Manutenção de Sistemas Desenvolvidos Utilizando Desenvolvimento Guiado por Testes

Vinícius H. S. Durelli [1], Matheus C. Viana [2], Rosângela Ap. D. Penteado

Departamento de Computação (DC) - Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – 13565-905 – São Carlos – SP – Brasil

{vinicius_durelli, matheus_viana, rosangel}@dc.ufscar.br

Abstract: *Software systems and the technology on which they are built remain in constant evolution. However, there are no methods and techniques well suited to support the evolution of these systems. This paper describes the benefits of well formed sets of automated tests that can be used during corrective, adaptive, predictive and perfective maintenance activities. It also contains an example describing a framework that has been reengineered from Smalltalk to Java by an iterative test driven process.*

Resumo: *Sistemas de software estão sujeitos à constante evolução, bem como as tecnologias sobre as quais eles são construídos. No entanto, faltam métodos e técnicas apropriados para dar apoio à evolução desses sistemas. Neste trabalho, são descritos os benefícios que podem ser obtidos com o uso de conjuntos consistentes de testes automatizados durante a realização de manutenção corretiva, adaptativa, preventiva e perfectiva. Um framework que passou pelo processo de reengenharia de Smalltalk para Java, de maneira iterativa e guiada por testes.*

1. Introdução

Reúso é o processo de construir software utilizando artefatos já existentes [Krueger 1992]. Pode-se reutilizar tanto experiência, por exemplo, documentada de forma padronizada, quanto artefatos de software como bibliotecas de componentes reutilizáveis e frameworks. Padrões de software podem ser citados como um dos exemplos de reutilização da experiência obtida e documentada por desenvolvedores experientes. A essência desses padrões é comunicar uma solução para determinado problema em um contexto específico [Gamma et al. 1995]. Existem padrões em vários níveis de abstração, como por exemplo: análise [Fowler 1996], projeto [Gamma et al. 1995], implementação [Beck 2007], engenharia reversa e reengenharia [Demeyer et al. 2002].

Frameworks proporcionam, além de reúso de código, também o reúso de projeto [Johnson 1997a]. São específicos de um domínio e constituídos por classes de software, abstratas e concretas, que fazem parte de um projeto abstrato. Esse projeto abstrato pode ser customizado por um desenvolvedor de aplicações com o objetivo de criar aplicações específicas em uma determinada linguagem de programação.

[1] Apoio financeiro CAPES.
[2] Apoio financeiro CNPq.

A utilização de frameworks para produzir novos sistemas de software resulta em: economia de recursos, aumento de produtividade e maior qualidade dos artefatos produzidos [Shiva e Shala 2007]. Todavia, requisitos e ambientes, em constantes mudanças, exercem influência sobre os sistemas de software. Assim, durante seu ciclo de vida, um sistema de software deve ser submetido a várias alterações. Caso contrário torna-se, progressivamente, menos útil nesse ambiente. À medida que um sistema de software evolui, sua complexidade aumenta e recursos extras são necessários para preservar e simplificar sua estrutura [Lehman 1996]. Essas alterações podem ter o propósito de corrigir problemas, realizar adaptações, aprimorar o código do sistema e/ou introduzir nova funcionalidade requerida pelo ambiente/usuário. Porém, outros problemas podem ser introduzidos durante a realização das alterações necessárias.

Uma das formas de se alterar sistemas de software orientados a objetos é por meio de refatoração, melhorando sua estrutura interna, de forma que o comportamento observável do código não seja alterado [Fowler et al. 1999]. Entre outras coisas, redistribui-se de classes, variáveis e métodos na hierarquia de classes, com o objetivo de facilitar futuras atividades de desenvolvimento ou manutenção [Mens e Tourwe 2004] [Demeyer et al. 2004]. Para a realização de refatorações, uma pré-condição é a existência de testes automatizados, utilizados para detectar a introdução inadvertida de problemas. Contudo, a maioria dos sistemas de software não tem um conjunto de testes que possa ser utilizado durante as atividades de refatoração.

Este trabalho tem o objetivo de apresentar como as atividades de manutenção podem ser conduzidas quando o sistema possui um conjunto de testes automatizados desde o início de seu desenvolvimento. Pois, posteriormente, esses testes juntamente com refatorações podem ser utilizados na realização de vários tipos de manutenção. O framework Gestão de REcursos de Negócios em Java (GRENJ) [Durelli 2007], implementado por um processo de reengenharia que emprega o desenvolvimento guiado por testes (*Test Driven Development*, TDD) [Beck 2002] [Koskela 2007], é utilizado para exemplificar as atividades de manutenção mencionadas.

Além desta Seção, este trabalho está organizado da seguinte forma: a Seção 2 apresenta os conceitos relacionados à prática denominada TDD; a Seção 3 apresenta o framework GRENJ e as ferramentas utilizadas em seu desenvolvimento; a Seção 4 comenta como os vários tipos de manutenção podem ser favorecidos com a existência de testes automatizados. A Seção 5 aborda alguns trabalhos relacionados e, por fim, a Seção 6 apresenta as considerações finais.

2. Test-Driven Development – TDD

TDD, juntamente com a refatoração e a programação em pares, figura entre as práticas fundamentais de *Extreme Programming* (XP) [Janzen e Saiedian 2005]. Usando TDD, casos de teste são criados de acordo com a funcionalidade que será implementada. Esses casos de teste são agrupados em uma lista de casos de teste (*list of test cases*) ou simplesmente lista de testes (*list of tests*) [Beck 2002] [Koskela 2007]. Os testes são implementados conforme se avança no desenvolvimento do sistema.

Considera-se que um teste foi satisfeito quando a funcionalidade avaliada por ele é corretamente implementada, o que deve ocorrer logo após a implementação do teste. Implementam-se os testes da lista de casos de testes até que a funcionalidade almejada tenha sido obtida, ou seja, até que todos os testes que a avaliam tenham sido satisfeitos.

Desse modo, utilizando-se TDD tanto o código referente aos testes quanto o código da funcionalidade, avaliada pelos testes, são produzidos em rápidas e pequenas iterações. No desenvolvimento em cascata e no desenvolvimento iterativo, os testes são criados após a implementação de toda ou de parcelas da funcionalidade, respectivamente. Abordagens desse tipo são consideradas "tradicionais" e denominadas abordagens *test-last* [Erdogmus et al. 2005]. Abordagens que empregam TDD são denominadas *test-first*. A Figura 1 contrasta o desenvolvimento de abordagens *test-last* e *test-first*.

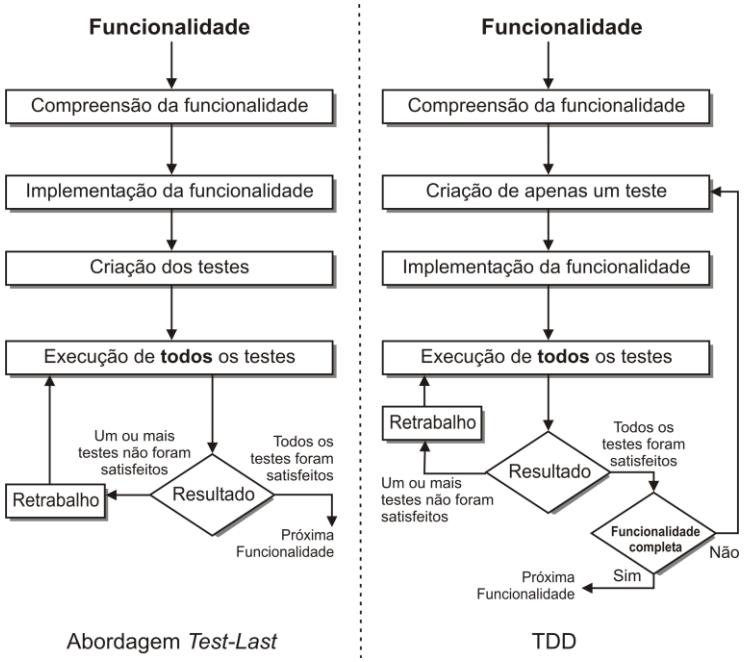

Figura 1. Abordagens *test-last* e TDD [Erdogmus et al. [2005].

3. O Framework GRENJ

GRENJ [Durelli 2007] é um framework resultante da reengenharia do framework Gestão de REcursos de Negócios (GREN) [Braga 2002] que foi implementado na linguagem Smalltalk com base nos quinze padrões de análise definidos pela Linguagem de Padrões para Gestão de Recursos de Negócio (GRN) [Braga e Masiero 2002]. Os sistemas que podem ser instanciados com esses frameworks são aqueles que têm operações do tipo: compra, venda, aluguel e manutenção. GRENJ foi implementado na linguagem Java e sua extensão para programação orientada a aspectos, AspectJ [Kiczales et al. 2001]. O processo de reengenharia aplicado é iterativo e utilizou os padrões de engenharia reversa [Demeyer et al. 2002] para extração do modelo de projeto do GREN. A implementação do GRENJ foi realizada empregando TDD.

O fato da linguagem Smalltalk ser dinamicamente tipada e a linguagem Java ser fortemente tipada implicou em várias refatorações durante o processo de implementação do GRENJ, pois a inferência dos tipos em linguagem dinamicamente tipada é complexa. Assim, a criação dos testes desde o inicio da reengenharia, devido à aplicação do TDD, facilitou a realização de refatorações. GRENJ, atualmente, é composto de pouco mais de vinte e oito mil linhas de código, das quais mais de dez mil estão relacionadas aos testes criados. O framework encontra-se dividido em duas camadas: a de negócios e a de persistência. A camada de negócio contém a implementação dos padrões da GRN. A camada de persistência contém as classes relacionadas à conexão com o banco de dados

e persistência dos objetos de forma geral. Essa camada foi implementada de acordo com o padrão *Persistence Layer* [Yoder et al. 1998].

Algumas ferramentas foram utilizadas, em conjunto com o IDE (*Integrated Development Environment*) Eclipse [Eclipse 2008], para facilitar o desenvolvimento do framework GRENJ. Algumas refatorações são automatizadas no próprio ambiente Eclipse, como: *Extract Method*, *Extract Superclass*, *Encapsulate Field*, entre outras. Além disso, o IDE mantém um histórico das refatorações aplicadas durante o desenvolvimento. Para a automatização do TDD e a criação dos testes, foi utilizado o framework JUnit [JUnit 2008]. O *plugin* EclEmma [Eclmma 2008] também foi utilizado para verificar quais os trechos de código que ainda não foram exercitados, possibilitando que a cobertura dos testes seja ampliada, aumentando assim a qualidade dos testes criados e do produto produzido.

4. Atividades de Manutenção Realizadas no Framework GRENJ

Atividades de manutenção podem ser realizadas para corrigir problemas (corretiva), realizar adaptações (adaptativa), aprimorar o projeto do sistema (preventiva) e introduzir novas funções (perfectiva).

As modificações necessárias durante as atividades de manutenção podem introduzir defeitos difíceis de serem detectados e gerar desperdício de tempo em sessões de depuração [Douce e Layzell 1999]. Em sistemas complexos, é difícil antecipar o relacionamento existente entre todas as suas camadas arquiteturais ou entidades. A quantidade de problemas introduzidos pode ser maior em sistemas com alto nível de acoplamento entre as suas classes e as camadas existentes.

Testes automatizados facilitam a realização de atividades de manutenção, apoiando a detecção de problemas e evitando, assim, que alterações causem efeitos colaterais inesperados. Testes executados após a realização de alterações no sistema são denominados testes de regressão e têm o propósito de detectar efeitos colaterais resultantes de alterações [Myers et al. 2004] [Meszaros 2007]. Um problema de regressão ocorre quando alguma funcionalidade do sistema deixa de realizar o comportamento esperado devido às alterações nele realizadas. A existência de testes possibilita que alterações na funcionalidade, no projeto e até mesmo na arquitetura do sistema sejam realizadas [Demeyer et al. 2002]. As Subseções seguintes descrevem como os testes, criados durante o desenvolvimento aplicando TDD, podem ser utilizados como testes de regressão, possibilitando, assim, a realização de atividades de manutenção utilizando refatorações.

4.1 Manutenção Corretiva

A manutenção corretiva quando realizada em sistema sem um conjunto de testes de regressão requer que testes manuais ou não automatizados sejam realizados a fim de reproduzir a situação problemática reportada. Quando o problema é identificado, as correções são realizadas e, novamente, a funcionalidade é avaliada de maneira não automatizada. Por exemplo, por meio da interface gráfica do sistema o desenvolvedor fornece os mesmos valores (parâmetros) que fizeram com que a funcionalidade se comportasse de maneira errônea anteriormente, avaliando se as alterações realizadas solucionaram o problema. Todavia, sem testes de regressão, não é trivial assegurar que, após as alterações necessárias serem realizadas, outros problemas não foram inseridos.

Quando o sistema que está com problema possui um conjunto de testes de regressão, é recomendável criar um teste automatizado para detectar o problema [Beck 2002] [Demeyer et al. 2002]. Durante a realização das modificações, necessárias para que o problema seja solucionado, o conjunto de testes de regressão é executado freqüentemente. Esse conjunto de testes apóia as alterações evitando que, inadvertidamente, o desenvolvedor altere trechos de código relacionados a outra funcionalidade que está corretamente implementada.

4.2 Manutenção Adaptativa

Os testes criados durante o desenvolvimento empregando TDD não avaliam questões relacionadas ao ambiente externo do sistema que está sendo desenvolvido. O framework GRENJ foi implementado na linguagem Java, assim, possíveis modificações de plataforma, Linux para Windows, por exemplo, podem ser efetuadas sem nenhuma alteração no framework.

4.3 Manutenção Preventiva

Sistemas de software são submetidos a várias alterações durante e após o seu desenvolvimento, as quais podem aumentar a complexidade do sistema. Alguns dos indícios de que a estrutura do sistema deve ser aperfeiçoada são: (i) simples alterações no sistema são custosas; (ii) usuários reportam problemas constantemente; (iii) o sistema possui código duplicado e outros *bad smells* [Fowler et al. 1999]. Aprimoramentos incrementais devem ser realizados de forma que o sistema possa continuar sendo utilizado e adaptado a um custo aceitável.

Uma das formas de facilitar a compreensão do sistema e aprimorar sua estrutura interna é por meio de refatorações. Para realizá-las com segurança, é indispensável a presença de testes automatizados [Fowler et al. 1999] [Demeyer et al. 2002] [Beck 2002]. Em sistemas sem esses testes é indicado o uso do padrão *Write Tests to Enable Evolution* [Demeyer et al. 2002]. Esse padrão estabelece que os custos e benefícios relacionados à introdução de testes devem ser balanceados. Portanto, constrói-se um conjunto de testes de regressão à medida que as atividades de manutenção preventiva são realizadas e somente para o trecho de código, camada, classe ou método que será aperfeiçoado.

Em sistemas com testes de regressão, caso o trecho de código que será aprimorado esteja coberto pelos testes, os desenvolvedores só precisam analisar quais *bad smells* [Fowler et al. 1999] existem e aplicar as refatorações apropriadas. Quando o código que será aprimorado é parcialmente coberto pelos testes, o padrão *Write Tests to Enable Evolution* pode ser aplicado para produzir os testes adicionais.

Padrões de projeto podem ser inseridos para tornar o sistema mais flexível à introdução de novas funções [Kerievsky 2004]. Ressalta-se que o código relacionado aos testes pode possuir *bad smells* [Deursen *et al* 2001] e deve ser refatorado quando necessário [Meszaros 2007].

4.4 Manutenção Perfectiva

A adição de funcionalidade, geralmente, pode introduzir problemas, considerando que raramente um desenvolvedor conhece todo o sistema e que é complexo prever quais conseqüências essa funcionalidade recentemente acrescentada pode causar.

No GRENJ a adição de funcionalidade pode ser realizada iterativamente seguida da execução dos testes de regressão. Além disso, é recomendável que a "nova" funcionalidade seja implementada utilizando TDD, que proporciona rápido *feedback* quanto à introdução de problemas e é, essencialmente, uma técnica de análise e projeto [Beck 2001] [Janzen e Saiedian 2005] [Jeffries e Melnik 2007]. O conjunto de testes de regressão também permite, quando necessário, que refatorações sejam realizadas a fim de "acomodar" a introdução dessa nova funcionalidade [Fowler et al. 1999].

5. Trabalhos Relacionados

Neste trabalho alterações no framework são realizadas por meio de refatorações. Alguns autores propõem, também, a utilização de regras de unificação (*unification rules*) para apoiar a manutenção e a evolução de frameworks [Cortes et al. 2004]. Regras de unificação são transformações que não preservam o comportamento externo e definem uma maneira de se incorporar novas abstrações na estrutura do framework. Portanto, poderiam ser introduzidas durante atividades de manutenção perfectiva. Todavia, alterações no comportamento externo podem influenciar alguns testes. O IDE Eclipse ainda não automatiza a aplicação de regras de unificação.

Alguns estudos analisam os benefícios da aplicação do TDD durante o desenvolvimento de software, bem como os benefícios derivados dos testes criados [Müller e Hagner 2002] [William et al. 2003] [Erdogmus et al. 2005] [Janzen e Saiedian 2006] [Müller e Höfer 2007]. Uma técnica para seleção do conjunto de testes de regressão em sistemas implementados na linguagem Java é apresentada por Orso et al. [2004].

Stroggylos e Spinellis [2007] examinam, por meio de métricas, a qualidade do código fonte de alguns sistemas após a realização de refatorações. Esse estudo aponta que nem todas as refatorações realizadas aprimoraram a qualidade dos sistemas. Dessa forma, apesar das refatorações possibilitarem a evolução dos sistemas, é responsabilidade do desenvolvedor avaliar quais refatorações devem ser realizadas e quais as conseqüências envolvidas.

6. Considerações Finais

A manutenção de sistemas pode ser facilitada quando algumas práticas são adotadas desde o desenvolvimento ou desde o início do processo de reengenharia. Uma delas é o TDD. Dessa forma, os testes automatizados criados durante o desenvolvimento do sistema podem, durante as atividades de manutenção, ser utilizados como testes de regressão. Outra prática é a refatoração, que pode ser utilizada tanto durante quanto após o desenvolvimento dos sistemas.

A fim de melhorar a cobertura dos testes, o *plugin* EclEmma [EclEmma 2008] foi empregado para obter informações sobre os trechos de código do framework GRENJ que não são exercitados pelos testes, criados durante seu desenvolvimento. Pois, mesmo utilizando TDD, nem toda a funcionalidade implementada tem cem por cento de cobertura. Caso um trecho de código não exercitado pelos testes necessite ser alterado, outros testes que exercitam esse trecho devem ser criados, assegurando que os efeitos colaterais introduzidos durantes as alterações sejam detectados.

A utilização de testes automatizados desde o início do desenvolvimento possibilita ao desenvolvedor cuidar atentamente de cada parte inserida ou modificada no

framework para que a funcionalidade original não seja alterada. As práticas TDD e refatoração possibilitam que o desenvolvedor realize qualquer tipo de manutenção de forma mais segura. Adicionalmente, os testes criados também servem como documentação do sistema [Beck 2002] [Demeyer et al. 2002].

Referências

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Professional.

Beck, K. (2007). *Implementation Patterns*. Addison-Wesley.

Braga, R. T. V. (2002). *Um Processo para Construção e Instanciação de Frameworks Baseados em uma Linguagem de Padrões para um Domínio Específico*. Tese de doutorado, ICMC/USP, São Carlos - SP.

Braga, R. T. V., Germano, F. S. R., e Masiero, P. C. (1999). *A Pattern Language for Business Resource Management*. Proceedings of the Annual Conference on Pattern Languages of Programs, 6:1–33.

Cortes, M., Fontoura, M., e Lucena, C. (2003). *Using Refactoring and Unification Rules to Assist Framework Evolution*. SIGSOFT Software Engineering Notes, 28(6):1–5.

Demeyer, S., Ducasse, S., e Nierstrasz, O. (2002). *Object-Oriented Reengineering Patterns*. Morgan Kaufmann.

Deursen, A., Moonen, L., Bergh, A., e Kok, G. (2001). *Refactoring Test Code*. In Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001).

Douce, C. R. e Layzell, P. J. (1999). *Evolution and Errors: An Empirical Example*. In ICSM ’99: Proceedings of the IEEE International Conference on Software Maintenance, páginas 493–498. IEEE Computer Society.

Durelli, V. H. S. (2007). *Reengenharia Iterativa do Framework GREN*. Documento de qualificação, Universidade Federal de São Carlos, São Carlos - SP.

EclEmma (2008). *EclEmma 1.3.1 – Java Code Coverage for Eclipse*. Disponível em: *http://www.eclemma.org/*. Acessado em 01 de Março de 2008.

Eclipse (2008). *The Eclipse Foundation*. Disponível em: *http://www.eclipse.org/*. Acessado em 20 de Fevereiro de 2008.

Erdogmus, H.,Morisio,M., e Torchiano,M. (2005). *On the Effectiveness of the Test-First Approach to Programming*. Software Engineering, IEEE Transactions on, 31:226-237.

Fowler, M. (1996). *Analysis Patterns: Reusable Object Models*. Addison-Wesley.

Fowler, M., Beck, K., Brant, J., Opdyke, W., e Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., e Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Janzen, D. e Saiedian, H. (2005). *Test-Driven Development: Concepts, Taxonomy, and Future Direction*. IEEE Computer, 38(9):43–50.

Jeffries, R. e Melnik, G. (2007). *Guest Editors' Introduction: TDD–The Art of Fearless Programming*. IEEE Software, 24(3):24–30.

Johnson, R. E. (1997). *Frameworks = (Components + Patterns)*. Communications of the ACM, 40.

JUnit (2008). *JUnit.org Resources for Test Driven Development*. Disponível em: *http://www.junit.org/*. Acessado em 01 de Março de 2008.

Kerievsky, J. (2004). *Refactoring to Patterns*. Addison-Wesley Professional.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., e Griswold, W. (2001). *Getting Started with AspectJ*. Communications of the ACM, 44:59–65.

Koskela, L. (2007). *Test Driven: TDD and Acceptance TDD for Java Developers*. Manning Publications.

Krueger (1992). *Software Reuse*. ACM Computing Surveys, 24.

Lehman, M. (1996). *Laws of Software Evolution Revisited*. In European Workshop on Software Process Technology, páginas 108–124. Springer.

Mens, T. e Tourwe, T. (2004). *A Survey of Software Refactoring*. Transactions on Software Engineering, IEEE, 30.

Meszaros, G. (2007). *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.

Müller, M. M. e Hagner, O. (2002). *Experiment about Test-First Programming*. IEEE Proceedings– Software, 149(5):131–136.

Müller, M. M. e Höfer, A. (2007). *The Effect of Experience on the Test-Driven Development*. Empirical Software Engineering, 12(6):593–615.

Myers, G. J., Sandler, C., Badgett, T., e Thomas, T. M. (2004). *The Art of Software Testing*. Wiley.

Orso, A., Shi, N., e Harrold,M. J. (2004). *Scaling Regression Testing to Large Software Systems*. In SIGSOFT '04: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, páginas 241–251. ACM.

Shiva, S. G. e Shala, L. A. (2007). *Software Reuse: Research and Practice*. International Conference on Information Technology, páginas 603–609.

Stroggylos, K. e Spinellis, D. (2007). Refactoring - does it improve software quality? In *ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops*. IEEE Computer Society.

Williams, L., Maximilien, E. M., e Vouk, M. (2003). *Test-Driven Development as a Defect-Reduction Practice*. In ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering, páginas 34–48. IEEE Computer Society.

Yoder, J. W., Johnson, R. E., e Wilson, Q. (1998). *Connecting Business Objects to Relational Databases*. Proceedings of the Annual Conference on Pattern Languages of Programs, Monticello-IL, EUA.