Conceitos e Benefícios do Test Driven Development

Eduardo N. Borges

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil
enborges@inf.ufrgs.br

Abstract. *This paper describes the development style called Test Driven Development (TDD). The origin, definition of the method, and some related research that is being currently carried through are boarded. The benefits of this boarding in terms of the implementation total time, final code quality and maintainability are presented. Finally, test scripts are demonstrated through a detailed example of the TDD use.*

Resumo. *Este artigo descreve o estilo de desenvolvimento de software orientado a teste chamado Test Driven Development (TDD). É abordada a origem do método, bem como sua definição e algumas pesquisas relacionadas que estão sendo realizadas atualmente. São apresentados os benefícios desta abordagem quanto ao tempo total de implementação, qualidade e manutenibilidade do código final. Por fim, são demonstrados scripts de teste através de um exemplo detalhado do uso do TDD.*

1. Introdução

Atualmente, as falhas de software são grandes responsáveis por custos e tempo no processo de desenvolvimento de software. Embora não seja possível remover todos os erros existentes em certa aplicação, é possível reduzir consideravelmente o número dos mesmos utilizando uma infra-estrutura de testes mais elaborada, que permita identificar e remover defeitos mais cedo e de forma mais eficaz.

Estes defeitos podem resultar de diversas causas como erros de conhecimento, comunicação, análise, transcrição, codificação, etc. Existem essencialmente três formas de tratar falhas de software:

1. Evitar falhas (*fault-avoidance*): com atividades apropriadas de especificação, projeto, implementação e manutenção sempre visando evitar falhas em primeiro lugar. Inclui o uso de métodos de construção de software avançados, métodos formais e reuso de blocos de software confiáveis.
2. Eliminar falhas (*fault-elimination*): compensação analítica de erros cometidos durante a especificação, projeto e implementação. Inclui verificação, validação e teste.
3. Tolerar falhas (*fault-tolerance*): compensação em tempo real de problemas residuais como mudanças fora da especificação no ambiente operacional, erros de usuário, etc.

Devido ao fato de *fault-avoidance* ser economicamente impraticável para a maioria das empresas, a técnica de eliminação de falhas geralmente é a adotada pelos fabricantes de software. Uma pesquisa publicada pelo *National Institute of Standards and Technology* (NIST) e pelo Departamento de Comércio dos Estados Unidos revela que os erros de software custam cerca de 60 bilhões de dólares à economia norte-americana a cada ano [NIST 2002]. Portanto, faz-se necessário o estudo de técnicas orientadas a testes, pois este pode proporcionar uma economia considerável para empresas de desenvolvimento e aumentar a qualidade do software produzido.

Visando esta economia e aumento da qualidade, o *Test driven development* (TDD) foi criado para antecipar a identificação e correção de falhas durante o desenvolvimento do software.

2. Definição

O TDD é um estilo de desenvolvimento de software ágil derivado do método *Extreme Programming* (XP) [Beck 2000] e do *Agile Manifesto* [Agile Alliance 2000]. É baseado também em técnicas de desenvolvimento utilizadas há décadas [Gelperin and Hetzel 1987] [Larman and Basili 2003]. A prática envolve a implementação de um sistema começando pelos casos de teste de um objeto. Escrevendo casos de teste e implementando estes objetos e métodos, surge a necessidade de outros métodos e objetos.

No TDD, desenvolvedores usam testes para guiar o projeto do sistema durante o desenvolvimento. Eles automatizam estes testes para que sejam executados repetidamente. Através dos resultados (falhas ou sucessos) julgam o progresso do desenvolvimento. Os programadores fazem continuamente pequenas decisões aumentando as funcionalidades do software a uma taxa relativamente constante. Todos estes casos de teste devem ser realizados com sucesso sucessivamente antes de o novo código ser considerado totalmente implementado.

O TDD pode ser visto como um conjunto de iterações realizadas para completar uma tarefa [Beck 2002]. Cada iteração envolve os seguintes passos, os quais são destacados na Figura 1.

- Escolher a área do projeto ou requisitos da tarefa para melhor orientar o desenvolvimento.
- Projetar um teste concreto tão simples quanto possível quando é requerida a orientação do desenvolvimento. Checar se o teste falha.
- Alterar o sistema para satisfazer este teste e outros possíveis testes.
- Possivelmente refatorar o sistema para remover redundância, sem quebrar nenhum teste.

Uma importante regra no TDD é: *"If you can't write a test for what you are about to code, then you shouldn't even be thinking about coding"* [Chaplin 2001]. Outra regra no TDD diz que quando um defeito de software é encontrado, casos de teste de unidade são adicionados ao pacote de teste antes de corrigir o código.

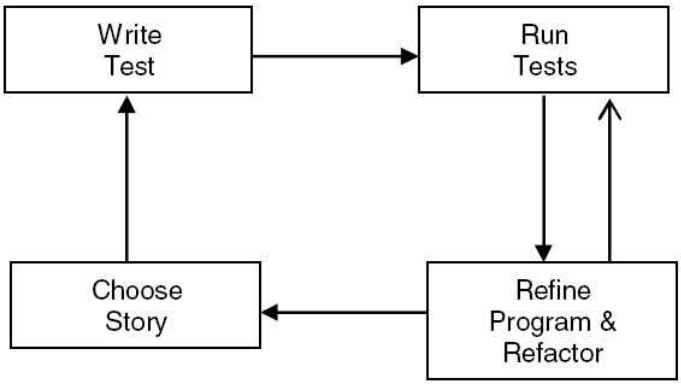

Figura 1. Passos do TDD.

Esta abordagem tende a ser mais coesa, pois o teste do código é parte íntima da codificação, e não um processo independente. Além disso, reduz o acoplamento dos componentes do software. Com isso, torna-se possível fazer decisões de projeto em cada estágio do desenvolvimento.

3. Pesquisa Relacionada

[Müller and Hagner 2002] conduziram um experimento estruturado comparando TDD com programação tradicional. Este experimento mediu a eficiência do TDD em termos do time de desenvolvimento, qualidade do código resultante e nível de entendimento do mesmo. Dada a especificação e a declaração de alguns métodos, os times deveriam completar o corpo destes métodos. O grupo que desenvolveu com TDD escreveu os casos de teste antes de começar a implementação. Já o grupo da programação tradicional escreveu os testes depois de completar o código.

A experiência ocorreu em duas fases, uma de implementação e outra de aceitação de teste. Ambos os grupos tiveram oportunidade de corrigir o código após a implementação. Apesar do tempo total de desenvolvimento dos métodos ter sido o mesmo para as duas equipes, o código do grupo TDD teve menos erros significantes quando reusado. Baseados nesse fato, os pesquisadores concluíram que a abordagem *test-first* aumenta substancialmente a qualidade do software e proporciona maior entendimento do código.

[Mugridge 2003] faz uma analogia entre o método científico e o TDD. O método científico é um modelo de como desenvolver teorias sobre quaisquer fenômenos. Primeiramente é definida uma hipótese. Após é projetado um experimento que comprove esta hipótese. Então, o experimento é executado e, baseado em seus resultados, são redefinidas as hipóteses até que seja comprovada a teoria. Esses passos são apresentados na Figura 2. Pode-se perceber a grande semelhança com as iterações no processo do TDD (ver Figura 1).

Escolher uma hipótese que comprove uma teoria é um problema bastante difícil. A primeira hipótese sugerida por um cientista pode ser extremamente subjetiva e não muito clara. Tal hipótese deve passar por refinamentos sucessivos até que seja definida uma teoria. No TDD, os primeiros testes são difíceis de escolher, mas ajudam a clarificar e especificar melhor o problema a ser resolvido. Portanto, o TDD é uma abordagem de desenvolvimento que segue as mesmas idéias do método científico, sendo

assim, uma técnica adequada para implementar grandes soluções que tendem a mudar com o passar do tempo. Além disso, é um método que permite o melhor entendimento do sistema.

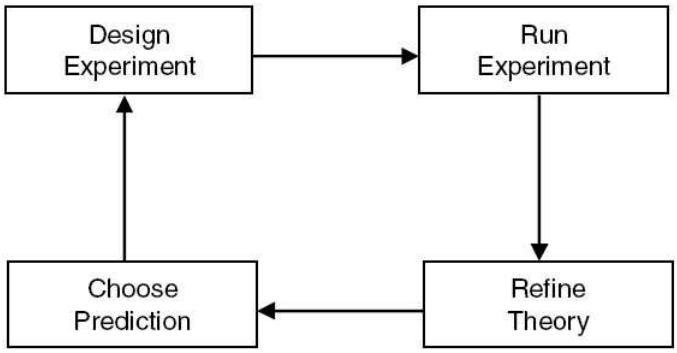

Figura 2. Passos do método científico.

Outro experimento realizado por [George and Williams 2004] mede a qualidade do software através do número de casos de teste caixa preta realizados com sucesso em ambas as abordagens de desenvolvimento. Todos os participantes implementam em duplas (*pair-programming*). São alocados de forma que cada dupla contenha um programador experiente na abordagem de desenvolvimento e outro iniciante. Apesar do tamanho do código desenvolvido ser relativamente pequeno (cerca de 200 linhas), e cada grupo ser formado por 6 duplas, os resultados foram bastante expressivos. A prática TDD forneceu código com qualidade superior quando comparada à prática de desenvolvimento tradicional. Além disso, programadores que praticam TDD codificaram mais rápido (16%) do que desenvolvedores que não a utilizaram. Este tempo foi medido em horas de trabalho considerando todo o processo de desenvolvimento.

O grupo de desenvolvimento da IBM responsável pela construção de *drivers* para vários dispositivos implantou o TDD como padrão no desenvolvimento de *releases* para uma nova plataforma [Williams et al 2003]. Foram criados 2400 casos de teste automatizados após a conclusão dos diagramas de classes e seqüência UML. O grupo realizou cerca de 40% de redução de tempo na verificação de defeitos em funções e métodos quando comparado ao grupo de desenvolvimento de *drivers* para plataformas já suportadas, o qual construía a 7ª *release* do *driver*. Isto comprovou o grande impacto de produtividade que o TDD proporciona.

4. Benefícios

TDD utiliza uma das técnicas do XP chamada *refactoring* [Beck 2000] para conseguir a compreensão do código e gerenciar a complexidade do mesmo. Como um grande programa ou sistema é continuamente modificado, ele torna-se muito complexo, sendo extremamente necessária a facilidade de manutenção [Lehman and Belady 1985]. Esta técnica é essencial para reduzir a complexidade do software e torná-lo manutenível.

A pequena granularidade do ciclo *test-then-code* dá um *feedback* contínuo ao programador. Falhas são identificadas mais rapidamente, enquanto o novo código é adicionado ao sistema. Assim, o tempo de depuração diminui compensado pelo tempo de escrita e execução dos casos de teste.

Alguns estudos indicam que cerca de 50% das tarefas no processo de manutenção de software são envolvidas no processo de entendimento do código [Corbi 1989]. A abordagem TDD ajuda na compreensão do programa porque os casos de teste e próprio código explicam melhor o funcionamento do sistema. Entretanto, esta prática permite somente o entendimento de uma parte do software. Para a compreensão da sua totalidade, é necessário que se faça uso de várias abstrações.

Os resultados de métodos ou funções são testados automaticamente. Estes valores são comparados aos resultados esperados ainda na etapa de codificação. Já na etapa de manutenção, as unidades de teste criadas anteriormente permitem avaliar mais facilmente novos defeitos que podem ter sido inseridos no software. Este benefício é essencial para o desenvolvimento e controle de novas *releases*, reduzindo a injeção de novas falhas no sistema.

5. Scripts de Teste

TDD geralmente faz uso de ferramentas e de um *framework* para a criação de unidades de teste orientadas a objetos. Um exemplo de *framework* é o JUnit (http://www.junit.org), onde unidades de caso de teste são adicionadas uma por classe pública, usualmente com o nome `<ClassName>TestCase`.

Para cada método público de uma classe existe pelo menos um teste `test<method>`. Múltiplos testes podem ser adicionados quando checam diferentes comportamentos do método. Antes da execução de cada teste, um método `setUp` é executado na classe `<ClassName>TestCase` para inicializar instâncias da `<ClassName>` as quais serão utilizadas na execução dos testes. Classes `TestCase` são usualmente agrupadas logicamente em classes `TestSuite`, as quais permitem a execução das classes `TestCase` em grupos.

5.1. Exemplo

O exemplo abaixo [Teles 2005] trata do algoritmo Crivo de Eratóstenes [Wikipédia 2006], o qual gera uma lista de números primos a partir de um número N que é passado como parâmetro de entrada. É utilizado o Framework JUnit e a IDE Java Eclipse.

Primeiramente é criada uma classe de teste geral com o teste mais simples possível que se possa imaginar. Espera-se que a classe gere uma string, com uma lista de números primos, separados por vírgula, menores ou iguais ao valor passado como argumento. Por exemplo, a busca por números primos até 10 retorna como resultado a string "2, 3, 5, 7".

```
import junit.framework.TestCase;
public class GeradorPrimosTeste extends TestCase {
  public void testePrimosGeradosAteNumeroDois() throws Exception {
    GeradorPrimos geradorPrimos = new GeradorPrimos();
    assertEquals("2", geradorPrimos.gerarPrimosAte(2));
  }
}
```

O método `assertEquals("2", geradorPrimos.gerarPrimosAte(2))` é o teste preliminar que checa se é possível gerar números primos corretamente até o número 2. O código ainda não é compilável pois não existe a classe `GeradorPrimos`. Isso é comum quando utilizada a abordagem TDD, pois escrever o código é conseqüência do sucesso ou falha dos testes. Então, é necessária a criação desta classe.

```
public class GeradorPrimos {
  public String gerarPrimosAte(int i) {
    return null;
  }
}
```

Isto já é suficiente para a compilação. Somente após a execução do teste que saberemos o que é necessário implementar. Executado o teste, verificamos que ele falha. Isto era de se esperar, pois o método `gerarPrimosAte()` ainda não resolve o problema. No TDD é importante ter certeza de que o teste falhará em situação que realmente deve falhar. Em seguida, é feita uma implementação simples do método `gerarPrimosAte()`.

```
public String gerarPrimosAte(int i) {
  return "2";
}
```

Executando o teste novamente percebe-se que tudo ocorre normalmente. Sendo assim, o programador tem a segurança de que o teste falha quando tem absoluta certeza de que deveria falhar, e que passa quando tem total confiança de que deveria passar. Isto ocorre através de soluções obviamente simples e depois, com a segurança de que o teste está correto, implementa-se o funcionamento complexo da classe. Esta técnica de incrementar soluções através de pequenos passos é conhecida como *baby steps* e é usada frequentemente no XP.

Então, cria-se novos casos de teste que geram a necessidade da implementação da classe `GeradorPrimos`. O código funciona para números até 3?

```
public void testePrimosGeradosAteNumeroTres() throws Exception {
  GeradorPrimos geradorPrimos = new GeradorPrimos();
  assertEquals("2, 3", geradorPrimos.gerarPrimosAte(3));
}
```

Executando o teste, naturalmente percebe-se que ele falha. Então são feitas modificações no método para que o teste passe.

```
public String gerarPrimosAte(int i) {
  if (i == 2) return "2";
  else return "2, 3";
```

```
}
```

Essa solução funciona para o argumento 3, mas nota-se que gera duplicação incômoda no código, onde os métodos `testePrimosGeradosAteNumeroDois()`, e `testePrimosGeradosAteNumeroTres()` possuem implementação quase idêntica. Além disso, a variável i não expressa bem sua intenção. Um princípio básico no TDD é chamado de *Don't Repeat Yourself* (DRY). Este princípio trata da importância de eliminar duplicações para tornar o código mais claro, coeso e manutenível. Para isto, utiliza-se a técnica de refatoração do XP (*refactoring*). Então, um método é extraído desta duplicação.

```
public void testePrimosGeradosAteNumeroDois() throws Exception {
  verificaPrimosGerados("2", 2);
}
public void testePrimosGeradosAteNumeroTres() throws Exception {
  verificaPrimosGerados("2, 3", 3);
}
private void verificaPrimosGerados(String listaEsperada,
  int numeroMaximo) throws Exception {
    GeradorPrimos geradorPrimos = new GeradorPrimos();
    assertEquals(listaEsperada,
      geradorPrimos.gerarPrimosAte(numeroMaximo));
}
...
public class GeradorPrimos {
  public static final int MENOR_PRIMO = 2;
  public String gerarPrimosAte(int valorMaximo) {
    if (valorMaximo == MENOR_PRIMO) return "2";
    else return "2, 3";
  }
}
```

Ainda é necessário criar testes de exceções para números menores o `MENOR_PRIMO`. Utilizando a técnica de refatoração para adicionar estes testes nota-se a necessidade de testes para números negativos. A refatoração normalmente demanda um pequeno investimento inicial, porém gera economia de tempo futura, mantendo o código organizado. É comum ocorrer situações em que são extraídos métodos que são utilizados inúmeras vezes em uma mesma classe de teste.

Então, cria-se novos casos de teste para os argumentos 4, 5, ... que geram a necessidade da implementação real do algoritmo Crivo de Eratóstenes. É preciso uma lista representando possíveis candidatos de números primos. Um vetor do tipo *boolean*, no qual *true* indica que o número é primo resolve o problema. Métodos de inicialização, verificação e atribuição precisam ser criados, sempre seguindo a abordagem TDD, onde os testes são gerados primeiramente e o códigos das classes e métodos são implementados consequentemente. Abaixo, uma possível solução para problema proposto (foram omitidas algumas etapas de criação de casos de teste e refatoração):

```
public String gerarPrimosAte(int valorMaximo)
  throws ValorMaximoInvalidoException {
    if (valorMaximo >= MENOR_PRIMO) {
      return numerosPrimos(valorMaximo);
    } else {
      throw new ValorMaximoInvalidoException();
    }
}

private String numerosPrimos(int valorMaximo) {
  boolean [] ehPrimo = inicializaListaCandidatos(valorMaximo);
  for (int valor = MENOR_PRIMO; valor <= valorMaximo; valor++) {
    if (ehPrimo[valor]) {
      for (int naoPrimos = MENOR_PRIMO * valor;
        naoPrimos <= valorMaximo; naoPrimos += valor) {
          ehPrimo[naoPrimos] = false;
      }
    }
  }
  return apresentaResultado(valorMaximo, ehPrimo);
}

private String apresentaResultado(int valorMaximo, boolean[] ehPrimo)
{
  String resultado = String.valueOf(MENOR_PRIMO);
  for (int i = MENOR_PRIMO + 1; i <= valorMaximo; i++) {
    if (ehPrimo[i]) {resultado += ", " + i;}
  }
  return resultado;
}

boolean[] inicializaListaDePrimosPotenciais(int valorMaximo) {
  boolean [] resultado = new boolean[valorMaximo + 1];
  resultado[0] = resultado [1] = false;
  for (int i = MENOR_PRIMO; i < resultado.length; i++) {
    resultado[i] = true;
  }
  return resultado;
}
```

6. Conclusões

Unido às técnicas como *pair-programming*, *refactoring* e outras, o TDD é responsável pelo sucesso dos projetos que utilizam XP. Diversas empresas já adotaram o TDD como padrão no desenvolvimento de software, pois aumentaram o nível de compreensão do código gerado economizando tempo de manutenção.

Com o uso desta técnica é possível reduzir a complexidade do software, aumentando a manutenibilidade do mesmo. Falhas são facilmente identificadas ainda na etapa de desenvolvimento graças ao contínuo *feedback* dado ao programador. As unidades de teste criadas permitem avaliar mais facilmente novos defeitos que podem ter sido inseridos no software facilitando o desenvolvimento de sucessivas *releases*.

Pesquisadores têm notado que técnicas e notações de desenvolvimento de software têm sido integradas ao processo de implementação [Perry, D.E. and Wolf 1992]. Tal integração tende a confundir projeto e implementação. A prática TDD também integra as diferentes fases do desenvolvimento: projeto, implementação e teste. É dada maior ênfase em como os elementos necessitam ser implementados e menor ênfase nas estruturas lógicas. Portanto, se adotado fielmente, o TDD pode resultar na falta do esboço do sistema. Decisões importantes de projeto podem ser perdidas devido à falta da documentação formal do projeto.

Referências

Agile Alliance, “The Manifesto for Agile Software Development”, vol. 2003: Agile Alliance, 2000.

Beck, K. “eXtreme Programming Explained”, Addison Wesley, 2000.

Beck, K. “Test Driven Development: By Example”, Addison Wesley, 2002.

Chaplin, D. “Test first programming”, TechZone, 2001.

Corbi, T.A. “Program understanding challenge for the 1990s”, IBM Systems Journal 28 (1989) 294–306.

Gelperin, D. and Hetzel, W. “Software quality engineering”, Fourth International Conference on Software Testing, Washington, DC, June 1987.

George, B. and Williams, L.A. “A structured experiment of test-driven development”. Information & Software Technology 46 (2004) 337–342

Larman, C. and Basili, V. “A history of iterative and incremental development”, IEEE Computer 36 (2003) 47–56.

Lehman, M. and Belady L. “Program Evolution: Processes of Software Change”, Academic Press, London, 1985.

Mugridge, R. “Test driven development and the scientific method”. Agile Development Conference, 2003. ADC 2003. 47- 52.

Müller, M.M. and Hagner, O. “Experiment about test-first programming”, Empirical Assessment In Software Engineering EASE’02, Keele, April 2002

National Institute of Standards and Technology (2002) “Software Errors Cost U.S. Economy $59.5 Billion Annually”, http://www.nist.gov/public_affairs/releases/n02-10.htm, June 28.

Perry, D.E. and Wolf, A.L. “Foundations for the study of software architecture”, ACM SIGSOFT 17 (1992) 40–52.

Teles, V. M. (2005) “Desenvolvimento Orientado a Testes”, disponível em http://www.improveit.com.br/xp/tdd.jsp.

Williams, L.; Maximilien, M.; Vouk, M. “Test-driven development as a defect-reduction practice”. IEEE International Symposium on Software Reliability Engineering, Denver, CO, 2003.

Wikipédia (2006) “Eratóstenes”, http://pt.wikipedia.org/wiki/Erat%C3%B3stenes.