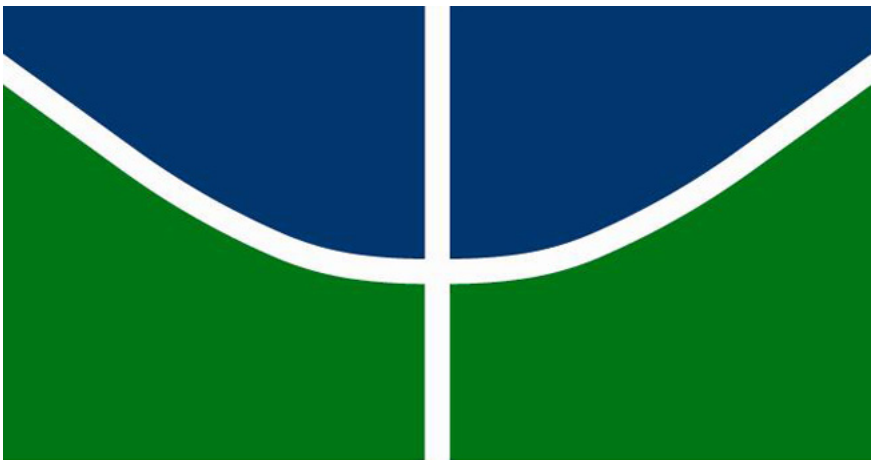

Universidade de Brasília - UnB

Instituto de Exatas

Departamento de Ciência da Computação

Rodrigo Chaves - 13/0132624

Gabriel Mesquita - 13/0009121

Test-driven Development

Brasília - DF

2016

Gabriel Mesquita de Araujo
Rodrigo de Araujo Chaves

Test-driven Development

Dissertação sobre por que Test-driven development
é pratica que melhora a qualidade
final do software apresentanda à disciplina
de Engenharia de Software da Universidade de Brasília.

Sumário

1 Introdução

Hoje, no Brasil, não há dados confiáveis sobre quantos reais são perdidos por software defeituosos, mas especialistas afirmam que 8 bilhões de reais é um valor bem próximo da realidade brasileira. Um exemplo que pode demonstrar o prejuízo de um software mal fabricado pode causar foi a sonda espacial Mars Climate Orbiter, perdida na atmosfera de Marte por errar a unidade em um cálculo, misturando as medidas de pés e metros.

Apesar desses danos de falhas serem custosos quando o software é colocado em produção, essas falhas também podem causar dor de cabeça as desenvolvedores e todos os stakeholders envolvidos durante a fase de desenvolvimento.

2 O que é Test-driven Development

Test-driven development é uma prática de desenvolvimento de software que tem sido usada esporádicamente por décadas. Com essa prática, um engenheiro de software passa por ciclos entre escrever um teste de unidade que falha e escrevendo a implementação do software para passar nesses testes. Test-driven development tem recentemente ressurgindo como uma prática crítica possibilitando metodologias de desenvolvimento ágil de software.

Quando discutimos sobre TDD, é considerado um conjunto de tarefas requiridas que podem ser implementadas em poucos dias ou menos. Na imagem 1, engenheiros de software produzem código de produção através de rápidas interações como as que seguem:

1. O primeiro passo é adicionar um teste simples o qual é suficiente para a suíte de teste falhar.
2. Depois executamos nossa suíte para confirmar que os testes realmente estão falhando.
3. Agora atualiza-se o código funcional afim de passar no novo teste.
4. Executamos a suíte de teste para verificarmos se agora realmente passamos no novo teste.

5. Agora com o teste passando: são removidos as duplicações de código afim de limpar o código.

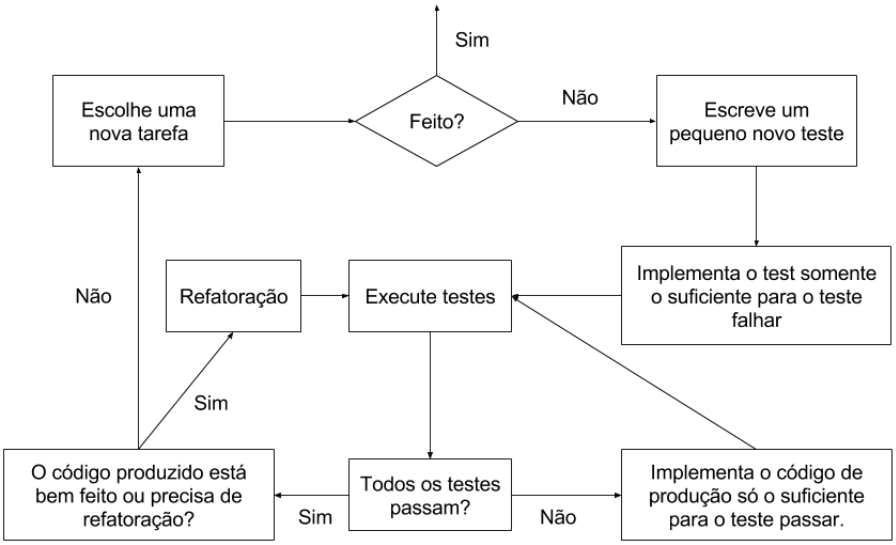

Figure 1: Fluxograma do TDD

3 Automação de Testes

No desenvolvimento orientado à Testes, uma ferramenta muito imporantante são os frameworks de testes automatizados para a criação de objetos orientados a testes de unidade.

Em Ruby, por exemplo, o framework `RSpec` é comumente usado onde cada Classe Pública tem um class correspondente `RSpec.describe <NomeDaClasse>`. Para cada método público dessa classe, é criado um teste `it " <NomeDoMétodo> "` onde o contrato do método é testado. Além disso, outros `it " <NomeDaFuncionalidade> "` são criados para validar o comportamento desse método com diferentes parametros. A cada execução de um `it`, um método `before` é chamado e esse é responsável por criar as variáveis de teste, criar a instância de `<NomeDaClasse>` que serão usadas em um ou mais métodos. Além disso, pode-se criar contextos onde um conjunto mais especifico de testes pode ser executado dentro de uma `context` e definir variáveis para esse contexto.

Uma prática comum dos framework de automação de testes é usar cores para expressar o resultados da suíte de testes. Caso algum teste não passe, a cor vermelha é usada para indicar a atenção do desenvolvedor. Caso todos os testes passem, usamos a cor verde para indicar os resultados dos testes. A imagem a baixo demonstra de forma

simplificada qual é o ritmo comum dentro do ciclo do Test-driven Development.

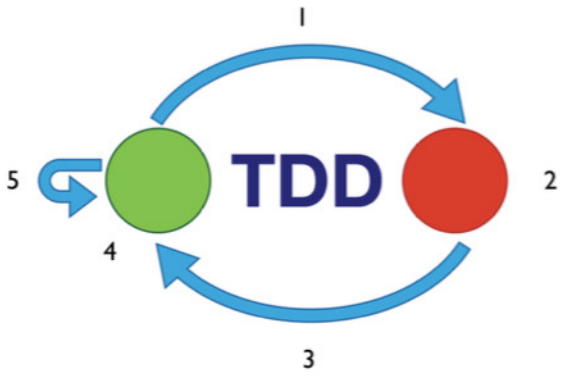

Figure 2: Colograma do TDD

4 Um exemplo de TDD

Para demonstrar como fazemos TDD, iremos usar o framework RSpec em Ruby para criar uma calculadora. Iniciamos nossa calculadora criando nosso primeiro teste:

```
# Arquivo Calculator_rspec.rb

RSpec.describe Calculator do
end
```

Executando esse teste, teremos esse resultado. Para executar a suíte de teste do RSpec, chamamos o comando rspec no terminal:

```
$ rspec
uninitialized constant Calculator (NameError)
```

Com esse, identificamos que o próximo passo é criar a classe `Calculator`

```
# Arquivo Calculator.rb

class Calculator
end
```

Executando a suíte de teste novamente.

```
$ rspec
No examples found.
```

```
Finished in 0.00043 seconds
(files took 0.15324 seconds to load)
0 examples, 0 failures
```

Passando pelo nosso primeiro teste, voltamos a escrever os testes. Vamos escrever um teste que espera que nossa calculadora faça uma soma simples de 1 mais 2 e encontre o resultado 3.

```
# Arquivo Calculator_rspec.rb

RSpec.describe Calculator do
  describe "#add" do
    it "returns the sum of its arguments" do
      expect(Calculator.new.add(1,2)).to eq(3)
    end
  end
end
```

Esse novo teste é descrito dentro de um bloco add onde todos os compartamentos desse método são testados. Executando a suíte de testes novamente.

```
$ rspec
F

Failures:

  1) Calculator#add returns the sum of its arguments
     Failure/Error: expect(@calculator.add(1,2)).to eq(3)

     NoMethodError:
       undefined method `add' for nil:NilClass
     # ./spec/calculator_spec.rb:10:in `block (3 levels)
     in <top (required)>'

Finished in 0.00069 seconds (files took 0.15787 seconds
to load) 1 example, 1 failure
```

```
Failed examples:

rspec ./spec/calculator_spec.rb:9 # Calculator#add returns
the sum of its arguments
```

Temos um novo erro. Agora temos um teste em vermelho, podemos codificar a nossa class `Calculator`. Vamos fazer um método que retorne a soma de dois parâmetros.

```
# Arquivo Calculator.rb

class Calculator
  def add a, b
    return a + b
  end
end
```

Executa-se novamente a suíte de testes.

```
$ rspec
.

Finished in 0.00158 seconds (files took 0.19148 seconds to
load)
1 example, 0 failures
```

Agora iremos testar se o nosso método add consegue somar um número negativo.

```
RSpec.describe Calculator do

  describe "#add" do
    it 'returns the sum of its arguments ' do
      expect(Calculator.new.add(1,2)).to eq(3)
    end

    it 'can calculate with negative param' do
      expect(Calculator.new.add(1, -2)).to eq(-1)
```

```
    end
  end
end
```

Executando novamente a suíte de testes.

```
$ rspec
..

Finished in 0.00231 seconds (files took 0.18369 seconds to
load)
2 examples, 0 failures
```

5 Design de Software em TDD

O Test-driven Development permite que o desenvolvedor que está trabalhando em um módulo pense como serão as responsabilidades, interfaces, serviços os quais serão disponibilizados por esse módulo, enquanto escreve os testes. Depois, quando vai escrever o código de produção, pode se preocupar somente em implementar o necessário para passar nos testes já feitos. Fazendo assim, criamos um ritmo entre codificação e teste até que todos os testes criados sejam implementados.

Pensando no design do projeto, os desenvolvedores do projeto podem criar classes e módulos mais coesos e menos acopladas por que, durante a fase de elaboração dos testes, conseguem visualizar a arquitetura geral da aplicação e melhorar como o componente que irão desenvolver conversa com os outros componentes. Em alguns testes, os desenvolvedores os escrevem usando o componente em desenvolvimento como já estivesse pronto e validam se esse consegue se comunicar com suas interfaces.

Em Test-driven Development, o código desenvolvido é mantido dentro do controle intelectual do desenvolvedor, já que o próprio escreveu os testes e ele ou ela está fazendo continuamente pequenas alterações de design e decisões de implementação, aumentando as funcionalidades do programa em um certo ritmo contínuo.

6 Desenvolvimento Ágil e Test-driven Development

Test-driven development é uma prática sugerida dentro do "Extremming Programming", também muito conhecido por XP, por Kent Beck. XP é surgiu nos Estados Unidos e tem ganhado bastante espaço no desenvolvimento de software pois é um conjunto de valores, principios e práticas que fazem os softwares serem produzidos em menos tempo e de forma mais econômica que o habitual.

Desenvolvimento incremental é uma prática que não só traz benefícios. Ao se adicionar novas funcionalidade a um software, há um risco de falhas serem introduzidas. O XP adotou o uso de Test-driven Development como um mecanismo de proteção impedindo que algo que já estivesse funcionando fosse quebrado e não detectado. "O desenvolvimento orientado a testes é uma forma de lidar com o medo durante a programação (BECK, 2003)."

Os testes automatizados criados pelos desenvolvedores formam uma base de testes que pode ser executada sempre que necessário visualizar se tudo está funcionando normalmente. Isso não impede que falhas sejam inseridas, porém é uma forma de detecção possibilitando uma correção efetiva e barata, impedindo que bugs se acumulem com o passar do tempo.

7 Objetivos ao se praticar TDD

TDD é uma prática de desenvolvimento que tem por objetivo garantir a qualidade e a confiabilidade do produto o mais rápido possível. Ao decorrer do desenvolvimento, todo o código elaborado é desenvolvido em conjunto com uma suíte de testes automatizados. Esses testes permitem uma segurança maior ao desenvolvedor quando precisa mudar algo que já foi implementado ou precisa refatorar o código.

Além disso, desenvolvedores experientes em TDD podem analisar se os testes produzidos estão difíceis de serem feitos e podem fazer refactoring ou mudanças.

8 Depuração

Quando é detectado um defeito em um software, é necessário consertá-lo. Nesse momento, entramos na fase de depuração. Se observamos os programadores realizando suas atividades, iremos perceber que eles levam mais tempo é depurando. Ou seja, procurando por falhas. Uma pequena parte do tempo de desenvolvimento é usado para a codificação propriamente dita. O restante do tempo é gasto projetando-se e entendendo o que deve ser feito.

Consertar a falha identificada é fácil, porém encontrar onde está acontecendo o problema é o que leva a fase de depuração ser tão longa. As equipes de desenvolvimento podem usar o Test-driven Development para serem capazes de identificar mais rapidamente essas falhas.

Além disso, ao se consertar uma falha local, existe uma possibilidade (entre 20 a 50 por cento) de se introduzir uma nova falha. As bases de testes automatizados facilitam a garantir que as correções feitas não introduzem novos problemas pois conseguem testar as outras funcionalidades do sistema ao ser executada.

9 Conclusão

Em vista dos fatos mencionados, é possível observar que o test-driven development sem dúvida auxilia o processo de desenvolvimento de software de diversas maneiras. Essa prática melhora a qualidade do software como já foi discutido e se utilizado corretamente diminui custos relacionados a correção de bugs ou refactoring de códigos complexos, boiler-plate e etc.

É de extrema importância mencionar que o TDD é uma prática útil, porém nem sempre é a melhor solução para um determinado problema. O contexto da empresa, do sistema e dos stakeholders devem ser analisados para avaliar se o TDD vai ser útil ou não. Ou seja, a falta desse tipo de análise pode gerar custos desnecessários ao desenvolvimento tanto monétarios quanto relacionados ao tempo de desenvolvimento.

10 Referências

Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, Laurie Williams, Realizing quality improvement through test driven development: results and experiences of four industrial teams. Disponível em http://link.springer.com/article/10.1007/s10664-008-9062-z#/page-1. Acessado em 26 de maio de 2016.

Andreas Augustin, Test-Driven Development: Concepts, Taxonomy, and Future Direction. Disponível em https://www.semanticscholar.org/paper/Test-Driven-Development-Concepts-Taxonomy-and-Janzen-Saiedian/bdcd570eb6a45d7a9107a18e25f54b741b92177f/pdf. Acessado em 26 de maio de 2016

Martin Fowler, Bill Venners: Test-Driven Development, A Conversation with Martin Fowler. Disponível em http://www.biology.emory.edu/research/Prinz/Cengiz/cs540-485-FA12/resources/testDrivenDev.pdf. Acessado em 30 de maio de 2016.

Mauricio Finavaro Aniche, Como a prática de TDD influencia o projeto de classes em sistemas orientados a objetos. Disponível em http://www.teses.usp.br/teses/disponiveis/45/45134/tde-31072012-181230/publico/dissertacao.pdf. Acessado em 30 de maio de 2016.

Roger Pressman, Software Engineering: A Practitioner's Approach, 7/e (McGraw-Hill, 2009), capítulo 3. Disponível em http://academic.brooklyn.cuny.edu/cis/sfleisher/Chapter_03_sim.pdf. Acessado em 1º de junho de 2016.

Vínicius Manhães Teles, Um estudo de caso da adoção das práticas e valores do extremme programming. Disponível em http://www.improveit.com.br/xp/dissertacaoXP.pdf. Acessado em 1º de junho de 2016.

Laurie Williams, E. Michael Maximilien, Mladen Vouk, Test-Driven Development as a Defect-Reduction Practice. Disponível em ftp://www.ufv.br/dpi/mestrado/TDD/willians_TDD_Defect-Reducion_Practice.pdf. Acessado em 1º de Junho de 2016.

André Faria Gomes, Agile, Desenvolvimento de Software com entregas frequentes e foco no valor de negócio. Casa do Código.