

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

Helton Luiz Marques¹

¹Univali – 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

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 aumentando aos poucos, porém com a certeza que há correção 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 de mesmo de escrever algum código. Desde a implementação, o projeto terá um nível de qualidade maior, já que esta técnica força pensar em problemas e suas soluções antes de qualquer código existir.

Além disso, traz 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 (Testes 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 a prevenção de defeitos.
- Menor tempo de debug: Com um menor número de falhas, obtêm-se um menor tempo de debug.
- A Documentação não mente: Com testes bem estruturados, é possível analisar a qualidade da documentação. Testes bem estruturados trazem uma forma mais visível de documentação, onde um exemplo funcional supri uma documentação ex-cassa.
- Paz de espírito: Um código exaustivamente testado com um conjunto abrangente de testes de regressão traz estabilidade e confiança aos desenvolvedores, trazendo conforto e melhores fins de semana.
- Aperfeiçoamento do design: Um bom design é um design testável. Funções muito grandes, forte acoplamento e condicionais complexas, levam a um 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: TDD traz imediatamente uma gratificação aos desenvolvedores. A cada codificação testada, traz a garantia de que o software está funcionando.

2. Test-First e Test-Last

Há duas abordagens em relação aos testes 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 uma 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 pareceu haver diferença entre os dois métodos. Houveram evidências de que a qualidade externa estaria mais relacionada ao número de testes escritos do que quando estes são escritos.

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, possivelmente 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 (Test-First e Test-Last). 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.

3. 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.

4. 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, é difícil convencer desenvolvedores e gestores sobre estas vantagens. Gestores preocupam-se sobre a demanda de tempo utilizada para esta atividade, que conflita com o cronograma do projeto, por outro lado, desenvolvedores não sentem-se confortáveis com revisões de código, principalmente por atingir seus egos e demonstrar suas falhas. Por outro lado, projetos que utilizam a revisão de código, trazem resultados extraordinários.

Aliado a testes unitários consisos e a rigorosos revisões de código, é possível concluir uma tarefa livre de falhas antecipadamente a inclusão dela no sistema de controle de versão. Há alguns métodos utilizados para revisão de código.

- **Revisão Tardia.** É realizado um evento em que será realizado a revisão de toda a equipe, ou até equipes responsáveis por outras áreas, para análise e correção de todo o sistema. Mas isso pode não ser a forma mais eficaz de realizar as revisões de código e não parece particularmente ágil. 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 foi escrito, compilado, testado e está pronto para check-in, o código é capturado para análise de outro desenvolvedor. As revisões de commit são designados para ser rápidos e sem extensas discussões, para garantir que o código é aceitável antes de inseri-lo ao projeto. Para eliminar quaisquer rotinas comportamentais ou vícios é ideal que a revisão seja realizada por desenvolvedores distintos a cada semana.
- **Programação em Par.** Utiliza a programação sempre em duplas, onde uma pessoa fica no teclado (o condutor) e uma pessoa se senta para trás e atua como navegador. De vez em quando, alternam-se os papéis. Aumentando a atenção e crítica sobre o desenvolvimento de uma funcionalidade. O ideal é integração entre desenvolvedores mais experientes, com menos os experientes.

Outra técnica útil é criar 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 ?

4.1. Subsections

The subsection titles must be in boldface, 12pt, flush left.

5. Figures and Captions

Figure and table captions should be centered if less than one line (Figure 1), otherwise justified and indented by 0.8cm on both margins, as shown in Figure 2. The caption font must be Helvetica, 10 point, boldface, with 6 points of space before and after each caption.

In tables, try to avoid the use of colored or shaded backgrounds, and avoid thick, doubled, or unnecessary framing lines. When reporting empirical data, do not use more



Figura 1. A typical figure

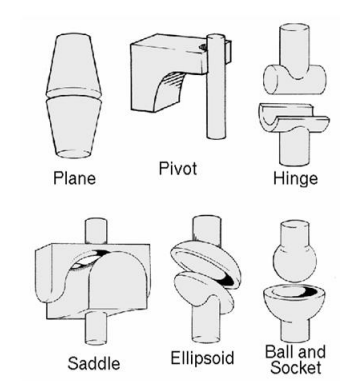


Figura 2. This figure is an example of a figure caption taking more than one line and justified considering margins mentioned in Section 5.

decimal digits than warranted by their precision and reproducibility. Table caption must be placed before the table (see Table 1) and the font used must also be Helvetica, 10 point, boldface, with 6 points of space before and after each caption.

6. Images

All images and illustrations should be in black-and-white, or gray tones, excepting for the papers that will be electronically available (on CD-ROMs, internet, etc.). The image resolution on paper should be about 600 dpi for black-and-white images, and 150-300 dpi for grayscale images. Do not include images with excessive resolution, as they may take hours to print, without any visible difference in the result.

7. Considerações e Trabalhos Futuros

Bibliographic references must be unambiguous and uniform. We recommend giving the author names references in brackets, e.g. [Knuth 1984], [Boulic and Renault 1991], and [Smith and Jones 1999].

Tabela 1. Variables to be considered on the evaluation of interaction techniques

	Chessboard top view	Chessboard perspective view
Selection with side movements	6.02 \pm 5.22	7.01 \pm 6.84
Selection with in- depth movements	6.29 \pm 4.99	12.22 \pm 11.33
Manipulation with side movements	4.66 \pm 4.94	3.47 \pm 2.20
Manipulation with in- depth movements	5.71 \pm 4.55	5.37 \pm 3.28

The references must be listed using 12 point font size, with 6 points of space before each reference. The first line of each reference should not be indented, while the subsequent should be indented by 0.5 cm.

Referências

- 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, D. E. (1984). *The T_EX Book*. Addison-Wesley, 15th edition.
- 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). On the effectiveness of test–first approach to programming. *IEEE Transactions on Software Engineering*, pages 226–237
- Jones, Capers (2007). *Estimating Software Costs*. McGraw-Hill, New York, 2nd edition.