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 do acréscimo de qualidade sobre o produto de software.

Além dos testes unitários, há também a revisão de código, que integrado a um sistema de testes automatizados e um sistema de controle de versão obtêm-se uma metodologia de trabalho eficaz e que traz velocidade e segurança no desenvolvimento de sistemas, principalmente em C/C++ onde 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, tem-se o *feedback* imediato a cada refatoração, e também a diminuição do *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, com 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:

- **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 escassa.
- **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 implementando 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, 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. Além disso, o risco da abordagem *Test-Last* está na perda do benefício da programação por iteração, 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% 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 e com conflitos com o cronograma do projeto, sem claro observar mais atentamente ao 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 concisos e a rigorosas sessões de revisão de código, é possível concluir uma tarefa livre de falhas antecipamente a inclusão dela no sistema de controle de versão.
Conforme [Subramaniam, Hunt 2006] Há alguns métodos utilizados para revisão de código, citados abaixo:

- **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 *pre-commit* são designados para serem rápidos e sem extensas discussões,

o que garante um código aceitável. Para eliminar quaisquer rotinas comportamentais ou vícios, o ideal é que a revisão seja realizada por desenvolvedores distintos a cada semana, através de um rodízio.

- **Programação em Par:** Utilizar a programação sempre em duplas, onde um desenvolvedor o 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], 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 ou um *branch* dedicado ao servidor de integração contínua para que execute os testes unitários e de integração, antes de ir para o repositório principal ou *branch* principal.
Isto é necessário porque há a possibilidade de desenvolvedores não executarem testes unitários ou funcionais após a conclusão de uma tarefa, enviando para o repositório uma possível falha. Este tipo de ação pode ser 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] 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

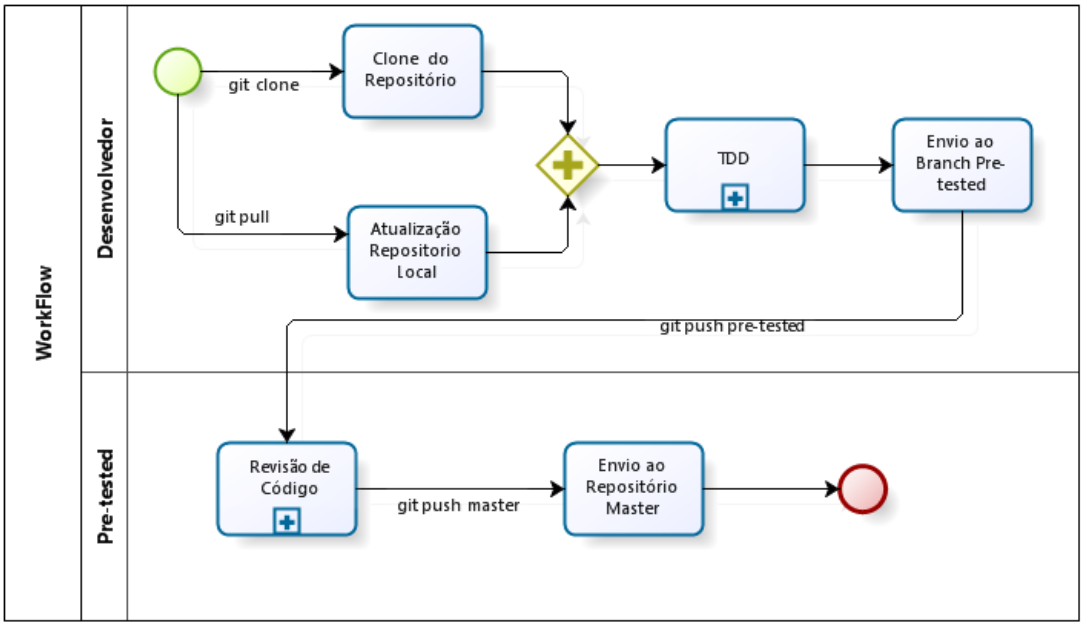

Figura 1. Workflow

O *workflow* apresentado na Figura 1 demonstra a integração entre o TDD e a utilização de *pre-tested commits*, que pode ser configurado em um repositório separado, ou conforme a política de versionamento adotado, através de um *branch* dedicado a availação e testes unitários e de integração, para trazer uma garantia do funcionamento pleno das tarefas implementadas, sem gerar também impacto no repositório principal.

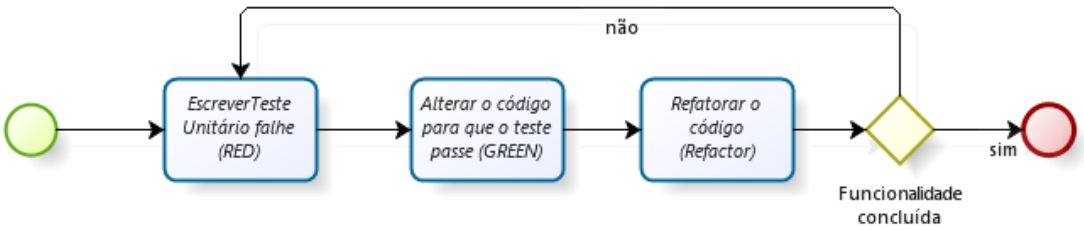

Figura 2. Workflow TDD

Na Figura 2 encontra-se o teste no ambiente local do desenvolvedor, que é responsável por implementar os testes unitários, executá-los, além de efetuar os testes funcionais para validação da tarefa.
Uma metodologia sugerida, é criar um branch local para a resolução de falhas, ou implementação de uma nova tarefa, aproveitando a excelente capacidade do merge do sistema de controle de versão Git. Segue o procedimento:

1. Clone do repositório principal: `git clone "repositório"`

2. Criar um branch para executar as alterações: `git branch -d "funcionalidadeX"`.
3. Realizar as alterações, executar os testes unitários localmente, e realizar commits: `git commit -a`.
4. Retornar ao repositório anterior: `git checkout "repositório"`.
5. Atualizar o repositório: `git pull`.
6. Merge entre o repositório principal e o branch ”funcionalidadeX”: `git merge "funcionalidadeX"`.
7. Resolução de conflitos e execução dos testes unitários.
8. Enviar o código ao repositório: `git push origin`.

Já na Figura 3 apresenta-se o workflow utilizado para a execução dos testes integrados utiizando o Jenkins, e na próxima ação a revisão de código pelos outros desenvolvedores utilizando o Gerrit. Após a votação, modificações caso necessário, e a aprovacão o código é enviado ao repositório ou *branch* principal.
Consequentemente após este ciclo, há um código muito mais estável e seguro aplicado ao repositório principal.

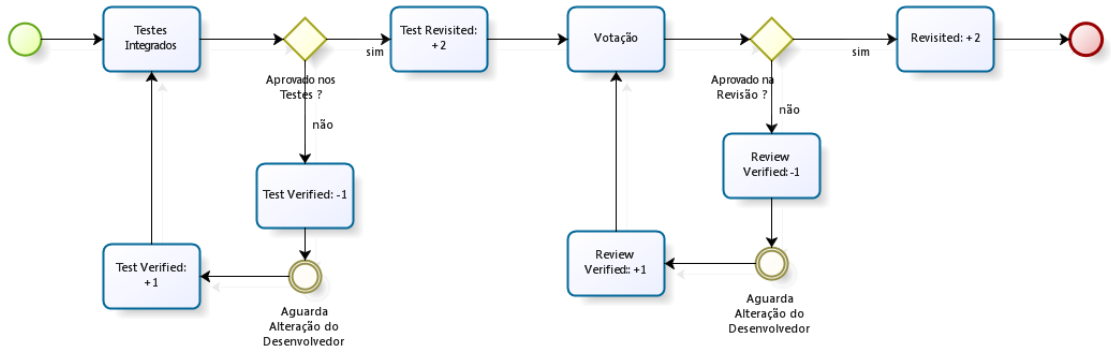

Figura 3. Workflow Revisão de Código

4. Avaliação

Escolhemos como referência para este trabalho demonstrar a utilização do framework criado pelo Google para Testes Unitários para C/C++, o Google Test. Este framework é utilizado pela própria equipe do Google para testes unitários em seus projetos em C/C++. Alguns relatados na página do projeto são:

- Chromium: `http://www.chromium.org`
- LLVM compiler: `http://llvm.org`
- Protocol Buffers: `http://code.google.com/p/protobuf/`

Baseado na arquitetura xUnit, portável para diversas arquiteturas (Linux, Mac OS X, Windows, Cygwin, Windows CE, e Symbian), e as principais funcionalidades para execução de testes unitários são, *assertions*, *death tests*, *value- and type-parameterized tests* e integração com o Jenkins para teste contínuo através de um XML. Uma outra característica muito interessante deste framework é que foi projetado para ser leve (*lightweight*) e com alta performance.
Em relação aos *asserts* a diferença básica entre eles é que no `ASSERT_*` o teste é

abortado ao encontrar uma diferença, e no `EXPECT_*` o teste continua, mas demonstra uma falha no relatório de testes. Alguns principais testes são baseados testes binários e testes de strings.

4.1. Caso de Uso

Como caso de uso partimos para um exemplo simples de integração do Google Test(C++) com um projeto em C. A grande vantagem deste framework é a flexibilidade da linguagem atender as duas arquiteturas (C e C++), e sendo possível utilizá-la em projetos de pequeno e grande porte, e também atendendo aos desenvolvedores de sistemas embarcados. Outra ferramenta utilizada no projeto é o make que é utilitário para verificar quais arquivos precisam ser compilados, além de automatizar o link de alguns diretórios, por exemplo, o diretório de arquivos de header (*.h).
O projeto tem a seguinte estrutura de diretórios:

- *include*: diretório com arquivos de header (*.h)
- *cadastro.h*: arquivo contendo os protótipos das funções do arquivo cadastro.c.
- *src*: diretório com arquivos de código fonte (*.c).
- *cadastro.c*: arquivo com as funções de cadastro.
- *main.c*: arquivo principal, obrigatório para projetos em C.
- *test*: diretório com arquivos com código de testes cpp.
- *cadastroTest.cpp*: arquivo com os testes das funções do arquivo cadastro.c.
- *main.cpp*: arquivo principal, obrigatório para utilização do Google Test.
- *Makefile*: responsável pela compilação dos arquivos de testes.
- *Makefile*: arquivo responsável por executar a compilação dos arquivos fontes, e arquivos de testes.

  Para este exemplo, temos a seguinte regra:

Valor do Imóvel
O itaú realizado o financiamento de imóveis residenciais a partir de `R$ 62 mil`, sem limite máximo. Foi criado um pequeno algoritmo que recebe o valor do financiamento, no formato de string, verificar se a string é válida (diferente de vazio), aplicando uma expressão regular para facilitar a validação da string, e verificar se o valor encontra-se acima de `R$62 mil` (`VALOR_IMOVEL_MIN`), retornando o valor já no formato float em caso de sucesso, ou o valor negativo em caso de erro.

  A partir da regra, temos a algoritmo do arquivo *cadastro.c*:

```c
#include <string.h>
#include <regex.h>

#include "cadastro.h"

#if (DEBUG >= 1)
#define DEBUG_MESSAGE(fmt, args...)    (fprintf(stderr, fmt, ##args))
#else
#define DEBUG_MESSAGE(fmt, args...)
#endif  /* (DEBUG >= 1) */

int check_regex(const char *er, const char *txt)
{
  regex_t regex;
  int ret;

  /** verify string to verify */
  if (!txt || !strlen(txt)) {
    return -1;
  }

  /** verify regular expression */
  if (!er || !strlen(er)) {
    return -1;
  }

  regcomp(&regex, er, REG_EXTENDED|REG_NOSUB);
  ret = regexec(&regex, txt,  0, NULL, 0);
  regfree(&regex);

  return ((ret != 0) ? -1 : 0);
}

float get_valor_imovel(const char * valor_imovel)
{
  float temp;
  /* expressão regular para convesão de strings para float */
  const char float_er[] = "[1-9][0-9]*\\.?[0-9]*([Ee][+-]?[0-9]+)?";

  if (check_regex(float_er, valor_imovel) != 0) {
    DEBUG_MESSAGE("\t\tvalor imóvel inválido %s\n", valor_imovel);
    return -1;
  }

  temp = atof(valor_imovel);
  if (temp < VALOR_IMOVEL_MIN) {
    DEBUG_MESSAGE("\t\tvalor do imóvel (%0.2f) abaixo do valor mínimo\n
        ", temp);
    return -2;
  }
  return temp;
}
```

Os testes aplicados sobre as funções contidas no arquivo cadastro.c, são colocados em um arquivo com o mesmo nome, com a extensão _test.cpp, esta convenção serve para uma melhor visualização dos arquivos de testes unitários.
Há duas funções de testes gerais, e dentro de cada uma há execução de diversos testes. É obrigatório a inclusão do header `<gtest/gtest.h>`. A função `EXPECT_FLOAT_EQ` verifica o retorno da função `get_valor_imovel`, igual a `R$ 65 mil`, conforme demonstrado no algoritmo abaixo:

```
1  #include <gtest/gtest.h>
2
3  #include "cadastro.h"
4
5  TEST(cadastroTest, cadastroCheckRegexTest)
6  {
7    const char float_er[] = "[1-9][0-9]*\\.?[0-9]*([Ee][+-]?[0-9]+)?";
8
9    EXPECT_EQ(0, check_regex(float_er, "65000.00"));
10   EXPECT_EQ(-1, check_regex(NULL, NULL));
11 }
12
13 TEST(cadastroTest, cadastroValorImovelTest)
14 {
15   EXPECT_EQ(-1, get_valor_imovel(NULL));
16   EXPECT_EQ(-2, get_valor_imovel("6300.00"));
17
18   EXPECT_FLOAT_EQ(65000.00, get_valor_imovel("65000.00"));
19 }
```

O projeto contem 2 arquivos de makefile, um arquivo encontra-se dentro do diretório test, e é executado para compilar os arquivos de testes.O arquivo Makefile na raiz do projeto compila o projeto (make all), e chama o arquivo Makefile contido no diretório de testes (make test).
Uma observação, quando citamos 'compilar' pelo arquivo Makefile, na verdade, o script make é responsável por chamar o compilador repassando de forma correta os parâmetros, automatizando a tarefa de compilação.

Abaixo o arquivo de compilação do diretório de Test.

```
CPP=g++

ALL_C := $(wildcard ../src/*.c)
ALL_CPP := $(wildcard *.cpp)

SOURCES := $(filter-out %main.c,$(ALL_C))
OBJS := $(addprefix ,$(SOURCES:.c=.o))
OBJS += $(addprefix ,$(ALL_CPP:.cpp=.o))

CFLAGS += -fprofile-arcs -ftest-coverage
CFLAGS += -I../include

LDFLAGS += -lgtest -lgcov -lpthread -lgtest_main

TARGET := json_test

# make all
all: $(TARGET)

$(TARGET): $(OBJS)
	@echo "[BIN] $@"
	@$(CPP) $^ $(LDFLAGS) -o $@
	@./$(TARGET)

%.o: %.c
	@echo "[CPP] $@"
	@$(CPP) $(CFLAGS) -o $@ -c $<

%.o: %.cpp
	@echo "[CPP] $@"
	@$(CPP) $(CFLAGS) -o $@ -c $<

clean:
	@echo "rm -rf $(OBJS)"
```

Ao executar os testes temos:

```
[CPP] ../src/cadastro.o
[CPP] cadastro_test.o
[CPP] main.o
[BIN] json_test
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from cadastroTest
[ RUN      ] cadastroTest.cadastroCheckRegexTest
[       OK ] cadastroTest.cadastroCheckRegexTest (0 ms)
[ RUN      ] cadastroTest.cadastroValorImovelTest
[       OK ] cadastroTest.cadastroValorImovelTest (0 ms)
[----------] 2 tests from cadastroTest (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (1 ms total)
[  PASSED  ] 2 tests.
```

Figura 4. Execução dos Testes Unitários

5. Conclusão

Neste trabalho apresentamos como funciona um ciclo de desenvolvimento utilizando TDD e como ele resulta em melhores resultados através de um melhor entendimento prévio do problema, avanço incremental do código criado, mais confiança no código escrito e uma melhoria no design projeto de software onde há a busca por um baixo acoplamento para facilitar o desenvolvimento dos testes unitários.

Para o uso do TDD são necessárias bibliotecas e ferramentas que possibilita a escrita dos testes e a sua execução. Neste trabalho apresentamos o Google Test, framework para as linguagens C e C++, que tem como vantagem ser leve, portável e gratuito. Detecta automaticamente os testes escritos, agilizando o desenvolvimento e evitando que sejam escritos testes que nunca serão executados.

Através de um exemplo prático, mostramos como adotar uma estratégia de desenvolvimento dirigida a testes e - principalmente - que este processo não precisa ser burocrático e demorado. Com uma ferramenta simples, como o Google Test, em poucos minutos qualquer desenvolvedor pode começar a escrever testes e, a partir deles, o código propriamente dito.

Referências

[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.

[Subramaniam, Hunt 2006] Subramaniam, Hunt (2006). Practices of an Agile Developer. The Pragmatic Bookshelf, Texas.

[TeamCity] Continous Integration for Everybody http://www.jetbrains.com/teamcity/

[Git] Free and open source distributed version control system http://git-scm.com/

[Jenkins] An extendable open source continuous integration server http://jenkins-ci.org/

[Google Test] Google C++ Testing Framework https://code.google.com/p/googletest/

[Gerrit] Gerrit Code Review https://code.google.com/p/gerrit/