Workflow de desenvolvimento em C/C++ baseado em TDD. Pre-tested commits

Helton Luiz Marques[1]

[1]Univali – Universidade do Vale do Itajaí
Pós–Graduação em Qualidade e Engenharia de Software
Florianópolis – SC – Brasil

helton.marx@gmail.com

Abstract. *This meta-paper describes the style to be used in articles and short papers for SBC conferences. For papers in English, you should add just an abstract while for the papers in Portuguese, we also ask for an abstract in Portuguese ("resumo"). In both cases, abstracts should not have more than 10 lines and must be in the first page of the paper.*

Resumo. *A proposta deste trabalho é apresentar o conceito de Test Driven Development, mostrar os benefícios que esta abordagem traz para os desenvolvedores, e também apresentar a revisão de código para análise e busca de falhas no código fonte, alterações na arquitetura do sistema, reimplementação de funcionalidades já existentes, duplicidade de códigos e melhorias de estética e padronização São apresentados os benefícios da utilização dessas metodologias e por fim um workflow de desenvolvimento em C/C++ baseado em TDD.*

1. Introdução

Com a popularização dos métodos ágeis. difundiu–se e muito o desenvolvimento orientado a testes (*Test Driven Development*), onde há uma busca constante em eliminar falhas de programação durante a codificação, com foco principalmente em design, pois produzir um código testável implica em baixo acoplamento, além da acréscimo de qualidade sobre produto de software.
Além dos testes unitários, há também a revisão de código, que integrado a um sistema testes automatizados e um sistema de controle de versão obtem-se uma metodologia de trabalho eficaz e que traz velocidade e segurança no desenvolvimento de sistemas, principalmente em C/C++ em que há uma gama enorme de plataformas e arquiteturas para esta linguagem.
Este artigo apresentará uma proposta de workflow para desenvolvimento em C/C++ baseado em TDD, utilizando *pre-tested commits*, sendo possível expandí-lo a outras linguagens de programação.

2. Fundamentação Teórica

Test Driven Development (TDD) é uma abordagem iterativa para desenvolvimento de software baseada em testes automatizados. A idéia é aplicar pequenos ciclos de **teste–codificação–refatoração**, também conhecido como *red–green–refactor* que consiste em criar um teste unitário que faça falhar o código desenvolvido (*Red*), alterar o código para que o teste passe da forma mais simples (*Green*) e refatorar o código com o objetivo de

melhorar a sua estrutura e design (*Refactor*). Este ciclo deve ser repetido em pequenos incrementos, até que a funcionalidade esteja pronta.
Assim em cada iteração, a complexidade vai aumentado aos poucos, porém com a certeza que há corretude em cada implementação.
Cada passo traz um benefício. Escrever o teste antes, prática conhecida como *Test First*, obrigatoriamente força com que o código produzido seja testável por construção, e código testável implica código com baixo acoplamento, aspecto extremamente importante no design de software.
Ao escrever testes previamente, faz com que o desenvolvedor pense antecipadamente no comportamento que a aplicação deve ter, antes mesmo de escrever algum código. Desde a implementação, o projeto terá um nível de qualidade mais apurado, já que esta técnica força pensar em problemas e suas soluções antes de qualquer código existir.
Além disso, obtem–se o *feedback* imediato a cada refatoração, e também suprime o *over–engineering*, ou seja, implementar mais do que o estritamente necessário.
O feedback acontece a cada ciclo, dando uma margem de segurança ao desenvolvedor, de o que está sendo implementado além de correto é necessário.
TDD pode ser aplicado tanto para pequenas partes que compõem um sistema (Teste Unitários) quanto para componentes (Testes de Integração), e traz benefícios tanto em design quanto em performance, pois depende da Arquitetura de Teste aplicada para garantir uma boa cobertura de testes, tem-se um tempo relativamente menor na resolução de falhas do sistema total.
Além de diminuir o custo na fase de manutenção do sistema, oferece também uma margem de segurança maior aos desenvolvedores.
Os testes automatizados também podem funcionar como uma forma de documentação, pois estão descrevendo em partes o comportamento do sistema, suas interfaces, métodos e arquitetura.
TDD é entendido como um dos princípios da Extreme Programming (XP).
Os principais benefícios do TDD são:

- **Diminuição de falhas:** Erros de lógica e também falhas de design podem ser encontrados rapidamente durante o TDD, ocorrendo previamente a prevenção de defeitos.
- **Menor tempo de debug:** Com um menor número de falhas, obtêm-se um menor tempo de debug.
- **Documentação não mente:** Com testes bem estruturados, é possível analisar a qualidade da documentação. Testes bem estruturados trazem uma forma mais vísivel através de um exemplo funcional, suprindo uma documentação excassa.
- **Paz de espírito:** Um código exaustivamente testado com um conjunto abrangente de testes de unitários traz estabilidade e confiança aos desenvolvedores, trazendo conforto e fins de semana despreocupados.
- **Aperfeiçoamento do design:** Um bom design é um design testável. Funções muito grandes, forte acoplamento e condicionais complexas tornam o código mais complexo e menos testável. Os desenvolvedores percebem previamente uma falha de design caso os testes não possam ser implementados.
- **Monitorar o progresso:** Os testes trazem exatamente um retrato do que está funcionando e uma porcentagem real do trabalho realizado.
- **Feedback imediato:** O TDD traz imediatamente uma gratificação aos desenvolvedores. A cada codificação testada, há uma garantia de que o software imple-

mentando está funcionando

2.1. Test-First e Test-Last

Há duas abordagens em relação aos teste unitários, que são o *Test-First* que força que o código de teste seja produzido antes da implementação do algoritmo, e o *Test-Last* que ao inverso, é a elaboração do teste unitário após a conclusão do algoritmo a ser testado.
[Erdogmus, H., Morisio, M., and Torchiano, M. 2005], resolveram investigar, por meio de um experimento as diferenças entre as abordagens *Test-First* e *Test-Last*. O experimento consiste em dois grupos em meio acadêmico, cada qual usando umas das abordagens. As conclusões foram as seguintes: o grupo *Test-First* acabou escrevendo 52% mais testes do que o grupo *Test-Last*, algo bastante significativo. Em termos de qualidade externa, pelo estudo, não houve diferença entre os dois métodos. Houveram sim, evidências de que a qualidade externa estaria mais relacionada ao números de testes escritos do que quando estes são escritos previamente.
Segundo os autores, apesar do método Test-First por si só não aumentar a qualidade, este pareceu ter um efeito positivo sobre a produtividade dos desenvolvedores, possívelmente por causa dos seguintes fatores:

- **Melhor compreensão do problema:** Especificar o teste antes força o desenvolvedor a refletir de maneira mais profunda e completa sobre o problema a ser resolvido. Criar cenários de testes antes ajuda a provar a robustez da solução proposta.
- **Melhor foco na tarefa a ser feita:** Existe uma menor carga cognitiva sobre o desenvolvedor, pois ele estará concentrado em resolver apenas uma pequena porção do problema, somente o suficiente para atender o teste existente falhando.
- **Aprendizado mais rápido:** O desenvolvedor saberá mais rapidamente se a funcionalidade implementada está de acordo com o esperado. Além disso, existe um critério inequívoco para saber quando o trecho de funcionalidade está realmente pronto.
- **Menos esforço de retrabalho:** Em uma abordagem *Test-Last*, corre-se o risco de pouca refatoração, desenvolvendo a funcionalidade por inteiro, incorrendo em retrabalho à medida que os testes forem criados posteriormente e apontarem outros problemas.

O estudo demonstra claramente uma qualidade maior ao utilizarmos *Test-First*, porém deve-se levar em consideração diversos fatores ao selecionar um método, principalmente quando há uma equipe mais experiente que saiba gerar produtividade unindo os dois métodos. Além disso, o risco da abordagem *Test-Last* está na perda do benefício da programação por interação, ou seja, evoluir o software a partir do ponto de vista de como se espera que seja usado, e também a ocorrência em demasia de *over–engineering*, isto é, escrever código para coisas que talvez não sejam necessárias naquele momento.

2.2. Testes Automatizados

São uma ferramenta extremamente valiosa de feedback, ajudando a garantir tanto a qualidade externa (aquela perceptível pelo usuário), quanto à qualidade interna (aquela perceptível pelo desenvolvedor).
A facilidade de poder executar continuamente os testes permite a identificação imediata de alterações indesejáveis no sistema. Por essa razão, os testes existentes para um sistema

também são conhecidos como Testes de Regressão, pois asseguram que um sistema não irá regredir do ponto atual em que se encontra. O acréscimo contínuo de funcionalidades pode ser realizado sem medo de quebrar algo já estabilizado.
Os testes também favorecem a prática saudável de refatoração, que consiste em aplicar pequenas transformações na estrutura interna do software, preservando o comportamento externo, com o objetivo de melhorar o código, tornando-o mais legível e mais aderente às manutenções pelas quais deverá passar. Entretanto é necessário a busca pelo equilíbrio da refatoração, que deve ser disciplinada para manter a qualidade do código escrito.
Deve-se observar que o Teste Unitário é a base para os Testes de Integração, pois trazem um feedback imediato, encorajando o desenvolvedor a executá-los mais frequentemente, até a execução de todos os testes do sistema em um servidor de integração contínua.
É vital que existam testes automatizados que sejam fáceis e rápidos de serem executados, pois caso contrário, os testes serão executados poucas vezes e os bugs serão descobertos de forma tardia no ciclo de desenvolvimento.

2.3. Revisão de código fonte

O melhor período para se encontrar falhas em código fontes é durante a codificação.
Conforme relatado em [Jones, Capers 2007], inspeções de código formais são cerca de duas vezes tão eficiente quanto qualquer forma conhecida de testes para encontrar profundamente falhas sutis de programação, e são o único método conhecido que tem uma média acima de 80 por cento na eficiência de remoção de defeitos.
Infelizmente, há dificuldades em convencer desenvolvedores e gestores sobre tais vantagens.Gestores preocupam-se sobre a demanta de tempo utilizada para esta atividade, que conflita com o cronograma do projeto, sem claro considerar o ganho obtido pela busca não tardia de falhas de programação, por outro lado, desenvolvedores sentem-se desconfortáveis com revisões de código, principalmente por atingir seus egos e demonstrar suas falhas. Aliado a testes unitários consisos e a rigorosas sessões de revisão de código, é possível concluir uma tarefa livre de falhas antecipamente a inclusão dela no sistema através do controle de versão.
Há alguns métodos utilizados para revisão de código.

- **Revisão Tardia:** É realizado um evento em que será realizado uma sessão de revisão de código por todas as equipes da empresa, o que demonstra não ser a forma mais eficaz de realizar as revisões de código, porque grandes equipes de avaliação tendem a entrar em longas discussões e uma revisão muito ampla não só pode ser desnecessário, mas pode até ser prejudicial para o progresso do projeto.
- **Jogo Pick-up:** Assim que algum código é escrito, compilado, testado e pronto para o *commit*, o código é capturado para análise por outro desenvolvedor. As revisão de *pré–commit* são designados para serem rápidos e sem extensas discussões, o que garante um código é aceitável.Para eliminar quaisquer rotinas comportamentais ou vícios, o ideal é que a revisão seja realizada por desenvolvedores distintos a cada semana.
- **Programação em Par:** Utilizar a programação sempre em duplas, onde um desenvolvedor fica no conduz a programação, utlizando o teclado e um outro desenvolvedor senta-se ao lado e atua como navegador, indicando melhorias e falhas. De vez em quando, alternam-se os papéis. Com isso, há um aumento de atenção e crítica sobre o desenvolvimento de uma funcionalidade. O ideal é integração entre

desenvolvedores mais experientes, com menos os experientes, para um aprendizado mútuo.

Outra técnica útil é utilizar um checklist para a revisão de código, por exemplo:

- É possível ler e compreender o código ?
- Há alguma possível falha ?
- Será que o código tem qualquer efeito indesejável em outras partes da aplicação ?
- Existe alguma duplicação de código ?
- É possível aplicar melhorias ou refatoração para melhorar o código ?

Além disso, é possível utilizar ferramentas de análise de código, como por exemplo, o Gerrit `https://code.google.com/p/gerrit/`, que integradas ao sistema de controle de versão executam revisões em grupo do tipo *post–commit*, isto é, o código recém enviado ao repositório sofre uma análise de um grupo de desenvolvedores, que aprovam ou não o código, indicando melhorias e alterações.

2.4. Pre-Tested Commits

A proposta do *pre-tested commit*, é dedicar um *repositório primário* ligado ao servidor de Integração Contínua para que execute os testes unitários e de integração, antes de ir para o repositório principal.
Isto é necessário porque normalmente desenvolvedores não executam testes após a conclusão de uma tarefa, enviando para o repositório uma possível falha. Este tipo de ação é justificada para um sistema muito complexo, com *n* testes de integração, e que demandam muito tempo, até horas para completar todo o processo de testes.
Obviamente. é mandatório que o desenvolvedor execute seus testes unitários, mas há também, uma possibilidade que seus testes não cubram uma falha gerada acidentalmente em outra parte do sistema. Portanto, para que não ocorra quebra do repositório principal o *pre-tested commit* é utilizado.
A utilização dele está fortemente ligado ao sistema de controle de versão, como o TeamCity `http://www.jetbrains.com/teamcity/` ou a um conjunto de ferramentas configuradas para tal função, como por exemplo a utilização do Git (SCV), Jenkins (Integração Contínua), Google Test (Testes unitários) e o Gerrit (Revisão de Código).

3. Workflow para C++ com TDD

4. Avaliação

5. Conclusões

Referências

[Boulic and Renault 1991] Boulic, R. and Renault, O. (1991). 3d hierarchies for animation. In Magnenat-Thalmann, N. and Thalmann, D., editors, *New Trends in Animation and Visualization*. John Wiley & Sons ltd.

[Knuth 1984] Knuth, D. E. (1984). *The TEX Book*. Addison-Wesley, 15th edition.

[Smith and Jones 1999] Smith, A. and Jones, B. (1999). On the complexity of computing. In Smith-Jones, A. B., editor, *Advances in Computer Science*, pages 555–566. Publishing Press.

[Erdogmus, H., Morisio, M., and Torchiano, M. 2005] Erdogmus, H., Morisio, M., and Torchiano, M.(2005). On the effectiveness of test–first approach to programming. IEEE Transactions on Software Engineering, pages 226–237

[Jones, Capers 2007] Jones, Capers (2007). Estimating Software Costs. McGraw-Hill, New York, 2nd edition.