Universidad de La Plata
Facultad De Informática

Refactoring de código estructurado

Trabajo de Especialización

Autor: Lic. Mariano MENDEZ

Director: Dr. Fernando G. TINETTI

Co-Director: Dra. Alejandra GARRIDO

A Jose Siaskiewicz, a quien le debo gran parte de lo que soy ...

Índice general

Capítulo 1

Introducción

Una de las características esenciales del software es la complejidad [8], es difícil encontrar un producto construido por el hombre que posea el grado de complejidad que tiene el software ya sea como proceso constructivo o como producto en sí mismo. Pensemos por un momento en la construcción de un sistema operativo, quizá un caso para tomar como ejemplo (por la época en que fue construido) sería el del OS/360 liderada por Fredrik P. Brooks en el cual unas 1000 personas formaron parte del equipo de desarrollo [9]; o en los sistemas de tiempo real o en los sistemas Open-Source, como el Linux, en los cuales han colaborado cientos e incluso miles de programadores en conjunto desde distintos lugares del mundo. Este tipo de hazaña es comparable a la construcción de grandes estructuras como los rascacielos, satélites y otros.

1.1. El Software

El cambio es un factor que ejerce continuamente presión sobre el software [8]. Lo sobresaliente de este caso es que una vez terminada la producción del mismo, el cambio sigue ejerciendo presiones sobre el producto, algo difícil de que suceda en la industria de la manufactura. Esto es debido, según Brooks, a dos motivos: el primero a que el software puede sufrir modificaciones más fácilmente que por ejemplo un edificio, el segundo a que el software está constituido por su funcionalidad, y justamente la funcionalidad es el aspecto que está sujeto a más presiones para cambiar.

Teniendo en cuenta éstas y otras características esenciales del software (Conformidad, Intangibilidad) el proceso por el cual se logra la construcción del mismo ha sufrido grandes cambios a lo largo de su corta existencia, unos

50 años de vida. Según las investigaciones hechas en los últimos años el porcentaje de proyectos de desarrollo que no logran completarse es muy elevado [53], como así también el sobreprecio y la modificación de calendarios interminables. Un ejemplo muy gráfico de este hecho es el sistema de entrega de equipaje del aeropuerto de Denver. Un proyecto que debía estar entregado para octubre de 1993 y en su lugar fue terminado en marzo de 1995, con un retraso de unos 16 meses aproximadamente y con un costo muy por encima del presupuestado [15].

La complejidad en la cual está sumergido el proceso de desarrollo de software se refleja en la rapidez de la evolución de los lenguajes de programación, herramienta fundamental este proceso. Hacia marzo del 2005 existían unos 7957 [17] lenguajes de programación, al día de hoy hay unos 8512 [49]. En unos 3 años y 7 meses el número de lenguajes de programación aumentó en 555 unidades, aproximadamente un 7 % más.
En la figura 1.1, figura 1.2 se pueden ver los árboles genealógicos de todos los lenguajes de programación, la primera actualizada al año 2005 y la última actualizada al mes de octubre de 2009. Ambas figuras en su formato original medían 3 x 6 metros. En la figura 1.3 se intenta realizar una ampliación de una de las imágenes para poder ver el grado de complejidad de éstas estructuras parentales.

En los últimos 50 años la ingeniería de software ha visto nacer, crecer, vivir y perder notoriedad a 5 paradigmas de programación [17]: el declarativo, el funcional, el estructurado, el orientado a objetos y el orientado a frameworks.

A partir de estos problemas detectados en el desarrollo de software, se han realizado estudios sobre el ciclo de vida del proceso de desarrollo encontrándose algunos aspectos notables. Uno de ellos, se centra en la etapa de mantenimiento del software, ésta constituye la fase que requiere de más esfuerzos dentro del ciclo de vida del desarrollo del producto. Entre el 60 % y el 70 % de todo el esfuerzo es utilizado en la etapa de mantenimiento. Esta etapa se divide en: mantenimiento correctivo, mantenimiento adaptativo, mantenimiento preventivo y mantenimiento perfectivo [26] [12].

- Mantenimiento Perfectivo: se aplican tareas de mantenimiento para perfeccionar el sistema de software en términos de performance, mantenibilidad, eficiencia de procesamiento, etc. Puede implicar la implementación de nuevos requerimientos.
- Mantenimiento Correctivo: se aplican tareas de mantenimiento para la corrección de errores en el sistema de software.

- Mantenimiento Adaptativo: se aplican tareas de mantenimiento para adaptar al sistema de software a cambios producidos en su entorno. Puede necesitar la implementación de nuevos requerimientos.

- Mantenimiento Preventivo: El propósito es prever averías o desperfectos.

Un error conceptual es pensar en el software como un producto al que puede definírsele una etapa de finalización, y que éste se "termina de construir" en un determinado momento. Un importante avance se ha hecho dentro de la disciplina de la ingeniería de software, cuando se logró entender que la primera versión de un producto no es más que eso, una "primera versión" dentro de un proceso de evolución continua [17].

A medida que el software evoluciona a través de sucesivas liberaciones de distintas versiones del producto, su código fuente también va cambiando. Es de vital importancia conocer qué componentes del producto siguen siendo estables y cuáles necesitan que se le aplique mantenimiento correctivo y cuáles no, pues los últimos son aquellos más susceptibles a que su código fuente decaiga en lo que se refiere a legibilidad, correctitud, calidad y otros aspectos internos. Estas sucesivas modificaciones hacen que el código empeore versión tras versión, tornándolo inmantenible debido a la introducción de bugs en nuevas correcciones, a la adaptación a nuevos requerimientos, etc. [42].

1.2. Software Restructuring

A partir de esta problemática una rama de la ingeniería de software se abocó a la investigación de cómo mejorar el código fuente, en base la aplicación de transformaciones al código ya existente. La reestructuración de código fuente nació como la "aplicación de modificaciones al código fuente para hacerlo más fácil de cambiar y de comprender, o hacerlo menos susceptible a errores cuando futuros cambios sean aplicados" [2]. Cabe destacar que en la definición utilizada por Arnold se excluye a la reestructuración para cualquier otro propósito, como la mejora del código para aumentar performance, la transformación de código para paralelizar, etc. Vista como tal, la reestructuración es una herramienta que podría ayudar a resolver el problema del enorme esfuerzo que se requiere en la fase de mantenimiento, dentro del ciclo de vida del desarrollo del software. Existen otros motivos por los cuales, según Arnold [2], la reestructuración del código fuente debe ser tenida en cuenta por los ingenieros de Software:

- Factores que están asociados a la comprensibilidad del código fuente :

    - Facilitar la documentación.
    - Facilitar las tareas de testing.
    - Facilitar las tareas de auditoría.
    - Reducir potencialmente la complejidad del software.
- Reducir el tiempo que necesitan los programadores para familiarizarse con el sistema antes de implementar tareas de mantenimiento.
- Hacer más fácil el reconocimiento de errores.
- Facilitar la introducción de nueva funcionalidad.
- Implementación de estándares en la estructura del código.

La reestructuración de programas surge como herramienta necesaria para ser aplicada en los procesos de mantenimiento debido a las características esenciales del software para reducir los costos de esta fase del proceso de desarrollo y como herramienta en la introducción de nueva funcionalidad dentro del ciclo evolutivo de las aplicaciones.

El objetivo de la reestructuración de software es principalmente mantener o aumentar el valor del mismo. Exteriormente, el valor se aumenta, mediante la satisfacción de los usuarios, ya estén éstos relacionados con el proyecto o no. A medida que los usuarios generan y peticionan cambios, debidos a las características del dominio de la aplicación, estas modificaciones son plasmadas durante la etapa de mantenimiento, haciendo que los programas se vuelvan más y más rígidos tornándose paulatinamente más difíciles de modificar. Internamente, el valor del software puede ser medido: a) por el ahorro en costo de mantenimiento que una aplicación puede aportar a otra; b) por el ahorro que proporciona la reutilización de componentes en otras aplicaciones; y c) por el ahorro incurrido en la ampliación del ciclo de vida del sistema previo a su reposición por uno nuevo. Como puede verse la reestructuración de código fuente reduce los costos de mantenimiento, a su vez aumenta la posibilidad de reutilización de componentes en otros sistemas y amplía el ciclo de vida del sistema aumentando de esta manera el valor del mismo [2]. Por ende, es importante a la hora de planear la aplicación de reestructuración de código evaluar como ese proceso de reestructuración aumentará el valor actual del sistema.

Puede encontrarse otra definición del concepto de reestructuración de software aportada por Chikofsky y Cross [13] : “es una transformación de una forma de representación a otra en el mismo nivel de abstracción, mientras se mantiene el mismo comportamiento externo del sistema en cuestión

(funcionalidad y semántica). Una transformación de reestructuración básicamente se aplica en la apariencia del sistema, como por ejemplo la alteración del código para mejorar su estructura, en el sentido tradicional del diseño estructurado. Mientras que la reestructuración crea nuevas versiones que implementan o proponen cambios al sistema, no hace modificaciones debido a variaciones en los requerimientos del sistema. Esto puede acercar a tener una adecuada comprensión del mismo que sugiera cambios que mejoren aspectos del sistema".

Arnold [2] propone distintas técnicas de restructuración de software que se listan a continuación :

1. Técnicas de reestructuración orientadas a modificaciones en el código fuente:

   a El estilo de programación: Se intenta reestructurar el código fuente de una aplicación para hacerlo más fácil de comprender:

   - *Pretty printing and code formatting*: Se mejora el aspecto del código fuente aplicando identación, espaciado, se lleva a una instrucción por línea, etc.

   - *Estandarización del estilo de programación*: Se modifica el código fuente del programa para hacerlo compatible con algún estándar, por ejemplo estándar de COBOL, FORTRAN, etc.

   - *Reestructuración con pre-procesador [29]*: Se sustituye código fuente con directivas más fáciles de comprender que posteriormente serán reemplazadas por el código original.

   b Paquetes / Código Reusable: Esta técnica consiste en la utilización de componentes, paquetes, bibliotecas ya utilizadas anteriormente.

   - *Reestructuración para reusabilidad [30]*: Se modifica el código fuente de forma tal que se modifique para poder ser aprovechado en otros programas.

- *Adquirir un paquete para sustituir un sistema obsoleto [11]*: Se sustituye un programa viejo con otro de similares características.

- *Adquirir un componente para sustituir un programa y además extender su funcionalidad [11]*: Se compra un componente nuevo que pueda extender incluso la funcionalidad del sistema a reemplazar. Esta técnica se utiliza cuando es muy riesgoso o económicamente inviable la construcción de una nueva aplicación para reemplazar una vieja.

- *Adquirir un componente para reemplazar parte de un sistema viejo [11]*: Igual que el anterior, pero el Nuevo componente sólo reemplaza una parte del Viejo programa.

- *Técnica del sistema sándwich [11]*: Esta técnica es utilizada en aplicaciones con una intrincada y compleja estructura. Se van haciendo reestructuraciones en capas, por ejemplo la interfaz de usuario, la capa de base de datos. Suponiendo una lógica de negocio compleja, se utiliza esta capa como una caja negra. Y se encierra en sándwich entre las otras capas.

c Control de flujo: Uno de los motivos por los cuales surge la reestructuración, es justamente para mejorar y hacer más sencillo el flujo de un programa. Esta categoría permitió desarrollar un gran número de herramientas. Dentro de las técnicas de reestructuración es una de las más complejas.

- *Eliminación de go-to*: todas las técnicas abajo enumeradas en este apartado, intentan eliminar los go-to no estructurados de un programa:
  - *Early goto-less approach [7].*
  - *Giant case statementapproach [3].*
  - *Boolean flagapproach [60].*
  - *Duplication of coding approach [60].*

- *Baker's graph-theoretic approach [5].*
- *Refined case statement approach [35].*

▪ Las herramientas enumeradas a continuación son utilizadas para reestructuración de programas COBOL de diversas empresas:

- *RETROFIT (tm)[36].*
- *SUPERSTRUCTURE (tm) [41].*
- *RECORDER (tm) [10].*
- *Cobol Structuring Facility (tm) [34].*
- *Delta STRUCTURIZER (tm).*
- *Doble conversión.*

d Datos: La reestructuración de software básicamente se centró en la reestructuración de las estructuras de control y no en los datos. Este hecho no implica la imposibilidad de aplicar reestructuring a datos en una base de datos. Por ejemplo cuando se aplica el proceso de normalización, pasar de 2da. forma normal a 3ra. forma normal, en ese proceso se puede decir que hay una reestructuración de los datos de una aplicación.

2. **Documentación o Actualización de la documentación**: Consiste en mantener actualizada la documentación de un sistema, facilitando así la reingeniería y la aplicación de ingeniería inversa.

▪ *Re-Modularización*: Estudia la forma de descomponer una aplicación en módulos que posean algún criterio de significación. Puede aplicarse a sistemas ya modularizados.

▪ *Auto documentación*: Utiliza herramientas que permitan agregar meta información dentro de los programas para mejorar la com-

prensibilidad del mismo.

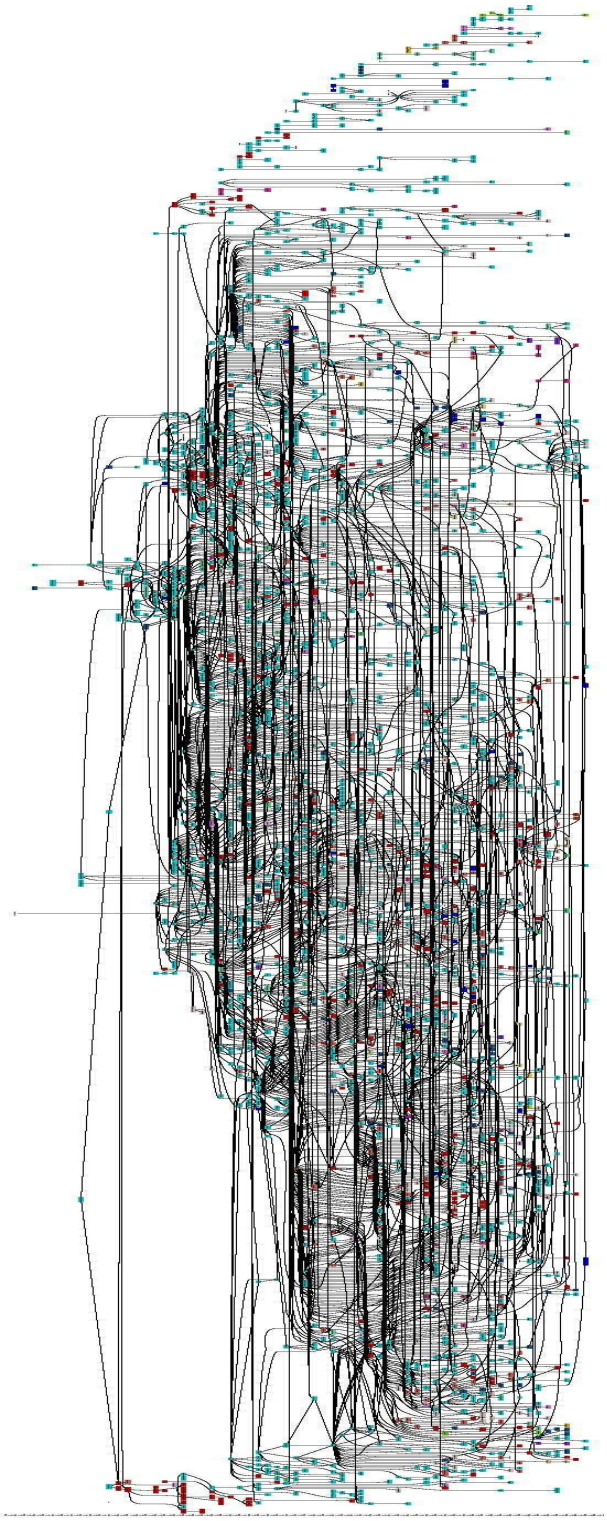

Figura 1.1: Árbol de Genealogía de los 7957 lenguajes de programación marzo 2005[17], el archivo pdf original mide 3 m x 6 m

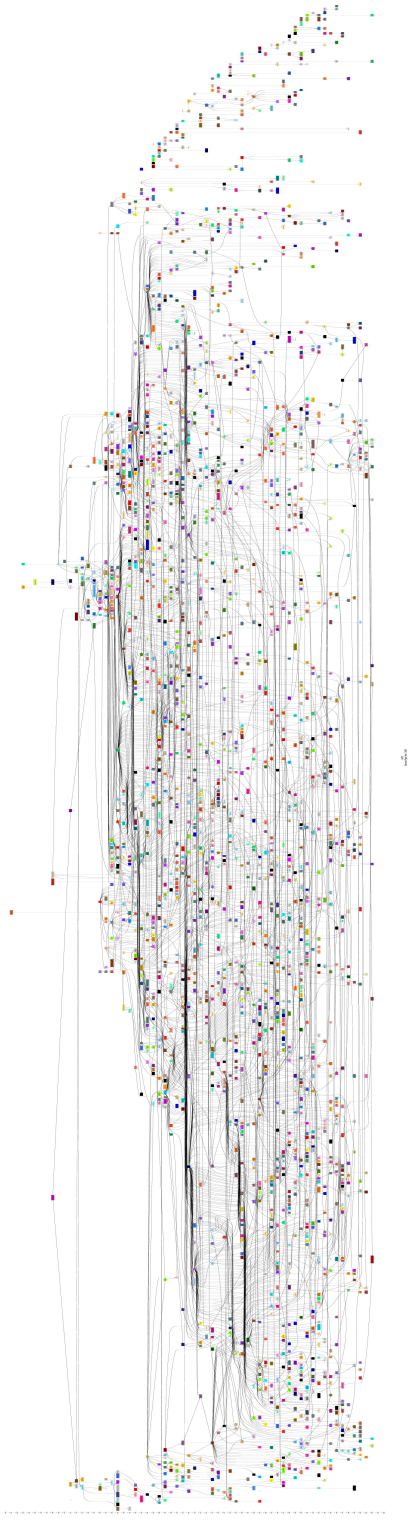

Figura 1.2: Árbol de Genealogía de los 8512 lenguajes de programación Octubre 2009, el archivo pdf original mide 3 m x 6 m

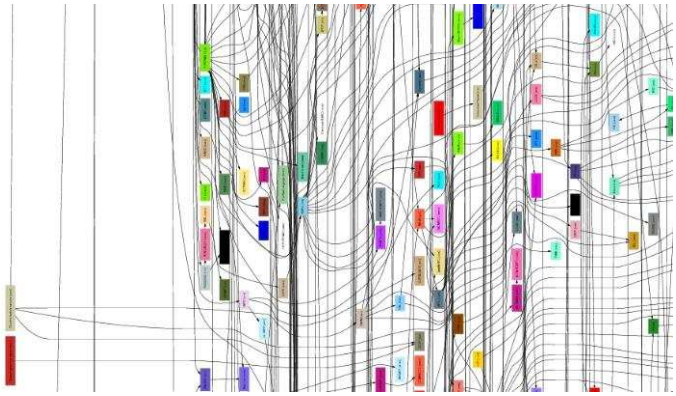

Figura 1.3: Ampliación de parte del árbol genealógico correspondiente a la Figura 1.1

Capítulo 2

Refactorización

Desde su creación los sistemas de software han sido utilizados, probados e incluso modificados. A lo largo de este período el software y los seres humanos han interactuado entre sí en forma cotidiana, no pudiéndose concebir la vida actual del hombre sin la presencia de los sistemas informáticos (comunicaciones, medicina, transporte, etc.). A tal punto que paulatinamente los procesos de negocio consumen cada vez más información procesada por los sistemas de software, incluso hasta a ser controlados y guiados por software. Por ende, las especificaciones y el diseño de estos sistemas requieren de supuestos acerca de la aplicación dada, del dominio de la aplicación y de su ámbito operativo, hecho que a su vez se refleja en el software.

2.1. La Decadencia del Código Fuente

Lehman [32] [31] propone una serie de leyes que guían la evolución de los sistemas de software:

1. **Cambio Continuo** - Los sistemas deben adaptarse continuamente de lo contrario se hacen progresivamente menos satisfactorios. Estas adaptaciones son el resultado del cambio en la operación del entorno en el cual la aplicación cumple una función.

2. **Complejidad Creciente** - A medida que evoluciona un programa, su complejidad se incrementa, a menos que se trabaje para mantenerla o reducirla. Esta ley implica un tipo de degradación o entropía en la estructura del programa. Esto a su vez conlleva a un aumento progresivo del esfuerzo de mantenimiento, a menos que se realice algún tipo de mantenimiento perfectivo a este respecto.

3. **Autoregulación** - El proceso de evolución del programa se autoregula con una distribución de medidas de atributos de producto y procesos cercana a la normal.

4. **Conservación de la Estabilidad Organizativa** - La velocidad de actividad global efectiva media en un sistema en evolución es invariante a lo largo del ciclo de vida del producto.

5. **Conservación de la Familiaridad** - Durante la vida activa de un programa en evolución, el contenido de las versiones sucesivas es estadísticamente invariante.

6. **Crecimiento Continuo** - El contenido funcional de un programa debe incrementarse continuamente para mantener la satisfacción del usuario durante su ciclo de vida.

7. **Calidad Decreciente** - Los sistemas serán percibidos como de calidad decreciente a menos que se mantengan de manera rigurosa y se adapten al entorno operativo cambiante.

En base a que el software tiene un comportamiento evolutivo se pudo observar que, en gran medida, muchos sistemas tienden a ser construidos/mantenidos con técnicas/prácticas que propician un camino casi hacia la antiiengeniería. Estas prácticas, que se desarrollan bajo la inercia cotidiana de las tareas destinadas a emparchar agujeros surgen a veces de la incomprensión del sistema, y otras, de la falta de bases de una buena arquitectura; esto es debido a la necesidad imperiosa del cumplimento de calendarios desnutridos, del ahorro de recursos costosos, de entregas construidas sobre estimaciones ficticias. Pruebas de estos hechos se encuentran dispersas en la vida cotidiana de los ingenieros de software. Estos sistemas tienden a convertirse a la larga en una "gran bola de barro" (big ball of mud), una masa amorfa de comportamiento errático muchas veces unida con delgados hilos de alambre, con una estructura (o más bien "des-estructura") interna más enmarañada que un plato de espaguetti [18]. Éstos van siendo presa del crecimiento desordenado (Piecemeal Growth), del código fuente escrito para salir del paso (Throwaway Code), etc. En este proceso gradual de decaimiento en el cual intervienen los usuarios y los programadores, los primeros solicitando continuos y pequeños cambios a ser efectuados con rapidez sobre el sistema, y los segundos haciendo realidad estas solicitudes a toda costa, existe un atributo del software que inexorablemente va perdiendo fuerza: la calidad. Al estar sometidos continuamente a las presiones del cambio, incluso los sistemas con una arquitectura asentada, pueden sufrir en su ciclo evolutivo, a tal punto

de que su estructura interna sea erosionada como una piedra sometida a los avatares del viento.

Este aumento entrópico al cual se ve sometido el software gracias a la pérdida paulatina de calidad, introducida por los pequeños cambios graduales incorporados de forma incremental en el software, ha llevado a autores a comparar este proceso con el que sufre a través del tiempo cualquier sustancia orgánica, ésta paulatinamente se descompone, es decir, con el tiempo va tomando mal olor! (bad Smell) [21].

2.2. Mal Olor en el Código Fuente

¿Cuándo un sistema de software comienza a tomar mal olor? Un programador experimentado puede intuir que su programa va camino a oler mal cuando hay [21]:

- Código duplicado. Se deben eliminar las líneas de código que son exactamente iguales en varios sitios, o bien eliminar líneas de código muy parecidas o con estructura similar en varios sitios.
- Métodos muy largos. En este caso hay que particionar el código fuente en trozos y extraerlos para crear métodos más pequeños, que sean más fáciles de mantener y de reusar.
- Clases muy grandes. En estos casos se debería tratar de identificar qué cosas hace esa clase, ver si realmente todas esas cosas tienen algo que ver la una con la otra y si no es así, hacer clases más pequeñas, de forma que cada una trate una de esas cosas. Por ejemplo, si una clase es una ventana, además realiza cálculos y escribe los resultados en una base de datos, ya está haciendo demasiadas cosas. Debería haber una clase que sea la ventana, otra que realice los cálculos y otra que sepa escribir en base de datos.
- Métodos que necesitan muchos parámetros. Suele ser buena solución hacer una clase que contenga esos parámetros y pasar la clase en vez de todos los parámetros. Especialmente si esos parámetros suelen tener que ver unos con otros y suelen ir juntos siempre.
- Instrucciones Switch-Case. Normalmente un switch-case se tiene que repetir en el código en varios sitios, aunque en cada sitio sea para hacer cosas distintas. Existen formas, usando el polimorfismo, que evitan

tener que realizar esta repetición a lo largo del código fuente, o incluso evitan tener que ponerlo en algún lado.

Éstas y otras pautas dan al programador la idea de que su código fuente puede ser susceptible a producir mal olor en un corto lapso de tiempo.

Dentro de este panorama casi desolador surge el concepto de refactorización de código fuente como una técnica que permite mejorar la comprensibilidad, la claridad, el diseño, la legibilidad y a su vez reducir la cantidad de errores. En definitiva una técnica para mejorar la calidad del software.

En un principio asociada a la programación orientada a objetos, el concepto de refactorización fue extendiéndose hacia otros paradigmas de programación, teniendo en cuenta la gran cantidad de código fuente escrito en los últimos 50 años de existencia del campo de la informática.

2.3. Refactoring

El primer uso conocido del término refactorización en la literatura se encuentra publicado en el artículo Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems por William F. Opdyke y Ralph E. Johnson [45]. La refactorización surge como un intento de mejorar la producción de software reusable. El desarrollo de software es un proceso complejo y continuo en el cual un producto transita, a lo largo de su construcción, por un proceso iterativo (desarrollo espiral, desarrollo por prototipos, desarrollo de tipo iterativo incremental, otros). El desarrollo de software re-usable es un proceso aun más complejo pues éste es el resultado de varias iteraciones de diseño, incluso cuando éste ya ha sido reusado, por ende los cambios a los que está sujeto, no lo afectan únicamente a él sino también al software que lo utiliza [44]. En base a ello la refactorización surge como "el proceso en el cual se aplican cambios en un sistema de software de forma tal que no altere el comportamiento externo del código, mejorando su estructura interna" [21].

Esta definición, que no deja de ser muy abierta, plantea una serie de cuestiones: ¿En qué consiste el comportamiento externo del software? ¿El área de aplicabilidad de la refactorización está restringida únicamente a software reusable? ¿Puede extenderse este concepto a otros productos del proceso de

desarrollo, o es solamente aplicable a código fuente? ¿En qué consiste la estructura interna? ¿Cuándo, dónde, por qué refactorizar?

2.3.1. El Comportamiento Externo

Dentro de la definición de refactorización, existe un concepto en el cual hay que detenerse un segundo: la idea de la preservación del comportamiento externo del software. Por definición el proceso de refactorización de código fuente debe preservar el comportamiento del software. Pero ¿qué se entiende por comportamiento? En la bibliografía no se encuentra una definición exacta sobre qué es el comportamiento externo del software. En su tesis Opdyke [44], sostiene que el conjunto de entrada y el conjunto de salida deben ser los mismos antes y después de aplicar el proceso de refactorización. Si bien en la definición de Opdyke los conjuntos de entrada y salida se conservan, Mens y Tourwé [39] agregan que ver la conservación del comportamiento sólo desde el punto de vista de las entradas/salidas es exiguo, pues existen otros aspectos dependiendo del dominio que hay que tener en cuenta. Este último enfoque permite ampliar la definición de comportamiento que debe ser o no preservado, dependiendo del dominio e incluso a veces dependiendo de las necesidades del usuario. Mens y Tourwé proponen tipos distintos de dominios:

- Software de tiempo real, en el cual los procesos de refactoring deben preservar las restricciones temporales. El tiempo es un factor que forma parte del comportamiento externo en este caso.
- Software embebido, el consumo de memoria en este caso forma parte del comportamiento a preservar en aplicaciones pertenecientes a este dominio.

Existen además otros dominios en los cuales se pueden definir otros aspectos que hacen al comportamiento externo. En las aplicaciones web, el contenido puede pasar a formar parte del comportamiento externo o como se especifica en [43] el conjunto de nodos y los links de navegación entre los nodos y el conjunto de operaciones disponibles para el usuario y la semántica de cada operación, pueden ser considerados comportamiento externo, desde el punto de vista de la refactorización.

A pesar de los intentos de formalización, no existe una definición formal de lo que es considerado comportamiento externo del software.

2.3.2. Características Internas y Externas del Software

Otro punto destacable aportado por el trabajo de [39] es el efecto de los procesos de refactorización sobre la calidad del software. El software posee características que se manifiestan en forma externa y otras que son propias de la estructura interna del mismo, definidas como características internas. En las primeras encontramos conceptos como robustez, extensibilidad, performance, reusabilidad, etc. Entre las características internas nos encontramos con los conceptos de comprensibilidad, legibilidad, correctitud, redundancia, etc. Los procesos de refactorización pueden afectar a las características internas: al aplicar reducción de código redundante, al aplicar cambios de nombres de métodos o variables. Pero también pueden afectar a factores o características externas que hacen a la calidad del software, por ejemplo la performance. Si bien se cree que la refactorización de código fuente afecta negativamente en cuanto a la performance [21], existen estudios que demuestran lo contrario. Refactorizaciones tendientes a reemplazar intrucciones if con polimorfismo mejoran la performance de la aplicación gracias a las optimizaciones que hacen los compiladores actuales [39].

2.3.3. Un Ejemplo de Refactorización

Éste es un ejemplo muy básico. Es un programa para calcular el valor total de una compra en un negocio de computación. A partir de que el programa sabe qué artículos se compraron para ser facturados, se calcula importe total de la factura. Existen tres tipos de artículos: los que son tipificados como software, los artículos de hardware y otros, que no entran en ninguna de las dos categorías anteriores. Además de calcular el costo de los artículos teniendo en cuenta si existen impuestos que regulen la operación, en este caso el IVA, se calculan una serie de bonificaciones según el tipo de artículo comprado. El diagrama de clases (figura 2.1):

A continuación se lista el código fuente de cada clase en C#

La clase Producto

Representa a un producto que es vendido en la tienda.

```
class Producto {
        public const int SOFTWARE=1;
        public const int HARDWARE=2;
```

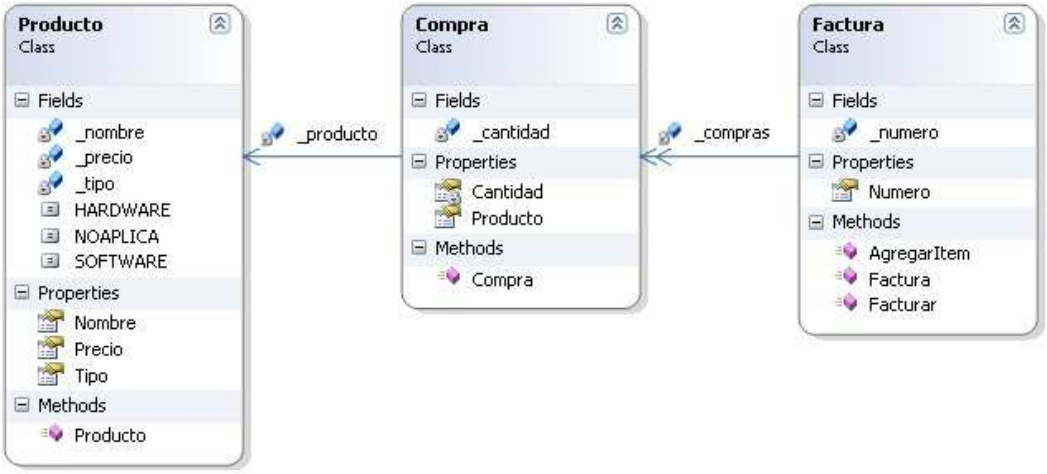

Figura 2.1: Diagrama de clases

```
public const int NOAPLICA=0;

private string _nombre;
private float _precio;
private int _tipo;


public Producto( string nombre, float precio, int tipo ) {
    _nombre = nombre;
    _precio = precio;
    _tipo = tipo;
}

public string Nombre
{
    get { return _nombre;}
    set { _nombre = value; }
}

public float Precio {
    get { return _precio; }
    set { _precio = value; }
}

public int Tipo {
    get { return _tipo; }
```

```
            set { _tipo = value; }
        }
}
```

La clase Compra

Representa la compra de un producto

```
class Compra {
        private Producto _producto;
        private int _cantidad;

        public Compra( Producto producto, int cantidad) {
            _producto = producto;
            _cantidad = cantidad;
        }

        public Producto Producto {
            get { return _producto; }
            set { _producto = value; }
        }

        private int Cantidad {
            get { return _cantidad; }
            set { _cantidad = value; }
        }

}
```

La clase Factura

Representa el importe a pagar por todas las compras realizadas: esta clase es la que se encarga de determinar el valor a pagar de todas las compras realizadas.
Esta clase posee un método llamado facturar() que es el encargado de calcular el valor total y estado de cuenta de la factura, aplicar el impuesto correspondiente y además calcular puntos por las compras realizadas. Ver diagrama de secuencia figura 2.2

```
class Factura {

        private List<Compra> _compras;
        private string _numero;
```

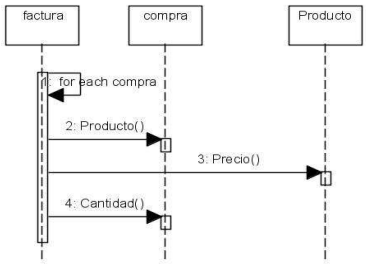

Figura 2.2: Diagrama de secuencia de método facturar()

```
public Factura(string numero) {
    _numero = numero;
    _compras = new List<Compra>();
}

public void AgregarItem(Compra item) {
    _compras.Add(item);
}

public String Numero {
    get { return _numero; }
    set { _numero = value; }
}

public string Facturar() {
    double totalFactura;
    IEnumerator it = _compras.GetEnumerator();
    int puntaje = 0;
    int ticket = "";


    totalFactura = 0;
    while (it.MoveNext) {
        double valorCompra = 0;
        Compra compra = it.Current;
        //Calcular el valor de una compra de la factura
        switch (compra.Producto.Tipo){
            case Producto.HARDWARE:
                valorCompra = compra.Producto.Precio * 10.5;
                break;
            case Producto.SOFTWARE:
                valorCompra = compra.Producto.Precio * 21;
                break;
            case Producto.NOAPLICA:
                valorCompra = compra.Producto.Precio;
                break;
        }
        // calcular bonificacion
        puntaje++;
        if (compra.Producto.Tipo() == Producto.HARDWARE && compra.Cantidad >1 ) puntaje++;
        // Calcular el valor Total de la factura
        totalFactura += valorCompra;
        // imprimir
        ticket= "\t" + compra.Producto.Nombre + "\t" + compra.Cantidad +"\t" +valorCompra + "\n";
     }
```

```
            // Totalizar
            ticket = "El total de su compra es :" + totalFactura.ToString() + "\n";
            ticket = "Su puntaje acumunlado en esta compra es: " + puntaje + "\n";
            }
}
```

El código fuente que se encuentra en el método facturar, sin dudas tiene algunos problemas, si quisiera ser visto desde la programación orientada a objetos. La escasa posibilidad de reutilización por ejemplo, habría que pensar qué sucedería si las reglas de cálculo de los impuestos como el IVA variaran; se debería reescribir completamente el método. Otro cambio al que podría estar sujeta la aplicación podría ser la variación en la clasificación de los tipos de productos.

Antes de producir alguna transformación al código fuente de esta aplicación, hay que tener en cuenta que la refactorización, por definición, no debe introducir ningún cambio en el comportamiento externo de la aplicación. Por ende hay que estar seguro de que esto no suceda, la forma de asegurarnos esto, es precisamente haciendo un muy buen conjunto de test de unidad para esta porción de código fuente [21].

Teniendo en cuenta las secciones anteriores, lo primero que salta a la vista en estas tres clases es la extensión del método facturar(). Existe una refactorización propuesta en este caso en [21] que es llamada "extracción de método" (extract method), que consiste básicamente en dividir un método muy extenso en dos más pequeños, con el objetivo de hacer más manejable el código fuente. Es un hecho comprobable, que es más fácil de seguir y comprender un método que entre en las 24 filas de un monitor que, aquel que ocupe más líneas y haga que sea necesario scroll para poder hacer el seguimiento.

En el caso de facturar() el fragmento de código en el cual es factible una división o extracción de un nuevo método independiente se encuentra en :

```
        public string Facturar() {
            double totalFactura;
            IEnumerator it = _compras.GetEnumerator();
            int puntaje = 0;
            int ticket = "";


            totalFactura = 0;
            while (it.MoveNext) {
                double valorCompra = 0;
                Compra compra = it.Current;
                //Calcular el valor de una compra de la factura
                switch (compra.Producto.Tipo){
                    case Producto.HARDWARE:
```

```
                    valorCompra = compra.Producto.Precio * 10.5;
                    break;
                case Producto.SOFTWARE:
                    valorCompra = compra.Producto.Precio * 21;
                    break;
                case Producto.NOAPLICA:
                    valorCompra = compra.Producto.Precio;
                    break;
            }
            // calcular bonifica
            puntaje++;
            if (compra.Producto.Tipo() == Producto.HARDWARE && compra.Cantidad >1 ) puntaje++;
            // Calcular el valor Total de la factura
            totalFactura += valorCompra;
            // imprimir
            ticket= "\t" + compra.Producto.Nombre + "\t" + compra.Cantidad +"\t" +valorCompra + "\n";
         }
        // Totalizar
        ticket = "El total de su compra es :" + totalFactura.ToString() + "\n";
        ticket = "Su puntaje acumulado en esta compra es: " + puntaje + "\n";
        }
```

Este nuevo método podría llamarse CalcularCompra(compra), quedando el nuevo fragmento de código así :

```
public string Facturar() {
        double totalFactura;
        IEnumerator it = _compras.GetEnumerator();
        int puntaje = 0;
        int ticket = "";


        totalFactura = 0;
        while (it.MoveNext) {
            double valorCompra = 0;
            Compra compra = it.Current;
            //Calcular el valor de una compra de la factura
            valorCompra =CalcularCompra(compra);
            // calcular bonifica
            puntaje++;
            if (compra.Producto.Tipo() == Producto.HARDWARE && compra.Cantidad >1 ) puntaje++;
            // Calcular el valor Total de la factura
            totalFactura += valorCompra;
            // imprimir
            ticket= "\t" + compra.Producto.Nombre + "\t" + compra.Cantidad +"\t" +valorCompra + "\n";
         }
        // Totalizar
        ticket = "El total de su compra es :" + totalFactura.ToString() + "\n";
        ticket = "Su puntaje acumunlado en esta compra es: " + puntaje + "\n";
        }

    private double CalcularCompra(Compra Item) {
        double valor = 0;
        switch (item.Producto.Tipo) {
            case Producto.HARDWARE:
                valor = item.Producto.Precio * 10.5;
                break;
            case Producto.SOFTWARE:
                valor = item.Producto.Precio * 21;
                break;
            case Producto.NOAPLICA:
```

```
                valor = item.Producto.Precio;
                break;
        }
        return valor;
    }
```

Luego de producido el cambio, el siguiente paso es correr los tests para el nuevo fragmento de código y verificar que todos los tests resulten favorables. Una vez hecho esto se puede observar como al reducir en tamaño el método, un cambio casi cosmético, se ha favorecido el código fuente en varios factores:

- Comprensibilidad: el código fuente ha quedado más fácil de comprender.
- Complejidad: el código fuente ha quedado menos complejo. Se acota la responsabilidad del método facturar.
- Legibilidad: el código fuente es ahora legible.
- Mantenibilidad: el código fuente es más fácil de mantener. Supongamos un cambio en la forma de cálculo de la facturación, ésta ha quedado aislada del resto de la funcionalidad de la clase en el método CalcularCompra().

Si bien este cambio ha aportado mejoras desde el punto de vista de la estructura interna del código fuente, y además se ha validado que el comportamiento externo del mismo no ha cambiado (testing) es innegable que se podría mejorar aun más esta estructura interna.

Si leemos más detenidamente el código resultante veremos que en CalcularCompra() no se está utilizando ningún dato que pertenece a la factura, por ende se puede llegar a la conclusión que el método CalcularCompra() está en el objeto equivocado, pues debería pertenecer a la clase Compra. He aquí la posibilidad de aplicar otra refactorización propuesta también en [21] llamada "Mover Método"que consiste en mover un método de una clase a otra. En este caso se debería mover CalcularCompra() de la clase Factura a la clase Compra.

```
class Compra {

    private Producto _producto;
    private int _cantidad;
```

```
        public Compra (Producto producto, int cantidad) {
            _producto = producto;
            _cantidad = cantidad;
        }



        private double CalcularCompra() {
            double valor = 0;
            switch (_producto.Tipo) {
                case Producto.HARDWARE:
                    valor = _producto.Precio * 10.5;
                    break;
                case Producto.SOFTWARE:
                    valor = _productoPrecio * 21;
                    break;
                case Producto.NOAPLICA:
                    valor = _productoPrecio;
                    break;
            }
            return valor;
        }
    }
```

Tras haber aplicado este cambio, que desde el punto de la programación orientada a objetos es más lógico que el método CalcularCompra pertenezca a la clase Compra que a la clase Factura. Otra vez se han aplicado cambios internos que no modifican el comportamiento externo, pero aportan a la estructura del código fuente mejoras sustanciales.

Ha resultado impresindible poder agrupar esta serie de cambios o refactorizaciones para tenerlas en cuenta cuando sean necesarias. En su libro Martin Fowler [21] ha proporcionado un catálogo de más de 90 refactorizaciones. A continuación se listan (http://www.refactoring.com/catalog/):

1. Add Parameter
2. Change Bidirectional Association to Unidirectional
3. Change Reference to Value
4. Change Unidirectional Association to Bidirectional

5. Change Value to Reference
6. Collapse Hierarchy
7. Consolidate Conditional Expression
8. Consolidate Duplicate Conditional Fragments
9. Convert Dynamic to Static Construction
10. Convert Static to Dynamic Construction
11. Decompose Conditional
12. Duplicate Observed Data
13. Eliminate Inter-Entity Bean Communication
14. Encapsulate Collection
15. Encapsulate Downcast
16. Encapsulate Field
17. Extract Class
18. Extract Interface
19. Extract Method
20. Extract Package by Gerard M. Davison
21. Extract Subclass
22. Extract Superclass
23. Form Template Method
24. Hide Delegate
25. Hide Method
26. Hide presentation tier-specific details from the business tier
27. Inline Class
28. Inline Method
29. Inline Temp

30. Introduce A Controller
31. Introduce Assertion
32. Introduce Business Delegate
33. Introduce Explaining Variable
34. Introduce Foreign Method
35. Introduce Local Extension
36. Introduce Null Object
37. Introduce Parameter Object
38. Introduce Synchronizer Token
39. Localize Disparate Logic
40. Merge Session Beans
41. Move Business Logic to Session
42. Move Class by Gerard M. Davison
43. Move Field
44. Move Method
45. Parameterize Method
46. Preserve Whole Object
47. Pull Up Constructor Body
48. Pull Up Field
49. Pull Up Method
50. Push Down Field
51. Push Down Method
52. Reduce Scope of Variable by Mats Henricson
53. Refactor Architecture by Tiers
54. Remove Assignments to Parameters

55. Remove Control Flag
56. Remove Double Negative by Ashley Frieze and Martin Fowler
57. Remove Middle Man
58. Remove Parameter
59. Remove Setting Method
60. Rename Method
61. Replace Array with Object
62. Replace Assignment with Initialization by Mats Henricson
63. Replace Conditional with Polymorphism
64. Replace Conditional with Visitor by Ivan Mitrovic
65. Replace Constructor with Factory Method
66. Replace Data Value with Object
67. Replace Delegation with Inheritance
68. Replace Error Code with Exception
69. Replace Exception with Test
70. Replace Inheritance with Delegation
71. Replace Iteration with Recursion by Dave Whipp
72. Replace Magic Number with Symbolic Constant
73. Replace Method with Method Object
74. Replace Nested Conditional with Guard Clauses
75. Replace Parameter with Explicit Methods
76. Replace Parameter with Method
77. Replace Record with Data Class
78. Replace Recursion with Iteration by Ivan Mitrovic
79. Replace Static Variable with Parameter by Marian Vittek

80. Replace Subclass with Fields
81. Replace Temp with Query
82. Replace Type Code with Class
83. Replace Type Code with State/Strategy
84. Replace Type Code with Subclasses
85. Reverse Conditional by Bill Murphy and Martin Fowler
86. Self Encapsulate Field
87. Separate Data Access Code
88. Separate Query from Modifier
89. Split Loop by Martin Fowler
90. Split Temporary Variable
91. Substitute Algorithm
92. Use a Connection Pool
93. Wrap entities with session

Fowler [21] considera que erróneamente puede llegarse a la conclusión, que refactorizar es aplicar pequeños cambios al código fuente variando imperceptiblemente o no el comportamiento externo del mismo. Solamente los cambios que hacen que el software sea más fácil de entender son considerados para el autor, como refactorización.

2.3.4. Otro Ejemplo Menos Trivial

El ejemplo que se presenta a continuación, es menos trivial que el anterior. Éste sirve para ver como mediante la refactorización de código fuente se puede además introducir por ejemplo patrones de diseño dentro de una aplicación ya escrita y funcionando. En el caso que a continuación se presenta una clase Empleado es refinada mediante herencia en tres subclases: Administrativo, Supervisor y Gerente. Cada una de ellas tiene que realizar ciertos cálculos en el momento de la liquidación de haberes y de las vacaciones, estos

cálculos varían según el tipo de empleado, por ende, los métodos LiquidarSueldo y LiquidarVacaciones de estas clases son distintos en cada caso. La clase Empleado además posee dos clases de soporte que son LiquidadorSueldo y LiquidadorVacaciones, que como su nombre lo indica, son responsables de calcular el sueldo y las vacaciones de un empleado .

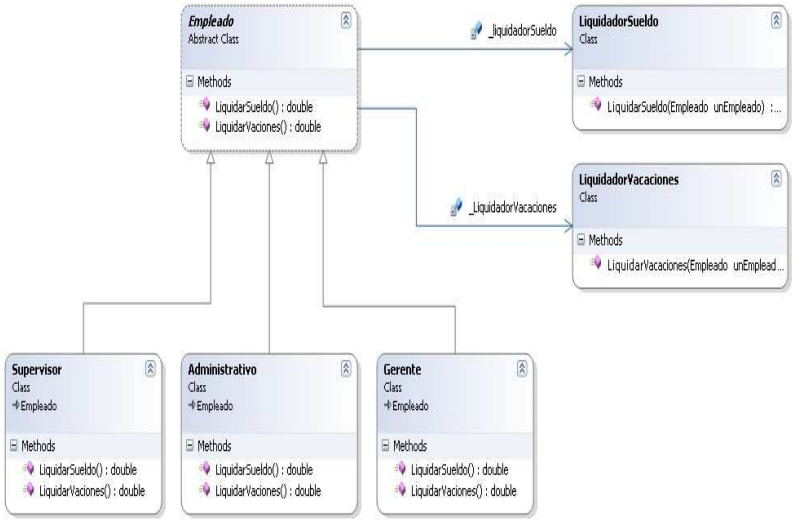

Figura 2.3: Diagrama inicial de clases

Este diseño no es óptimo, pues diferente funcionalidad de la clase empleado está diseminada a lo largo de las clases hijas. Llegado el caso de necesitar agregar nueva funcionalidad a la clase Empleado, se deberían realizar cambios a lo largo de todas las subclases, además se necesitaría una clase de soporte para realizar la nueva funcionalidad. Este tipo de evolución incrementa además la complejidad y reduce la comprensibilidad del diseño [39]. Es posible evitar este inconveniente aplicando refactorización al diseño del sistema, esto podría lograrse introduciendo un patrón de diseño gof: Visitor [22] el cual mantendría la funcionalidad cohesionada en la clase Empleado.

¿Cuales serían los pasos a seguir para lograr transformar el diseño existente? Para ello se deberán aplicar algunas de las refactorizaciones propuestas por Fowler:

1. Mover el método LiquidarSueldo() de las subclases de Empleado a la clase LiquidadorSueldo, aplicando Move Method.

2. Renombrar el método LiquidarSueldo() movido en 1 con un nuevo nombre (visitar), usando Rename Method.

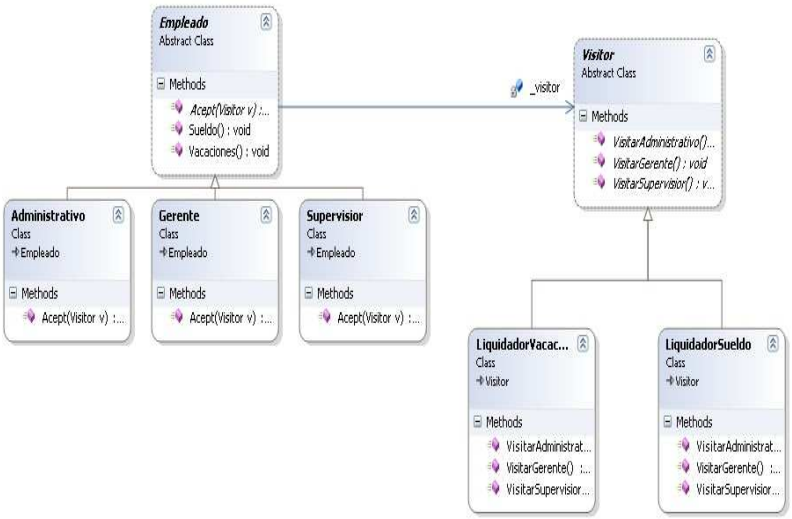

Figura 2.4: Diagrama de clases refactorizado

3. Mover el método LiquidarVacaciones() de las tres subclases de Empleado a la clase LiquidadorVacaciones, aplicando también en este caso Move Method.

4. Renombrar el método LiquidarVacaciones() movido en 1 con un nuevo nombre (visitar), usando Rename Method.

5. Introducir una nueva clase abstracta llamada Visitador, como superclase de LiquidadorSueldo y LiquidadorVacaciones, aplicando la refactorización Add Class.

6. Agregar tres nuevos métodos abstractos llamados VisitarAdministrativo(), VisitarSupervisor() y VisitarGerente() aplicando la refactorización llamada Add Method.

7. Agregar un nuevo método llamado Aceptar() en la clase Empleado, aplicando Add Method.

Haciendo una combinación de las refactorizaciones de la lista de Fowler se puede lograr introducir un patrón gof a un diseño ya existente [28]. Esta herramienta se torna de fundamental importancia en las tareas de mantenimiento de sistemas heredados (Legacy Systems), dado que permite mediante la utilización compuesta de refactorizaciones obtener un diseño mejorado de aplicaciones que se encuentran corriendo en ambientes de producción; además, gracias a la definición de refactorización se puede asegurar que el

comportamiento externo de la misma no varíe. Asimismo, es posible extraer otras refactorizaciones a partir de un conjunto primitivo si se toman, por ejemplo, los pasos 1 al 7 se podría componer una refactorización llamada MoveMethodsToVisitor [28][39][44]. En el libro Refactoring to Patterns [28] se puede encontrar una lista propuesta de composiciones de refactorizaciones que logran arribar a la mejora del diseño introduciendo patrones gof:

- Chain Constructors
- Compose Method
- Encapsulate Classes with Factory
- Encapsulate Composite with Builder
- Extract Adapter
- Extract Composite
- Extract Parameter
- Form Template Method
- Inline Singleton
- Introduce Null Object
- Introduce Polymorphic Creation With Factory Method
- Limit Instantiation with Singleton
- Move Accumulation to Collecting Parameter
- Move Accumulation to Visitor
- Move Creation Knowledge to Factory
- Move Embellishment to Decorator
- Replace Conditional Dispatcher with Command
- Replace Conditional Logic with Strategy
- Replace Constructors with Creation Methods
- Replace Hard-Coded Notifications with Observer
- Replace Implicit Language with Interpreter

- Replace Implicit Tree with Composite
- Replace One/Many Distinction/s with Composite
- Replace State-Altering Conditionals with State
- Replace Type Code with Class
- Unify Interfaces with Adapter
- Unify Interfaces

Algunos de ellos son autodescriptivos, por ejemplo, Replace Implicit Tree with Composite, permite reemplazar una estructura claramente similar a la de un árbol por el patrón Composite. Otro ejemplo interesante de observar es cómo sustituir código enmarañado que varía según estados por el patrón State, o extraer un método de cálculo con Strategy(figura 2.5).

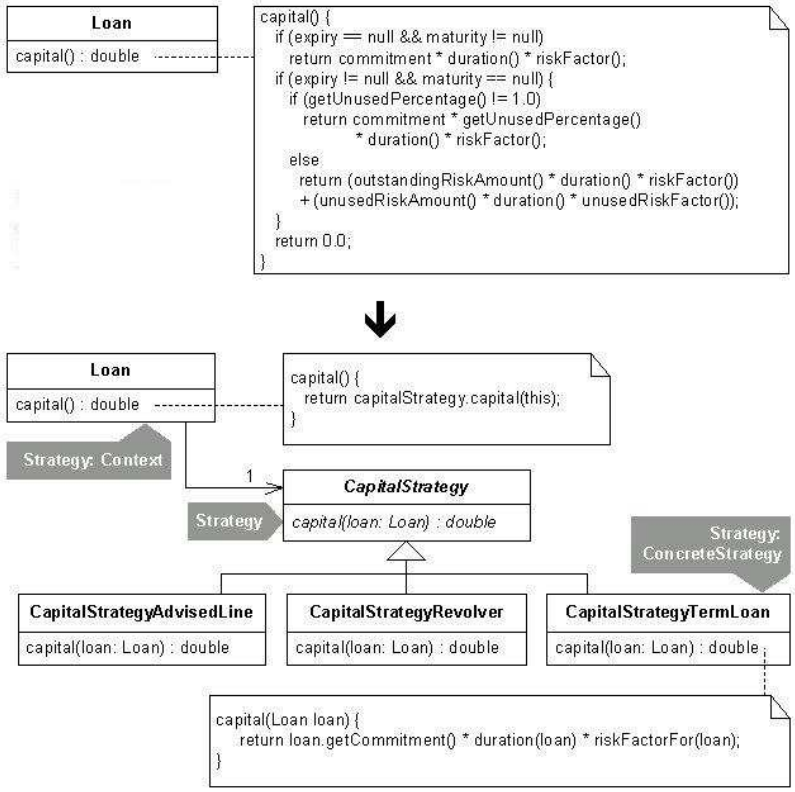

Figura 2.5: Refactorización de método de cálculo aplicando Strategy

Capítulo 3

Herramientas de Refactorización

Durante sus dieciocho años de vida el concepto de refactorización se ha extendido más allá del ámbito para el cual fue creado (OOP). Su área de aplicabilidad se ha extendido a tal punto que se puede encontrar el concepto de refactoring utilizado en muchas áreas de la ingeniería de software. Dada su utilidad como herramienta en la etapa de mantenimiento se han desarrollado herramientas automatizadas para un gran número de lenguajes de programación, así como también para distintos tipos de artefactos que se generan a lo largo del proceso de desarrollo de software.

3.1. Entornos Integrados de Desarrollo

En la actualidad, la mayoría de los sistemas integrados de desarrollo, comúnmente llamados IDEs, poseen opciones automatizadas para realizar refactorizaciones. Estas utilidades están disponibles para los programadores como opciones de menú. Estas herramientas asisten al programador en el proceso de refactorización y favorecen la escritura de buen código fuente brindando la posibilidad de refactorizar en el mismo momento en el cual el código fuente es escrito. Ejemplos de estos productos se encuentran en entornos como Eclipse, Visual Estudio .Net, net.Beans, etc. Todos ellos proporcionan opciones para aplicar técnicas de refactorización (Figura 3.1, Figura 3.2).

A partir de que esta técnica se introdujo en el proceso de desarrollo de software ha surgido una gran cantidad de herramientas automatizadas para ser aplicadas algunas como plug-in de entornos de programación y otras como refactoring browsers.

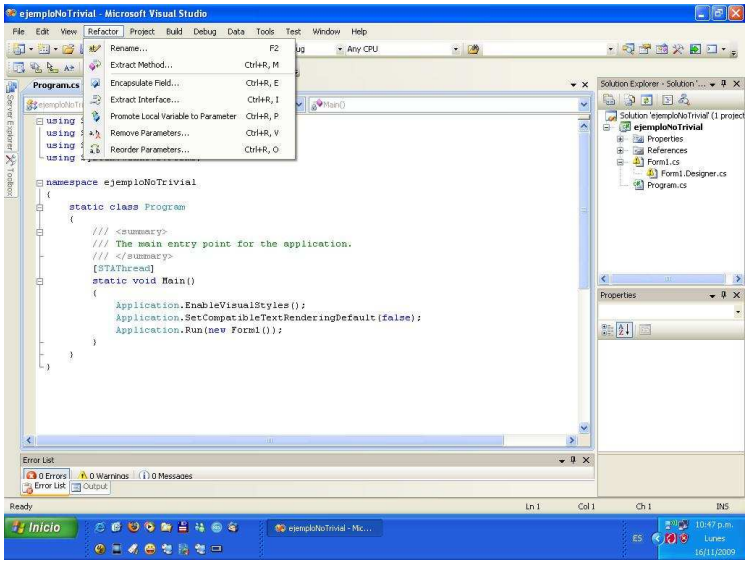

Figura 3.1: Microsoft Visual Studio

3.2. Herramientas Comerciales

A continuación se detalla una lista de las herramientas comerciales existentes en la actualidad para distintos lenguajes de programación:

3.2.1. Smalltalk

- Smalltalk Refactoring Browser

  Esta herramienta se desarrolló para VisualWorks, VisualWorks/ENVY, y IBM Smalltalk. Incluye las características del entorno de ventanas estándar de Smalltalk además con técnicas de refactorización. (http://st-www.cs.uiuc.edu/users/brant/Refactory/RefactoringBrowser.html).

3.2.2. Java

- InteliJ Idea IDE que es capaz de aplicar algunas técnicas de refactoring como: renombrar, extraer métodos, introducir variables, etc. (http://www.jetbrains.com/idea/index.html)

- JFactor

  Es en realidad una familia de productos que provee herramientas para aplicar técnicas de refactorización. Éste es capaz de integrarse al

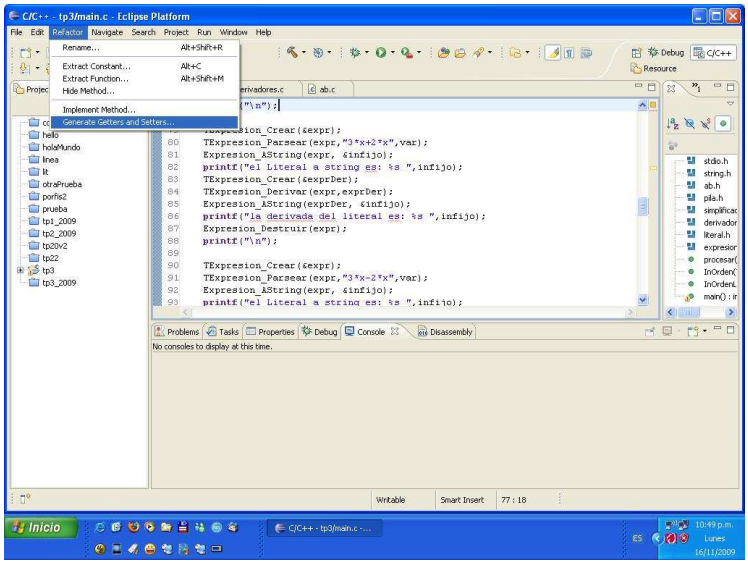

Figura 3.2: Eclipse

entorno de desarrollo VisualAge. Este producto permite aplicar las siguientes refactorizaciones: extract method, rename method variables, introduce explaining variable, inline temp, replace magic number with symbolic constant, inline method, rename method, safe delete method, pull up method, push down method, introduce foreign method, rename field, pull up field, push down field, encapsulate field, extract superclass, extract interface.
(http://old.instantiations.com/jfactor/default.htm)

- XRefactory

  Es una herramienta de desarrollo de software para C y Java que proporciona la posibilidad de aplicar técnicas de refactorización. Una de sus notables características es que este producto es un plug-in para Emacs. Además soporta otros lenguajes de programación como C (figura 3.3). Otra característica de este producto es que sus creadores afirman que ha sido desarrollado para trabajar en proyectos de gran envergadura, proyectos con millones de líneas de código fuente.
  Este producto permite aplicar: extracción y renombramiento de paquetes, clases, parámetros, variables, inserción, borrado y movimiento de parámetros, atributos y métodos; pushing down y pulling up atributos y métodos; encapsulamiento de atributos, entre otros.
  (http://www.xref.sk/xrefactory/main.html)

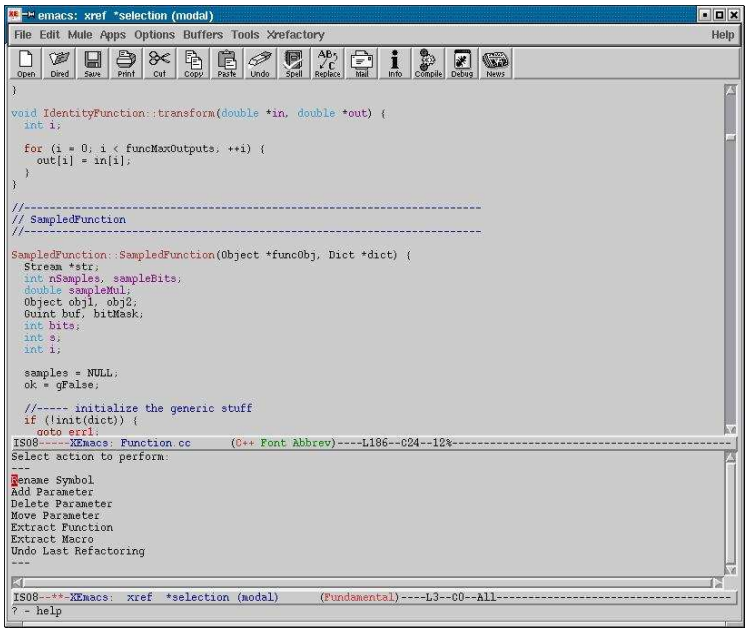

Figura 3.3: XRefactory

- JBuilder

  Es un entorno integrado de desarrollo para java que además permite aplicar técnicas de refactorización. El software fue creado en 1995. La versión 2006 (Borland JBuilder 2006) tiene 3 ediciones: Enterprise (para aplicaciones J2EE, Web Services y Struts), Developer (para el desarrollo completo de aplicaciones Java) y Foundation (con capacidades básicas para iniciarse en el desarrollo de aplicaciones Java y por el momento es gratuito).

- RefactorIt

  Es una herramienta para desarrolladores Java, que posee la habilidad de aplicar 30 técnicas distintas de refactorización de código fuente. Además proporciona un analizador gráfico de dependencias y más de 100 métricas para medir calidad del software. Puede ser utilizado como herramienta stand alone o integrada como plug-in a Eclipse, NetBean y Jbuilder.
  (http://freshmeat.net/projects/refactorit/)

- JRefactory

  Chris Seguin ha desarrollado este software durante el período 1999-2002, llegó a la versión 2.6.40. Desde entonces, Mike Atkinson se ha

hecho cargo llegando a liberar la versión 2.8 en octubre de 2003 con mejoras importantes. El software es gratuito y se proporciona "tal cual".

IDEs soportados por la herramienta

- jEdit (4.1final and 4.2pre11).
- Netbeans (3.6) - parcialmente soportado.
- JBuilder X - parcialmente soportado.
- Ant (1.5.4) - solamente pretty printing.
- También funciona como herramienta stand alone.

La próxima versión soportará más IDEs y con un mayor nivel de integración para los siguientes IDEs: Eclipse, InteliJ y Emacs JDE.

Características:

JRefactory permite aplicar las siguientes refactorizaciones: Move class between packages (repackage), Rename class, Add abstract parent class, Add child class, Remove empty class, Extract interface, Push up field, Push down field, Rename Field, Push up method, Push up abstract method, Push down method, Move method, Extract method, Rename Parameter.
Se presenta como aplicación de línea de comando, stand alone y como plug-in para Jedit, Jbuilder, NetBeans y Elixir.
(http://jrefactory.sourceforge.net/)

- Transmogrify

  El propósito principal de esta aplicación es la de refactorizar código fuente, pudiendo ésta aplicar:

  - Rename Symbol Extract Method
  - Replace Temp With Query
  - Inline Temp
  - Pull up field

  El producto se utiliza como plug-in de Jbuilder y Forte4Java. El mecanismo de funcionamiento consiste en parsear los archivos de un proyecto y a partir de la generación de una ventana con todo el código fuente, basta con posicionarse en el símbolo en el cual se desea aplicar alguna

refactorización y seleccionar la misma en el menú de la herramienta. El producto analiza el código fuente, selecciona y valida si la refactorización seleccionada es aplicable. De no serlo muestra un diálogo de error. En el caso que sea aplicable se realiza el cambio y se realiza copia de seguridad de los fuentes.
(http://transmogrify.sourceforge.net/)

- JavaRefactor

  Este producto es un plug-in para Jedit. Permite aplicar refactorización basado en un catálogo de refactorizaciones para Java. Con este producto se puede renombrar clases, atributos, métodos y paquetes; hacer PushDown y PullUp de métodos y atributos a través de una jerarquía de clases. Otras refactorizaciones serán incluidas en futuras versiones según sus creadores.
  (http://plugins.jedit.org/plugins/?JavaRefactor)

- CodeGuide

  Este producto es un entorno integrado de desarrollo para Java que ofrece la posibilidad de aplicar refactoring al código fuente Java escrito con esta herramienta.
  (http://www.omnicore.com/en/)

- CloneDR

  Este producto detecta y elimina código fuente duplicado en aplicaciones de gran envergadura. Según sus creadores, se repite entre el 10 - 25 % del código fuente escrito en aplicaciones con un gran número de líneas de código. Esta herramienta permite detectar esta redundancia y modificar el mismo para reducir las repeticiones (figura 3.4).

3.2.3. .NET

- C# Refactory
  Esta es una herramienta para aplicar refactorización al entorno de programación de Visual Studio .Net como un plug-in. El cátalogo de refactorizaciones que proporciona es reducido: extract method, decompose conditional, extract variable, introduce explaining variable, extract superclass, extract interface, copy class, push up members, rename type, rename member, rename parameter, rename local variable.
  (http://www.xtreme-simplicity.net/CSharpRefactory.html)

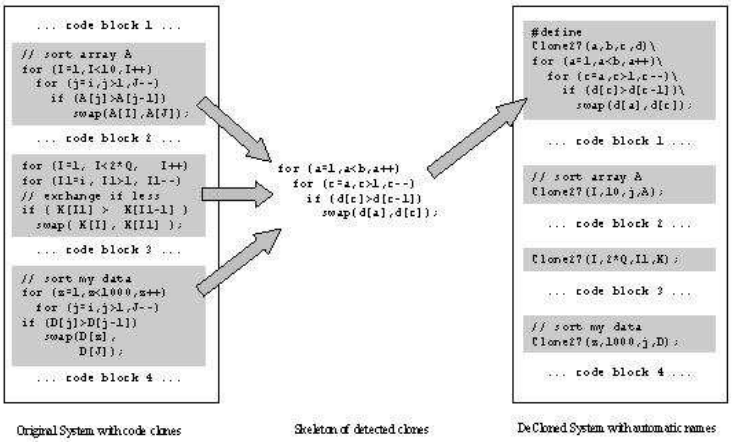

Figura 3.4: CloneDR

- Refactor! - Ver descripción en el apartado de Visual Basic
- Visual Assist X

  Esta herramienta es un plug-in para Visual Studio. Permite realizar las siguientes refactorizaciones: Rename, Extract Method, Encapsulate Field, Change Signatura, Move Implementation to Source File, Add Member, Add Similar Member, Document Method, Create Declaration, Create Implementation. (http://www.wholetomato.com/)

- JustCode!

  Herramienta que agrega más refactorizaciones a las que vienen por defecto en Visual Studio .Net. Es distribuida como plug-in. Soporta refactorizaciones para C#, Visual Basic.net y ASP.net.
  (http://www.omnicore.com/en/justcodefeatures.htm)

3.2.4. C/C++

- SlickEdit

  Editor multi-lenguaje y multi-plataforma, dentro de sus características proporciona la posibilidad de la aplicación de refactorizaciones al código fuente escrito con esta herramienta.

  (http://www.slickedit.com/)

- Ref++
  Plug-in para Visual Estudio .net que proporciona características de refactorización para C++.

- XRefactory - ver descripción en el apartado de Java

3.2.5. Visual Basic

- Refactor!

  Refactor! para Visual Basic es una herramienta en formato de plug-in, que permite a los programadores aplicar refactorizaciones a código fuente escrito para este lenguaje. Según sus creadores es capaz de aplicar 30 refactorizaciones. El producto es para Visual Basic .net no aplicable a Visual Basic 6.
  Existe una versión gratuita del producto que permite aplicar las siguientes refactorizaciones al código fuente: reorder parameters, extract method, extract property, create overload, encapsulate field, reverse conditional, simplify expression, introduce local, introduce constant, inline temp, replace temp with query, split temporary variable, move initialization to declaration, split initialization from declaration, move declaration near reference (figura 3.5).

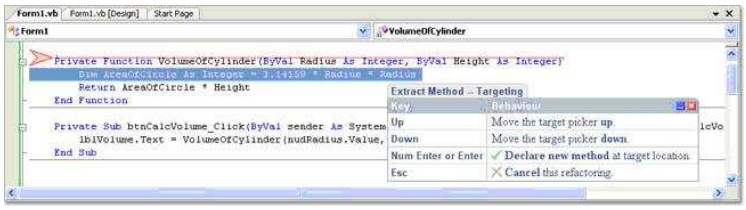

Figura 3.5: Refactor!

(http://msdn.microsoft.com/es-es/vbasic/ms789083.aspx)

- Aivosto Project Analyzer

  Esta aplicación, según sus creadores, es un revisor de código y una herramienta para analizar la calidad del código fuente. Unas de las características de la misma es que realiza un análisis de impacto antes de realizar algún cambio al código fuente de la aplicación (figura 3.6).

  (http://www.aivosto.com/project/project.html)

3.2.6. Delphi

- ModelMaker

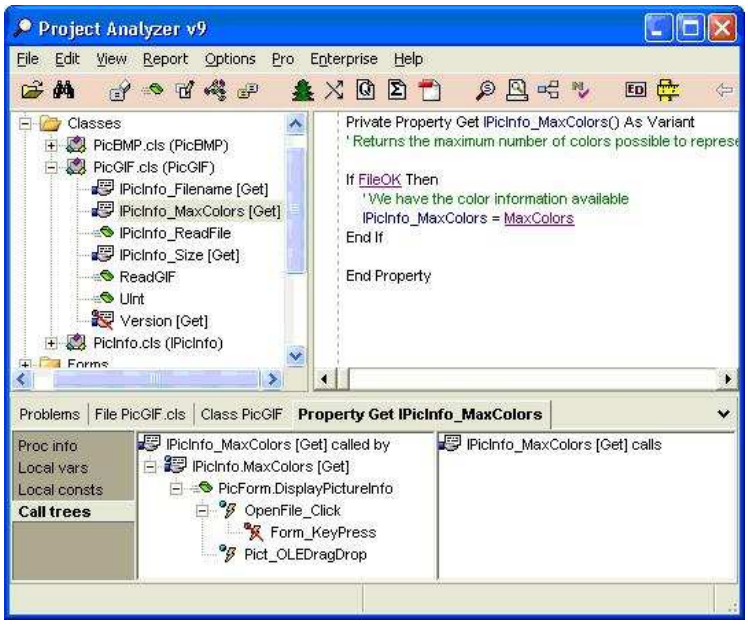

Figura 3.6: Aivosto Project Analizer

Esta herramienta permite aplicar técnicas de refactorización a programas pascal o delphi. Permite realizar refactorizaciones como por ejemplo: Extract Method, Extract Class / Interface, Rename Parameter etc.

3.2.7. Erlang

Erlang es un lenguaje diseñado por Ericsson, está orientado a la programación concurrente. En su origen fue creado para construir aplicaciones distribuidas, tolerantes a fallas, soft-real-time y de funcionamiento continuo. Está considerado como lenguaje funcional.

- Wrangler

  Es una herramienta de refactorización construida para Erlang, que permite aplicar refactorizaciones al código Erlang en forma interactiva. Aun se encuentra en su fase de prototipo. Esta herramienta no es comercial.

  (http://www.cs.kent.ac.uk/projects/forse/wrangler/doc/overview-summary.html)

3.2.8. Haskell

- Hare [33]

Es un prototipo de herramienta para refactorizar programas escritos en Haskell. Hare permite realizar refactorizaciones básicas como rename, delete, promote level, etc. La aplicación corre una interfaz gráfica amigable.

3.2.9. Resumen

Puede verse cómo en los últimos años se ha incrementado la cantidad de herramientas de desarrollo que facilitan la aplicación de refactorización de código fuente. Básicamente existen tres variantes. La primera como plug-in adosable a un entorno de desarrollo. La segunda formando parte de un entorno integrado de desarrollo (IDE). La tercera, como una aplicación standalone. Las dos primeras variantes son las más encontradas. Cabe destacar que si bien la cantidad de herramientas de refactorización son numerosas, esta técnica es muy reciente y es de esperar que las herramientas evolucionen en un número aun mayor.

3.3. Otras Aplicaciones de Refactoring

3.3.1. Refactorización de Modelos

Desde el punto de vista de la investigación académica, la refactorización de modelos es un área que aun está poco desarrollada. Existen muchas preguntas que poseen una respuesta abierta en este ámbito de investigación. Desde un ángulo pragmático no son muchas las herramientas automáticas capaces de aplicar refactorización a modelos. Uno de los problemas fundamentales es que la refactorización de modelos es un proceso que dista mucho de ser trivial. Se ha de tener en cuenta que un modelo construido consta de distintas vistas que a su vez usan distintas notaciones (diagramas de clases, diagramas de estados, diagramas de interacción, etc). Un ejercicio interesante para hacer es suponer que parte de este modelo es refactorizado. Una de las tareas a cumplir es la de mantener el modelo consistente, resincronizando todas las vistas por ejemplo, además de mantener consistente los cambios que pueden haberse producido dentro del código fuente de la aplicación [38]. Ésta es sólo una parte del complejo proceso que se requiere en la refactorización de modelos. Desde el punto de vista de la refactorización, los modelos UML son aquellos candidatos más apetecibles para ser refactorizados. Cabe destacar que existen varios trabajos realizados en esta dirección. Desde la perspectiva de los modelos de comportamiento, no es sencillo poder probar que los cambios ejercidos por una refactorización preservan el comportamiento del modelo. Existen pocos

trabajos hechos sobre refactorización de diagramas de estados, por ejemplo.

3.3.2. Refactorización y la Web

Otra área de investigación en la cual se está introduciendo la refactorización es la de las aplicaciones web. Si bien es complejo poder encuadrar la definición de refactorización dentro de este modelo de aplicaciones, en el cual el comportamiento externo es un factor que trae controversias por ser inherente a la aplicación. Existen trabajos como [50] [52] [43] en los cuales se introduce la refactorización aplicada al modelo como herramienta de mejora de la aplicación web.

3.3.3. Refactorización de Programas Orientados a Aspectos

A medida que la refactorización como técnica para mejorar el código fuente fue evolucionando, también se desplazó del área de aplicación original a tal punto que se extendió a otras áreas de la programación orientada a objetos, como la programación orientada a aspectos. En [27] se propone un enfoque sistemático para aplicar refactoring a la programación orientada a aspectos. Asimismo se investiga el impacto de las refactorizaciones que propone Fowler [21] en los programas orientados a aspectos. En [54] además de explorar el conjunto de transformaciones propuestas por [44] se introducen nuevas refactorizaciones que aplican exclusivamente a la programación orientada a aspectos. Trabajos como el de Monteiro [40] intentan proponer un catálogo de refactorizaciones que sean exclusivamente aplicables a la programación orientada a aspectos. A continuación se detallan algunos ejemplos:

- Extend marker interface with signature.
- Generalise target type with marker interface.
- Introduce aspect protection.
- Replace intertype field with aspect map.
- Extract superaspect.
- Push down pointcut.

3.3.4. Economía de la Refactorización

Así como la refactorización se ha introducido en la vida cotidiana de los programadores ya sea a la hora de un nuevo desarrollo como en el proceso de mantenimiento del software, nace de esta situación un factor que puede resultar de importancia dentro del ciclo de vida de un producto de software: el factor económico. El factor económico de la refactorización tiene que ver con los costos de la aplicación de esta técnica y sobre el efecto de la refactorización en el costo de adicionar nueva funcionalidad al producto. Existe una línea investigativa que se interesa en determinar cómo se mide el costo de la aplicación de refactorización a un producto, o del estudio del costo de la introducción de nueva funcionalidad aplicando o no refactorización ([56] y otros).

Capítulo 4

Refactorización de Código Estructurado

Hacia fines de la década de los sesenta, la ingeniería de software, recientemente creada, ya translucía problemas dentro de sus filas. Debido a la maleabilidad del software muchas personas y organizaciones adoptaron la filosofía de "codificar y arreglar" (Code and Fix) como técnica dentro del proceso de desarrollo de software [6]. Este hecho podría considerarse como el aspecto característico que representa al desarrollo de software en la década de los sesenta y que encausó a la Ingeniería de Software en la búsqueda de métodos más formales.

4.1. La Programación Estructurada

Hacia el principio de la década de los setenta comienza a gestarse la necesidad de organizar la forma en que los programadores realizaban su trabajo. Éste movimiento tiene sus bases en la carta enviada por Dijkstra, E.W. a ACM Communications, conocida como "Goto Statement Considered Harmful" [16] en la cual el autor argumenta los motivos del por qué las instrucciones del tipo Go To deben ser abolidas de los lenguajes de alto nivel. En esa misma época Böhn-Japonini [7] publican un trabajo del cual se desprende el teorema que lleva el mismo nombre, en el cual se demuestra que los programas secuenciales pueden ser construidos sin instrucciones Go To. Los programas pueden ser construidos a partir de tres tipos de estructuras, llamadas estructuras de control: Secuenciales, Iterativas y Selectivas.

Este movimiento derivó posteriormente en el surgimiento de otros con-

ceptos como el de Cohesión y Acoplamiento reforzados por Constantine [14], conocidos como los principios de la modularidad.

Al día de la fecha existen miles de millones de líneas de código fuente de programas escritos basándose en los principios de la programación estructurada. Muchos de estros programas se encuentran principalmente escritos en COBOL, C o FORTRAN. Un informe publicado en 2003 por Gartner Group [55] muestra como aun sigue habiendo producción de nuevas aplicaciones en lenguajes de programación como COBOL, desarrollados básicamente bajo el paradigma de la programación estructurada. Unos 210.000 millones de líneas de código fuente de programas escritos en COBOL siguen siendo utilizados actualmente en ambiente de producción. Son unas 5000 millones de líneas de código COBOL escritos anualmente de aplicaciones totalmente nuevas.

4.2. Lenguajes de Programación

La programación estructurada puede ser vista como un subconjunto o una subdisciplina de la programación imperativa. Si bien existen muchos lenguajes de programación en los cuales se pueden utilizar los preceptos de la programación estructurada, tres de ellos son por sus características los más destacables.

4.2.1. C

Éste es considerado uno de los lenguajes más importantes dentro de la historia de los lenguajes de programación. C es un lenguaje de programación de propósito general desarrollado entre 1969 y 1973 por Dennis Ritchie, investigador de los Laboratorios Bell, para el sistema operativo Unix.

Es el lenguaje que comúnmente se utiliza en la programación de drivers o componentes del núcleo de sistemas operativos de la familia Unix, a tal punto que su evolución está estrechamente ligada a este lenguaje. C es un sucesor del lenguaje B desarrollado por Ken Thompson en 1973, el cual a su vez deriva de BCPL. Como la mayoría de los lenguajes imperativos, siguiendo la tradición de ALGOL, C fue diseñado para ser compilado usando un compilador relativamente sencillo, para tener acceso de bajo nivel a la memoria y para que el set de instrucciones mapeen muy fácilmente a instrucciones de código máquina. Éstas son algunas de las características básicas de C cuyo

diseño fue orientado para ser un lenguaje eficiente, con un núcleo relativamente pequeño que además estaba provisto de bibliotecas que manejaban archivos y cálculos matemáticos.

En 1978 se publicó "The C Programming Language " que fue adoptado como una especificación informal del lenguaje, hasta que en 1983 la ANSI formó un comité para la especificación del estándar del lenguaje de programación C [1]. Ese mismo estándar fue adoptado en 1990 por la ISO como ISO/SEC 9899:1990 conocida vulgarmente como C90. Muchos lenguajes de programación han tomado a C como base, el ejemplo más claro es C++ de Bjarne Stroustrup cuya especificación es de 1985.

4.2.2. FORTRAN

El 15 de octubre de 1956 fue publicado el primer borrador de la especificación de The IBM Mathematical Formula Translating System [4], este lenguaje de programación nació con un conjunto de 32 instrucciones, la mayor parte de ellas para controlar los dispositivos de entrada/salida (READ DRUM, PUNCH).
Esta primera versión de FORTRAN no estaba preparada para soportar programación de tipo procedural y sólo podían definirse arrays de 1, 2 ó 3 dimensiones de sólo dos tipos de datos: enteros (fixed point) o de punto flotante (floating point). Admitía la declaración implícita de tipos de variable, las comenzadas en i, j, k, l, m o n eran de tipo entero y las de punto flotante comenzaban con cualquier otro carácter. No admitía la conversión implícita de tipos de datos, es decir, no podían usarse variables enteras y de punto flotante en una misma expresión, si bien la conversión estaba permitida.
FORTRAN II fue la versión posterior, liberada en 1958. Entre algunas de las características nuevas que FORTRAN II poseía se encuentran:

- La inclusión de SUBROUTINE, FUNCTION y END.
- CALL y RETURN.
- COMMON.
- Tipos de datos de Doble precisión y complejos (COMPLEX).

En ese mismo año (1958) también fue liberada la versión de FORTRAN III, en la cual no se aportaron mejoras significativas al lenguaje. Ésta además poseía características que la hacía muy dependiente de la máquina donde se

compilaba; éstas fueron eliminadas en la versión de FORTRAN IV en 1961. Gracias a la gradual relevancia obtenida por este lenguaje de programación, hacia 1966 la American Starndadr Association (en la actualidad ANSI) conformó un comité que redactó la primera versión del estándar de FORTRAN conocida como FORTRAN 66. El mismo se basa principalmente en FORTRAN IV. En 1978 y debido a la popularidad adquirida por el lenguaje y a las modificaciones que introdujeron algunos fabricantes de compiladores, la ANSI se vio obligada a hacer una revisión del estándar. En abril de 1978 fue publicada una nueva versión llamada FORTRAN 77 [19]. Esta versión se convertiría en el dialecto de FORTRAN más utilizado a lo largo de unos 30 años.
Hacia 1992, con la aparición de la programación orientada a objetos, el estándar de FORTRAN se somete a una nueva revisión, que concluye con la publicación del estándar denominado FORTRAN 95 [20].
Cabe destacar que en la revisión de 1992 no se elimina ninguna característica respecto de la versión anterior (en el Appendix B.1 se indica: The list of deleted features in this standard is empty."). La última revisión del estándar es la de FORTRAN 2003 en la cual se introdujeron todas las propiedades de la programación orientada a objetos como el polimorfismo, la herencia, etc. FORTRAN es posiblemente el lenguaje con mayor bagaje histórico entre los lenguajes de programación.

4.2.3. COBOL

Podría decirse que éste es el lenguaje de programación más conocido por la personas que trabajan en ambientes de procesamiento de datos. Su nombre es una abreviatura de COmmon Business-Oriented Language, que fue construido principalmente para implementar sistemas fuertemente relacionados con el ámbito financiero, económico y administrativo. En mayo de 1959 fue creada la comisión CODASYL, conformada por el Departamento de Defensa de los Estados Unidos, por fabricantes de hardware y por usuarios.

La definición y formalización del lenguaje fue aprobada por la comisión en enero de 1960 (seis meses después de la conformación de dicha comisión). COBOL nace con el aporte de dos lenguajes de programación Flow-Matic de Grace Hopper y el IBM COMTRAN de Bob Bemer, ambos parte del proyecto.

Este lenguaje de programación se popularizó muy rápidamente, a tal punto que fue revisado entre 1961 y 1965 con el objetivo de añadirle mayor

funcionalidad. En 1968 surge la primera versión ANSI de COBOL llamada COBOL ANSI-68, con revisiones posteriores COBOL ANSI-74, COBOL ANSI-85 posiblemente la versión más utilizada de este lenguaje de programación y una última revisión en el año 2002 llamada COBOL ANSI-2002. A partir del año 2007 se está preparando una nueva revisión del lenguaje.

Según estudios realizados por la Gartner Group en el 2003 COBOL sigue siendo un lenguaje de programación con gran producción de líneas de código fuente, para ese año se calculaban unos 5000 millones de líneas de código. Una de las características entre las versiones de los estándares COBOL-74 y COBOL-85 que resultó en una de las más grandes críticas al lenguaje es la incompatibilidad que existe entre las dos versiones del estándar.

4.3. Refactorización de Código Fuente Estructurado en C

Si bien el concepto de refactorización surge para la programación orientada a objetos, la aplicación de este concepto fue migrando a otros paradigmas de programación como el estructurado. En el año 2000 se publica la tesis "Software refactoring applied to C programming language" [23] en la cual es publicado el primer catálogo de Refactorizaciones para un lenguaje de programación utilizado en el paradigma de la programación estructurada como el lenguaje C. Este trabajo es uno de los pioneros en la aplicación de técnicas de refactoring a código estructurado. Además de presentar la primera herramienta automatizada para refactorizar código estructurado. El catálogo aquí presentado es de suma importancia pues es una guía de referencia para los programadores de dicho lenguaje [23]:

1. Adding a Program Entity:
   - **a** Add a variable
   - **b** Add a parameter to a function
   - **c** Add a typedef definition encapsulating an existing type
   - **d** Add a field to a structure
   - **e** Add a pointer to a variable
2. Deleting a Program Entity
   - **a** Delete unused variable

b Delete unused parameter

c Delete a function

3. Changing a Program Entity:

   a Rename variable

   b Rename constant

   c Rename user-defined type

   d Rename structure field

   e Rename function

   f Replace the type of a program entity

   g Contract variable scope

   h Extend variable scope

   i Replace value with constant

   j Replace expression with variable

   k Convert variable to pointer

   l Convert pointer to direct variable access

   m Convert global variable into parameter

   n Reorder function arguments

4. Complex refactorings:

   a Group a set of variables in a new structure.

   b Extract function

   c Inline function

   d Consolidate conditional expression

   e For into while

   f While into for

   g While into do while

A continuación se presenta un ejemplo extraído de [23] en el cual se puede observar la aplicación de varias refactorizaciones propuestas en el catálogo:

```
#define XMAX 100
#define YMAX 100
#define TRUE 1
#define FALSE 0
#include <stdio.h>
typedef struct {
   float x;
   float y;
} Point;

typedef struct {
   Point pt1;
   Point pt2;
} Rectangle;

Point middle;

Point make_pt(float x, float y) {
   Point temp;
   temp.x = x;
   temp.y = y;
   return temp;
}

char less_than(Point pt1, Point pt2) {
   if (pt1.x < pt2.x) {
      return TRUE; }
   else if (pt1.x == pt2.x && pt1.y <= pt2.y) {
      return TRUE; }
   else
   return FALSE;
}

Point middle_point(Rectangle *r) {
   Point m;
   m = make_pt((r->pt1.x + r->pt2.x)/2,
   (r->pt1.y + r->pt2.y)/2);
   return m;
}

void main() {
```

```
   Rectangle sc;
   float x1, x2;
   scanf("Enter upper left x coordinate: %f", x1);
   scanf("Enter upper left y coordinate: %f", x2);
   sc.pt1 = make_pt(x1, x2);
   if (x1 < (XMAX-10) && x2 < (YMAX-10)) {
      float x1, x2;
      scanf("Enter lower right x coordinate: %f", x1);
      scanf("Enter lower right y coordinate: %f", x2);
      sc.pt2 = make_pt(x1, x2);
   }
   else
      sc.pt2 = make_pt(XMAX, YMAX);
   middle = middle_point(&sc);
}
```

Después de aplicar:

1. Rename structure-field al campo x de la estructura Point renombrándolo como xcoor.

2. Rename structure-field al campo y de la estructura Point renombrándolo por ycoor.

3. Se renombró pt1 de la estructura Rectangle aplicando Rename structure-field por el nombre upperleft.

4. Rename structure-field al campo pt2 de la estructura Rectangle renombrándolo por lowerright.

Se obtiene el código fuente refactorizado:

```
#define XMAX 100
#define YMAX 100
#define TRUE 1
#define FALSE 0
#include <stdio.h>
typedef struct {
   float x_coord;
   float y_coord;
} Point;
typedef struct {
   Point upper_left;
```

```
   Point lower_right;
} Rectangle;
Point middle;

Point make_pt(float x, float y) {
   Point temp;
   temp.x_coord = x;
   temp.y_coord = y;
   return temp;
}

char less_than(Point pt1, Point pt2) {
   if (pt1.x_coord < pt2.x_coord) {
   return TRUE;
   }
   else if (pt1.x_coord == pt2.x_coord && pt1.y_coord <= pt2.y_coord) {
      return TRUE;
   }
   else return FALSE;
}

Point middle_point( Rectangle *r) {
   Point m;
   m = make_pt((r->upper_left.x_coord + r->lower_right.x_coord) / 2,
   (r->upper_left.y_coord + r->lower_right.y_coord) / 2);
   return m;
}
void main() {
   Rectangle sc;
   float x1, x2;
   scanf("Enter upper left x coordinate: %f", x1);
   scanf("Enter upper left y coordinate: %f", x2);
   sc.upper_left = make_pt(x1, x2);
   if (x1 < (XMAX - 10) && x2 < (YMAX - 10)) {
      float x1, x2;
      scanf("Enter lower right x coordinate: %f", x1);
      scanf("Enter lower right y coordinate: %f", x2);
      sc.lower_right = make_pt(x1, x2);
   }
   else sc.lower_right = make_pt(XMAX, YMAX);
   middle = middle_point(&sc);
```

```
}
```

A simple vista puede observarse cómo la transformación producida mejora la comprensibilidad y la claridad del código fuente C, cumpliendo esta transformación con la definición de refactorización provista por Fowler [21].

En la actualidad son muy pocos los entornos de desarrollo integrados destinados a la construcción de software para el lenguaje C de programación que no posean capacidades de refactorización para este lenguaje. Un ejemplo de entorno de programación integrado (IDE) utilizado para desarrollar en C que provee capacidades de refactorización para el lenguaje C es Eclipse. Esta herramienta de desarrollo proporciona una serie de refactorizaciones que están enumeradas en el catálogo de Garrido [23] como por ejemplo Extract function, Replace value with constant y Rename en todas sus versiones.

El trabajo de Garrido [23], además de ser el primero en presentar un catálogo de refactorización para un lenguaje de programación procedural, sienta las bases para la aplicación del concepto en refactorización al paradigma de la programación estructurada, sirviendo éste como línea de partida para la construcción, medición y comparación de herramientas de refactorización para el lenguaje de programación C y para otros lenguajes procedurales.

Otros trabajos han sido publicados sobre refactorización de código fuente escrito para el lenguaje de programación C, algunos de ellos dirigidos a la semi-automatización del proceso de refactorización [48] y otros orientados a la refactorización en C con directivas de preprocesamiento [24] [25] [37].

4.4. Refactorización de Código Fuente Estructurado en COBOL

Pese a ser uno de los lenguajes más populares y sobre los cuales la producción de código fuente en la actualidad es aun importante, no se ha podido encontrar un catálogo sobre refactorización de código fuente COBOL.

4.5. Refactorización de Código Fuente Estructurado en FORTRAN

A pesar de ser un lenguaje de programación con casi unos 50 años de vida con cientos de miles de líneas de código escritas, y además uno de los lenguajes más utilizados para cómputo paralelo y de altas prestaciones [46] y uno de los lenguajes más evolucionados desde la fecha de su creación hasta el día de hoy, FORTRAN cuenta con un muy escaso desarrollo de herramientas y técnicas de refactorización.

Al día de la fecha, FORTRAN, no posee un catálogo asentado de refactorizaciones, a pesar de que posee una característica particular en la especificación de los distintos estándares producidos a lo largo de toda su historia. Todos los estándares revisados desde la versión de FORTRAN 77 en adelante son compatibles entre sí hacia atrás: es posible escribir código fuente orientado a objetos para el estándar de FORTRAN 2003 en el formato de tarjetas de 80 columnas (si bien esto no es del todo recomendable, es posible [46]).
Esta compatibilidad "backward" hace que pueda escribirse el mismo código FORTRAN de diversas formas, un ejemplo muy ilustrativo de esto son los ciclos DO-LOOP.
En la versión de FORTRAN 90/95 es posible escribir [59]:

```
      DO 100 J=1,10              DO J=1,10
      A(J)=A(J)+ B(J)              A(J)=A(J)+ B(J)
100   CONTINUE                   END DO
```

Éstas son dos posibles formas de escribir el mismo ciclo Do-Loop en FORTRAN 90/95, la primera forma pertenece al estándar de FORTRAN 77 [19] y la segunda la versión corregida o revisada de FORTRAN 90/95 (ambas válidas para este último). Por ende, en un programa escrito en FORTRAN 90 [20] podemos encontrar la misma estructura de control escrita en dos formatos distintos, esto si bien permite la compatibilidad de versionado entre los distintos estándares del lenguaje, por otro lado hace más compleja la comprensión y la claridad del código fuente.

Una característica que poseen las dos versiones de F77 y F90 en los loops son los llamados Shared Do loops Terminations, en este caso se recomienda eliminarlos por las formas más estructuradas de escribir, que es con los correspondientes END-DO [58]:

```
      DO 101 k=1,l
      DO 101 i = 1,imax
      X(i,k)=temp(i,k)-h25e2
      Y(I,k)=x(i,k) * x(I,k)
  101 CONTINUE
```

La forma propuesta de transformación de este fragmento de código a una versión mucho más clara y estructurada es la siguiente:

```
DO  k=1,l
   DO  i = 1,imax
      X(i,k)=temp(i,k)-h25e2
      Y(I,k)=x(i,k) * x(I,k)
   END DO
END DO
```

Nuevamente la forma propuesta provee mucha más claridad y comprensibilidad al código fuente que la original, siendo ambas formas totalmente amparadas por el estándar de FORTRAN 90.

En el trabajo realizado por Tinetti [58] se citan algunas otras mejoras a implementarse en la estructura interna del código fuente de una aplicación FORTRAN, en este caso específicamente para un sistema de predicción climático [57], donde todas cumplen con la definición de refactorización de Fowler [21]; se pueden encontrar algunas de ellas también propuestas por Ralph Johnson en [47]. En su trabajo, Tinetti [58] se propone mejorar el código fuente de un sistema legacy construido totalmente en FORTRAN, cuyo código está escrito para FORTRAN 77 con la intención de mejorar su estructura interna mediante la migración del programa al estándar de FORTRAN90.

Otro caso interesante de estudio sobre las características del código fuente de FORTRAN lo encontramos en los operadores lógicos. A partir de la versión de FORTRAN 90 [20] es posible escribir de dos formas los operadores lógicos:

```
F77: .GE. .GT. .EQ. .NE. .LE. .LT.
F90: >=   > =   =   /=    <=   <
```

A continuación se muestran distintas formas en las cuales se puede encontrar código fuente FORTRAN escrito:

```
IF (X .GT. 0) THEN
   IF (X .GE. Y) THEN
      WRITE(*,*) 'X ES POSITIVO Y X >= Y'
   ELSE
      WRITE(*,*) 'X ES POSITIVO PERO, X < Y'
   ENDIF
ELSEIF (X .LT. 0) THEN
   WRITE(*,*) 'X ES NEGATIVO'
ELSE
   WRITE(*,*) 'X ES CERO'
ENDIF
```

Código fuente escrito en el formato F77 [19]

```
IF (X > 0) THEN
  IF (X >=0 Y) THEN
      WRITE(*,*) 'X ES POSITIVO Y X >= Y'
     ELSE
        WRITE(*,*) 'X ES POSITIVO PERO, X < Y'
     ENDIF
ELSEIF (X < 0) THEN
     WRITE(*,*) 'X ES NEGATIVO'
ELSE
     WRITE(*,*) 'X ES CERO'
ENDIF
```

Código fuente escrito en el formato F90 [20] (ambos válidos para F90).

Cabe además destacar que se han hecho trabajos de grado que han demostrado que la aplicación de ciertos cambios al código fuente de programas escritos en FORTRAN mejoran, además, otras características del mismo como por ejemplo la performance. En este trabajo se llega a medir una variación en el rendimiento de ciertas rutinas de la aplicación de hasta un 22,94 %, cambiando el código fuente escrito a la notación matricial donde podía realizarse el cambio[51][58] .

Un aspecto importante que hay que destacar es la casi total ausencia de herramientas que permitan la aplicación de estas transformaciones a programas FORTRAN. Uno de los proyectos que más avances ha realizado sobre este tema es el equipo de PHOTRAN, plug-in de Eclipse para FORTRAN. Siguiendo la línea de este IDE, la herramienta proporciona un muy buen motor para realizar las refactorizaciones de código fuente (figura 4.1). En las

últimas versiones se han introducido algunas de las refactorizaciones descriptas en este apartado. Una de las razones de la importancia de este punto, se basa en las características de los programadores FORTRAN, si bien un entorno gráfico proporciona claridad, usabilidad, comprensibilidad, etc. es cierto que, además, los programadores de este lenguaje están muy acostumbrados a entornos de tipo consola, en los cuales el formato de la herramienta a utilizar cambia radicalmente.

Muchas de las transformaciones de código fuente propuestas anteriormente podrían ser aplicadas como reglas automáticas. El problema de la aplicación batch de estas reglas reside en la necesidad de que estos cambios deban ser efectuados por una persona, dado que es la persona quien tiene control sobre cuáles deberían ser las transformaciones aplicables para lograr la mejora dentro de la estructura interna. Por ello es de vital importancia que la herramienta que realiza las refactorizaciones interaccione con el programador. Este es un aspecto importante dentro de la construcción de herramientas dirigidas a refactorizar código fuente.

Por otro lado, existen muy pocos trabajos que apliquen técnicas de refactorización dentro de la programación estructurada orientados a áreas más específicas, como por ejemplo la High Performance o la Programación en Paralelo. Este es un campo dentro de las técnicas de refactorización de código estructurado en el cual no se han hecho muchos trabajos sobre el tema, un muy buen punto de partida para futuras investigaciones.

Si bien al parecer las técnicas de refactoring han tenido su auge durante la década del 2000, aun existen varias áreas de aplicabilidad en las cuales estas técnicas no han sido explotadas.

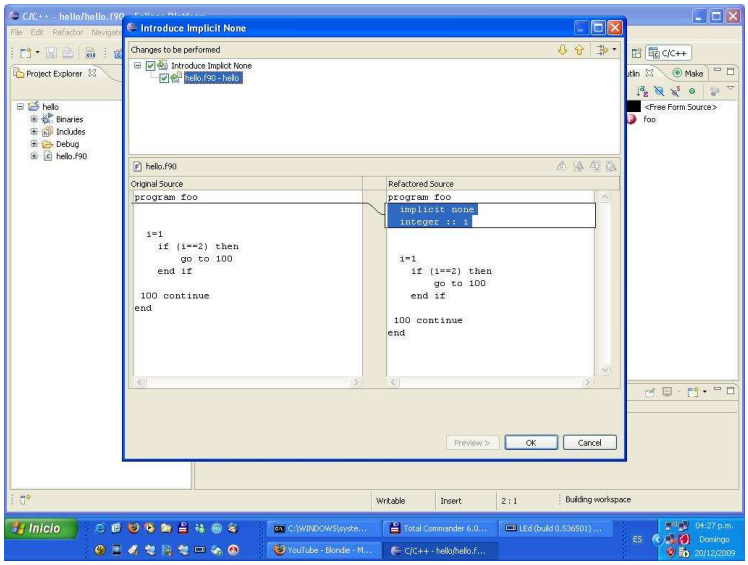

Figura 4.1: Herramienta de refactorización Photran

Capítulo 5

Conclusiones

En el presente trabajo se ha realizado un estudio actualizado del estado del arte sobre refactoring y se estudió con más detalle la aplicación de este concepto a la programación estructurada.

El software en sí mismo posee características que lo transforman en un producto muy particular, en el trabajo se revisaron estas características particulares y los motivos históricos que sentaron los precedentes para el surgimiento de técnicas como la refactorización de código fuente.

Un producto de software de calidad debería ser producido a partir de un buen diseño, seguido de un proceso ordenado de implementación. Ésta no es una práctica habitual, y nos encontramos con software construido a partir de un proceso de desarrollo muy poco ordenado. Existen incluso aplicaciones que han sido desarrolladas por personas con escasos conocimientos de técnicas de producción de software, como por ejemplo, aplicaciones desarrolladas por físicos, matemáticos, etc. que utilizaron lenguajes de programación como FORTRAN para implementar modelos numéricos de cálculo.

Se han enumerado los motivos principales por los que un sistema de software necesita apoyarse en técnicas que permitan mejorar el producto, mediante la aplicación de pequeños cambios en forma paulatina.

A partir de la definición y presentación de técnicas de refactorización se hace referencia en este trabajo a distintos catálogos de refactorizaciones que han sido elaborados para ser aplicados en distintas áreas de la ingeniería de software.

Asimismo se ha realizado un estudio de las distintas herramientas dispo-

nibles, hasta la edición de este trabajo, que proveen técnicas automatizadas para la realización de estas transformaciones.

Dentro de las aplicaciones actuales de las técnicas de refactoring el trabajo se centró en la aplicación de la misma a la programación estructurada. Dentro de esta área se observó la necesidad de ahondar aun más en el estudio y aplicación de las técnicas de refactorización en lenguajes de programación como COBOL y FORTRAN.

Además se presentan casos de estudio como el de [58] en el cual se han aplicado estas técnicas orientadas a la mejora de programas ya escritos con el objetivo de poder paralelizarlos.

Otro hecho que se desprende de este estudio es la necesidad de profundizar en el área de las herramientas automatizadas específicas para código fuente estructurado. En la actualidad se cuenta con un número muy reducido de herramientas que aplique estas técnicas.

En resumen, la refactorización de código estructurado se encuentra aun en desarrollo y esconde un gran potencial que necesita ser explotado.

5.1. Futuros Trabajos

Si bien existen trabajos realizados sobre las técnicas de refactorización para código estructurado como el de [23] existen lenguajes de programación en los cuales estas técnicas no han sido explotadas, como FORTRAN Y COBOL.

Más precisamente en FORTRAN es necesario determinar un catálogo que sirva como referencia a los programadores sobre las posibles transformaciones al que el lenguaje escrito en este código fuente puede ser sometido.

Además es necesario contribuir a la construcción de una herramienta para FORTRAN que permita aplicar las refactorizaciones a los sistemas de software ya escritos. Dicha herramienta debe convertirse en un apoyo en el momento de la aplicación de estas transformaciones, que la mayoría de las veces no pueden realizarse en forma manual.

Es necesario continuar investigando sobre las posibles incumbencias de la refactorización y de los límites de aplicabilidad de la misma dentro del cam-

po de la ingeniería de software. Asimismo es importante determinar cuáles son los límites dentro de los que una serie de transformaciones pueden ser llamadas refactorizaciones o no. Además es necesario formalizar los conceptos dentro de la definición de refactoring donde aun hoy existen opiniones controversiales de cuándo una determinada transformación es o no una refactorización.

Bibliografía

[1] A. ANSI. Standard X3. 159-1989, Programming Language C, ANSI. *Inc., NY*, 1989.

[2] RS Arnold. Software restructuring. *Proceedings of the IEEE*, 77(4):607–617, 1989.

[3] E. Ashcroft and Z. Manna. The translation of'goto'programs to'while'programs. *Information Processing*, 71:250–255, 1972.

[4] J. Backus. The history of Fortran I, II, and III. *ACM SIGPLAN Notices*, 13(8):165–180, 1978.

[5] Brenda S. Baker. An algorithm for structuring flowgraphs. *J. ACM*, 24(1):98–120, 1977.

[6] B. Boehm. A view of 20th and 21st century software engineering. In *Proceedings of the 28th international conference on Software engineering*, page 29. ACM, 2006.

[7] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, 1966.

[8] F.P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE computer*, 20(4):10–19, 1987.

[9] M. Broy and E. Denert. *Software pioneers: contributions to software engineering*. Springer Verlag, 2002.

[10] E Bush. The automatic restructuring of cobol. In *The Institute of Electrical and Electronics Engineers, Inc on Conference on software maintenance–1985*, pages 35–41, Piscataway, NJ, USA, 1985. IEEE Press.

[11] R. Canning. Rejuvenate your old systems. *EDP Analyzer*, 22(3):1–16, 1984.

[12] N. Chapin, J.E. Hale, K.M. Khan, J.F. Ramil, and W.G. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1), 2001.

[13] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.

[14] LL Constantine. Programming Profession, Programming Theory, and Programming Education. *Computer and Automation,*, 17(2):14–19, 1968.

[15] R. de Neufville. The baggage system at Denver: prospects and lessons. *Journal of Air Transport Management*, 1(4):229–236, 1994.

[16] E.W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[17] J.M. Favre. Languages evolve too! changing the software time scale. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE), Washington, DC, USA, IEEE Computer Society*, pages 33–44, 2005.

[18] B. Foote and J. Yoder. Big ball of mud. *Pattern languages of program design*, 4(654-692):99, 2000.

[19] A. FORTRAN. X3. 9-1978. *American National Standards Institute, New York*, 1978.

[20] A. FORTRAN. X3.198-1992. *American National Standards Institute, New York*, 1992.

[21] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[22] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Q.R.H. Montreal, and I.R.J. Urbana. Back DESIGN PATTERNS-Elements of Reusable Object-Oriented Software.

[23] A. Garrido. *Software refactoring applied to C programming language*. PhD thesis, University of Illinois, 2000.

[24] A. Garrido and R. Johnson. Refactoring C with conditional compilation. In *18th IEEE Int. Conf. on Automated Software Engineering*, 2003.

[25] A. Garrido and R. Johnson. Program refactoring in the presence of preprocessor directives. *University of Illinois at Urbana-Champaign, Champaign, IL*, 2005.

[26] S. Hussain, M.Z. Asghar, B. Ahmad, S. Ahmad, C.A. Furia, B. Meyer, M. Dalmau, P. Roose, S. Laplace, S. Bouzefrane, et al. A Step towards Software Corrective Maintenance Using RCM model. *USA*, 2009.

[27] M. Iwamoto and J. Zhao. Refactoring aspect-oriented programs. In *4th AOSD Modeling With UML Workshop, UML*, 2003.

[28] J. Kerievsky. *Refactoring to patterns*. Pearson Education, 2005.

[29] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. Computing Mcgraw-Hill, January 1978.

[30] R. G. Lanergan and C. A. Grasso. Software engineering with reusable designs and code. pages 187–195, 1989.

[31] M.M. Lehman et al. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[32] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Turski. Metrics and laws of software evolution-the nineties view. In *4th International Software Metrics Symposium (METRICS'97)*, 1997.

[33] H. Li, C. Reinke, and S. Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 27–38. ACM New York, NY, USA, 2003.

[34] RC LINGER. MILLS˜ HD A case study m Cleanroom software engineering: The IBM˜ Cobol restructuring facility. *Proceedtngs of COMPSAC*, 88.

[35] R.C. Linger, B.I. Witt, and HD Mills. *Structured Programming; Theory and Practice the Systems Programming Series*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1979.

[36] Michael J. Lyons. Salvaging your software asset: (tools based maintenance). In *AFIPS '81: Proceedings of the May 4-7, 1981, national computer conference*, pages 337–341, New York, NY, USA, 1981. ACM.

[37] Bill McCloskey and Eric Brewer. Astec: a new approach to refactoring c. *SIGSOFT Softw. Eng. Notes*, 30(5):21–30, 2005.

[38] T. Mens, G. Taentzer, and D. Muller. Challenges in model refactoring. In *Proc. 1st Workshop on Refactoring Tools, University of Berlin*, volume 98, 2007.

[39] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[40] M.P. Monteiro and J.M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the 4th international conference on Aspect-oriented software development*, page 122. ACM, 2005.

[41] HW Morgan. Evolution of a software maintenance tool. In *Proceedings of the 2nd Natwnal Conference EDP Software Maintenance*, pages 268–278, 1984.

[42] M.C. Ohlsson, A. Von Mayrhauser, B. McGuire, and C. Wohlin. Code decay analysis of legacy software through successive releases. In *Proc. IEEE Aerospace Conf*, pages 69–81. Citeseer, 1999.

[43] L. Olsina, A. Garrido, G. Rossi, D. Distante, and G. Canfora. Web Application evaluation and refactoring: A Quality-Oriented improvement approach. *Journal of Web Engineering*, 7(4):258–280, 2008.

[44] W.F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, Citeseer, 1992.

[45] W.F. Opdyke and R.E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, 1990.

[46] Jeffrey Overbey, Spiros Xanthos, Ralph Johnson, and Brian Foote. Refactorings for fortran and high-performance computing. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 37–39, New York, NY, USA, 2005. ACM.

[47] J.L. Overbey, S. Negara, and R.E. Johnson. Refactoring and the evolution of Fortran. *Urbana*, 51:61801.

[48] J. Peng. Semi-Automated Refactoring Applied to the C Programming Language.

[49] D. Pigott. An interactive historical roster of computer languages. *hopl. murdoch. edu. au, last visited: March*, 2005.

[50] Y. Ping and K. Kontogiannis. Refactoring web sites to the controller-centric architecture. In *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR'04)*, volume 1534, pages 20–00. Citeseer.

[51] Diego Luis Rodrigues. Optimizacin de Software Mediante BLAS Aplicado a un Modelo Climtico. 2008.

[52] G. Rossi, M. Urbieta, J. Ginzburg, D. Distante, and A. Garrido. Refactoring to Rich Internet Applications. A Model-Driven Approach. In *Proceedings of the Eighth International Conference of Web Engineering, ICWE*, 2008.

[53] D. Rubinstein. Standish group report: Theres less development chaos today. *Software Development Times*, 1, 2007.

[54] S. Rura. *Refactoring aspect-oriented software*. PhD thesis, WILLIAMS COLLEGE, 2003.

[55] G. Shiffler, C. Smulders, JN Correia, JK Hale, and W. Hahn. Gartner dataquest market databook. *Gartner Dataquest Inc*, 2003.

[56] Q.D. Soetens. Refactoring Economics.

[57] Fernando G. Tinetti, Pedro G. Cajaraville, Juan C. Labraga, and Mónica A. López. Ingeniería Inversa Aplicada a Software Numérico: Modelos Climáticos. *XI Workshop de Investigadores en Ciencias de la Computacin 2009*, 51:223–227.

[58] Fernando G. Tinetti, Mónica A. Lopez, and Pedro G. Cajaraville. Fortran Legacy Code Performance Optimization: Sequential and Parallel Processing with OpenMP . *World Congress on Computer Science and Information Engineering*, pages 471–475, 2009.

[59] U.C. UNPSJB. Fortran Legacy Code Performance Optimization: Sequential and Parallel Processing with OpenMP.

[60] Edward Yourdon. *Techniques of Program Structure and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986.

Índice de figuras