A Relação Entre TDD e Qualidade de Software

InfoQ Br

Postado por **Cássio Landim Ribeiro** em 05 Abr 2010

http://www.infoq.com/br/articles/relacao-tdd-qualidade

TDD é uma prática que visa aumentar a velocidade da entrega de produtos através da simplificação das atividades de desenho de software. [Koskela 2008] resume a filosofia do TDD em uma frase -- somente escreva código para fazer um teste falho passar. Enquanto isso, [Astels 2003] define o TDD como sendo um estilo de desenvolvimento onde:

- Uma suíte exaustiva de testes de programadores é mantida;
- Nenhum código entra em produção a não ser que tenha testes associados;
- Os testes são escritos antes;
- Os testes determinam que código precise ser escrito.

Qualidade não pode ser alcançada através da avaliação de um produto já feito. O objetivo, portanto, é prevenir defeitos de qualidade ou deficiências em primeiro lugar, tornando os produtos avaliáveis através de medidas de garantia de qualidade [Lewis 2004].

[Beck 1999] cita alguns exemplos de riscos relacionados ao desenvolvimento de software. Dos riscos citados, dois estão diretamente ligados a qualidade de software e podem ser tratados através da utilização de TDD:

- Taxa de defeitos -- o software é colocado em produção mas a taxa de defeitos é tão alta que ele acaba não sendo utilizado. TDD eleva a validação de um software a um patamar superior, testando-o função por função;
- Deterioração do sistema -- o software é colocado em produção com sucesso, porém após algum tempo o custo de se fazer modificações ou a taxa de defeitos aumenta tanto que o sistema precisa ser substituído. TDD mantém o programador focado na solução, de forma que o software não fica carregado de códigos desnecessários, duplicados ou de difícil manutenção, impedindo a deterioração do sistema.

Nesta mesma obra, [Beck 1999] elaborou três frases de impacto, que servem como um ponto de partida para entendermos como o TDD afeta a qualidade de um software:

- Toda vez que alguém toma uma decisão e não a testa, existe uma grande probabilidade de que esta decisão esteja errada;
- Funcionalidades de software que não podem ser demonstradas através de testes automatizados simplesmente não existem;
- Testes nos dão à chance de pensar sobre o que queremos, independente da forma como a solução será implementada.

Ao utilizar TDD, devemos escrever testes para cada solução implementada. Dessa forma, diminuímos a probabilidade de tomarmos uma decisão errada. Ao mesmo tempo, temos a

oportunidade de experimentar várias implementações diferentes para o mesmo problema e escolher aquela mais limpa, elegante e que apresente o melhor desempenho.

Desenho Simplificado e Evolucionário

Escrevendo somente o necessário para os testes e removendo toda a duplicação, você automaticamente obtém um desenho que é perfeitamente adaptado para os requisitos atuais e igualmente preparado para todas as futuras funcionalidades [Beck 2002].

Design simplificado reduz os custos porque ao escrever menos código para atender os requisitos, menos código existirá para ser mantido no futuro. Design simplificado é mais fácil de se construir, manter e entender.

Refatoração

Os testes lhe dão a confiança de que grandes refatorações não mudarão o comportamento do sistema, o que se conclui que, quanto maior a confiança, mais agressivamente você poderá conduzir refatorações em larga escala que estenderão a vida de seu sistema. A refatoração torna a elaboração dos próximos testes muito mais fácil [Beck 2002].

Custos são reduzidos porque a refatoração contínua evita que o desenho se degrade com o passar do tempo, assegurando que o código continue fácil de ser entendido, mantido e modificado.

Feedback Constante

[Beck 2002], no último capítulo de sua publicação, afirma que TDD o ajuda a dar atenção aos problemas certos na hora certa, de forma que o desenho do software fica mais limpo e com muito menos defeitos. O TDD faz com que o programador ganhe confiança sobre seu código com o passar do tempo, isto é, à medida que os testes vão se acumulando (e melhorando), ele ganha confiança no comportamento do sistema. E ainda, à medida que o desenho é refinado, mais e mais mudanças se tornam possíveis.

Outra vantagem do TDD que [Beck 2002] acredita poder explicar seus efeitos positivos, é a forma como ele encurta o ciclo de feedback sobre as decisões de desenho. Ele dura apenas segundos ou minutos, e é seguido pela reexecução dos testes dezenas ou centenas de vezes por dia. Ao invés de se projetar um desenho e então esperar semanas ou meses para outra pessoa sentir as dores ou glórias de sua consequência, o feedback emerge em segundos ou minutos, enquanto você tenta traduzir suas idéias em interfaces plausíveis.

Suíte de Testes (Regressão)

Usando TDD, os testes unitários são criados num momento onde a funcionalidade a ser implementada está mais bem definida na mente do programador, e depois podem e devem ser utilizados para fazer testes de regressão.

Uma suíte de testes automáticos feita por programadores reduz os custos de um software funcionando como uma rede de segurança de testes que capturam defeitos, problemas de comunicação e ambigüidades antes e permitem que o desenho possa ser modificado de forma incremental.

Esta suíte de testes gerada pelo TDD é fundamental para viabilizar procedimentos de Integração Contínua.

Documentação Para Programadores

A suíte de testes serve como uma documentação voltada para o programador que tem um entendimento mais rápido e facilitado do que uma parte do código faz através do código que

o testa. Cada teste unitário especifica o uso apropriado de uma classe de produção [Langr 2005].

Conclusões

Esta técnica de desenvolvimento produz desenhos menos acoplados que são mais fáceis de manter, reduz altamente a quantidade de defeitos, e reforça a construção e manutenção apenas do que é realmente necessário. Finalmente, testes bem escritos atuam como um tipo de requisitos “executáveis” que ajudam a manter o entendimento compartilhado da equipe de desenvolvimento, sobre como o sistema de software representa os problemas do mundo real.

Por outro lado, o fato de se ter um grande número de testes unitários passando com sucesso, pode passar uma falsa sensação de segurança, resultando na implementação de menos atividades de garantia de qualidade, como testes de integração e testes de conformidade.

É importante ressaltar também que, esta técnica não garante a obtenção de níveis aceitáveis em certos aspectos do software final, como usabilidade e desempenho. Além disso, TDD não consegue mitigar riscos relacionados com a falta de requisitos ou com requisitos erroneamente definidos.

Referências

Astels, D. (2003). Test-Driven Development: A Practical Guide. Prentice Hall PTR.

Beck, K. (1999). Extreme Programming Explained: Embrace Change. Addison Wesley.

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

Koskela, L. (2008). Test Driven: Pratical TDD and Acceptance TDD for Java Developers.Manning.

Langr, J. (2005). Agile Java Crafting Code with Test-Driven Development. Prentice Hall PTR.

Lewis, W. E. (2004). Software Testing and Continuous Quality Improvement. Auerbach, 2 edition.