un software perciben una bajada de calidad en los sistemas que no se mantienen de forma rigurosa (7L). Se hace necesaria la reingeniería para conseguir aumentar la fiabilidad del mismo.
- Prepararse para una mejora de la funcionalidad.- 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 (7L). 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

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).

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.

2.1.5. Reingeniería
Conceptos de reingeniería

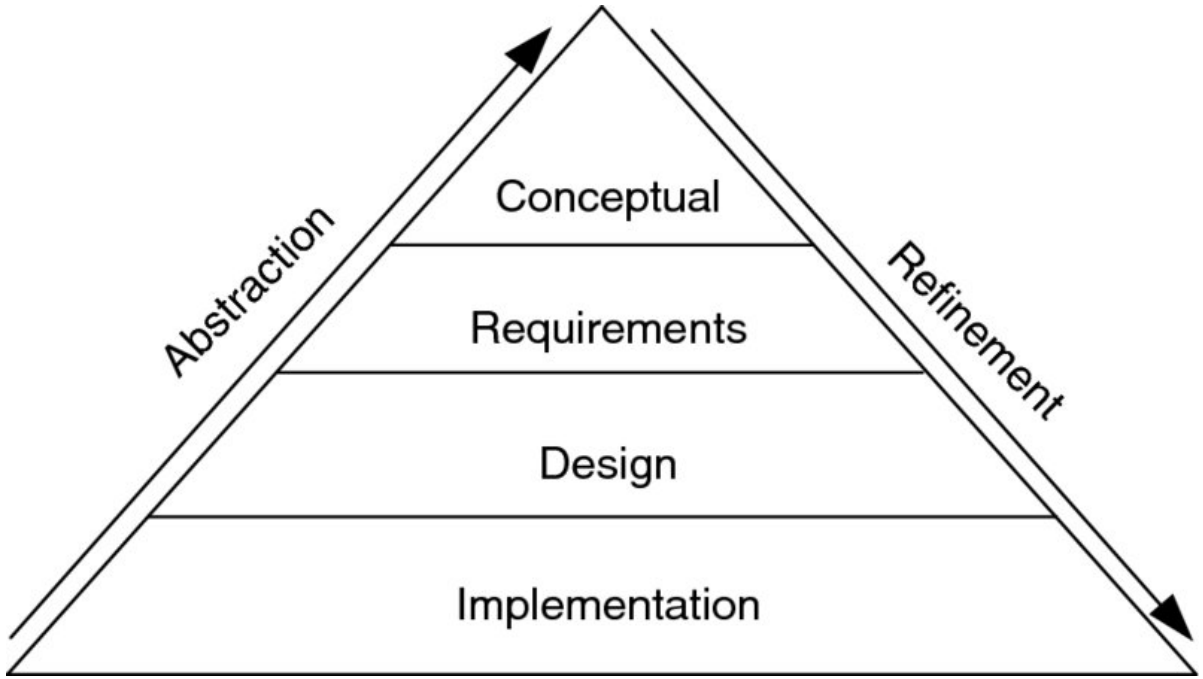

Figura 1: Niveles de abstracción y refinamiento. [Fuente: (Tripathy and Naik, 2014, cap. 4)].

2.1.5. Reingeniería
Conceptos de reingeniería

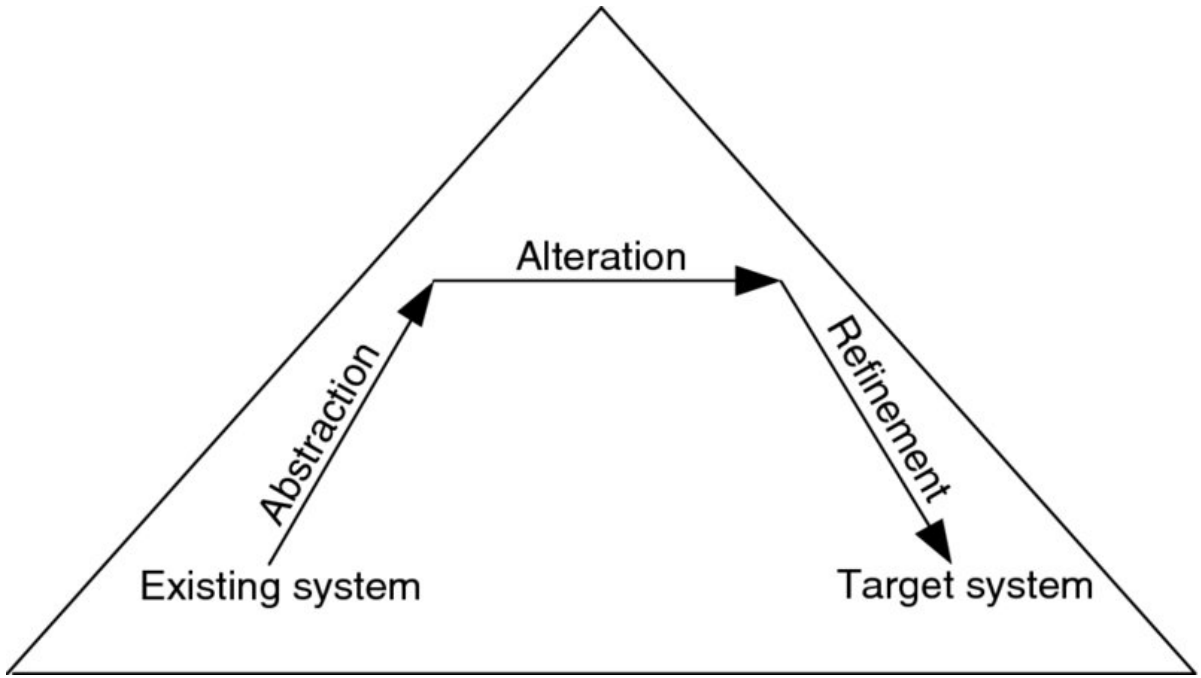

Figura 2: Bases conceptuales en el proceso de reingeniería. [Fuente: (Tripathy and Naik, 2014, cap. 4)].

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

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 *Descripción Arquitectónica (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.

2.1.5. Reingeniería
Conceptos de reingeniería

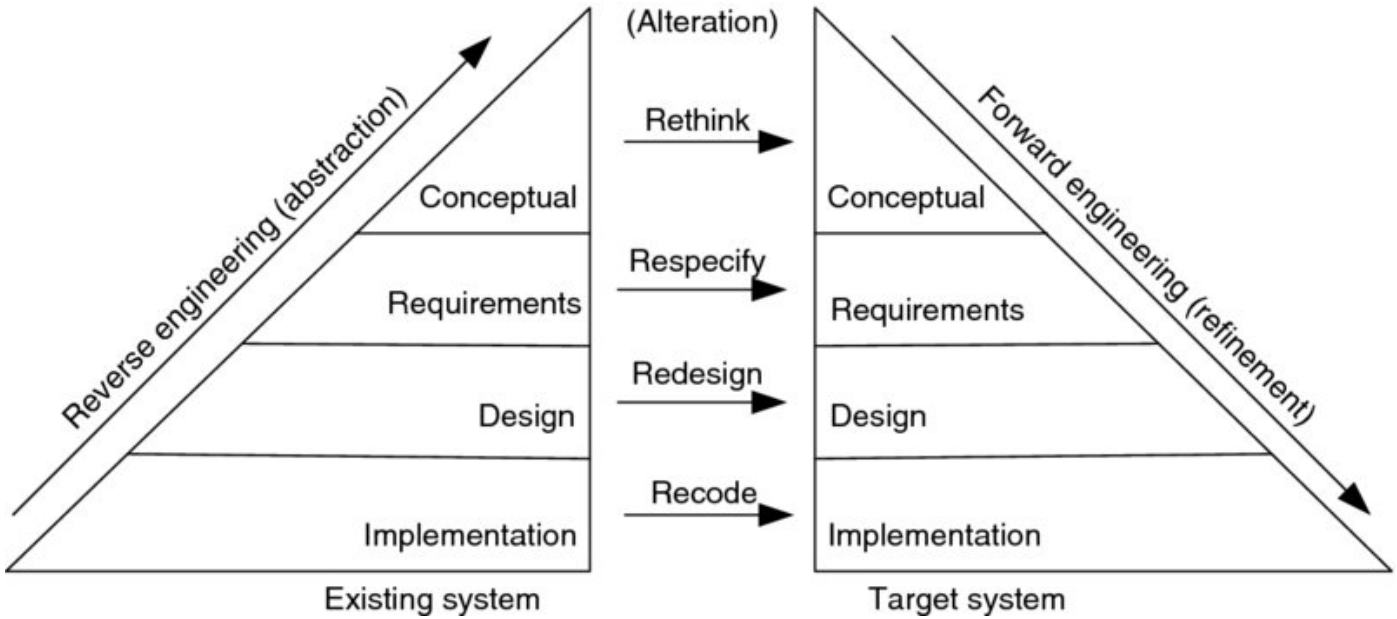

Figura 3: Bases conceptuales en el proceso de reingeniería. [Fuente: (Tripathy and Naik, 2014, cap. 4)].

2.1.5. Reingeniería
Conceptos de reingeniería

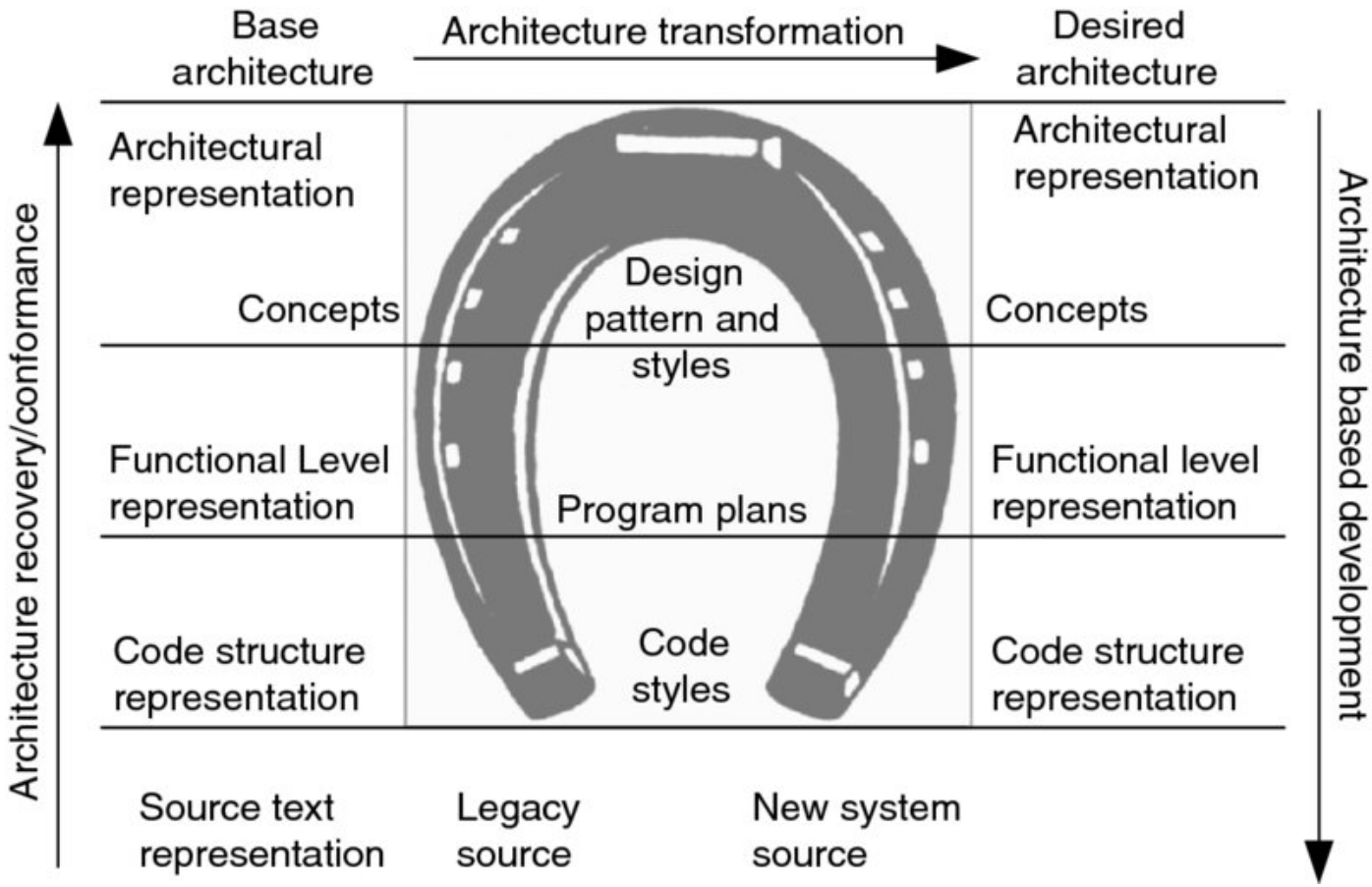

Figura 4: Bases conceptuales en el proceso de reingeniería. [Fuente: (Tripathy and Naik, 2014, cap. 4)].

2.1.6. Software heredado (legacy software)

Software heredado

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

Dos operaciones posibles:

- Envoltura (wrap).- Alternativa al mantenimiento. 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–. Tiene los siguientes elementos:
  - 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.

2.1.6. Software heredado (legacy software)

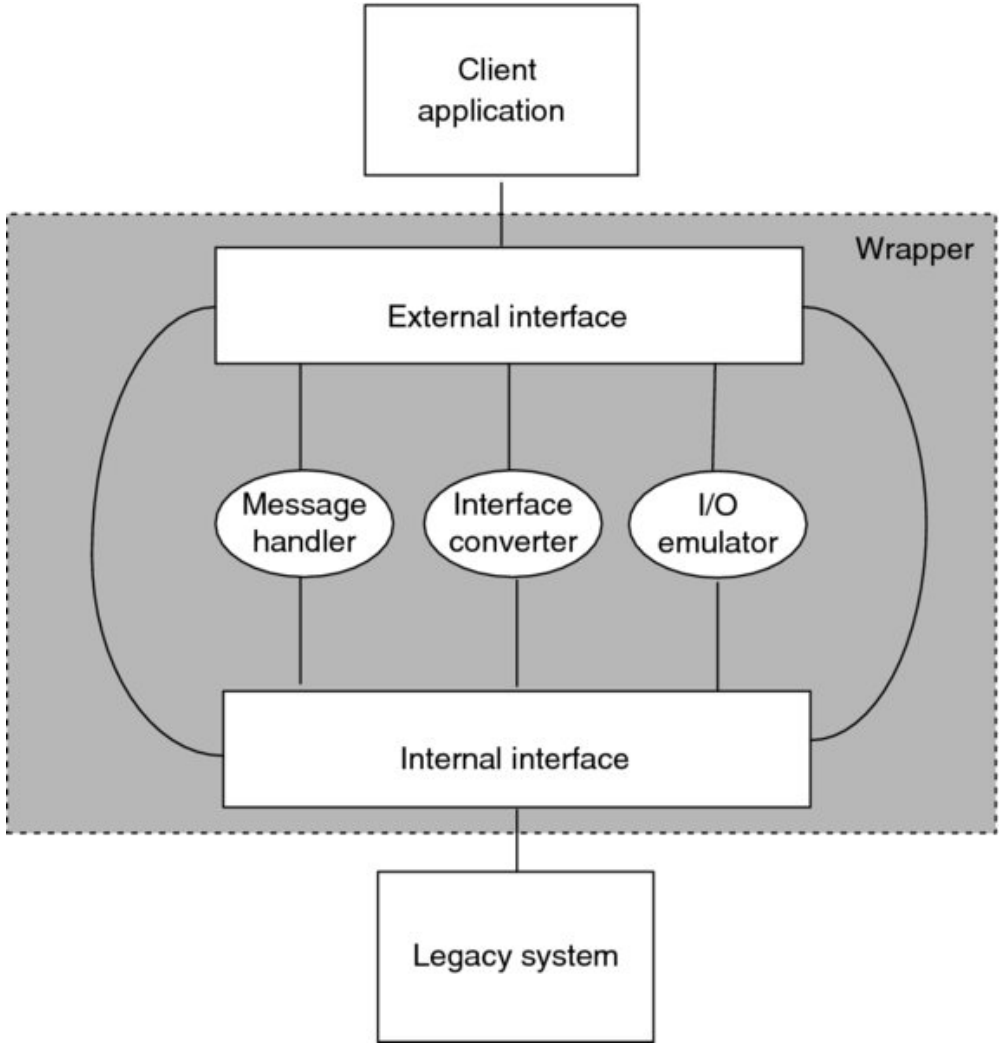

Figura 5: Módulos de un entorno envoltorio. [Fuente: (Tripathy and Naik, 2014, cap. 5)].

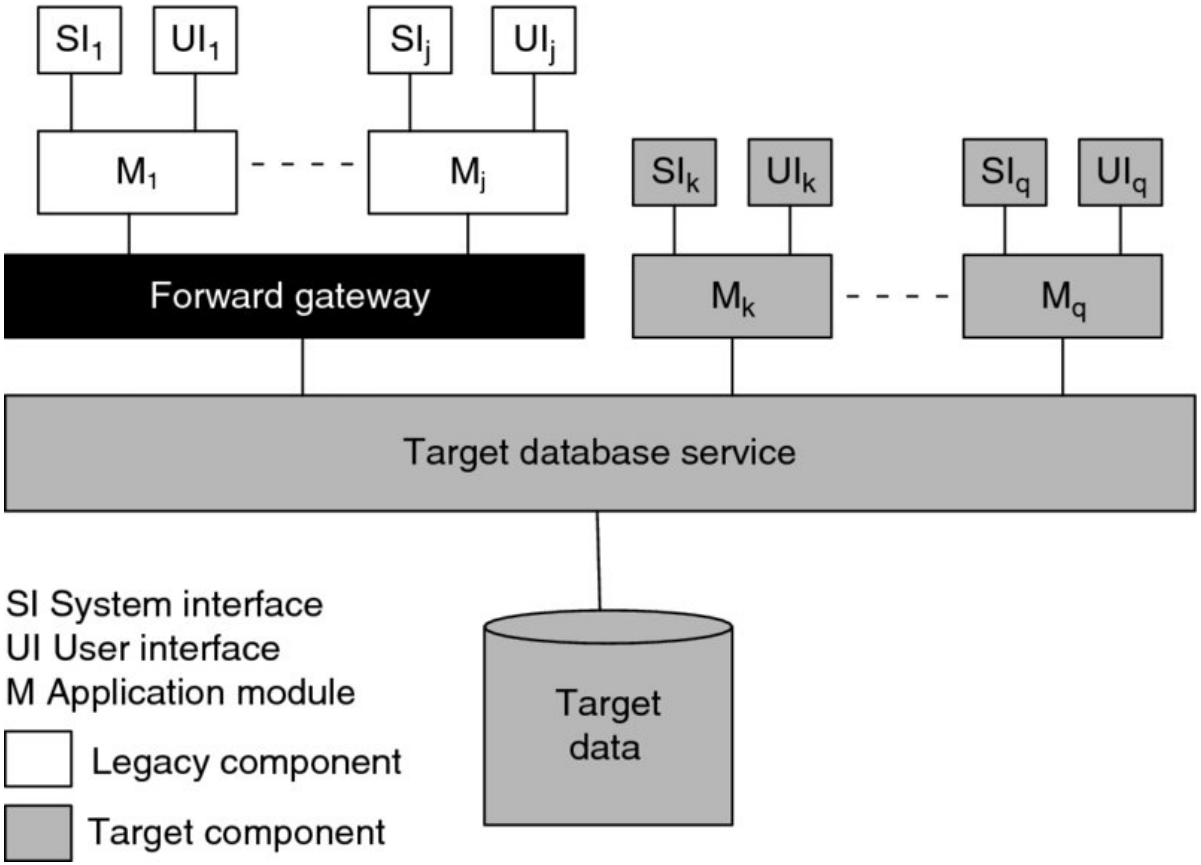

Figura 6: Enfoque *Base de datos primero*. [Fuente: (Tripathy and Naik, 2014, cap. 5)].

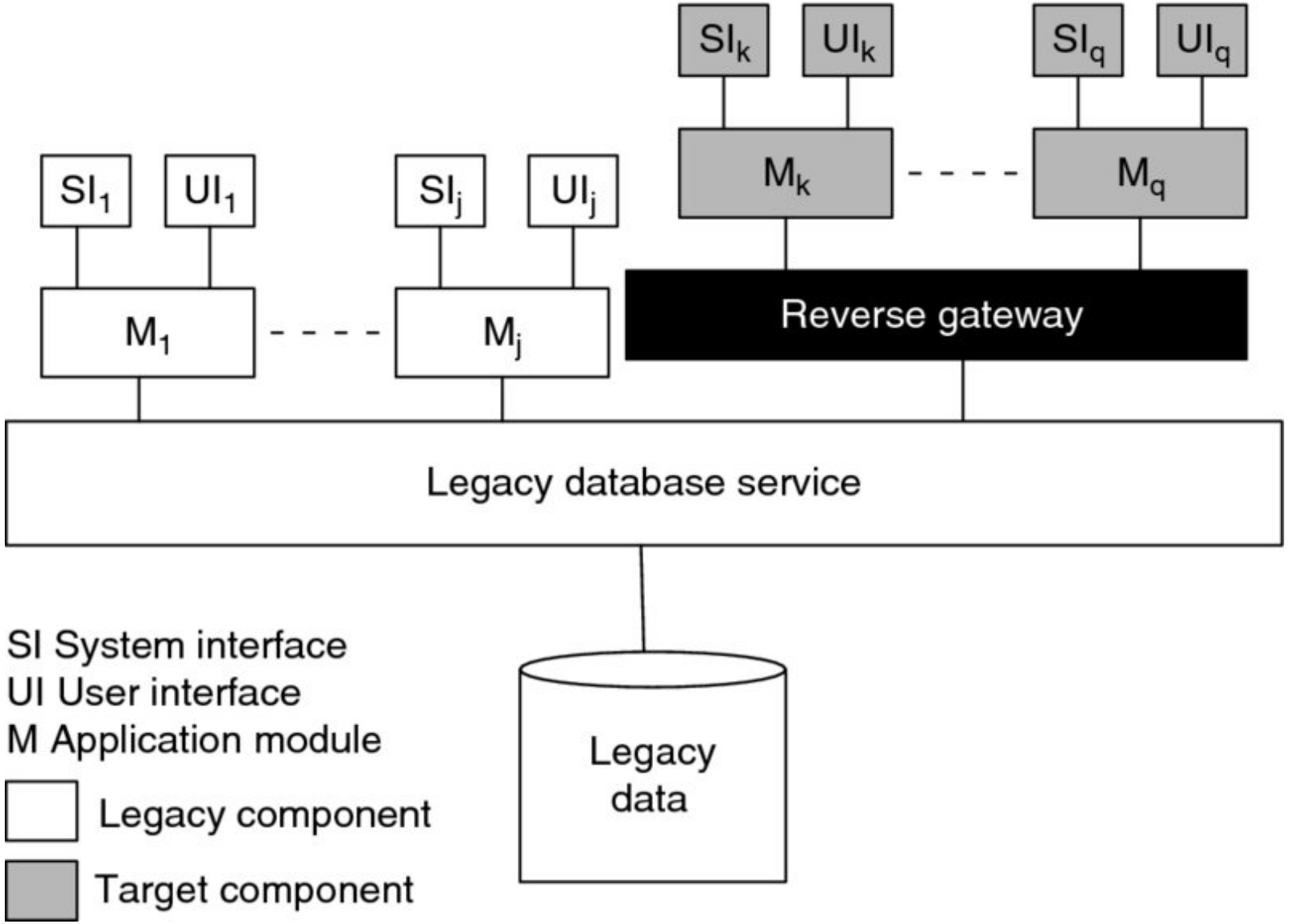

Figura 7: Enfoque *Base de datos después*. [Fuente: (Tripathy and Naik, 2014, cap. 5)].

2.1.6. Software heredado (legacy software)
Métodos de migración

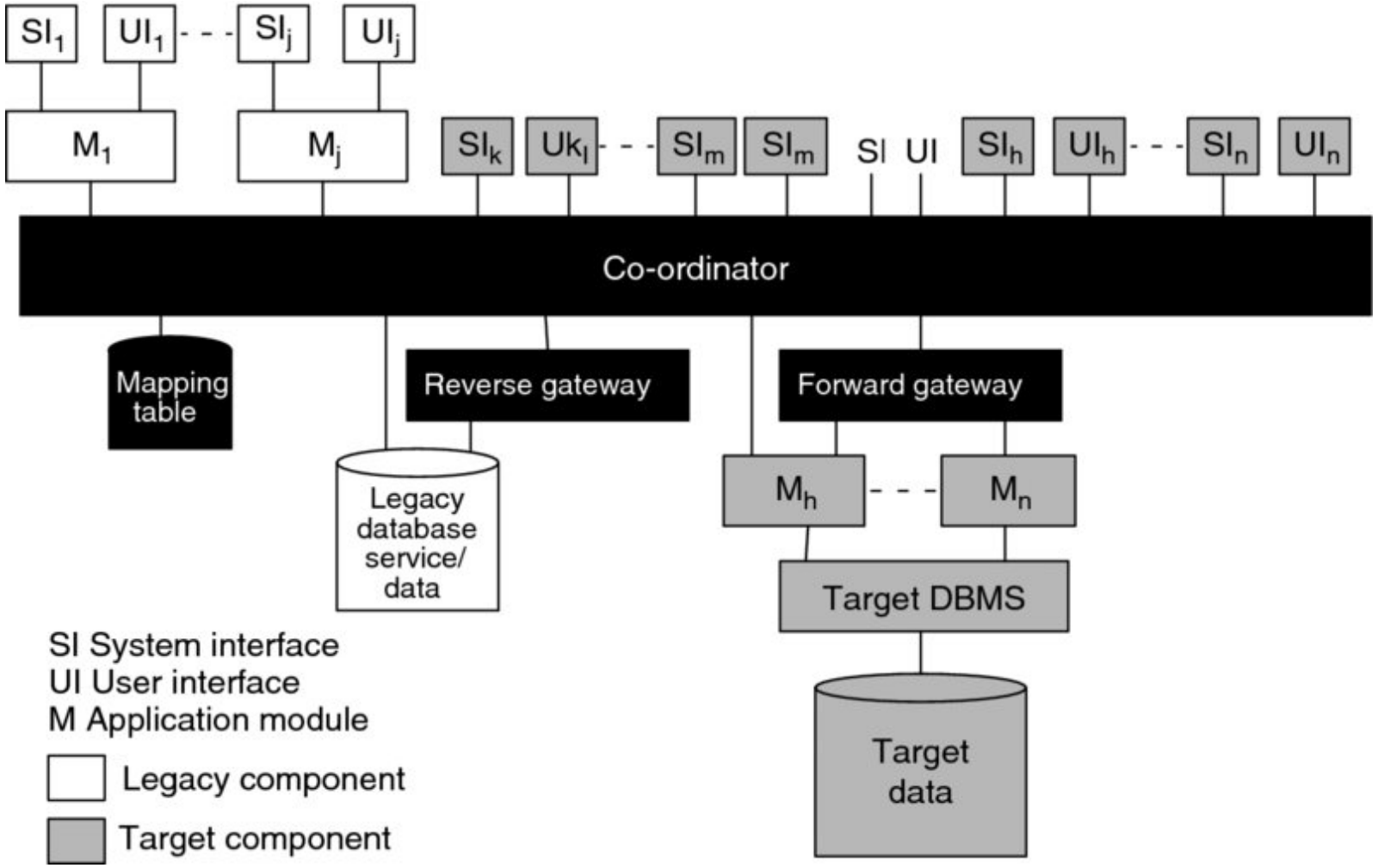

Figura 8: Enfoque *Base de datos compuesta*. [Fuente: (Tripathy and Naik, 2014, cap. 5)].

2.1.7. 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*.

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

Identificar qué refactorizar

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 está 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.

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.

- 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 (alrededor del 50 % del esfuerzo total)
- La comprensión del programa implica la construcción de modelos mentales de un sistema subyacente en diferentes niveles de abstracción
- Un paso clave en el desarrollo de modelos mentales es generar hipótesis o conjeturas e investigar su validez. Las hipótesis se deben formular de forma explícita
- Estrategias para llegar a hipótesis significativas: de abajo hacia arriba (*bottom-up*), de arriba hacia abajo (*top-down*) y combinaciones entre ellas
- 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.

Referencias I

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