2
Revisão Bibliográfica

2.1.
Desenvolvimento Dirigido por Comportamentos

Como evolução das técnicas de desenvolvimento dirigido por testes, Dan North sugeriu uma série de melhorias que deram origem ao que chamou de desenvolvimento dirigido por comportamentos (DDC) [8]. O termo se originou de sua percepção de que o desenvolvimento dirigido por testes não tem como única vantagem a criação de uma forma mais eficiente de realizar testes, sendo também importante para a especificação dos módulos em construção [9]. Ao criar os casos de teste para o código que será produzido, o desenvolvedor está, ao mesmo tempo, descrevendo o comportamento esperado de cada módulo, contribuindo desta forma para um melhor entendimento dos resultados desejados. Essa prática pode ser considerada como uma forma de redundância de raciocínio [6], onde o programador é obrigado a pensar mais de uma vez sobre uma mesma questão, cada qual utilizando formas diferentes de raciocínio. Essa prática é interessante por criar uma nova oportunidade para o desenvolvedor encontrar problemas que possam ter passado despercebidos pela etapa de especificação do sistema.

Para dar maior ênfase aos comportamentos, duas mudanças são propostas:

- Mover o foco dos testes para os comportamentos, através de modificações no vocabulário utilizado e na divisão das unidades de código.
- Utilizar linguagens mais próximas do domínio da aplicação, facilitando a compreensão dos testes por integrantes não técnicos da equipe.

2.1.1.
Movendo o Foco Para os Comportamentos

Segundo North, o vocabulário atualmente utilizado na prática de desenvolvimento dirigido por testes é muito ligado à verificação do que foi construído, principalmente através da palavra “teste” [8]. Desta forma, aqueles

que utilizam as técnicas são guiados a pensar somente nas vantagens obtidas através dos testes [4]. Para modificar o foco para a especificação, um primeiro passo é a não utilização da palavra “teste”, buscando termos mais ligados ao comportamento de sistemas, como por exemplo, o emprego da expressão “deveria fazer”. Dizer que um módulo “Deveria fazer ação x” torna muito mais explicito os comportamentos que se espera do sistema em desenvolvimento, valorizando as funcionalidades desejadas. Outra questão que necessita de mudanças é a divisão dos casos de teste para a cobertura do código. Atualmente o desenvolvimento dirigido por testes induz os desenvolvedores a dividirem seus casos de teste de acordo com a estrutura do código [9]. Ao invés disso, é interessante que os casos de teste sejam divididos de acordo com os comportamentos esperados do sistema [9]. Esta mudança também auxilia na percepção do valor agregado pelo que está sendo desenvolvido, uma vez que os comportamentos implementados em cada ciclo são derivados dos requisitos que o integrarão.

2.1.2. Utilizando Linguagens Mais Próximas do Domínio da Aplicação

A outra mudança está ligada à linguagem utilizada na especificação dos comportamentos. A utilização de uma linguagem mais próxima do domínio da aplicação facilita a compreensão da especificação dos comportamentos por todos os integrantes da equipe de um projeto, diminuindo a distância entre membros técnicos e não técnicos. Dessa forma, cria-se o que é chamado por North de linguagem ubíqua do domínio [8]. A linguagem proposta por Dan North segue um formato similar a cenários [8], utilizando a estrutura a seguir: **Dado** um contexto inicial, **quando** um evento ocorre, **então** determinado resultado é obtido. O uso das palavras **dado**, **quando** e **então** deve fornecer uma estrutura bastante livre para facilitar a especificação dos comportamentos, e ao mesmo tempo possuir um nível de formalidade suficiente para que seja possível automatizar tarefas utilizando estas especificações [8]. Embora existam variações nas linguagens utilizadas, uma coisa comum entre elas [10,11,12] é a presença de:

- um contexto, ou seja, uma forma de explicitar pré condições que devem ser válidas para que o comportamento ocorra;
- uma ação, que causa o comportamento que se deseja observar;

- e uma resposta, ou seja, um comportamento observado que foi causado por uma ação específica em um dado contexto.

A união desses três elementos cria o que é chamado de especificação, ou seja, uma descrição do comportamento que se deseja obter. Na figura 1 é apresentado um exemplo caso de teste que descreve um comportamento, utilizando o formato proposto por Dan North.

Dado que a conta possui crédito
e o cartão é válido
e o caixa possui dinheiro

Quando o cliente solicita dinheiro

Então garante que a conta é debitada
e garante que o dinheiro é entregue ao cliente
e garante que o cartão é devolvido

Figura 1: Exemplo de caso de teste descrevendo comportamento

Existem trabalhos mencionando a utilização de diagramas de caso de uso como linguagem de especificação de testes [13,14]. Considerando-se o fato de diagramas de caso de uso ser uma forma de modelagem dos requisitos do que se deseja construir, utilizar uma linguagem baseada em casos de uso como linguagem de especificação de testes pode contribuir para a criação de casos de testes mais focados nos comportamentos do sistema, e menos na estrutura interna do código, conforme proposto por Dan North em DDC [8]. A utilização de casos de uso para especificar testes também seria provavelmente uma forma de se criar especificações em uma linguagem mais compreensível para todos os membros interessados, sejam técnicos ou não, da equipe do projeto que as mais tradicionais especificações em linguagens textuais ou em código nativo do sistema.

Entretanto, a linguagem original de casos de uso, como definida em UML 2 [15], possui baixo grau de formalidade, o que dificulta a especificação dos comportamentos de um sistema de maneira que esses possam ser interpretados automaticamente por uma ferramenta de testes. Uma possível solução apresentada [13] é a utilização de formulários relacionados a cada caso de uso, contendo informações adicionais como pré condições, descrições das ações do caso de uso em linguagens mais formais, pós condições, entre outras. Essas informações adicionais servem para dar maior grau de formalidade aos casos de uso, tornando mais simples sua interpretação por outras ferramentas.

2.2. O Futuro das Ferramentas de Testes

Em 2007, Jennitta Andrea apresentou uma série de características importantes para futuras ferramentas de testes [16]. Inicialmente, o trabalho apresenta algumas metas que novas ferramentas deveriam alcançar, para depois listar possíveis formas de se alcançar estas metas.

As metas a serem alcançadas por estas novas ferramentas podem ser divididas em três grupos, estando relacionadas à escrita, leitura e execução dos testes. Ao utilizar artefatos de teste como elemento principal de especificação, deve-se ter cuidado para que estes o façam de forma suficientemente clara. Para que os testes realizem a especificação de forma efetiva, eles devem ser escritos de forma mais declarativa, em contraste com as formas tradicionais que costumam ser imperativas, elevando o nível de abstração de forma que sejam mais curtos e apresentem maior densidade de informações. Desta forma, leitores dos testes podem ver o contexto no qual cada teste se insere de forma mais clara e objetiva. Na figura 1, por exemplo, encontramos um exemplo de caso de teste escrito em linguagem semelhante à proposta por Dan North como parte da implementação das idéias de BDD [8], e utilizada por ele na ferramenta Jbehave [10]. Neste exemplo, encontramos uma maior densidade de informações em comparação as formas usuais de escrita de testes em TDD. Os comportamentos desejados do código sendo produzido também são descritos de forma bastante clara e direta.

2.2.1. Objetivos Relacionados à Escrita de Testes

É muito importante que os ambientes para a escrita dos testes apóiem o desenvolvedor tanto quanto os ambientes para código de produção. Muitas vezes os ambientes para a escrita de testes não são melhores que um editor de texto comum, não oferecendo ao desenvolvedor funcionalidades como recursos de *autocomplete*, verificações estáticas em tempo real, execução do artefato produzido de forma integrada ao ambiente e apresentação dos possíveis erros encontrados nessas diversas tarefas. Tais facilidades permitem que o desenvolvedor poupe tempo com tarefas repetitivas e facilmente automatizáveis, diminuindo ou eliminando a ocorrência de erros humanos em diversas delas, e contribuindo para a qualidade do código produzido. Vale lembrar que é de

extrema importância escrever código de testes correto uma vez que estes usualmente não dispõem dos mesmos artifícios utilizados para se garantir a qualidade do código de produção. Os testes devem ser simples o suficiente para que a probabilidade de inserção de defeitos no código de testes seja minimizada, e quando estiverem demasiadamente grandes, é preciso buscar formas de dividi-los em diversos testes menores. A presença de defeitos nos códigos de testes é especialmente ruim por mascarar falhas ou apontar falhas inexistentes no código de produção. A dificuldade em escrever código de testes também pode fazer com que esta prática se torne uma espécie de gargalo para o projeto, muitas vezes sendo deixada de lado ou feita sem o cuidado necessário.

2.2.2. Objetivos Relacionados à Leitura de Testes

Considerando-se que, por ser utilizado como especificação do sistema, o código de testes será lido com maior freqüência que o código de produção, ele deve ser mais facilmente localizável e compreensível. Diferentes membros da equipe do projeto com diferentes papéis devem ser capazes de ler o código de testes de forma eficiente entendendo de forma clara e inequívoca as capacidades do sistema, independente de seus conhecimentos técnicos. As linguagens utilizadas ao escrever os testes são de grande importância para que isso seja viável, e devem sempre que possível estar próximas de linguagens utilizadas no domínio da aplicação. Idealmente, diferentes integrantes da equipe devem poder visualizar o mesmo código de testes em diferentes linguagens de acordo com cada ocasião. Deve ser possível localizar testes individuais ou grupos de testes, de acordo com funcionalidades ou grupos de módulos do sistema. As múltiplas possibilidades de visualização dos testes permitem que os diversos integrantes de um projeto analisem as especificações de acordo com as necessidades específicas de suas funções na equipe, com o nível de abstração e escopo adequados, possibilitando sua utilização como documentos de requisitos dos artefatos que devem ser construídos.

2.2.3. Objetivos Relacionados à Execução de Testes

Deve ser possível executar os testes em diferentes ambientes durante as diversas fases de um projeto, como por exemplo, desenvolvedores em seus

ambientes de desenvolvimento, ferramentas de execução automática periodicamente ou especialistas no domínio utilizando navegadores para verificar o correto funcionamento de alguma funcionalidade. Os resultados gerados após a execução devem sempre ser claros, indicando se houve ou não sucesso em cada teste, e em caso de insucessos, apresentar informações sobre a falha observada, como o local onde ocorreu e outras informações que possam auxiliar na descoberta do defeito que a originou. Também devem ser apresentadas evidências da execução como seqüências de telas ou chamadas de métodos. A criação de interfaces adaptáveis entre o sistema e o código de testes permitiria que o mesmo fosse executado em implementações utilizando diferentes tecnologias, tornando possível a especificação dos testes antes mesmo que esta opção seja feita.

2.2.4. Como Alcançar os Objetivos?

Para alcançar essas metas, o trabalho identifica três pontos fundamentais que devem ser melhorados: o ambiente de desenvolvimento de testes, o suporte a múltiplas linguagens de especificação e suporte a múltiplas interfaces com o sistema. O ambiente de desenvolvimento de testes deve fornecer as mesmas facilidades que os ambientes para código de produção, e deve possuir ferramentas que permitam ao desenvolvedor encontrar os testes, seja por seu nome, seu conteúdo ou por meta dados. Durante a execução dos testes, o ambiente deve fornecer facilidades para a depuração, como execução passo a passo, visualização de valores de variáveis, entre outras.

A ferramenta deve permitir a utilização de múltiplas linguagens de especificação, permitindo que seus usuários utilizem diferentes formatos, de acordo com as necessidades de cada ocasião. Os testes não devem ser dependentes da linguagem em que foram escritos, podendo ser visualizados em qualquer uma das linguagens disponíveis. Idealmente novas linguagens seriam adicionadas através de extensões da ferramenta que seguem uma interface para um editor da linguagem e um formatador. Os resultados da execução também devem ser separados da forma como são exibidos, de forma que seja possível modificar a apresentação dos resultados.

A possibilidade de gerar testes através de técnicas de programação por demonstração [17] seria interessante para criar testes para sistemas legados de

forma mais simples. Os testes gerados a partir do registro da utilização do sistema deveriam ser transformados em código, podendo ser alterado da mesma forma que os testes escritos manualmente. O registro das operações poderia ser realizado tanto no nível da interface com o usuário como no nível das chamadas internas. A lógica do código de teste deveria ser bem separada do código de acesso à interface do módulo em teste, de forma que este código de acesso à interface possa ser modificado para que os testes sejam executados em diferentes implementações do sistema que utilizem diversas tecnologias de forma mais simples. As diversas implementações do código de acesso à interface devem coexistir no projeto, e a seleção de qual delas será utilizada deve ser feita através de configurações.

2.3. Frameworks de Testes

Diversos *frameworks* foram criados levando em consideração as idéias do desenvolvimento dirigido por comportamentos, possuindo também muitas características semelhantes ao que foi proposto por Jennitta Andrea como evolução para ferramentas de testes [16]. Dentre as mais relevantes podemos citar Jbehave[10], Rspec[11] e Instinct[12].

2.3.1. JBehave

O *framework* Jbehave [10] foi criado por Dan North para introduzir as idéias que deram nome ao desenvolvimento dirigido por comportamentos, e foi desenvolvido utilizando a linguagem Java [18]. Executa os testes a partir de *scripts* em arquivos escritos em linguagem mais próxima da linguagem natural, utilizando a estrutura descrita anteriormente de “Dado, Quando e Então” para descrever respectivamente o contexto em que o comportamento ocorre, o evento que o origina e o resultado esperado. Para cada comportamento descrito no arquivo, deve existir um método em uma classe Java responsável por executá-lo. Desta forma o código de testes pode ser facilmente reutilizado e fica suficientemente separado do *script*, de forma que o *script* sirva como documentação do funcionamento dos artefatos em desenvolvimento [8]. As figuras 2 e 3 apresentam respectivamente um exemplo de script para esta ferramenta, e seu módulo de testes. O exemplo foi extraído do *site* da ferramenta.

```
Given I am not logged in
When I log in as Liz with a password JBehaver
Then I should see a message, "Welcome, Liz!"
```

Figura 2: Exemplo de script de entrada para a ferramenta JBehave.

```
package my.domain.app;

// Ensure extends JUnit's Assert
// It works just like JUnit with Hamcrest's matchers
// or you can create your own

import static org.jbehave.Ensure.ensureThat;

import org.jbehave.scenario.steps.Steps;
import org.jbehave.scenario.annotations.Given;
import org.jbehave.scenario.annotations.When;
import org.jbehave.scenario.annotations.Then;

public class LoginSteps extends Steps {

    // Some code to set up our browser and pages
    // ...

    @Given("I am not logged in")
    public void logOut() {
        currentPage.click("logout");
    }

    @When("I log in as $username with a password $password")
    public void logIn(String username, String password) {
        currentPage.click("login");
    }

    @Then("I should see a message, \"$message\"")
    public void checkMessage(String message) {
        ensureThat(currentPage, containsMessage(message));
    }
}
```

Figura 3: Exemplo de módulo de testes para a ferramenta JBehave.

Espera-se que a utilização de uma linguagem mais próxima de linguagem natural para descrever os comportamentos facilite a leitura e as revisões por parte de membros não técnicos da equipe.

Entretanto, a linguagem natural pode não ser sempre a melhor linguagem para descrever o comportamento de sistemas. Diferentes linguagens podem ser mais eficientes para diferentes domínios de aplicações [16].

2.3.2. RSpec

Desenvolvido para a linguagem de programação Ruby por Dave Astels e Steven Baker entre outros, o Rspec [11] possui uma linguagem para a definição de comportamentos em código, através dos elementos *Describe* e *It*. O *Describe* é utilizado para descrever um comportamento específico do artefato em teste e diversos elementos *It* o compõem para definir os resultados esperados para este comportamento. Existem ainda elementos *before* e *after*, que são executados antes e depois de cada comportamento, respectivamente. A figura 4 mostra um exemplo de utilização da ferramenta para a classe da figura 5, e foi retirado do *site* da ferramenta.

```
# bowling_spec.rb

require 'bowling'


describe Bowling do

  it "should score 0 for gutter game" do

    bowling = Bowling.new

    20.times { bowling.hit(0) }

    bowling.score.should == 0

  end

end
```

Figura 4: Exemplo de especificação de comportamento para a classe da figura 5.

```
# bowling.rb

class Bowling

  def hit(pins)

  end


  def score

    0

  end

end
```

Figura 5: Classe descrita pelo exemplo da figura 4.

Do ponto de vista de funcionalidades, este *framework* é bastante semelhante ao Jbehave [10], entretanto sua especificação de comportamentos em código pode dificultar a leitura das descrições, principalmente por parte de integrantes não técnicos da equipe do projeto.

O fato da ferramenta sugerir uma divisão de comportamentos por classe também torna o código de testes um pouco mais vinculado à estrutura interna do programa construído, o que de certa forma vai contra aquilo que se é proposto em BDD [8].

2.3.3. Instinct

O Instinct [12] é um *framework* para desenvolvimento dirigido por comportamentos desenvolvido na linguagem Java [18]. Nele a especificação de comportamentos se dá através de *Annotations*, um mecanismo de Java que permite adicionar meta informações ao código. Através destas informações o *framework* identifica marcações como *@Subject*, que define o atributo que contém o objeto que será testado, e *@Specification*, que marca métodos que serão executados para a verificação de comportamentos. A figura 6 apresenta um exemplo de especificação de comportamento, extraído do *site* da ferramenta.

```
import static com.googlecode.instinct.expect.Expect.expect;
import com.googlecode.instinct.marker.annotate.BeforeSpecification;
import com.googlecode.instinct.marker.annotate.Specification;
import com.googlecode.instinct.marker.annotate.Subject;

public final class AnEmptyStack {
    @Subject private Stack<Object> stack;

    @BeforeSpecification
    void setUp() {
        stack = new StackImpl<Object>();
    }

    @Specification
    void mustBeEmpty() {
        expect.that(stack.isEmpty()).isEqualTo(true);
    }
}
```

Figura 6: Exemplo de classe do *framework Instinct* descrevendo comportamentos.

Uma característica interessante dessa ferramenta é a divisão dos casos de teste em classes que representam as pré-condições necessárias para a realização dos testes. Na figura 6, por exemplo, aparece uma classe chamada *AnEmptyStack*, que realiza a verificação dos comportamentos para uma pilha vazia, o que neste caso é a pré-condição para os testes do módulo de pilha.

Sua forma de especificação de comportamentos é bastante interessante por ser bem simples e pouco intrusiva, tornando fácil aprender a utilizá-lo. Entretanto, sua especificação em código dificulta a leitura das especificações por membros não técnicos da equipe.