"Visualización del Impacto de la Refactorización de Código"

Alumno:

Juan Andrés Alcorta

Director:

Mg. Santiago A. Vidal

Co-Directora:

Dra. Claudia Marcos

Construcción de sistemas

Se puede observar que en gran medida el software tiende a ser mantenido utilizando prácticas que generan consecuencias negativas [9]. Estas prácticas se basan en tareas destinadas a corregir inconsistencias y errores que surgen en muchos casos de la incomprensión del sistema, y en otros casos de la falta de estándares y buen diseño [1]. En este proceso gradual de degradamiento, los desarrolladores tratan de insertar nuevos cambios en el sistema en la menor cantidad de tiempo posible. El resultado es la pérdida progresiva de uno de los atributos más importantes del software: la calidad.

El proceso en el que el software va perdiendo su calidad ha llevado a autores a comparar este proceso con el que sufre a través del tiempo cualquier sustancia orgánica: se descompone, es decir, con el tiempo va tomando mal olor (bad smell). Un bad smell (también llamado Code Smell) se refiere a cualquier síntoma que puede indicar que el código fuente presenta problemas [2]. Estos síntomas no son necesariamente defectos o impedimentos de que el programa funcione correctamente. En cambio, indican deficiencias en el diseño que pueden incrementar el costo de mantenimiento y evolución [2]. Algunas señales de code smell son: código duplicado, métodos muy largos, clases muy grandes, métodos que reciben muchos parámetros.

Para solucionar los code smells, el código debe ser refactorizado [2]. En ingeniería de software, refactoring describe el proceso de modificar el código fuente sin cambiar su comportamiento. Es

realizado a menudo como parte del proceso de desarrollo de software y se cree que la falta de la mismo incurre en el incremento de una deuda técnica generando mayores costos de mantenimiento [3].

Beneficios de refactoring de código:

- Código auto-documentado, incrementando legibilidad y mantenibilidad. En la mayoría de los casos, es la única documentación que siempre permanece actualizada. Permite la comprensión rápida del código ya que el mismo comunica su propósito fácilmente.
- Mejora en el diseño del software. Durante el proceso de desarrollo, la calidad del diseño de software puede decrecer rápidamente añadiendo gran dificultad para introducir nuevos requerimientos. Software con un diseño pobre requiere una mayor cantidad de código para realizar una misma tarea ya que probablemente el código se repita en distintos lugares.
- Generalización de código. Promueve el re-uso de código y permite aplicarlo a una variedad más amplia de problemas.

Los beneficios anteriores contribuyen con la mejora de aspectos de calidad como flexibilidad, mantenibilidad y reusabilidad, lo que resulta en un código más robusto.

Los desarrolladores tienden a tener opiniones firmes sobre refactoring [4]. Algunos (tipo A) están convencidos de la pésima calidad del trabajo de los otros desarrolladores y quieren refactorizar todo el código, sin embargo esto no siempre es posible debido al riesgo de la incorporación de errores. Otros (tipo B) siempre toman el enfoque de aversión al riesgo de hacer la menor cantidad posible de modificaciones que completarán su tarea. En general, muy pocos desarrolladores (Tipo C) se acercan a un equilibrio entre la adición de nueva funcionalidad y la incorporación de refactoring. En cualquiera de los casos de tipo A, B y C la falta de herramientas que determinan el impacto real de rafactoring y sus efectos secundarios contribuyen con la decisión de posponer el refactoring reiteradas veces y en la mayoría de los casos nunca es ejecutado [4].

Se cree que la falta de refactoring incurre en la inclusión de funcionalidad no deseada y el incremento de costos en testing [3]. Por ejemplo, eXtreme Programming afirma que el refactoring ahorra tiempo en desarrollo y mejora la calidad del software [5]. Por otro lado, existe la idea de que los ingenieros de software a menudo evitan el refactoring cuando se encuentran limitados con respecto a recursos y tiempos de entrega [3].

Estudios recientes muestran resultados contradictorios: Ratzinger et al. [6] encontró que si el número de refactorings crece el número de defectos decrece; sin embargo, Weißgerber y Diehl encontraron que en un alto porcentaje de refactorings usualmente desencadena un incremento del

reporte de defectos [7]. Aunque el refactoring se define como la modificación del código preservando la semántica del mismo, Murphy-Hill et al. encontró que desarrolladores usualmente ejecutan refactorings junto con otros cambios que alteran el comportamiento, además que los refactorings son aplicados en forma manual en lugar de utilizar una herramienta que automatiza el proceso [8]. Este tipo de prácticas es propenso a introducir errores en el código.

Solución propuesta

El objetivo de este trabajo final es desarrollar una aplicación que pueda asistir a desarrolladores en la aplicación de varios refactorings simultáneos con el fin de mejorar la calidad del código. Un ejemplo de estas mejoras es la solución de code smells. Para esto, se extenderá el entorno de desarrollo Eclipse (el cual provee la funcionalidad de aplicar varios tipos de refactoring), de forma tal que es posible la resolución de code smells mediante la aplicación de un conjunto de refactorings.

Dado un posible refactoring o conjunto de refactorings que se desean aplicar, esta aplicación proveerá una visión global de los cambios y el impacto de los mismos mediante la utilización de diagramas UML. Los diagramas mostrarán detalles a nivel de paquetes y de clases, haciendo énfasis en las partes del sistema donde se encuentra la mayor cantidad de cambios. Una vez que el usuario visualiza en forma gráfica los cambios se proveerá la opción de continuar aplicando el refactoring o deshacer los cambios.

Adicionalmente, se mostrarán métricas sobre los módulos impactados por el refactoring.

De esta manera el usuario puede ser alertado de cambios que no estaban siendo considerandos y podrían ser propensos a introducir defectos en la aplicación.

Objetivos específicos:

- Permitir al usuario seleccionar y aplicar un conjunto de refactorings.
- Mostrar gráficamente el impacto que tendrán un conjunto de refactorings en el código.
- Dado el impacto, brindar al usuario la opción de proceder aplicando el refactoring o descartarlo.
- Generar métricas con las áreas de código más afectadas.

Plan de trabajo

- Investigación y análisis de los principios de refactoring. (1 mes)
    - Definición y tipos de refactoring.
    - Razones por las cuales aplicar refactoring.
    - Situación en las que aplicar un refactoring.
    - Problemas en la aplicación de refactoring.
- Investigación y análisis de herramientas de modelado UML. (1 mes)
- Investigación del funcionamiento de los módulos de refactoring de Eclipse. (1 mes)
- Integración de herramienta de modelado UML con módulos de Refactoring. (3 meses)
- Desarrollo de funcionalidad: múltiples refactorings, layout de diagramas UML, métricas. (4 meses)
- Escritura del informe. (2 meses)

Bibliografía

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

[2] M. Fowler and K. Beck. refactoring: improving the design of existing code. Addison-Wesley Professional. 1999.

[3] L. A. Belady and M. Lehman. A Model of Large Program Development. IBM Systems Journal. 1976.

[4] Emerson Murphy-Hill and Andrew P. Black. Breaking the barriers to successful refactoring: observations and tools for extract method. In ICSE '08: Proceedings of the 30th International Conference on Software Engineering, pages 421–430, New York, NY, USA, 2008. ACM.

[5] K. Beck. extreme Programming explained, embrace change. Addison-Wesley Professional, 2000.

[6] J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In MSR ’08: Proceedings of the 2008 international working conference on Mining software repositories, pages 35–38, New York, NY, USA, 2008. ACM.

[7] C. Gorg and P. Weißgerber. Error detection by refactoring ¨ reconstruction. In MSR ’05: Proceedings of the 2005 international workshop on Mining software repositories, pages 1–5, New York, NY, USA, 2005. ACM Press.

[8] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In ICSE ’09: Proceedings of the 31st International Conference on Software Engineering, pages 287– 297, Washington, DC, USA, 2009. IEEE Computer Society.

[9] W. Brown, R. Malveau, H. McCormick III, T. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons. 1998.