Introducción a la Reingeniería de Software Mediante Patrones de Diseño

Tomás Balderas Contreras
balderas@ccc.inaoep.mx

Instituto Nacional de Astrofísica, Óptica y Electrónica
Coordinación de Ciencias de la Computación
Programación Genérica y Patrones de Software

19 de julio, 2003

Contenido

1 Introducción **2**
1.1 Ingeniería inversa 2
1.2 Análisis y rediseño 4
1.3 Ingeniería avanzada 5

2 Refactorings **5**

3 Identificación de patrones en programas en C++ **8**
3.1 Maisa 8
3.2 Colombus 9

4 Conclusiones **11**

Resumen

El *código legado* (legacy code) de una corporación es un conjunto de programas en código fuente que fueron desarrollados varios años atrás, que han sido usados durante todo ese tiempo y que ahora deben ser modificados para satisfacer nuevos requerimientos. Los problemas inherentes al mantenimiento de este software surgen si los miembros del equipo técnico actual no trabajaron en su desarrollo; y no se aplicó una metodología estricta durante su diseño. En estas circunstancias, el software se convierte en un verdadero "código ajeno" al grupo de desarrollo, el cual cuenta con documentación incompleta sobre el sistema y sus cambios, o bien carece completamente de ella. El proceso de reingeniería de software tiene como objetivo extraer información de diseño de un sistema y emplear tal información para reconstruirlo, mejorar su desempeño o extender sus funcionalidades. La identificación de patrones en el código fuente también es de gran ayuda para mejorar el diseño del nuevo sistema. Este artículo explora la forma en que se pueden obtener patrones de diseño mediante un proceso de reingeniería de software.

1 Introducción

La reingeniería de software consiste en examinar el diseño e implantación de un sistema existente y en aplicar diferentes técnicas y métodos para rediseñarlo y transfromarlo, en un esfuerzo de mejorar su calidad [5]. Se espera que el software resultante tenga la misma funcionalidad que el software existente y que, además, haya sido enriquecido con nuevas funciones y su rendimiento general haya mejorado.

El equipo que lleva a cabo el proceso de reingeniería debe tener en consideración las necesidades actuales y futuras del usuario. Se debe garantizar que requerimientos como mantenibilidad, usabilidad, funcionalidad y otros sean preservados apropiadamente. Realizar este proceso no es una tarea sencilla, y la situación se complica o se vuelve intratable si no se dispone de documentación o esta es insuficiente y si el código fuente cuenta con módulos de miles de líneas y pocos comentarios.

La figura 1 muestra las fases que sigue el proceso de reingeniería de software, las cuales se describen en seguida [3].

1.1 Ingeniería inversa

La ingeniería inversa es un proceso de recuperación de información sobre el diseño del código legado, que tiene como objetivo construir una repre-

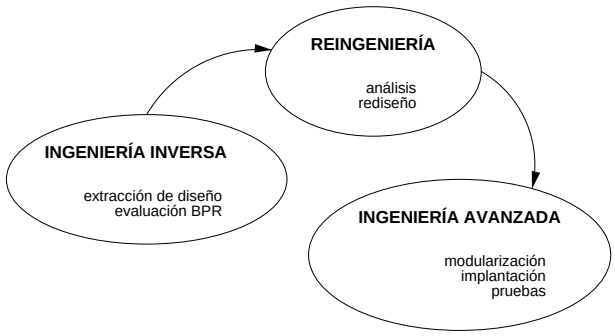

Figura 1: Las fases del proceso de reingeniería.

sentación del sistema a un nivel de abstracción mayor que el código fuente del software. Una ingeniería inversa furctífera desemboca en una o más especificaciones de diseño y la localización de los componentes reutilizables del software.

El equipo de desarrollo puede utilizar herramientas CASE para extraer información sobre el diseño del código fuente, pero el *nivel de abstracción*, la *completitud* de la información recabada, el *grado de interacción* entre las herramientas y los usuarios y la *direccionalidad* del proceso son factores que varían considerablemente.

El empleo de la ingeniería inversa tiene las siguientes ventajas:

- Desarrolla información sobre el diseño, en caso de que nunca antes se haya realizado
- Recupera información perdida con el transcurso del tiempo
- Proporciona diferentes clases de documentación sobre el sistema existente e identifica componentes reutilizables
- Permite mejorar la calidad del sistema y minimizar los gastos al identificar componentes reutilizables

No obstante, existen varias dificultades a vencer presentes durante este proceso:

- El código fuente del sistema puede esar pobremente estructurado
- Es necesaria una investigación exhaustiva cuando se tiene una gran cantidad de código fuente pero un conocimiento insuficiente del soft-

ware y poca o ninguna documentación. En este caso los costos son muy altos

- Las herramientas CASE actuales tienen límites en su operación, siendo incapaces de extraer suficiente información del código fuente

1.2 Análisis y rediseño

Esta es la segunda etapa del proceso de reingeniería y está muy vinculada con las fases de análisis y diseño tradicionales del proceso de ingeniería de software. Sin un análisis y un entendimiento del sistema apropiados no es posible completar un proyecto que proporcione resultados satisfactorios.

El equipo de reingeniería comienza el *análisis* identificando las necesidades del usuario y transformándolas en requerimientos altamente especificados. Otro aspecto que se debe tomar en cuenta es realizar una etapa de análisis de costo contra beneficio. Por ejemplo, en ocasiones puede ocurrir que los cambios al sistema sean tan pequeños que no produzcan un sistema muy diferente al original, o que sean tan extensos para implantar en una cantidad de tiempo razonable o que el costo de los cambios en la implantación sea muy grande.

Los resultados del análisis deben ser monitoreados y documentados cuidadosamente. Un análisis bien documentado no solo facilita el diseño e implantación, sino que también proporciona información útil al equipo de desarrollo en el futuro.

Durante la fase de *rediseño* el equipo de desarrollo intenta cambiar el sistema legado de acuerdo a los resultados del análisis, añadiendo nuevos elementos de diseño para incorporar los cambios a la funcionalidad del sistema. La diferencia con la etapa de diseño de un proyecto de ingeniería de software ordinario es que el rediseño debe considerar los elementos de diseño anteriores y cómo serán afectados por los cambios.

El rediseño a menudo es difícil debido a los serios conflictos entre el nuevo diseño y el original. O bien ambos diseños pueden ser compatibles, lo que facilita la implantación del nuevo software y reduce el tiempo. En cada caso el diseño del nuevo sistema debe ser cuidadosamente documentado.

Los cambios a las interfaces deben ser mínimos pues dichos cambios pueden provocar incompatibilidades con los sistemas en el ambiente, los cuales deben ser actualizados para adaptarse a los cambios. Otra razón para limitar los cambios es evitar a los usuarios tener que aprender una nueva interfaz, lo que les puede consumir mucho tiempo.

1.3 Ingeniería avanzada

Esta etapa coincide con el proceso usual de ingeniería de software, emplea el diseño de alto nivel desarrollado durante la etapa anterior y, progresivamente, lo transforma en un diseño de bajo nivel para su implantación. Esta etapa se realiza en tres pasos:

Modularización: Es el proceso de dividir datos, construir capas de aplicación y producir grupos de clases que lleven a cabo la funcionalidad deseada. Los resultados de la modularización incluyen un programa de tamaño y complejidad reducidos, lo que implica un mantenimiento sencillo del sistema.

Implantación: Un objetivo principal durante este paso es no construir clases muy grandes o muy pequeñas. Cada clase debe implantar un solo concepto, si la clase implementa más de un concepto probablemente tenga baja cohesión. Este diseño de bajo nivel debe garantizar que la cohesión y el acoplamiento permitan la reusabilidad y la mantenibilidad.

Pruebas: Tiene como objetivo encontrar errores que posiblemente ocurren durante la ejecución de un programa y optimizar algunos aspectos que funcionan bien.

El resto de este reporte está organizado de la siguiente manera: la sección 2 discute la tecnología de refactorings, que son transformaciones de código fuente empleadas para el rediseño de sistemas de software, la sección 3 ilustra una propuesta para identificar patrones de diseño en un sistema de software escrito en C++ y, finalmente, la sección 4 proporciona las conclusiones de este documento.

2 Refactorings

Durante el ciclo de desarrollo de software se deben llevarse a cabo dos tipos de transformaciones entre representaciones del sistema. Las transformaciones *especificación-a-fuente* convierten una especificación de alto nivel del sistema a código fuente compilable. Las transformaciones *fuente-a-fuente* convierten un programa escrito en un lenguaje dado en un nuevo programa escrito en el mismo lenguaje. Un *refactoring* es una transformación fuente-a-fuente parametrizada, aplicada a programas orientados a objetos, que tiene

la propiedad de preservar el comportamiento del sistema original; es útil para rediseñar software legado [6, 4].

Un refactoring está completamente definido por los siguientes componentes:

- Propósito
- Argumentos
- Descripción
- Condiciones de habilitación
- Estado inicial
- Estado objetivo

La transformación verifica el cumplimiento de las condiciones de habilitación para asegurarse que el comportamiento del programa será preservado. Después identifica el segmento de código fuente que será afectado por los cambios y, a continuación, efectúa las modificaciones. Los programas se rediseñan mediante la aplicación de una secuencia de refactorings. Si cada refactoring individual preserva el comportamiento del sistema original, entonces la secuencia de refactorings también lo preserva.

La figura 2 muestra la definición del refactoring **Inherit**. Esta transformación establece una relación de herencia entre las clases no relacionadas **Base** y **Derived**. Desde el punto de vista de un diagrama de clases, la transformación convierte a la clase indicada por **Derived** en una subclase de la clase indicada por **Base**. Sin embargo, el refactoring modifica también el código fuente de la aplicación para reflejar este cambio.

Se define un conjunto de *invariantes* que, cuando se mantienen, aseguran que dos programas se ejecutarán de forma idéntica. Cuando un refactoring corre el riesgo de violar un invariante, las condiciones de habilitación se añaden para garantizar que el invariante se conserva. En [6] se identifican los siguientes siete invariantes, requeridos para preservar el comportamiento de programas en C++ y Smalltalk:

1. **Superclase única**: Después de aplicar el refactoring la clase debe tener siempre a lo más una superclase directa, y esta no puede ser también una subclase.

2. **Nombres distintos para clases**: Después de aplicar el refactoring cada clase debe tener un nombre único y las clases no deben anidarse.

Name:

Inherit| *Base*, Derived |

Purpose:

To establish a superclass-subclass relationship between two existing classes.

Arguments:

Base - superclass name
Derived - subclass name

Description:

Inherit|| makes ***Base*** a superclass of **Derived**.

Enabling Conditions:

1. ***Base*** must not be a subclass of **Derived** and **Derived** must not have a superclass.
2. Member variables of **Derived** must have distinct names from member variables of ***Base*** and its superclasses.
3. A member function of **Derived** which overrides a function must have the same type signature as the function it overrides.
4. Subclasses of ***Base*** must implement any pure virtual methods if objects of that class are created.
5. Initializer lists must not be used to initialize **Derived** objects.
6. For all inherited instance variables whose type is a class, the constructors for those classes cannot have any side-effects outside of object initialization if **Derived** is instantiated.
7. Program behavior must not depend on the size or layout of **Derived**.

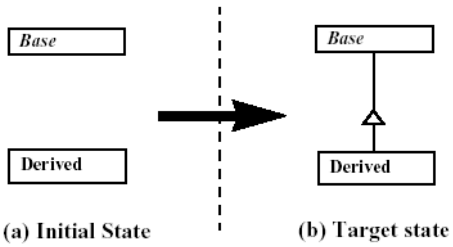

Figura 2: El refactoring **Inherit**.

3. **Nombres distintos para miembros**: Después de aplicar el refactoring todas las variables y funciones miembro dentro de una clase deben tener nombres distintos.

4. **Variables miembro heredadas no redefinidas**: Una variable instancia heredada de una superclase no puede ser redefinida en ningua superclase.

5. **Selectores compatibles en la redefinición de funciones miembro**: Después de aplicar el refactoring, si una función miembro definida en una superclase se redefine en una subclase, debe mantener el mismo selector o firma.

6. **Compatibilidad de tipo en las asignaciones**: Después de aplicar el refactoring, el tipo del resultado de cada expresión que se asigne a una variable debe ser una instancia del tipo de la variable.

7. Operaciones y referencias semánticamente equivalentes

3 Identificación de patrones en programas en C++

En [1] se presenta un método para identificar patrones de diseño a partir del código fuente de programas de aplicación escritos en C++. El método empleado para llevar a cabo el reconocimiento consiste en la interacción de las herramientas Colombus y Maisa. Colombus lleva a cabo la transformación del código fuente de un programa en C++ a su representación como diagramas de clase de UML. Maisa analiza la calidad de un diseño de software, a partir de su representación mediante diagramas UML, y realiza la minería de patrones de diseño en dicha representación.

3.1 Maisa

Maisa es una herramienta desarrollada en la Universidad de Helsinki, Finlandia, con la finalidad de calcular métricas de software en un sistema y buscar instancias de patrones de diseño en su arquitectura, expresada como diagramas UML.

La herramienta realiza la minería de patrones mediante una técnica de *satisfacción de restricciones*. Un problema de satisfacción de restricciones (CSP) está definido a partir de un conjunto de variables x_i, cada una sobre un dominio finito D_i y un conjunto de restricciones que limitan los valores que pueden ser asignados a las variables. Las *restricciones unitarias* (P_i)

limitan los valores para una sola variable y las *restricciones binarias* (P_{ij}) representan una condición para un par de variables. El CSP se puede establecer formalmente como sigue:
$(\exists x_1 \in D_1)(\exists x_2 \in D_2)\ldots(\exists x_n \in D_n)P_1(x_1) \wedge P_2(x_2) \wedge \ldots \wedge P_n(x_n) \wedge P_{12}(x_1, x_2) \wedge P_{13}(x_1, x_3) \wedge \ldots \wedge P_{n-1n}(x_{n-1}, x_n)$ con P_{ij} tal que $i < j$. El CSP se modela frecuentemente con un grafo en el que los nodos representan a las variables y los arcos representan las restricciones.

La técnica de minería o identificación de patrones se define como un CSP de la siguiente manera:

- Las variables (nodos) representan los roles de un patrón.
- Los dominios de las variables contienen todos los nombres (identificadores) presentes en los diagramas UML que se analicen.
- Las restricciones unitarias representan las condiciones para un solo rol (e.g. el elemento en el role X debe ser de un tipo Clase).
- Las restricciones binarias representan condiciones entre dos roles (e.g. la clase en el rol X debe ser una subclase de la clase en el rol Y).

Un *enlace* (binding) es un par $\{rol, elemento\}$, donde *rol* es el nombre de un rol en el patrón y *elemento* es el nombre de un componente de los diagramas UML asignado al rol. Para cada patrón se calcula un *resultado*, que es un conjunto de enlaces que describe al patrón, el número de estos enlaces depende de la estructura de cada patrón. Para cada resultado se evalúan las condiciones, si no se satisfacen el resultado se descarta y se prueba otro mediante *backtracking*.

3.2 Colombus

Colombus es una herramienta para ingeniería inversa desarrollada por el Grupo de Investigación en Inteligencia Artificial en Szeged, Hungría, en conjunto con el Laboratorio de Tecnología de Software del Centro de Investigación de Nokia. Puede extraer modelos de clase en UML de proyectos grandes en C/C++, además de permitir manipulación de proyectos, extracción de datos, representación de datos, almacenamiento de datos, filtrado y visualización. El sistema puede ser extendido para realizar otras actividades de ingeniería inversa mediante la adición de módulos (plug-ins).

Los siguientes tres tipos de módulos llevan a cabo la operción básica del sistema:

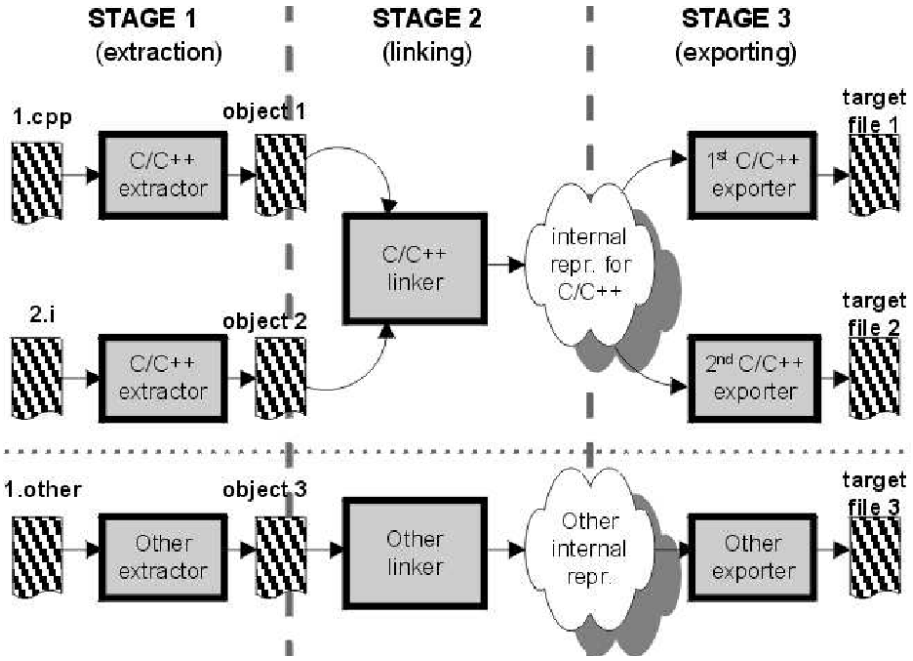

Figura 3: Ejecución de los tres módulos del sistema Colombus.

- **Módulos de extracción**: Analiza un archivo de código fuente de entrada y producen un archivo de salida que contiene la información extraída.
- **Módulos de enlace**: Construye y filtra la representación interna del proyecto. El proceso se lleva a cabo sobre los archivos creados por el módulo anterior.
- **Módulos de exportación**: Exporta la representación interna construída por el módulo anterior a un formato de salida específico.

La figura 3 ilustra el proceso llevado a cabo por los tres módulos, los archivos de entrada recibidos y los archivos intermedios y de salida generados. Colombus ofrece una API (Application Programming Interface) al usuario para que pueda desarrollar sus propios módulos en forma de bibliotecas DLL y para tener acceso a la información extraída de los archivos de código fuente.

4 Conclusiones

Se discutió el concepto de reingeniería de software y se describieron las etapas que lo componen: ingeniería inversa, análisis y rediseño e ingeniería avanzada. Se proporcionó una breve introducción a la tecnología de refactorings como un medio de llevar a cabo el proceso de rediseño en software legado. Por último se describió una propuesta para identificar patrones de diseño en el software, basada en el uso de las herramientas Colombus (para ingeniería inversa) y Maisa (para cálculo de métricas y minería de patrones).

El desarrollo de herramientas CASE que automaticen completamente el proceso de reingeniería de software y que permitan la identificación y el rediseño mediante patrones de diseño de software aun es incipiente. Hace falta mucho trabajo de investigación y desarrollo para encontrar nuevas técnicas que permitan convertir, de manera eficiente, todo el software legado existente en sistemas modernos y reusables.

La intención de este breve documento es plantear la problemática existente y algunas propuestas que dan solución a algunos aspectos. El lector interesado puede profundizar en algún tema y desarrollar su propia línea de investigación.

Referencias

[1] Ferenc, R., et al. 2002. Recognizing Design Patterns in C++ Programs with the Integration of Columbus and Maisa. En *Acta Cybernetica Journal*. Vol. 15. Szeged, Hungary. 669-682.

[2] Gamma, E., et al. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Massachusetts: Addison-Wesley Publishing Company.

[3] Gjörwell, D., S. Haglund y D. Sandell. 2002. Reengineering and Reengineering Patterns. Reporte Técnico. Departamento de Ciencias de la Computación e Ingeniería. Mälardalens Högskola. Västerås, Suecia.

[4] Opdyke, W. F. 1992. Refactoring Object-Oriented Frameworks. Ph. D. Thesis. University of Illinois at Urbana Champaign. Illinois, EUA.

[5] Pressman, R. 1993. *Ingeniería del Software: Un Enfoque Práctico*. Tercera Edición. España: McGraw-Hill.

[6] Tocuda, L. A. 1999. Evolving Object-Oriented Designs with Refactorings. Ph. D. Thesis. The University of Texas at Austin. Texas, EUA.