REFACTORING: SECUENCIA DE APLICACIÓN

[1] Ing. Pablo Vilaboa; [2]Dra. Alejandra Garrido;
[1] CAETI, Facultad de Tecnología Informática, UAI; [2] LIFIA, Fac. Cs. Exactas, UNLP;
[1] Pablo.vilaboa@uai.edu.ar; [2] garrido@sol.info.unlp.edu.ar;

RESUMEN

Este trabajo intenta demostrar la posibilidad de diseñar un orden lógico o una metodología coherente que consienta la teoría para formalizar algún criterio práctico o racional que determine un orden en la ejecución de refactoring definido en un catálogo. Dicha secuencia debe asegurar transformaciones sucesivas respetando cambios legibles y mantenibles que aseguren los niveles de comprensión de modo tal que el responsable de transformar el código legacy pueda lograr un resultado optimo al culminar el proceso.

Palabras Claves: Refactoring, Fortran, Secuencialidad, legibilidad, Mantenimiento

CONTEXTO

El trabajo descripto se encuentra enmarcado en la aplicación de un enfoque sistémico, disciplinado y cuantificable del mantenimiento del software subscripta a la línea de investigación de Algoritmos y software del centro de altos estudios en tecnología informática.

INTRODUCCIÓN

La claridad del código fuente está íntimamente vinculado con la comprensión, el grado de claridad es directamente proporcional a la distancia que existe entre el desarrollador del código y quien lo interpreta. Por lo tanto la evaluación que pudiese hacerse sobre la claridad del código tendría un alto grado de susceptibilidad, y es con el uso de refactoring que se puede disminuir la brecha de susceptibilidad. Todos los profesionales que se desempeñan en el área de sistemas han experimentado horas de frustración al heredar soluciones informáticas. El mantenimiento y la escalabilidad del producto logran involucrar miles de horas hombre para alcanzar muchas veces soluciones inútiles. En la actualidad los sistemas por diferentes motivos se vuelven ilegibles y difíciles de mantener, sobre todo cuando los desarrolladores son algo indiferentes a las arquitecturas de diseño, buenas prácticas y criterios adecuados de programación. El código producidos en un ambiente poco estandarizado se convierte en código inmanejable (Foote & Yoder, 2000)

Foote en su trabajo titulado "*Big ball of Mud*" expone que las empresas deben conocer las causas que comprometen la ocurrencia de sistemas enmarañados y buscar los caminos alternativos para solucionarlos. El autor trata de encontrar una respuesta a la realidad del por qué buenos programadores terminan creando programas ilegibles, y es allí donde recae la necesidad de encontrar esquemas que permitan recuperar el código fuente de los sistemas de información de la empresa.

El código heredado, es algo que posiblemente haya pasado de moda, pero funciona y la organización no desea desprenderse de él. El software legado no necesariamente está íntimamente vinculado a un lenguaje obsoleto, seguramente un sistema de algunos pocos años haya pasado por tanto programadores que puede convertirse en un legacy mucho más pronto de lo que uno pude imaginarse (Demeyer, Ducasse, & Nierstrasz, 2009) Según lo planteado por Foote en su trabajo *Big ball of mud,* se pueden clasificar algunas razones por las cuales los sistemas Legacy se convierten en una enorme bola de barro. Dichas razones no son triviales y enumeran

los pecados capitales en los cuales incurren generalmente las empresas desarrolladoras de software.

Históricamente los desarrolladores han madurado junto a cientos de metodologías en el desarrollo de software, las modas fueron cambiando y los protagonistas que forman parte del grupo de trabajo también. Esto lleva a largas discusiones sobre el cumplimiento real de los requerimientos, para asegurar que el producto final es exactamente lo que el cliente desea.

La generación de productos bajo una demanda exigida fomenta la creación de código rápido que luego debería ser desechado, en general si funciona no se toca y rara vez es documentado. El código rápido convive con el resto de la aplicación arrastrándose a lo largo del tiempo y sufriendo adaptaciones de todo tipo. La acumulación de código rápido perjudica la claridad y disminuye la posibilidad de escalar la aplicación y su mantenimiento.

Los sistemas miden su complejidad en función de diversos factores: como la cantidad de áreas que abarcan, su tamaño y la cantidad de personas que participan. Los sistemas de información complejos promueven un alto impacto por la cantidad de recursos necesarios que son utilizados, exigiendo una coordinación adecuada entre todas las tareas. (Laudon & Laudon, 2000). Las empresas para mejorar su nivel competitivo utilizan los recursos en otros países con mejores ofertas económicas (Porter, 1991). Estos países tienen un alto grado de rotación en el personal, y sumado a la falta de documentación se pierde un contexto de interrelaciones que constituyen la base de conocimiento de la solución.

Comúnmente el uso de la programación orientada a objetos permite el aporte de uno de sus pilares fundamentales, la encapsulación. El uso de encapsulación permite ocultar la implantación desarrollada en el interior a los ojos observadores del exterior. El uso de fachadas colabora en el ordenamiento de una solución basada en capas eliminando las dependencias complejas (Gamma, Helm, & Johnson, c2003). A pesar de las ventajas que promueve el uso de patrones, es utilizado también para ocultar el código desprolijo.

El ciclo de vida de los sistemas de información impone cambios naturales en la evolución del software. (Lehman, 2000) Propone un conjunto de leyes que fueron identificadas en el estudio de sistemas entre 1968 y 1985. Los mecanismos que aseguran el crecimiento y madures del software son con el tiempo justificantes en el aumento de complejidad de los legacy.

En base a que los sistemas tienden a evolucionar según las leyes de Lehman, el software, en su mayoría, no respetan las buenas prácticas aplicando técnicas que favorecen los riesgos de aumentar el mal olor en el código fuente (Mendez, Overbey, & Garrido, 2010). Lo anteriormente mencionado identifica varias de las prácticas que, a lo largo del tiempo, promueven las razones por las cuales el código heredado puede ser difícil de mantener. La falta de buenas prácticas en el ciclo de vida de un sistema de información convierte el código fuente en una malla compleja de interrelaciones que lo vuelven difícil de mantener y que junto a la falta de testeos de unidad catapultan los modelos de refactoring como soluciones indispensables.

Con el afán de asegurar aplicaciones legibles, varios autores (Larman, c2003) (Gamma, Helm, & Johnson, c2003) (Kerievsky, 2009) (Fowler, Rice, & Foemmel, 2010) han trabajado sobre la confección de un catalogo de patrones que predominan en la comunidad con el fin de mejorar el código de los sistemas. Sin embargo, los criterios que mueven el uso de patrones no siempre son adecuados, el uso indiscriminado de patrones no siempre es considerado como buenas prácticas (Poveda Villalon, Suárez-Figueroa, & Suárez-Figueroa, 2009) y sumado al hecho que las

aplicaciones con cierto grado de madurez no lo aplican desde sus origines transforma las aplicaciones Fortran en un verdadero problema de mantenibilidad. Los cambios cognitivos y las diferentes habilidades que pueden tener los programadores se vuelve un multiplicador del problema. A pesar que estos programadores basen su andamiaje de conocimiento en los tres conceptos básicos finito, preciso y definido (Joyanes Aguilar, 1998) no asegura un código legible sin la auditoría del trabajo que realizan. Es aquí donde aparecen los refactoring, los mismos permiten asegurar transformaciones en el código sin modificar los resultados obtenidos en dicho código (Overbey, Negara, & Johnson, 2009).

Luego de varios años de evolución, Fortran se ha convertido en un lenguaje de programación aplicable en disciplinas como meteorología, física y matemáticas. Las aplicaciones desarrolladas en este lenguaje han evolucionado variando sus requerimientos y ajustándolos a las necesidades actuales. Fortan a evolucionado en los últimos años asegurando la convivencia de todas las versiones del lenguaje, cada fragmento de código es un modulo complejo que depende del momento que fue programado y la versión del lenguaje que se utilizó. La magnitud de las tareas de mantenimiento se incrementan por la evolución de las versiones del lenguaje y por los ajustes de los requerimientos funcionales convirtiendo las actividades de mantenimiento en un verdadero reto (Mendez, Overbey, & Garrido, 2010). Por esto último, La legibilidad pasó a ser considerada un atributo importante de las piezas de programación.

La preocupación por transformar la mantenibilidad, legibilidad o el riesgo de un sistema de información en valores cuantificables no es una tarea sencilla debido a la compleja interoperabilidad entre diferentes elementos poco similares que conviven con el código fuente como estilos de programación, cambios en la conceptualización de las buenas prácticas de moda, evolución en las versiones de la semántica del lenguaje o la discontinuidad u obsolescencia del conjunto de instrucciones. Estas relaciones complejas se reflejan en los estilos de programación creando zonas de conflicto al momento de escalar o mantener los servicios de la aplicación. El uso de métricas es un camino factible para alcanzar el objetivo de cuantificar la secuencia de aplicación de Refactoring.

La secuencialidad puede expresarse con la combinación de dos factores, la legibilidad y la mentenibildiad. Raymond expone que la legibilidad está sujeta a diversas características descriptivas, tan relativas como el propósito del software, la lógica de programación y la documentación que acompaña al código. Uno de las consideraciones del autor es analizar la legibilidad en pequeños fragmentos de código, algo que en nuestro modo de entender se aplicaría perfectamente a los ejemplos de Refactorinzación. (Raymond & Westley, 2008). El estudio de campo de estudiantes de ingeniería centrado en el comportamiento compulsivo al programar y las interpretaciones intelectuales de la semántica facilitan el hallazgo de estadísticas de legibilidad.

La IEEE define mantenibiliad como La facilidad con la que un sistema o componente de software puede ser modificado para corregir fallos, mejorar su funcionamiento u otros atributos o adaptarse a cambios en el entorno (Institute of electrical and Electronics Engineers, 1990). Considerar mantenible un fragmento del código fuente asegura la factibilidad de cambiar el código despues de concluir su implementación. El mantenimiento del código esta sujeto según Sommerville en 3 tipos claramente diferenciados 1) por fallas, 2) Por Adaptaciones, 3) Por nuevas funcionalidades (Sommerville, 2002).

A diferencia de la subceptibidliad relacionada a la legibildiad, la mantenibilidad puede calcularse en función del índice de mantenibilidad expresado como una conjunción de varias métricas (Oman & Hagemeister, 1992) (Kaur & Singh, 2011) (Aldekoa, Trujillo, Sagardui, & Diaz, 2006) (Samoladas, Stamelos, Lefteris, & Apostolos, 2004).

$$MI = 171 - 5{,}2\ln(avgV) - 0{,}23\ avgV(g) - 16{,}2$$

En donde:

avgV: es el Volumen promedio por modulo de Halstead.

avgV(g): es el promedio extendido por modulo del cálculo de complejidad ciclomática

avgLOC: es el promedio de la cantidad de líneas de código por modulo

PerCM: es el promedio del porcentaje por modulo de líneas para comentarios.

Originalmente el cálculo de MI fue definido sin considerar las líneas de comentarios, pues en función de su participación en la ecuación el valor es insignificante. Con lo cual el índice de mantenibilidad podría calcularse como:

$$MI = 171 - 5{,}2\ln(avgV) - 0{,}23\ avgV(g) - 16{,}2\ln(avgLOC)$$

Esta métrica analizada por los ingenieros de HP, pretende ser un índice porcentual que indica la complejidad de mantener el código de un modo general, con el fin de poder localizar de manera factible los posibles problemas. El valor obtenido por las formular de esfuerzo de Halstead brinda los valores posibles para determinar el índice de mantenibilidad, sin embargo el Cálculo del índice se complementa con los resultados de una combinación de métricas. (Coleman, Ash, Lowther, & Oman, 1994).

Líneas de investigación y desarrollo

Este proyecto tiene como objetivo principal encontrar una vinculación entre la legibilidad y la mantenibilidad para determinar secuencialidad en la aplicación de Refactoring.

Nuestro estudio elabora un sentido para la construcción de un algoritmo que permita esta estimación basado en la generación de criterios que asocien la legibilidad y la mantenibildiad de una pieza de código apoyado en estadísticas de legibilidad y métricas precisas que expresen numéricamente el índice de mantenibilidad.

Buscamos focalizar los requerimientos para reducir la incertidumbre que promueve la necesidad de participación humana en la toma de decisiones de modo tal que la secuencia de aplicación de Refactoring en ciclos iterativos asegure una continuidad en los valores de legibilidad y mantenibildiad desde el inicio del proceso hasta su culminación.

Resultados y Objetivos

Durante este proyecto se trabajó en 1) selección de un catálogo de refactoring para Fortran, 2) Formalizar un estudio comparativo de las estadísticas de Legibilidad y las métricas de mantenibilidad para establecer una secuencia de aplicación.

Este análisis dio lugar a la confección de una lista de refactring ordenada en función de los objetivos planteados. El estudio comparativo permite disminuir la brecha de incertidumbre en la implementación de un criterio para aplicar los Refactoring en una solución legacy.

Actualmente, sólo con el índice de mantenibildiad es posible ordenar los refactoring, la experiencia de evaluar conceptos y compartimientos referidos a la legibilidad puede mejorar el orden de aplicación de los Refactoring.

Formación de recursos humanos

Los trabajos elaborados en la presente

investigación forman parte del desarrollo de tesis para optar al grado de Magister en ingeniería de sistemas de software (UAI).

REFERENCIAS

Aldekoa, G., Trujillo, S., Sagardui, G., & Diaz, O. (2006). Experience Measuring Maintainability in Software Product lines. Brcelona: XV Jornadas de Ingeniería del software y Base de Datos.

Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994). Using Metrics to Evaluate Software System Maintainabilitv. *27* (8).

Demeyer, S., Ducasse, S., & Nierstrasz, O. (2009). *Object-Oriented Reengineering Patterns.*

Fowler, M., Rice, D., & Foemmel, M. (2010). *Patterns of enterprise application architecture* (Vol. 1). Boston, MA: Addison Wesley.

Foote, B., & Yoder, J. W. (2000). Big ball of mud. In *Pattern Languages of Program Design* (Vols. volume 4, pages 654–692.). Addison Wesley.

Gamma, E., Helm, R., & Johnson, R. (c2003). *Patrones de diseño : elementos de software orientado a objetos reutilizable* (Vol. 1). Madrid: Pearson Educación.

Institute of electrical and Electronics Engineers. (1990). *IEEE Standar Computer dictionary: A compilation of IEEE Standard Computer Glossaries.* New York, USA: IEEE.

Joyanes Aguilar, L. (1998). *Programación en turbo /borland : Pascal 7* (Vol. 3a). Madrid: Osborne/McGraw-Hill.

Kaur, K., & Singh, H. (2011). Determination of Maintainability Index for Object Oriented systems. *36* (2).

Kerievsky, J. (2009). *Refactoring to patterns* (Vol. 1). Boston, MA : Addison Wesley.

Laudon, K. C., & Laudon, J. P. (2000). *Sistemas de información gerencial: administración de la empresa digital* (Vol. 8). Mexico, DF: Pearson Educación.

Larman, C. (c2003). *UML y patrones : una introducción al análisis y diseño orientado a objetos y al proceso unificado* (Vol. 2). Madrid: Pearson Educación.

Lehman, M. (2000). *Rules and tools for software Evolution Planning and Management FEAST 2000 International Workshop: Feedback in software and Business Processes.* London: Imperial College.

Mendez, M., Overbey, J., & Garrido, A. (2010). *A Catalog and Classification of fortran Refactoring.* La Plata: Universidad de la Pplata.

Overbey, J., Negara, S., & Johnson, E. (2009). Refactoring and the evolution of Fortran. (IEEE, Ed.) *roceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering, ser. SECSE '09* , 28-34.

Oman, P., & Hagemeister, J. (1992). *Metrics for Assesing a Software system's Maintainability.* Moscow: University of Odaho.

Poveda Villalon, M., Suárez-Figueroa, M., & Suárez-Figueroa, G. (2009). *Malas prácticas en ontologías.* Sevilla, España.: CAEPIA-TTIA.

Porter, J. (1991). *La ventaja competitiva de las naciones.* Buenos aires: Javier Vergara.

Samoladas, I., Stamelos, I., Lefteris, A., & Apostolos, O. (2004). Open Source software Development Should Strive for Even Greater Code Maintainability. *47* (10).

Sommerville, I. (2002). *Ingeniería de Softaware.* México: Pearson.

Raymond, P. B., & Westley, R. W. (2008). *A metric for software readability* (Vol. 10.1145/1390630.1390647). Seattle: International symposium on software testing and analysis.