2
Estado da arte

2.1.
Desenvolvimento dirigido por comportamentos

O desenvolvimento dirigido por comportamentos foi proposto por Dan North (North, 2006) ao perceber que, ao praticar DDT, o desenvolvedor é muitas vezes obrigado a identificar quais classes irá precisar, quais as colaborações entre elas e o que elas devem fazer para atingir seu objetivo. Nesse caso, o DDT estaria sendo usado como uma ferramenta para modelar o que os módulos devem fazer e como se relacionam. Evidentemente, isso dificulta sobremaneira a participação do cliente (com pouca formação em computação) na formulação das especificações e dos correspondentes testes. O problema passaria a ser então, muito mais o de especificar comportamentos e relações em um nível de abstração mais alto, ao invés de no nível de módulos individuais. Ou seja, passar-se-ia a praticar o DDC.

Para dar ênfase aos comportamentos é proposta uma mudança no vocabulário utilizado, primeiramente substituindo a palavra teste, geralmente usada como prefixo para nomear métodos em DDT, por nomes de ações que descrevam a funcionalidade que um método do teste deve verificar. Também deve ser feita uma mudança na forma de pensar ao escrever os testes: enquanto em DDT os desenvolvedores dividem os testes de acordo com a estrutura do código, em DDC os testes devem ser divididos de acordo com as funcionalidades do sistema.

Outra mudança é a definição de uma linguagem comum que deve refletir o domínio da aplicação, como é proposto em DDD (*Domain Driven Design*) (Evans, 2003). Essa linguagem deve ser livre de termos técnicos relacionados com a implementação em computador e deve ser compreensível por todos os envolvidos no projeto, tanto desenvolvedores quanto clientes, a fim de diminuir a distância de comunicação entre desenvolvedores e não técnicos. Esse vocabulário deve enfatizar a descrição das funcionalidades e dos valores agregados por elas ao sistema.

Dan North sugeriu uma linguagem semi-estruturada composta por três palavras chaves: “Dado”, “Quando” e ”Então”. Essas palavras devem ser empregadas de acordo com a seguinte estrutura: “dado um contexto, quando ocorrer determinado evento, então se deve obter determinado resultado”. Essa estrutura formaria um cenário. Como exemplo, tem-se o seguinte cenário:

Dado que estou autenticado como Administrador
Quando eu clicar no botão Listar usuários
Então será exibida a lista de todos os usuários cadastrados
Quando eu selecionar nesta lista o usuário José e clicar no botão apagar
Então será exibida a lista de usuários cadastrados e esta lista não conterá o usuário José

Figura 1 - Exemplo de cenário em DDC.

Associada a um cenário, também pode existir uma estória de usuário com as palavras “como”, “eu quero” e “para” disposta no seguinte formato: “como um ator, eu quero determinada funcionalidade, para atingir determinado objetivo”. Associada ao cenário exemplificado acima se pode ter a seguinte estória de usuário:

Como Administrador do sistema
Eu quero ser capaz de adicionar, alterar e excluir os dados dos usuários
Para poder manter o sistema atualizado

Figura 2 - Exemplo de estória de usuário em DDC.

Os cenários definiriam os testes necessários para validar a funcionalidade identificada na estória de usuário. Vários cenários podem estar associados a uma mesma estória.

A estória apresentada corresponde a várias características, como a capacidade de criar, acessar, atualizar e excluir usuários e evidencia a necessidade de vários cenários para que possa ser considerada completamente implementada. Também é necessário tratar casos anômalos, por exemplo, podem existir usuários pertencentes a categorias que proíbem a sua exclusão (e.g. o Administrador em Windows). De maneira geral, estas características não transparecem nas estórias e cenários e, consequentemente, correm o risco de não serem implementadas,

comprometendo a qualidade do software tal como percebida pelo usuário após a entrega do sistema. Idealmente, deveria existir um cenário para cada possível fluxo de execução de uma funcionalidade.

2.1.1. *Frameworks* de testes em DDC

Diversos frameworks foram criados para auxiliar o desenvolvimento dirigido por comportamentos. Dentre as soluções mais conhecidas, podemos citar JBehave (JBehave, 2008) e RSpec (RSpec, 2008).

2.1.1.1. JBehave

O framework JBehave começou a ser desenvolvido por Dan North em 2003 para pôr em prática as suas ideias sobre desenvolvimento dirigido por comportamentos. Esta ferramenta é uma extensão de JUnit (JUnit, 2009) e, como consequência, a linguagem utilizada na implementação dos testes é Java.

Nesta ferramenta os cenários são escritos em um arquivo texto com as palavras chaves "**Given**", "**When**" e "**Then**", para denotar, respectivamente, um contexto, um evento e um resultado esperado após o evento. Também é possível adicionar novas palavras chaves. Esse arquivo deve servir como uma especificação do sistema de fácil leitura para desenvolvedores e clientes. Como exemplo de um cenário em JBehave para a estória de usuário da seção 2.1, temos, em inglês:

```
Scenario: Administrator successfully deletes user

Given I am logged in as Administrator
When I click in the button List users
Then a list containing all the registered users is showed
When I select in this list the user José and click button erase
Then the users list is showed and the user José should not be in the list
```

Figura 3 - Exemplo de cenário em JBehave.

Para cada arquivo contendo cenários deve existir uma classe em Java que implementa o teste descrito no script. Esta classe deve ter o mesmo nome do arquivo. Por exemplo, se o arquivo contendo a cenário acima for salvo com o nome "**admin_successfully_deletes_user**" então a classe Java que implementa

este cenário deve ser nomeada "**AdminSuccessfullyDeletesUser**". A classe que implementa o cenário deve conter uma instância de outra classe que conterá os métodos que implementam cada passo do cenário. Cada método desta outra classe deve ser mapeado através de *annotations* em Java, que são marcações usadas para inserir meta-informação no código. Essas *annotations* irão determinar qual método é mapeado em qual passo do script de testes. Por exemplo, para mapear um método que inicializa um contexto, usa-se, antes da declaração do método, a *annotation* "**@Given**" seguida de uma cadeia de caracteres que servirá para identificar de qual passo do arquivo de scripts este método é responsável. Da mesma forma existem as *annotations* "**@When**" e "**@Then**" para mapear métodos que implementam, respectivamente, uma ação e verificam um resultado esperado.


```
public class AdminSucessfullyDeletesUser extends Scenario
{
    public AdminSucessfullyDeletesUser()
    {
        super( new DeleteUserSteps() );
    }
}
```

Figura 4 - Classe em Java que implementa um cenário JBehave.

```
public class DeleteUserSteps extends Steps
{
    @Given("I am logged in as Administrator")
    public void loginAsAdmin()
    {
        //código necessário para fazer autenticação
    }

    @When("I click in the button $buttonLabel")
    public void clickButton( String buttonlabel )
    {
        //código necessário para clicar um botão
    }

    //código necessário para implementar os métodos restantes para cada passo
}
```

Figura 5 - Classe que implementa os passos descritos em um cenário JBehave.

Assim, JBehave lê cada passo de um arquivo contendo as descrições de um cenário e procura, através de *annotations* no código Java, o método responsável por executar este determinado passo.

Entretanto, um arquivo texto com a estrutura usada por JBehave pode não ser a melhor opção para capturar todos os comportamentos de um sistema.

2.1.1.2. RSpec

RSpec é um framework de DDC para a linguagem de programação Ruby. A partir dos testes implementados, esta ferramenta gera um arquivo texto que descreve os comportamentos testados, mas, diferentemente de JBehave, esse arquivo só é gerado depois que os testes já foram escritos.

Para implementar o módulo de testes, RSpec utiliza o método "*describe*" para especificar a classe que será testada e pode conter diversas especificações para validar um determinado contexto.

Em cada teste para a classe especificada, deve existir um método "it". Esse método recebe como parâmetro uma cadeia de caracteres que deve descrever o comportamento que será testado no corpo do método.

```
class Agua
    # código da classe Agua
end

describe Agua do
    it "deve estar no estado sólido a -1°C e 1 atm" do
        # atribui temperatura e pressão para objeto do tipo Agua
        # verifica estado
    end

    it "deve estar no estado líquido a 25°C e 1 atm" do
        # atribui temperatura e pressão para objeto do tipo Agua
        # verifica estado
    end
end
```

Figura 6 - Teste em RSpec.

```
Agua
- deve estar no estado sólido a -1°C e 1 atm
- deve estar no estado líquido a 25°C e 1 atm

Finished in 0.0 seconds

2 examples, 0 failures
```

Figura 7 - Saída das especificações testadas em RSpec.

O fato da descrição do comportamento ser escrita dentro da codificação e apenas só poder ser visualizada fora do código a partir da execução do teste, torna difícil a leitura dos requisitos por parte de membros não técnicos da equipe.

2.2. Geração de testes a partir de "*capture and replay*"

Ferramentas como Squish for Web (Squish, 2010) e Selenium IDE (Selenium, 2009) oferecem a capacidade de geração automática de testes para interfaces de sistemas web através de técnica conhecida como "*capture and replay*" (Araújo e Staa, 2009). Nesta técnica, as ações do usuário sobre a interface do sistema são gravadas por uma ferramenta e é gerado um código que reproduz essas ações. Esse código então pode ser editado para refatoração e inserção de verificações, tais como assertivas.

Um ponto fraco desta abordagem é que o código gerado automaticamente acaba sendo mais difícil de compreender e manter do que o código escrito manualmente, caso não haja uma refatoração posterior à geração automática do código. Isso acontece porque, como todas as ações do usuário na interface são codificadas, acaba-se por existir código desnecessário, oriundo, por exemplo, de cliques de mouse acidentais em elementos que não são de interesse para o teste, como uma área em branco da tela, ou ainda, codificação de ações que o usuário executou por engano. Outro fator negativo é que, caso o usuário tenha gravado vários casos de teste em uma única execução da ferramenta, estes casos de teste estarão todos em uma mesma função no código automaticamente gerado, o que requer um trabalho adicional de refatoração de código para extrair cada caso de teste para uma função separada. Caso isso não seja feito, a manutenção do código pode tornar-se difícil, devido à falta de organização do mesmo.

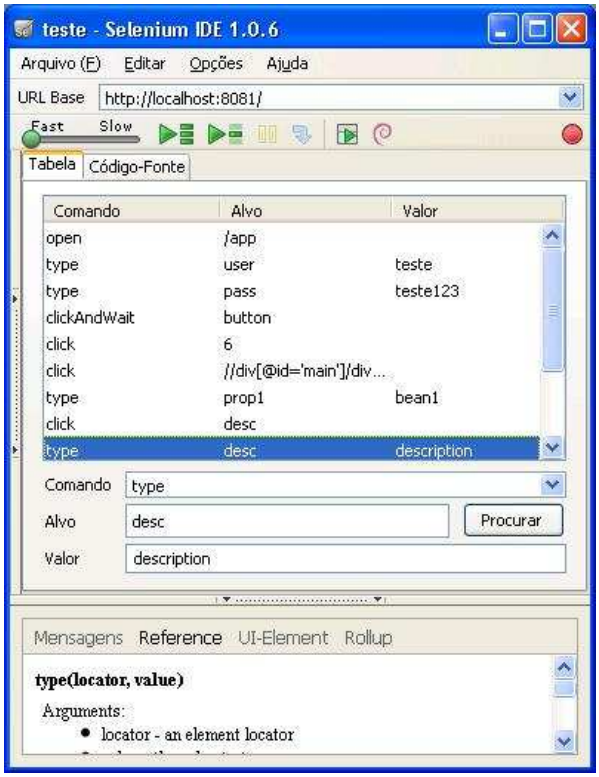

Figura 8 – Interface Selenium IDE.

2.3.
Geração de testes a partir de casos de uso

Alguns artigos propõem que a descrição do caso de uso sirva de insumo para a identificação e criação de casos de teste. No artigo (Heumann, 2001) esta abordagem é proposta através da execução de três passos: identificar, para cada caso de uso, um conjunto de cenários; para cada cenário, identificar pelo menos um caso de teste e as condições que devem ser atendidas ou não, para que este cenário aconteça; finalmente, para cada caso de teste, identificar os dados necessários. Por cenário entende-se uma execução do fluxo principal ou dos fluxos secundários (ou uma combinação de ambos) do caso de uso. O artigo (Wood e Reis, 2007) segue proposta semelhante ao artigo anterior, onde cada caso de teste é identificado a partir dos vários fluxos de execução (principal, secundários, exceções) presentes na descrição do caso de uso. O artigo (Ahlowalia, 2002) também defende o uso dos fluxos de execução para identificar casos de teste, porém diferencia-se ao sugerir que os possíveis caminhos sejam classificados de acordo com sua frequência e importância, e que esta classificação seja usada para escolher quais caminhos serão testados. Entretanto, nenhum destes

artigos aborda como a descrição e redação dos fluxos pode auxiliar a geração de casos de teste.

Um trabalho interessante é apresentado no artigo (Im K, Im T. e McGregor, 2008) onde casos de teste são automaticamente extraídos de casos de uso definidos a partir de uma linguagem específica de domínio. Primeiramente é definida uma linguagem específica do domínio. Em seguida, os casos de uso são escritos usando elementos desta linguagem. Os testes são gerados a partir dos casos de uso e de *templates* de testes. Porém, os dados utilizados nos casos de teste são valorados a partir de um editor do caso de uso, o que torna necessário configurar várias vezes o caso de uso para gerar testes com diferentes valores. A cobertura de todos os valores também é difícil de ser verificada com esta abordagem. Outro ponto que não fica claro é quanto à adequação desta proposta em um sistema real, visto que o exemplo utilizado no artigo é didático.

2.4. Geração de testes a partir de tabelas de decisão

Tabelas de decisão são usadas para expressar qual conjunto de condições é necessário satisfazer para que determinadas ações ocorram. São representadas na forma de tabelas no formato da figura abaixo, podendo ser divididas em três áreas:

- Uma área representando as condições;
- Uma área representando as ações;
- Finalmente, uma área de regras, que são as combinações dos valores das condições e as respectivas ações a serem tomadas para essa combinação de condições.

Condições	**Regra 1**	**Regra 2**	**Regra 3**	**Regra 4**
Condição 1	V	V	F	F
Condição 2	V	F	V	F
Ações				
Ação 1	X		X	
Ação 2		X		X

Tabela 1 – Exemplo de tabela de decisão.

Dependendo da tabela, suas regras podem assumir outros valores que não verdadeiro (V) e falso (F). É também comum o uso dos valores “---” ou “-” para

indicar que, independentemente da condição ser verdadeira ou falsa, o resultado esperado é o mesmo. Este recurso é usado para simplificar as tabelas de decisão, uma vez que duas colunas em potencial tornam-se uma só.

O número de regras de uma tabela de decisão pode ser calculado pela fórmula $m^n = c$, onde 'm' é o número de valores que as condições podem assumir, 'n' é o número de condições existentes e 'c' é o número de colunas (ou regras). Esta fórmula é válida se todas as condições puderem assumir o mesmo número de valores. Regras com "---" (indiferente) tem seu peso multiplicado por dois para cada "indiferente" que existir na regra. Assim uma regra com dois valores "indiferente" conta por quatro regras. Se o número de regras, calculado através da fórmula apresentada, for diferente do existente na tabela (levando-se em conta o peso de regras com valores "indiferente"), então a tabela de decisão está incompleta.

Embora muito utilizadas em outras áreas, há pouca literatura sobre como tabelas de decisão podem ser utilizadas para auxiliar a geração de testes. No artigo (Ferriday, 2007) o autor defende o uso de tabelas de decisão para a identificação de casos de teste a partir das regras da tabela de decisão. Também demonstra como tabelas de decisão podem ser validadas quanto a sua completeza, consistência e não ambiguidade. Em (Myers, 2004) as tabelas de decisão são montadas a partir dos grafos de causa-efeito, sendo utilizadas como forma auxiliar para a identificação dos testes, onde as colunas da tabela são os casos de teste. Entretanto, não é discutido como elas podem ser usadas para a geração automática dos testes.

Na dissertação (Lachtermacher, 2010) a tabela de decisão é usada para automatizar a geração de casos de teste para interfaces implementadas na linguagem Java. Nesse trabalho o usuário, através de ferramentas desenvolvidas pela autora, é capaz de montar uma tabela de decisão que é convertida em casos de teste com geração automática de dados a partir de uma gramática geradora, e em seguida para scripts de teste na linguagem de programação Java. Os testes gerados são para interfaces em Java.

2.5. Limitações das propostas publicadas

Os scripts de testes funcionais devem ser construídos com base em alguma especificação que possa validar se os testes estão corretos e em conformidade com as especificações do sistema. É importante que esta linguagem seja de fácil compreensão para pessoas não técnicas, a fim de que seja possível diminuir as barreiras de comunicação entre desenvolvedores, testadores e clientes. Uma linguagem com estas características poderia servir tanto de especificação para as funcionalidades do sistema quanto para a elaboração dos testes destas funcionalidades.

A linguagem utilizada no desenvolvimento por comportamentos poderia ser uma solução para este problema, porém, em muitos casos, o uso da estrutura sugerida por DDC para descrever funcionalidades através de cenários pode ser excessivamente simples, podendo deixar de lado uma série de aspectos, como requisitos não-funcionais ou características que o cliente ignora, mas que são relevantes para os desenvolvedores. Em algumas aplicações, uma única funcionalidade, como a reserva de passagem em uma companhia aérea, pode apresentar várias opções e alternativas de execução. Isso implica em um grande número de cenários interdependentes, cuja leitura e redação podem tornar-se difícil. Além disso, essa interdependência de cenários pode não estar explícita.

Uma alternativa seria usar casos de uso para descrever as funcionalidades do sistema, pois estes conseguem exibir de forma mais compacta os fluxos de execução possíveis para uma dada funcionalidade do sistema. Porém os casos de uso, como definidos na UML (UML, 1997), são excessivamente informais e pouco estruturados para que se possa elaborar uma especificação para testes. Para auxiliar na geração de testes deveriam obedecer a um padrão mais formal, com um vocabulário restrito e estruturado, contando com estruturas de controle na descrição dos fluxos de execução, como laços e condicionais.

Sistemas reais podem requerer um grande número de casos de teste para que sejam completamente testados. Por exemplo, um formulário com cinco campos obrigatórios para preenchimento e dois botões mutuamente exclusivos necessitariam de 64 casos de teste para abranger todas as combinações possíveis de dados (levando-se em conta que apenas o preenchimento importa, e não o valor

dos dados preenchidos). As dificuldades de redação e elicitação de todas as possibilidades de casos de teste seriam facilitadas com a adoção de uma tabela de decisão onde cada condição seria uma linha da tabela e cada coluna um caso de teste. A tabela de decisão permite uma melhor visualização dos casos de teste a serem gerados e também possui propriedades que permitem verificar se esta está completa, consistente e sem ambiguidades.