PS-159

A TOOL FOR AUTOMATING ACCEPTANCE TESTS BASED ON LANGUAGE LETA

Rogério Iokoi (Instituto de Pesquisas Tecnológicas, SP, Brasil)
rogiok@gmail.com
Paulo Sérgio Muniz Silva (Instituto de Pesquisas Tecnológicas, SP, Brasil)
paulo.muniz@poli.usp.br

Executable Acceptance Test-Driven Development helps to discover problems before the construction of software by tests. However, quality analysts need training and learning time to understand a programming language and tools to implement an acceptance test. In addition, implementation time can be long and affects the delivery of a software feature. This article presents a tool to automate the acceptance tests with the support of the language called LETA (Linguagem para Especificação de Testes de Aceitação). The language is similar to natural language with a few words of English. It also incorporates and validates business rules, because it is based on the SBVR (Semantics of Business Vocabulary and Business Rules) specification. The results obtained comparing the LETA with another tools, are the possibility of a quality analist to use a tool without knowing programming language, and the automatic generation of some test component, relieving its implementation by the programmer.
Keywords: agile method, acceptance test, automated test, test-driven development, SBVR.

UMA FERRAMENTA PARA A AUTOMATIZAÇÃO DE TESTES DE ACEITAÇÃO BASEADA NA LINGUAGEM LETA

O Desenvolvimento Dirigido por Testes de Aceitação Executável ajuda a descobrir os problemas antes da construção do software por meio dos testes. Mas, analistas de qualidade precisam de treinamento e tempo para dominar uma linguagem de programação e as ferramentas para implementar um teste de aceitação. Além disso, a implementação pode ser longa, prejudicando a entrega do software. Este artigo apresenta uma ferramenta para automatizar os testes de aceitação com a LETA (Linguagem para Especificação de Testes de Aceitação). A linguagem possui poucas estruturas e palavras do inglês, sendo próxima da natural, e incorpora regras de negócio, baseando-se na SBVR (Semantics of Business Vocabulary and Business Rules). Os resultados, obtidos da comparação da LETA com outras ferramentas, são a possibilidade de um analista de qualidade utilizar uma ferramenta sem conhecer a linguagem de programação e geração automática de alguns componentes de teste, aliviando a sua implementação pelo programador.
Palavras-chave: processo ágil, teste de aceitação, teste automatizado, desenvolvimento dirigido por testes, SBVR.

I. Uma Ferramenta para a Automatização de Testes de Aceitação Baseada na Linguagem LETA

A. Introdução

Encontra-se em processos ágeis de desenvolvimento de software, como o *Extreme Programming* (XP) apresentado por Kent Beck (Beck, 2004: 1), a técnica denominada Desenvolvimento Dirigido por Testes ou *Test-Driven Development* (TDD) (Beck, 2002: 2). A técnica consiste em especificar e implementar os testes antes da implementação de uma funcionalidade do software. A técnica também pode ser utilizada para projetar um software (Janzen & Saiedian, 2008: 77), pois um teste pode ajudar tanto na sua definição estrutural, tais como classes, objetos, métodos e outros, quanto na sua definição comportamental, uma vez que um comportamento é avaliado a partir dos resultados gerados pelo software de acordo com os dados fornecidos a ele (Janzen & Saiedian, 2006: 1).

Além disso, o TDD diminui, mas não elimina, a distância entre as decisões de implementação do projeto e o resultado destas decisões, pois o código está sendo executado e testado em seguida. Também há a possibilidade de identificar os defeitos e suas causas com mais facilidade, pois o defeito pode estar no código escrito recentemente. A técnica estimula os programadores a escreverem códigos testáveis, como métodos e funções que retornam valores e que são verificados e validados (George & Williams, 2003: 1136).

Alguns estudos sobre a técnica destacam o aumento de qualidade no software, baseados na quantidade de defeitos. Um deles compara o TDD com outro método que implementa os testes após a codificação de uma funcionalidade e demonstra que o TDD consegue obter 28% a menos de defeitos (VU, Niklas, Shenkel-Therolf, & Janzen, 2009: 233). Outro estudo apresenta uma queda de 67% (valor médio) de defeitos em códigos legados (Nagappan, Maximilien, Bhat, & Williams, 2008: 297).

Existem algumas variações da técnica TDD. Uma delas é mencionada por Park e Maurer (2008: 19) e chamada Desenvolvimento Dirigido por Teste de Aceitação Executável ou *Executable Acceptance Test-Driven Development* (EATDD). É uma técnica que melhora a comunicação entre os clientes e desenvolvedores, e ajuda a descobrir problemas antes da construção do sistema, elevando o nível de abstração dos testes para além do código que o implementa. A execução dos testes de aceitação pode indicar o progresso do desenvolvimento, pois o sucesso de um conjunto de testes pode indicar que a implementação de uma funcionalidade foi concluída. Além disso, há a geração de uma documentação atualizada do software, porque os testes descrevem os comportamentos dele.

Nos processos ágeis, as histórias de usuário são amplamente utilizadas para descrever as funcionalidades do software. Normalmente, o cliente escreve e prioriza as histórias que devem ser desenvolvidas, pois ele é a melhor pessoa a expressar as funcionalidades desejáveis do software. Os testes de aceitação são elaborados a partir delas e são usados para determinar se o software está completo e se pode ser aceito pelo cliente. Além disso, esses testes podem conter informações ou detalhes que não estão nas histórias e que ajudam no desenvolvimento de uma funcionalidade. Depois, eles são executados para validar a funcionalidade e para concluir se ela está de acordo com o desejado ou

especificado (Cohn, 2004: 68). De acordo com Beck (2004: 74), o papel do analista de qualidade é ajudar o cliente a definir os critérios de aceitação e os testes.

Existem diversas ferramentas utilizadas para implementar os testes de aceitação. Uma delas, denominada FitNesse (Martin, 2011), permite a colaboração de usuários, analistas de qualidade e programadores para a especificação de testes com exemplos. Com uma interface Wiki, um usuário ou analista de qualidade pode editar as páginas e colocar os casos de teste numa tabela com os seus dados de entrada e dados esperados.

Outras ferramentas, do tipo grava e executa, como o Selenium (ThoughtWorks, 2011), simulam um conjunto de ações do usuário numa interface gráfica, tais como cliques em botões numa página HTML, ou preenchimento de campos de texto de um formulário. Além disso, existem comandos que validam os dados ou componentes da interface gráfica, como uma mensagem na página HTML.

No entanto, um analista de qualidade pode encontrar dificuldades ao implementar os testes de aceitação por meio dessas ferramentas. Dentre elas pode-se citar o custo de treinamento e tempo de aprendizado para dominá-las, a necessidade de conhecer uma linguagem de programação para implementar os adaptadores (ou *fixture*) na ferramenta FitNesse, assim como, para utilizar a API da ferramenta Selenium e, por fim, a própria API do software que está sendo testado, seja uma página ou um componente na forma de objetos e métodos. Além disso, em alguns casos, é necessário utilizar mais de uma ferramenta conforme o tipo de teste que pretende elaborar.

Outra dificuldade ocorre durante a implementação dos testes. O tempo dessa implementação pode ser longo que impede a entrega das funcionalidades de uma aplicação. Além disso, conforme aumenta a pressão dos prazos para entregar uma funcionalidade, a construção de alguns testes são descartados ou reduzidos. Outro fator para essa perda de prioridade é o aumento de testes que ocorre conforme a aplicação cresce (Rendell, 2008: 300).

Este artigo apresenta uma ferramenta baseada na linguagem LETA (Linguagem para Especificação de Testes de Aceitação) para a automatização dos testes de aceitação que pretende minimizar algumas dessas dificuldades. Analistas de qualidade podem utilizar diretamente a linguagem LETA para a especificação de testes de aceitação sem a necessidade de conhecimentos técnicos de programação, pois ela oferece um nível de abstração apropriada para eles. Assim, espera-se uma diminuição do aprendizado, treinamento e da definição dos testes.

A partir de um caso de teste especificado na linguagem LETA, a ferramenta consegue gerar automaticamente variações deste teste na forma de códigos executáveis e escrito em outra linguagem, como Java, pois a ferramenta baseia-se na técnica denominada Testes Baseados em Modelos ou *Model-Based Testing* (MBT). Nessa técnica, os testes são gerados a partir de um modelo simplificado do sistema focalizando apenas o que precisa ser testado (Utting & Legeard, 2006: 28). Desta maneira, a ferramenta alivia uma parte da implementação do teste que normalmente é realizada por um programador, restando apenas implementar a comunicação do teste com o software que está sendo testado.

O artigo está dividido da seguinte forma: a Seção 2 apresenta um exemplo do processo de especificação e implementação de um teste escrito na linguagem LETA; a Seção 3 apresenta uma breve descrição das características da linguagem; a Seção 4 discute a comparação de um caso de teste escrito em diversas ferramentas, inclusive com a linguagem LETA e os resultados desta comparação; a Seção 5 apresenta os diferenciais e as limitações da ferramenta e da linguagem; e a Seção 6 conclui e sugere alguns trabalhos futuros.

II. Apresentação Geral da Ferramenta e da Linguagem LETA

A linguagem LETA é utilizada para a especificação de testes de aceitação. Caracteriza-se por ser textual, declarativa e por incorporar conceitos de negócios na forma de termos, fatos e regras. Ela é baseada na notação especificada na *Semantics of Business Vocabulary and Business Rules* (SBVR) do Object Management Group (2008).

A linguagem é declarativa no sentido de que permite dizer o que se deseja fazer, em vez de como fazer, pois os testes de aceitação devem representar um comportamento desejado do software do ponto de vista do cliente ou do analista de qualidade (Crispin & Gregory, 2009: 99). Além disso, durante a elaboração dos testes de aceitação, que ocorre antes da codificação, nem sempre existem todos os detalhes para a implementação deles, apenas uma ideia do que se precisa testar.

Basicamente, um caso de teste possui duas partes. A primeira corresponde aos dados que são enviados a um componente de software a ser testado. A segunda consiste nos critérios de sucesso que comparam os dados esperados com os resultados gerados pelo componente (Myers, 2004: 14).

Para exemplificar a utilização da ferramenta e da sua linguagem, considera-se uma empresa que efetua vendas pela Internet e deseja aplicar um desconto de 10% nas vendas com valores acima de R$ 200,00. Isso pode implicar numa nova funcionalidade de um software que pode ser escrita na história: “Os pedidos acima de R$ 200,00 recebem um desconto de 10% no seu valor total”.

Um dos possíveis testes de aceitação definido por um analista de qualidade é: “Verificar o desconto de 10% num pedido com valor total igual ou maior que R$ 200,00”. O analista pode escrever esse teste de aceitação formalmente utilizando a linguagem LETA de acordo com a Figura 1.

```
1  Test ValidarDescontoNoPedido
2  Verify
3  Pedido tem Desconto 10
4  When
5  Pedido tem ValorTotal >= 200
```

Figura 1. Exemplo de teste que gera dois casos de testes.

Ao definir esse teste na linguagem LETA num arquivo e, em seguida, executar o comando “leta”, a ferramenta irá gerar automaticamente um código escrito numa outra linguagem, como Java.

Esse código gerado contém uma classe utilizada pelo programador para implementar a única parte manual do teste: uma subclasse dela responsável pela comunicação do código gerado com o software que está sendo testado - o Sistema Sob Teste (*System Under Test* – SUT) no jargão de teste. Por exemplo, a Figura 2 apresenta a implementação da subclasse **ValidarDescontoNoPedidoAdaptador** que herda o método, chamado **sendToSut**, da classe gerada **ValidarDescontoNoPedidoTest**. Esse método recebe como parâmetro os dados do testes, representado pelo objeto **input**, que são repassados ao componente do software que processa o valor total de um pedido, denominado **PedidoSUT**. Em seguida, é solicitado ao objeto **pedido** o total novo. Depois, há um cálculo para obter o desconto que retorna pelo método **sendToSut** e pelo objeto **output**. O desconto obtido é comparado com o valor especificado para indicar se houve falha ou sucesso no teste.

Por fim, após a implementação do adaptador pelo programador, basta executar a classe **ValidarDescontoNoPedidoTest** por meio do JUnit.

```
1  public class ValidarDescontoNoPedidoAdaptador
2      extends ValidarDescontoNoPedidoTest {
3
4      @Override
5      TCResult sendToSut(TCInput input) {
6
7          // Define os valores do pedido a partir do teste
8          PedidoSUT pedido = new PedidoSUT();
9
10         int valorTotal = input.whenPedidoTemValorTotal().intValue();
11
12         pedido.setTotal(valorTotal);
13
14         // Recupera o valor processado pelo PedidoSUT
15         TCResult result = new TCResult();
16
17         result.verifyPedidoTemDesconto(
18             (1 - (pedido.getTotalCalculado() / valorTotal)) * 100
19             );
20
21         return result;
22     }
23 }
```

Figura 2. Exemplo de adaptador implementado pelo programador.

Um das características da ferramenta é a geração de dois ou mais testes a partir de um caso de teste escrito na linguagem LETA. Isso ocorre quando são utilizados os operadores relacionais: **>=**, **<=** ou **!=.** Por exemplo, o caso de teste apresentado na Figura 1, que utiliza o operador >= para o **ValorTotal** (linha 5), resulta em dois subcasos de teste. Um desses casos contém o valor total igual a 200 e outro com o valor 430, um valor maior que 200 e gerado aleatoriamente.

A Figura 3 ilustra esse exemplo, começando com a definição do teste (>= 200) e os dois subcasos gerados (= 200 e = 430) e, por meio do adaptador, os dois valores são enviados ao SUT, representando o componente que está sendo testado.

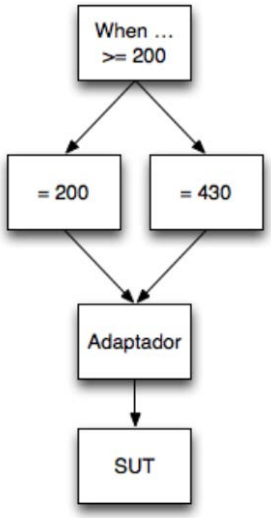

Figura 3. Geração de dois testes executáveis

III. Breve Descrição da Linguagem LETA

Um caso de teste na linguagem LETA é composto por três partes: o seu nome, um critério de sucesso e um cenário do teste.

Para nomear um caso de teste é necessário utilizar a palavra reservada ***Test*** e um identificador. Não existe uma distinção entre dois identificadores quando são usados letras maiúsculas e minúsculas. Por exemplo, os identificadores **departamento** e **Departamento** não são considerados distintos.

A palavra reservada ***When*** define um cenário de teste. Este cenário pode conter regras de negócios e comportamentos do sistema que se desejam testar. A palavra reservada ***Verify*** especifica um critério de sucesso do teste. Essas duas palavras são seguidas de fatos. Um fato é composto por termos e complementos, que também são definidos por meio de identificadores.

Há três tipos de fatos: unário, binário e n-ário. O primeiro é formado por um termo e um complemento. O segundo tipo se caracteriza por ser formado por dois termos e um complemento entre eles. O último é formado por uma intercalação de complementos entre termos. O exemplo na Figura 4 contém a definição de um teste com os três tipos de fatos: um unário na linha 3, dois binários nas linhas 6 e 7, e um n-ário na linha 5.

```
1 Test ValidarComprador
2 Verify
3 Comprador éVálido
4 When
5 Comprador tem Identificador com DígitoVerificador And
6 Comprador tem EndereçoEntrega And
7 Comprador tem CartãoCrédito
```

Figura 4. Um caso de teste na linguagem LETA.

A. Literais e Instâncias de Termos

Há cinco tipos de literais: a cadeia de caracteres, os números inteiros, os números com casas decimais (tendo o ponto como separador), a data com hora e o valor nulo (**null**).

Cada termo pode ter instâncias, que são indivíduos ou casos particulares deste termo. Uma instância é definida por meio de um dos literais. A Figura 5 apresenta cinco termos e suas instâncias.

Figura 5. Exemplo de termos e instâncias de termos.

B. Operadores Lógicos, Relacionais e Fórmulas Matemáticas

Os operadores lógicos ***And***, para conjunção lógica, e ***Or***, para a disjunção, são utilizados para combinar os fatos. A Figura 4 apresenta três fatos com operadores lógicos ***And***.

Os operadores relacionais são usados para expressarem uma comparação entre um termo e uma instância deste termo. Os operadores <, >, <=, >= e **!=** são utilizados para os

valores numéricos e para as datas. Porém, somente o operador **!=** é utilizado para as cadeias de caracteres. Não existe o operador de igualdade entre o termo e sua instância, pois ela está implícita. A Figura 6 apresenta alguns operadores.

Figura 6. Exemplo de operadores relacionais.

Os operadores matemáticos são utilizados nas fórmulas matemáticas. Se um comportamento do sistema consiste em cálculos aritméticos, as fórmulas em um teste ajudam a especificar e a validar esses cálculos. Os operadores são: + (adição), - (subtração), / (divisão), * (multiplicação), ** (potência), % (resto da divisão) e = (igualdade).

A fórmula matemática deve seguir o formato: TR = T1 Op1 ... OpN TN. Em que TR é um termo que pode ter uma instância, T1 e TN são termos com instâncias, e Op1 e OpN são operadores, exceto o operador de igualdade '='. Há também parênteses para modificar as precedências na fórmula matemática. A Figura 7 apresenta uma fórmula.

Figura 7. Exemplo de fórmula.

C. Lista de Valores

Durante a especificação de um teste, pode ocorrer a necessidade de se declarar diversas instâncias dos termos num fato que, consequentemente, pode implicar a repetição destes termos, mas com valores ou instâncias diferenciados.

Para diminuir essa duplicidade de termos e fatos, existe a capacidade de definir um conjunto de valores para cada termo. Foram adotados alguns caracteres especiais para delimitar um conjunto: início ({) e final (}), e o separador de itens (,) para definir listas e valores. Por exemplo, o conjunto **{{1, 1}, {2, 2}, {3, 3}}** tem três listas: **{1, 1}**, **{2, 2}** e **{3, 3}**, com dois valores em cada.

Desse modo, é necessário definir associações entre os termos e a ordem dos valores de uma lista. Para isso, utiliza-se o caractere especial arroba (@) seguido de um número inteiro que indica a ordem do valor. Por exemplo, a partir da lista **{100, 80, 20}** e do fato **Resultado @1 = ValorA @2 + ValorB @3,** o termo **Resultado** está associado ao primeiro valor da lista, o termo **ValorA** associado ao segundo valor e o termo **ValorB** associado ao terceiro valor.

Pode-se utilizar operadores relacionais entre o termo e a ordem para expressar uma relação entre eles. No exemplo **Calculadora tem Resultado >= @1**, é utilizado o operador >= para relacionar o termo **Resultado** com a primeira ordem da lista.

Enfim, a palavra reservada ***Set*** é utilizada para definir o conjunto de listas dentro de um caso de teste. Na Figura 8, existem duas listas com dois valores em cada, sendo o primeiro associado ao termo Apelido (linha 5) e o segundo valor da lista associado ao termo Senha (linha 6).

```
1  Test ValidaUsuário
2  Verify
3  Usuário éValido
4  When
5  Usuário tem Apelido @1 And
6  Usuário tem Senha @2
7  Set
8  {
9      {"usuario1", "senhaA"},
10     {"usuario2", "senhaB"}
11 }
```

Figura 8. Exemplo de lista de valores.

IV. Análise e Resultados de Uso da Ferramenta e da Linguagem LETA

Para avaliar a ferramenta e a linguagem LETA, foi elaborada uma análise comparativa preliminar e informal com outras ferramentas, tais como *JUnit*, *Selenium* e *FitNesse*. Essa avaliação consiste na implementação de um teste de aceitação a partir de uma história encontrada nos exemplos apresentados por Cohn (2004).

A. JUnit

O caso de teste **"Testar com um cartão não permitido: American Express"** implementado em *JUnit* (Beck, 2002) é apresentado na Figura 9. O teste define a classe **CartaoSUT** com os atributos: **nome**, **tipo**, **número** e **vencimento**. Depois, o método **validarCartao** é executado para verificar os dados do cartão e retorna um valor indicando se o cartão é válido ou não. Por fim, é realizada a comparação entre um valor esperado com um valor resultante do método.

Para esse teste, é necessário que a classe **CartaoSUT**, apresentada na Figura 10, seja criada com as suas interfaces ou métodos para que o código do teste seja compilado.

Figura 9. Caso de teste em JUnit.

Figura 10. Classe construída para o teste em JUnit.

A ferramenta JUnit não oferece um meio para dividir a implementação de um teste em duas ou mais partes. Assim, um profissional, seja um analista de qualidade ou programador, pode especificar o caso de teste num único bloco com os dados a serem enviados ao SUT e os critérios de sucesso na forma de assertivas. Também é necessário conhecer a linguagem de programação Java para construir as classes e métodos do teste e para utilizar a API do SUT.

B. Selenium

O caso de teste **"Testar com um cartão não permitido: American Express"** implementado com a API do *Selenium* é apresentado na Figura 11. Esse teste interage com uma página HTML contendo os campos: **tipo-cartao**, **nome-cartao**, **numero-cartao**, **vencimento-cartao** (linhas 9 a 12). Depois que esses campos são preenchidos com dados inválidos de um cartão, o botão **enviar** é acionado (linha 15). Passados dois segundos, é realizada uma validação para verificar se a mensagem de retorno está de acordo com o esperado (linha 21).

Figura 11. Caso de teste implementado com o Selenium.

Novamente, a ferramenta não disponibiliza uma forma clara de divisão do teste. Dessa maneira, um profissional pode implementar o caso de teste com os dados a serem enviados ao SUT e seus critérios de sucesso num único bloco. Além disso, é necessário conhecer a linguagem Java e a programação em HTML com os seus elementos visuais.

C. FitNesse

Uma parte da implementação do teste na ferramenta FitNesse, na forma de tabela, pode ser escrita de acordo com a Tabela 1.

Tabela 1. Tabela na ferramenta FitNesse contendo os dados do cartão inválido.

exemplo.CartaoFixture				
nome	**tipo**	**numero**	**vencimento**	**validarCartao?**
João da Silva	American Express	344168608834057	01/2015	false

A classe adaptadora do teste, chamada **CartaoFixture**, encontra-se na Figura 12. O método **validarCartao** implementado nessa classe adaptadora apresenta a interação com as classes do SUT (linhas 25 a 30).

Para esse teste, é necessário que a classe **CartaoSUT**, apresentada na Figura 10, seja criada com as suas interfaces ou métodos para que o código do teste seja compilado e executado.

Figura 12. Classe adaptadora ou *fixture* para o teste em FitNesse.

Diferentemente das ferramentas anteriores, essa ferramenta divide um teste em duas partes, a definição dos casos de teste e o adaptador (ou *fixture*). Assim, é possível

considerar uma divisão de tarefas entre profissionais de perfis diferentes. Dessa forma, um analista de qualidade pode especificar os casos de teste numa interface Wiki, sendo necessário conhecer os comandos para a construção de uma tabela. Além disso, um programador pode implementar os adaptadores e, para isso, necessita conhecer a linguagem Java para os adaptadores e para utilizar a API do SUT.

D. LETA

A implementação do teste que valida um cartão na linguagem LETA pode ser escrita conforme apresentado na Figura 13. Na cláusula ***Verify***, há a descrição da validação do cartão e a cláusula ***When*** apresenta os dados do cartão que são enviados ao SUT.

Figura 13. Caso de teste na linguagem LETA.

A implementação manual do código adaptador, que herda da classe gerada e contém as classes do SUT para validar o cartão, está apresentada na Figura 14.

```
1  public class CartaoInvalidoAdaptadorTest extends CartaoInvalidoTest {
2
3      @Override
4      TCResult sendToSut(TCInput input) {
5
6          // API do SUT
7          CartaoSUT cartaoSut = new CartaoSUT();
8
9          cartaoSut.setNome(input.whenCartaoTemNome().cartao().temNome().stringValue());
10         cartaoSut.setTipo(input.andCartaoTemTipo().cartao().temTipo().stringValue());
11         cartaoSut.setNumero(input.andCartaoTemNumero().cartao().temNumero().stringValue());
12         cartaoSut.setVencimento(input.andCartaoTemVencimento().cartao().temVencimento().stringValue());
13
14         // Classe contendo a resposta do SUT
15         TCResult result = new TCResult();
16
17         // Recupera o valor da validação do cartão
18         result.verifyCartaoNaoEValido(cartaoSut.validarCartao());
19
20         return result;
21     }
22 }
```

Figura 14. Classe adaptadora que faz a comunicação com o SUT.

A ferramenta LETA também divide um teste em duas partes: a especificação de um caso de teste e o seu adaptador. Assim, um analista de qualidade pode especificar um teste utilizando apenas a linguagem LETA, sem a necessidade de conhecer uma linguagem de programação, como a Java e, ao mesmo tempo, definir uma validação para uma regra de negócio, implícita no teste. A única parte que requer conhecimento de linguagem de programação, e portanto, requerendo o trabalho de um programador, é o adaptador que utiliza a API do SUT.

E. Síntese dos Resultados

De acordo com Uting e Legeard (2006), um teste pode ter as seguintes partes ou componentes: modelo, teste de aceitação, *script* ou procedimento de teste, adaptador e a comunicação com a aplicação.

O modelo representa a parte do software que se deseja testar. O teste de aceitação é um teste de alto nível de abstração, contendo os comportamentos que se deseja testar e detalhes que o modelo não tem. O *script* ou procedimento de teste é um teste executável contendo os passos e comportamentos do teste de aceitação, porém escrito numa linguagem de programação, como Java. O adaptador é um subcomponente de um teste executável que repassa os dados do modelo e do caso de teste para a aplicação (SUT) e recupera os dados processados dela. A comunicação do teste com o software é representada pela API ou por um componente da aplicação.

Assumindo essa categorização, foi elaborada uma relação entre os componentes de um teste e as ferramentas acima analisadas, focalizando o melhor perfil de profissional para implementar um componente de um teste, seja um analista de qualidade e/ou programador. A Tabela 2 apresenta essa relação.

Tabela 2. Implementação de um Componente de Teste por um Profissional e uma Ferramenta de Teste

Componente de teste	**Ferramenta de teste**			
	JUnit	***Selenium***	***FitNesse***	***LETA***
Modelo	Não se aplica	Não se aplica	Não se aplica	O analista de qualidade define formalmente as regras de negócio com os cenários e os critérios de sucesso de um teste
Teste de aceitação	Não se aplica	Não se aplica	O analista de qualidade define casos de teste em tabelas	
Script de teste	O programador escreve o código do teste na forma de classes e métodos	O programador utiliza a API da ferramenta	Não se aplica, pois a ferramenta possui um motor que interpreta uma página HTML ou comandos Wiki	A ferramenta gera os testes executáveis
Adaptador	Não se aplica	Não se aplica	O programador precisa implementar os adaptadores ou *fixtures*	O programador necessita implementar os adaptadores
Comunicação	O	O programador	O programador	O programador

Componente de teste	**Ferramenta de teste**			
	JUnit	***Selenium***	***FitNesse***	***LETA***
com o SUT	programador necessita conhecer a API da aplicação	necessita conhecer a página HTML	precisa conhecer a API do software	precisa conhecer a API da aplicação

De acordo com a Tabela 2 é possível definir três abordagens que são encontradas nas ferramentas. A primeira é apresentada na Figura 15a, em que um teste de aceitação é escrito informalmente, normalmente em linguagem natural, por um analista de qualidade a partir de uma história. Em seguida, o programador deve interpretar a história e implementar manualmente todo o *script* de teste contendo os seus dados, passos e o adaptador para a comunicação com o SUT. Os casos de teste dependerão exclusivamente da interpretação do programador, o que pode se constituir num ponto frágil da atividade de teste. Essa é a abordagem encontrada nas ferramentas JUnit e Selenium.

A Figura 15b se diferencia da anterior devido ao teste de aceitação ser formal, podendo ser escrito diretamente pelo analista de qualidade numa tabela ou por outro meio contendo seus dados e/ou passos. Diminui-se a fragilidade acima mencionada, mas vários detalhes que permitem a realização do teste devem ser definidos de modo informal e manual na forma de comentários em texto livre, requerendo também a interpretação do programador. Assim, esses detalhes e o adaptador são codificados no *script* por um programador. Essa abordagem é encontrada na ferramenta FitNesse.

A última abordagem apresentada na Figura 15c, representa um teste de aceitação contendo um modelo que define as regras de negócio envolvidas num teste, escritas em uma linguagem formal com expressividade muito próxima à linguagem de negócios, bem como os dados concretos do teste. Em seguida, os *scripts* de teste contendo os dados e passos são gerados automaticamente. A única parte manual dessa atividade, que requer a intervenção de um programador, é o código do adaptador. A ferramenta baseada na linguagem LETA adota essa abordagem.

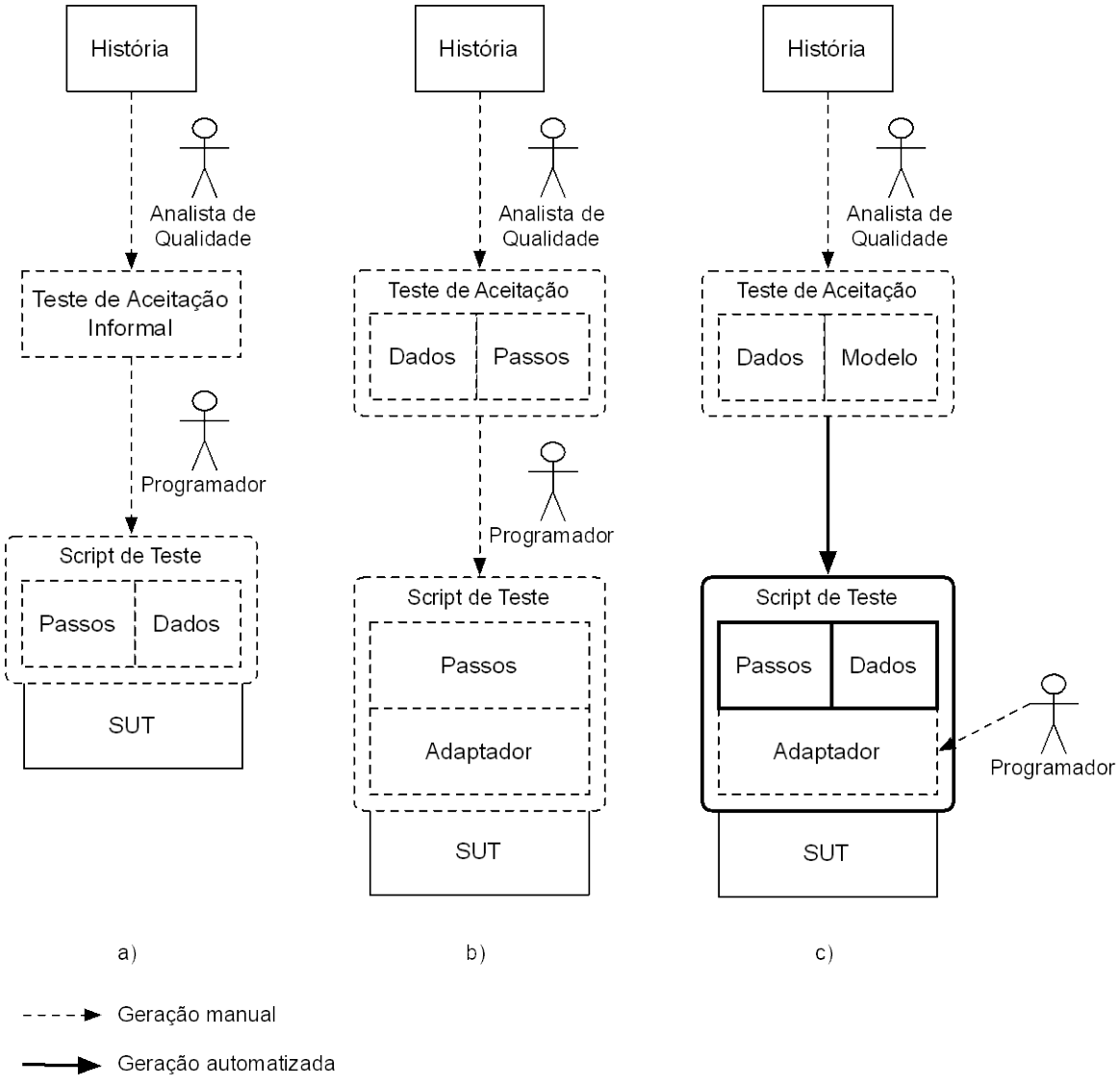

Figura 15. Três abordagens encontradas nas ferramentas de teste

V. Diferenciais e Limitações

A. Diferenciais da Ferramenta e da Linguagem LETA

Na especificação de um teste, um analista de qualidade, que adota a ferramenta e a linguagem LETA, pode definir um caso de teste sem a necessidade de programação, como Java. Diferentemente das ferramentas JUnit e Selenium que necessitam o conhecimento da linguagem Java para especificar um teste e para utilizar a sua API.

Além disso, com a linguagem LETA, há uma independência do teste com respeito à implementação de um funcionalidade durante a especificação de um teste. Com a ferramenta Selenium, na maioria das vezes, é necessária alguma implementação das páginas HTML, ou até um protótipo, para definir a interação dos elementos destas páginas e para a implementação dos critérios de sucesso nos testes.

Por outro lado, as validações de regras de negócio, que estão envolvidas num teste, nem sempre estão claras no código de um teste. Às vezes, é preciso recorrer a outros meios para descrever as regras, as validações necessárias para estas regras e os objetivos de um teste. Com a ferramenta Selenium, há a possibilidade de descrever essas regras na forma de comentários em seu código, um meio informal e dependente da interpretação do programador. A linguagem LETA oferece um meio formal para descrever as regras e as suas validações com base na notação encontrada na especificação SBVR (OMG, 2008), que procura manter a redação dos enunciados o mais próximo possível do discurso corrente dos negócios. A notação permite descrever regras de negócios na forma de termos, fatos e regras. A cláusula **Verify** indica as validações e os objetivos de um teste e a cláusula **When** descreve as regras que estão sendo validadas.

Pode-se observar que a linguagem LETA e FitNesse são muito semelhantes num ponto, em que ambas não requerem que um analista de negócio conheça uma linguagem de programação. A FitNesse oferece um interface Wiki com um conjunto de comandos para formatação e definição de tabelas, contendo os dados que são enviados ao SUT e as validações dos resultados gerados pelo SUT.

A diferença mais relevante entre as duas ferramentas ocorre após a definição de um teste. A ferramenta com a linguagem LETA gera automaticamente combinações de subcasos de teste a partir de um caso de teste escrito na linguagem LETA, o que não ocorre com a FitNesse.

Por exemplo, se definido na clásula **When** um valor de compra maior ou igual (>=) a R$ 1000,00, são gerados automaticamente dois subcasos de teste, um deles contendo o valor R$ 1000,00 e outro o valor R$ 1430,00 (um valor aleatório e acima de R$ 1000,00), ambos utilizados como dados de entrada para um componente do software a ser testado. A Figura 16a ilustra a definição do teste (>= 1000), definido por um analista de qualidade, e a geração de dois subcasos de teste (= 1000 e = 1430) que são enviados ao SUT pelo adaptador.

A ferramenta FitNesse não gera automaticamente casos de teste a partir de uma tabela. A tabela, escrita manualmente por um analista de qualidade, precisa conter todas as possibilidades de testes, assim, uma linha contendo o valor R$ 1000,00 e outra linha contendo o valor R$ 1430,00. A Figura 16b ilustra os dois casos de teste que são enviados ao SUT pelo adaptador.

Todavia, essas duas ferramentas necessitam de adaptadores para repassar os valores definidos nos casos de teste ao SUT. Desta maneira, a ferramenta FitNesse repassa os dados da tabela para o SUT por meio de um adaptador, codificado por um programador, e, num sentido contrário, os resultados do SUT são lidos pelo adaptador e comparados com os dados definidos na tabela. Os adaptadores na ferramenta com a LETA também são codificados por um programador para repassar os dados dos testes de aceitação ao SUT. Eles também recuperam os resultados e os comparam com os valores definidos nos testes de aceitação.

Entretanto, em alguns casos, as regras de negócio não ficam claras numa tabela na FitNesse. Assim, é preciso recorrer ao meio informal e descrever as regras, em linguagem natural, em uma página da ferramenta. Por outro lado, a linguagem LETA tenta aliviar essas situações oferecendo um meio formal para descrever as regras de negócio num teste de aceitação.

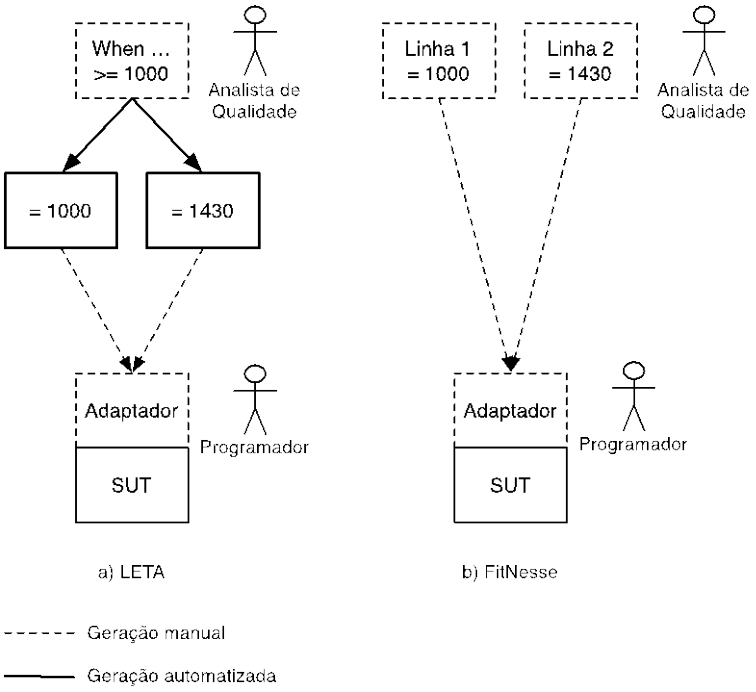

Figura 16. Geração de subcasos de teste na ferramenta LETA (a) e as duas linhas numa tabela da ferramenta FitNesse (b).

B. Limitações da Ferramenta e da Linguagem LETA

A ferramenta e a linguagem LETA, no seu estágio atual, apresentam algumas limitações, como a dependência de um outro mecanismo de execução dos testes. Nesse caso, há uma dependência da ferramenta JUnit, pois os testes executáveis gerados utilizam a sua API e seu mecanismo de execução. Assim, o sucesso de um teste de aceitação, escrito na LETA, depende do sucesso de todos os testes executáveis em JUnit.

Outra limitação encontra-se na geração dos subcasos de teste, em que são gerados apenas aqueles com valores válidos, pois não há critérios de sucesso para valores inválidos. Por exemplo, se houver um software que aplica 10% de desconto em compras maiores ou iguais a R$ 1000,00, podemos definir um teste de aceitação em que o seu critério de sucesso é verificar se o SUT retorna o desconto de 10% para valores acima ou iguais a R$ 1000,00. Mas, não há a geração de um subcaso de teste para valores menores que R$ 1000,00, pois não há um critério de sucesso para estes valores, ou, não se sabe qual desconto aplicar para estes valores.

VI. Conclusões e Trabalhos Futuros

A ferramenta para testes de aceitação baseada na linguagem LETA pode reduzir o tempo de treinamento e aprendizado de um analista de qualidade por meio de uma linguagem simplificada, que não exige conhecimentos de programação. Além disso, é possível reduzir o tempo de implementação dos casos de teste com a geração automática de testes executáveis. Dessa maneira, requer-se apenas a intervenção de um programador para implementar o código adaptador para a comunicação dos testes com o SUT.

Para futuros trabalhos, existem vários pontos de melhorias na linguagem LETA, tais como melhorias na sintaxe para diminuir a repetição de termos, complementos ou outros elementos de um caso de teste.

Outra melhoria consiste em estender a linguagem LETA para interagir com páginas HTML, como a ferramenta Selenium. Desse modo, pode-se gerar automaticamente código para testes executáveis que simulem ações do usuário, como preencher um campo ou pressionar um botão de um formulário em HTML.

Também é possível estendê-la para gerar automaticamente códigos de teste em outras linguagens, como C, C++ e outras.

Outro trabalho futuro consiste na realização de um estudo de caso da aplicação da ferramenta e da linguagem LETA em um ambiente real de teste de aceitação, para mensurar os custos, tempos de aprendizado e treinamento, tempos de implementação dos testes com a técnica EATDD, identificação de outras qualidades ou limitações da linguagem durante a sua utilização, e outros.

VII. Referências

Beck, K. (2004). *Extreme Programming Explained: Embrace Change.* Boston: Addison Wesley.

Beck, K. (2002). *Test-Driven Development: By Example.* Boston: Addison Wesley.

Cohn, M. (2004). *User Stories Applied: For Agile Software Development.* Boston: Addison Wesley.

Crispin, L., & Gregory, J. (2009). *Agile Testing: A Practical Guide for Testers and Agile Teams.* Boston: Addison Wesley.

George, B., & Williams, L. (2003). An Initial Investigation of Test Driven Development In Industry. *ACM Symposium on Applied Computing* (pp. 1135-1139). New York: ACM.

Janzen, D. S., & Saiedian, H. (2008, Maio-Abril). Does Test-Driven Development Really Improve Software Design Quality? *IEEE Software*.

Janzen, D. S., & Saiedian, H. (2006). On the Influence of Test-Driven Development on Software Design. *19th Conference on Software Engineering Education & Training* (p. 8). New York: IEEE.

Martin, R. C. (2011). *FitNesse*. Retrieved 07 10, 2011, from FitNesse: www.fitnesse.org

Mugridge, R., & Cunningham, W. (2005). *Fit for Developing Software: Framework for Integrated Tests.* Upper Saddle River: Prentice Hall PTR.

Myers, G. J. (2004). *The Art of Software Testing.* Hoboken: John Wiley & Sons, Inc.

Nagappan, N., Maximilien, E. M., Bhat, T., & Williams, L. (2008, Fev). Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering* , pp. 289-302.

OMG. (2008). *Semantics of Business Vocabulary and Business Rules.* Needham: OMG.

Park, S. S., & Maurer, F. (2008). The Benefits and Challenges of Executable Acceptance Testing. *2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral* (pp. 19-22). New York: ACM.

Rendell, A. (2008). Effective and Pragmatic Test Driven Development. *Agile 2008 Conference* (p. 6). New York: IEEE.

ThoughtWorks. (2011). *Selenium*. Retrieved 07 10, 2011, from Selenium: seleniumhq.org

Utting, M., & Legeard, B. (2006). *Practical Model-Based Testing: A Tools Approach.* San Francisco: Elsevier.

VU, J. H., Niklas, F., Shenkel-Therolf, C., & Janzen, D. S. (2009). Evaluating Test-Driven Development in an Industry-sponsored Capstone Project. *Sixth International Conference on Information Technology: New Generations* (pp. 229-234). New York: IEEE.