Uma linguagem de domínio específico externa para implementação de testes de unidade e de integração em softwares orientados a objetos

Rafael de Melo Lima Santini[1]

[1]Universidade do Sul de Santa Catarina (UNISUL) – Tubarão, SC – Brasil

rafael@santini.eti.br

Resumo. *Testes de unidade e de integração implementados na própria linguagem utilizada no desenvolvimento do software podem resultar em códigos pouco legíveis e mais verbosos. Nesse contexto, este artigo descreve o modelo de uma linguagem de domínio específico externa e faz a comparação entre testes implementados na linguagem Java com os equivalentes à linguagem proposta. A pesquisa conclui que os testes implementados na nova linguagem são menos verbosos por apresentarem uma quantidade de símbolos e linhas menor em relação aos testes escritos em Java e são mais legíveis ao incorporar as funcionalidades do framework JUnit e da biblioteca Mockito.*

Introdução

Tradicionalmente, no contexto da qualidade de software, testes de unidade e de integração são implementados na própria linguagem utilizada no desenvolvimento do software. Essa prática pode resultar num código de teste pouco legível, pois a utilização de uma linguagem de propósito geral (GPL - *general-purpose language*), como Java, tende a resultar num código verboso, no sentido de que é necessário escrever mais código do que o necessário caso houvesse uma linguagem de domínio específico (DSL - *domain-specific language*) para implementação de testes de unidade e de integração. Além disso, para suportar a execução dos testes, realizar asserções e criar dublês, é necessário adotar frameworks e bibliotecas, aumentando, desta forma, a complexidade na implementação de testes de unidade e de integração.

Assim, profissionais que não estão familiarizados com a linguagem de programação empregada no desenvolvimento do software tendem a encontrar dificuldades na implementação de testes de unidade e de integração. Para Otto (2017, p. 32), "Linguagens de propósito geral possuem muitos recursos, e dificultam o aprendizado e o uso. Uma DSL, ao contrário, deve suportar um mínimo de recursos necessários para resolver o problema do domínio.".

Diferentemente de linguagens de propósito geral, linguagens de domínio específico têm como finalidade resolver um determinado tipo de problema, como a SQL (*Structured Query Language*) para sistemas de gerenciamento de bases de dados relacionais.

[1] Artigo apresentado como trabalho de conclusão de curso de Ciência da Computação da Universidade do Sul de Santa Catarina (Unisul), como requisito parcial para a obtenção do título de Bacharel. Professor Orientador: Charbel Szymanski, Mestre em Engenharia de Automação e Sistemas.

Ao escrever os testes de unidade e de integração numa linguagem de domínio específico externa, de forma autônoma da linguagem utilizada no desenvolvimento do software, torna os testes de unidade e de integração independentes da tecnologia. Dessa forma, caso seja necessário reescrever o software em outra linguagem, os testes já existentes poderão ser utilizados de forma a assegurar que as mesmas funcionalidades e situações cobertas pelos testes estejam presentes na nova implementação.

Pode-se imaginar a seguinte situação: uma empresa possui um software desenhado com diagramas da UML (*Unified Modeling Language*), restrições definidas com a OCL (*Object Constraint Language*), implementado na linguagem Java e com os testes de unidade e de integração também implementados na mesma linguagem empregada no desenvolvimento do software. Por uma situação comercial ou tecnológica, essa empresa precisa reescrever esse software para a plataforma .NET usando a linguagem C#. Como os testes de unidade e de integração são dependentes de outra tecnologia, essa empresa enfrentará dificuldades em reusar o que foi especificado com testes de unidade e de integração.

Nesse sentido, no contexto da qualidade de software, este trabalho propõe uma linguagem de domínio específico externa para implementação de testes de unidade e de integração que possa servir como um modelo a ser adotado no desenvolvimento de software.

Objetivos

Objetivo geral

O objetivo geral deste trabalho é criar uma linguagem de domínio específico externa para implementação de testes de unidade e de integração em softwares orientados a objetos.

Objetivos específicos

Este trabalho tem os seguintes objetivos específicos:

- Definir a linguagem com seus elementos de ordem léxica, sintática e semântica;
- Abranger todas as funcionalidades da arquitetura do framework JUnit;
- Contemplar todos os recursos oferecidos pela biblioteca Mockito;
- Implementar um compilador tendo como alvo a JVM (Java Virtual Machine).

Justificativa

Inicialmente softwares eram testados manualmente, num processo no qual após o programador codificar a solução, o programa era executado de forma manual para observar se a saída esperada era obtida. Se algum erro fosse verificado, o programador realizava a alteração no código e repetia o processo.

Para agilizar o processo de teste, programadores começaram a escrever programas para testar partes do software, de forma a agilizar esse processo. Com isso, os testes podiam ser executados de forma mais rápida, visto que não era necessário informar os parâmetros manualmente a cada execução, e funcionalidades específicas eram testadas separadamente. Porém, não havia uma arquitetura para esses testes e em alguns casos a execução e verificação ainda eram realizados manualmente.

Em 1994, Kent Beck, criador da técnica de desenvolvimento guiado por teste (TDD - Test-Driven Development), escreveu o seu primeiro framework para testes de unidade chamado SUnit (para Smalltalk), que deu origem a arquitetura xUnit, da qual vários frameworks para testes de unidade são baseados, como JUnit (para Java), DUnit (para Delphi), NUnit (para linguagens da plataforma .Net, como C#), dentre outros. (ANICHE, 2012).

Otto (2017, p. 39, grifo nosso) também coloca um aspecto importante de uma linguagem de domínio específico:

> Os especialistas de domínio sem conhecimentos de programação não vão se tornar programadores por usarem DSLs, mas serão capazes de entender as abstrações usadas para implementar as regras do negócio, e se elas são adequadas para cobrir todos os seus cenários. **Como elas são desenvolvidas em um alto nível de abstração, quem as utiliza não precisa se preocupar com detalhes de implementação de baixo nível**.

Assim, nota-se que o estado da arte dos testes de unidade e de integração é composto pelos frameworks da arquitetura xUnit tendo os programas de testes implementados na mesma linguagem na qual o software é desenvolvido, fazendo uso de bibliotecas para auxiliar a implementação dos testes, como bibliotecas para criação de dublês. Porém, observa-se uma oportunidade de inovação, que é a de abstrair os detalhes de baixo nível encontrados quando testes de unidade e de integração são implementados em linguagens de propósito geral e tornar esses testes independentes da linguagem na qual o software é implementado, melhorando, também, a legibilidade do código e tendo esse como uma forma de documentação.

Metodologia

O desenvolvimento da linguagem foi baseado num processo iterativo e incremental, no qual cada iteração resultou na criação e implementação de um conjunto de recursos da linguagem. A adoção desse processo teve como propósito obter um retorno rápido do que estava sendo pesquisado e implementado.

A pesquisa consistiu na avaliação de testes implementados na linguagem Java com o uso do framework JUnit e da biblioteca Mockito. A amostra foi composta por testes no contexto de regras de negócio de sistemas de informação, de exemplos contidos nas documentações do JUnit e do Mockito e de exemplos contidos na literatura.

O processo de especificação da linguagem consistiu nos seguintes passos:

1. A partir da amostra de testes, esses eram reescritos de forma hipotética visando um código menos verboso, mais legível e com a incorporação das funcionalidades do JUnit e do Mockito como comandos na linguagem proposta.
2. A partir dos testes reescritos na linguagem ainda hipotética, os elementos de ordem léxica e sintática eram definidos tendo nesta etapa o primeiro nível de validação da linguagem.
3. De forma iterativa, o processo recomeça no primeiro passo. Em alguns casos os elementos de ordem léxica e sintática já definidos eram ajustados para contemplar as novas necessidades da linguagem.

Revisão teórica

Qualidade de software

É comum as pessoas empregarem o termo qualidade no dia a dia, principalmente ao avaliarem produtos e serviços. Porém, como definir qualidade? Para algumas pessoas, um produto ou serviço terá qualidade caso atenda às suas necessidades e expectativas. Esse conceito é interessante para ser aplicado na definição de qualidade de software. Inthurn (2001, p. 22) define que "Qualidade de software é um conjunto de propriedades a serem satisfeitas de modo que o software atenda às necessidades de seus usuários.".

Um software é desenvolvido para atender a uma necessidade que normalmente está relacionada à resolução de um problema. Assim, espera-se que o software cumpra com as funcionalidades para as quais ele foi projetado e desenvolvido. Dessa forma, pode-se dizer que um software possui qualidade caso satisfaça as necessidades propostas.

O teste de software surge como uma maneira de assegurar a qualidade do software. É por meio dele que é possível verificar se o software está respondendo ou não de acordo com o esperado. Assim, acredita-se que o software possui qualidade caso os testes não apresentem erros. Entretanto, cabe colocar que a não identificação de erros não garante a ausência deles.

SOFTEXRECIFE (2011) coloca que a qualidade de um sistema ou produto é influenciada pelo processo empregado durante o seu desenvolvimento. Nesse contexto, diversos modelos de referência para desenvolvimento e teste de software surgiram. No Brasil existem os modelos Melhoria do Processo de Software Brasileiro (MPS.Br) e Melhoria do Processo de Teste Brasileiro (MPT.Br). Ambos modelos têm como base outros modelos internacionais.

Teste de unidade

O objetivo de um teste de unidade em um software orientado a objetos é testar uma classe sem a dependência de outras classes. Ou seja, procura-se testar a classe de forma isolada. Porém, na maioria das vezes isso não é possível, pois a classe possui algum acoplamento. Embora o cenário ideal seja classes com alta coesão e baixo acoplamento, é praticamente impossível desenvolver um sistema sem nenhum acoplamento. Para resolver esse problema, empregam-se algumas técnicas, como a utilização de dublês.

Aniche (2012, 8) define que "Um teste de unidade testa uma única unidade do nosso sistema. Geralmente, em sistemas orientados a objetos, essa unidade é a classe.".

Inthurn (2001, p. 58) coloca ainda que:

> Em um software que não foi desenvolvido sob os princípios da orientação a objetos, o teste de unidade concentra-se no esforço de verificação da menor unidade de projeto de software – o módulo ou função. Já num software orientado a objetos, torna-se mais complexo trabalhar com as funções de forma isolada, isto é, testar uma função sem levar em conta a sua classe. Além disso, a herança nos permite abrir um leque maior de possibilidades quando falamos em unidade, pois tanto uma classe quanto uma hierarquia de classe podem ser vistas como uma unidade.

Objetos dublês

Como visto anteriormente, um teste de unidade significa testar uma classe sem a dependência de outras classes. Porém, na prática, um software é composto de diversas classes que interagem entre si. Para resolver esse problema, adota-se objetos dublês que são utilizados para resolver as dependências da classe a ser testada, de forma a manter o isolamento e poder reproduzir a situação desejada. É possível encontrar na literatura cinco tipos de objetos dublês: dummy, fake, stub, mock e spy.

Dentre as definições de Fowler (2013) pode-se citar três:

- Dummy: são objetos que não são de fato utilizados e servem apenas para preencher parâmetros de métodos, como um objeto do tipo String;
- Fake: são objetos que possuem implementações funcionais, porém inadequadas para serem utilizadas em produção;
- Stub: são objetos onde apenas os métodos utilizados pelo teste são implementados, cujas respostas são adequadas ao teste em questão.

Objetos mock podem ser criados manualmente ou através de bibliotecas desenvolvidos especificamente para esse propósito, como jMock e Mockito. Dependendo da linguagem de programação, a criação de objetos mock manualmente implica na utilização de uma interface de forma a definir um contrato em comum entre a classe real e a classe que será utilizada para criar o objeto mock. A desvantagem dessa forma é a necessidade de implementar cada método da interface. Por outro lado, as bibliotecas para criação de objetos mock retornam a partir de uma interface ou classe um objeto mock contendo uma implementação padrão para cada método e lhe permitem especificar o que cada método deve retornar.

Massol (2005, p. 155) traz a seguinte definição de um objeto mock:

> Um objeto mock (ou mock resumidamente) é um objeto criado para ficar no lugar de um objeto com o qual o seu código estará colaborando. O seu código pode chamar métodos no objeto mock, o qual fornecerá resultados conforme foi determinado pelos seus testes.

MOCKITO (2013) define que um objeto spy é aquele que chama as implementações reais do objeto, a menos que alguma seja sobrescrita.

Teste de integração

É comum pensar que um teste de integração é um teste de unidade ou vice-versa. Isso ocorre porque um teste de unidade e um teste de integração podem ser escritos da mesma forma. Frameworks da arquitetura xUnit, como o JUnit, podem e, tradicionalmente, são utilizados nesses dois tipos de teste. Além disso, o simples fato de utilizar um objeto real em vez de um dublê num teste de unidade, o tornará, conceitualmente, um teste de integração.

A definição dada por Inthurn (2001, p. 59) diz que “O teste de integração verifica basicamente se as unidades testadas de forma individual executam corretamente quando colocadas juntas, isto é, quando integradas.”.

O teste de integração é, portanto, o teste de uma classe com as suas dependências satisfeitas por objetos reais, de forma a verificar que o conjunto funciona como o esperado, aproximando-se, assim, do cenário real de uso.

Massol (2005, p. 191, grifo nosso) traz uma definição interessante na tentativa de criar um conceito que defina a aproximação entre um teste de unidade e um teste de integração: "**Testes de unidade de integração** se situam entre os testes de unidade lógicos (testando métodos individuais) e os testes funcionais (testando interações entre métodos).". Porém, a maioria da literatura separa claramente os conceitos de teste de unidade, teste de integração e teste funcional.

Teste funcional

Tradicionalmente testes funcionais são especificados a partir de casos de uso, os quais são resultantes dos requisitos funcionais. Para essa especificação dá-se o nome de casos de teste. Os casos de testes, portanto, visam reproduzir a maneira pela qual os usuários utilizarão o software.

Cabe colocar, entretanto, que um teste funcional não implica que esse seja necessariamente executado sob a ótica de uma pessoa utilizando o software através de uma interface gráfica de usuário. Softwares também podem se comunicar entre si. Um software que faz uso de um webservice, por exemplo, é um usuário. Nesse caso, um teste executado sobre esse webservice é considerado um teste funcional.

Assim como existem ferramentas para automação de testes de unidade e de testes de integração, como os frameworks da arquitetura xUnit, existem ferramentas para automação de testes funcionais para interfaces gráficas de usuário. Essas ferramentas permitem que o testador grave as ações a serem executadas sobre uma interface para serem reproduzidas posteriormente. Para sistemas web, uma ferramenta popularmente emprega é o Selenium.

Segundo Inthurn (2001, p. 56), o teste funcional procura descobrir basicamente funções incorretas ou ausentes e erros de interface. Coloca ainda que:

> O teste funcional concentra-se nos requisitos funcionais do software. Através dele torna-se possível verificar as entradas e saídas de cada unidade. Este tipo de teste, preocupa-se com o que, e não como a unidade está sendo executada.

Outros tipos de teste

Os testes de unidade, de integração e funcional formam a estrutura básica para garantir a qualidade do software no que tange as suas funcionalidades. Alguns autores, inclusive, costumam fazer analogia a uma pirâmide, na qual os testes de unidade estão na base, os testes de integração no meio e no topo os testes funcionais. Além desses 3 tipos de teste, existem outros, como:

- Teste de validação: verifica que o software funciona como o cliente espera;
- Teste de sistema: assemelha-se ao teste funcional;
- Teste de recuperação: que visa verificar a tolerância a falhas;
- Teste de segurança: se existem mecanismos de segurança adequados;
- Teste de estresse: como o sistema se comporta com uma carga de uso alta;
- Teste de desempenho: empregado em sistemas de tempo real.

Como o objetivo do trabalho não é tratar sobre qualidade e teste de software, bem como quando empregar cada tipo de teste, optou-se por conceituar os tipos de teste que

estão mais relacionados a arquitetura xUnit, da qual a linguagem de domínio específico irá abordar.

Linguagem de domínio específico

Embora o termo linguagem de domínio específico (DSL - *domain-specific language*) possa parecer novo, linguagens desse tipo existem desde que linguagens de programação começaram a surgir. Expressões regulares, por exemplo, que também são usadas como ferramenta para a construção de compiladores, é uma linguagem de domínio específico. Inclusive, expressões regulares são um ótimo exemplo de como uma linguagem de domínio específico pode ser mais expressiva e legível do que o equivalente numa linguagem de propósito geral (GPL - *general-purpose language*).

Segundo Fowler (2010), linguagens de domínio específico são pequenas linguagens focadas num particular aspecto de um software. Elas podem ser principalmente de duas formas: externa e interna. Uma DSL externa é aquela que é independente da linguagem de propósito geral, como expressões regulares. DSLs do tipo interna, também chamadas de embutidas, são uma forma particular de API (*Application Programming Interface*), frequentemente referidas como *fluent interface*, como a utilizada no framework jMock.

O framework jMock é um exemplo de uma linguagem de domínio específico interna e da aplicação da técnica *fluent interface*. No artigo *Evolving an embedded domain-specific language in Java*, os autores descrevem como o jMock, que era uma biblioteca inicialmente composta de classes para teste de software, evoluiu para uma linguagem de domínio específico embutida. (FREEMAN; PRYCE, 2006).

Parr (2010, p. 16, tradução nossa) coloca que:

> Uma linguagem de domínio específico é nada além do que: uma linguagem de computador projetada para tornar usuários particularmente produtivos em um específico domínio. Exemplos incluem matemática, shell scripts, wikis, UML, XSLT, makefiles, PostScript, gramáticas formais, e até mesmo formatos de arquivo de dados como valores separados por vírgula e XML. O oposto de uma DSL é uma linguagem de propósito geral como C, Java ou Python. No uso comum, DSLs também normalmente têm a conotação de ser menor por causa do seu foco. Isso nem sempre é o caso, como SQL, por exemplo, que é muito maior do que a maioria das linguagens de propósito geral.

Modelo proposto

O modelo proposto visa descrever os principais elementos da linguagem, em especial os recursos referentes ao domínio específico, na forma de diagramas de sintaxe. Buscou-se adotar palavras do domínio específico e coloca-las de uma forma sintática que pudesse tornar o código menos verboso e mais legível.

Características da linguagem

É uma linguagem de domínio específico externa, textual, de tipagem estática com inferência de tipo, com coletor de lixo, assim como o Java, e não orientada a objetos, embora faça uso de classes e objetos. Os comandos do domínio específico são baseados no framework JUnit e na biblioteca Mockito e os comandos de propósito geral são baseados na linguagem Pascal.

Os comandos da linguagem foram pensados de forma a possibilitar uma leitura do código de forma mais fluída, como se fossem sentenças numa linguagem natural, embora não exista a pretensão de caminhar nesse sentido. Por isso adotou-se os comandos da linguagem Pascal em vez da linguagem Java, pois as construções do tipo *if* <*expressão*> *then* <*comando*> estão mais em consonância com os comandos do domínio específico, como *assert that* <*expressão*> *is equals to* <*expressão*>.

Aspectos léxicos

A linguagem possui as seguintes palavras reservadas:

assert that; at least; at least once time; at most; case; do; else; end; equals to; false; for; for each; happens; if; in; in order; is; mock; never happens; not; null; of; one time; repeat; return; spy; the same; then; throw; times; to; true; until; verify that; when; while; with.

Os elementos léxicos, tais como identificadores, literais e números, seguem os mesmos padrões da linguagem Java, não cabendo aqui reproduzi-los.

Aspectos semânticos

A linguagem possui os seguintes aspectos semânticos:

- Verificação estática de tipos;
- Fracamente tipada: não é necessário declarar o tipo, porém, não faz conversão automática;
- Escopo de bloco: identificadores são restritos ao bloco no qual foram declarados;
- Identificadores e comandos são case-sensitive.

Estrutura da linguagem

Assim como uma classe está para uma linguagem orientada a objetos, um caso de teste está para a linguagem proposta. Um caso de teste é composto por um ou mais testes e pelos eventos de inicialização e finalização. A figura a seguir mostra a sintaxe do caso de teste em notação EBNF (*Extended Backus-Naur Form*).

```
CasoDeTeste ::= 'test case' Literal CorpoDoCasoDeTeste 'end'

CorpoDoCasoDeTeste ::= AntesDoCasoDeTeste? AntesDoTeste?
DepoisDoTeste? DepoisDoCasoDeTeste? (Teste)+ 'end'

AntesDoCasoDeTeste ::= 'before test case do' BlocoDeCodigo 'end'

AntesDoTeste ::= 'before test do' BlocoDeCodigo 'end'

DepoisDoTeste ::= 'after test do' BlocoDeCodigo 'end'

DepoisDoCasoDeTeste ::= 'after test case do' BlocoDeCodigo 'end'

Teste ::= 'test' Literal BlocoDeCodigo 'end'

BlocoDeCodigo ::= Comando* 'end'
```

Figura 1. Sintaxe do caso de teste em notação EBNF. Fonte: Elaboração do autor, 2017.

A seguir cada não-terminal é descrito.

Não-terminal CasoDeTeste

Um caso de teste é composto por um ou mais testes e pelos eventos de inicialização e finalização.

Não-terminal CorpoDoCasoDeTeste

Contém o corpo do caso de teste que é composto pelos testes e pelos eventos de inicialização e finalização (antes do caso de teste, antes do teste, depois do teste, depois do caso de teste).

Não-terminal AntesDoCasoDeTeste

Contém o bloco de código que será executando antes do caso de teste iniciar. É executado apenas uma vez antes de qualquer teste ser executado. É indicado para realizar inicializações que serão compartilhadas pelos testes. Por exemplo: se o caso de teste necessita acessar um banco de dados, a conexão pode ser realizada apenas uma vez neste evento.

Não-terminal AntesDoTeste

Contém o bloco de código que será executado antes de cada teste. Ou seja, se o caso de teste possui quatro testes, esse evento irá ocorrer quatro vezes, uma vez antes de cada teste. Pode ser utilizado para realizar alguma ação antes de um teste ser executado. Por exemplo: se os testes acessam um banco de dados, a transação pode ser iniciada neste evento.

Não-terminal DepoisDoTeste

Contém o bloco de código que será executado após cada teste. É útil para realizar alguma ação após um teste ter sido executado. Por exemplo: se os testes realizam modificações no banco de dados, este evento pode ser utilizado para cancelar as modificações realizadas na transação.

Não-terminal DepoisDoCasoDeTeste

Contém o bloco de código que será executado após o caso de teste, ou seja, após todos os testes terem sido executados. Este evento pode ser utilizado para finalizar algum recurso alocado no caso de teste. Por exemplo: fechar a conexão com o banco de dados.

Não-terminal Teste

Um caso de teste pode ser composto de um ou mais testes. Cada teste é composto por um bloco de código.

Não-terminal BlocoDeCodigo

Contém as sentenças que serão executados.

Exemplo de um caso de teste

A figura a seguir mostra o exemplo completo de um caso de teste.

```
test case 'Calculadora'

before test case do
 calculadora = new Calculadora
end

before test do
 calculadora.inicializa
end

after test do
 // ...
end

after test case do
 // ...
end

test 'Soma'
 calculadora.operando1 = 10
 calculadora.operando2 = 30
 resultado = calculadora.soma
 assert that resultado is equals to 40
end

test 'Subtração'
 calculadora.operando1 = 50
 calculadora.operando2 = 25
 resultado = calculadora.subtrai
 assert that resultado is equals to 25
end

end
```

Figura 2. Exemplo do código-fonte de um caso de teste. Fonte: Elaboração do autor, 2017.

A seguir são apresentados os comandos do domínio específico.

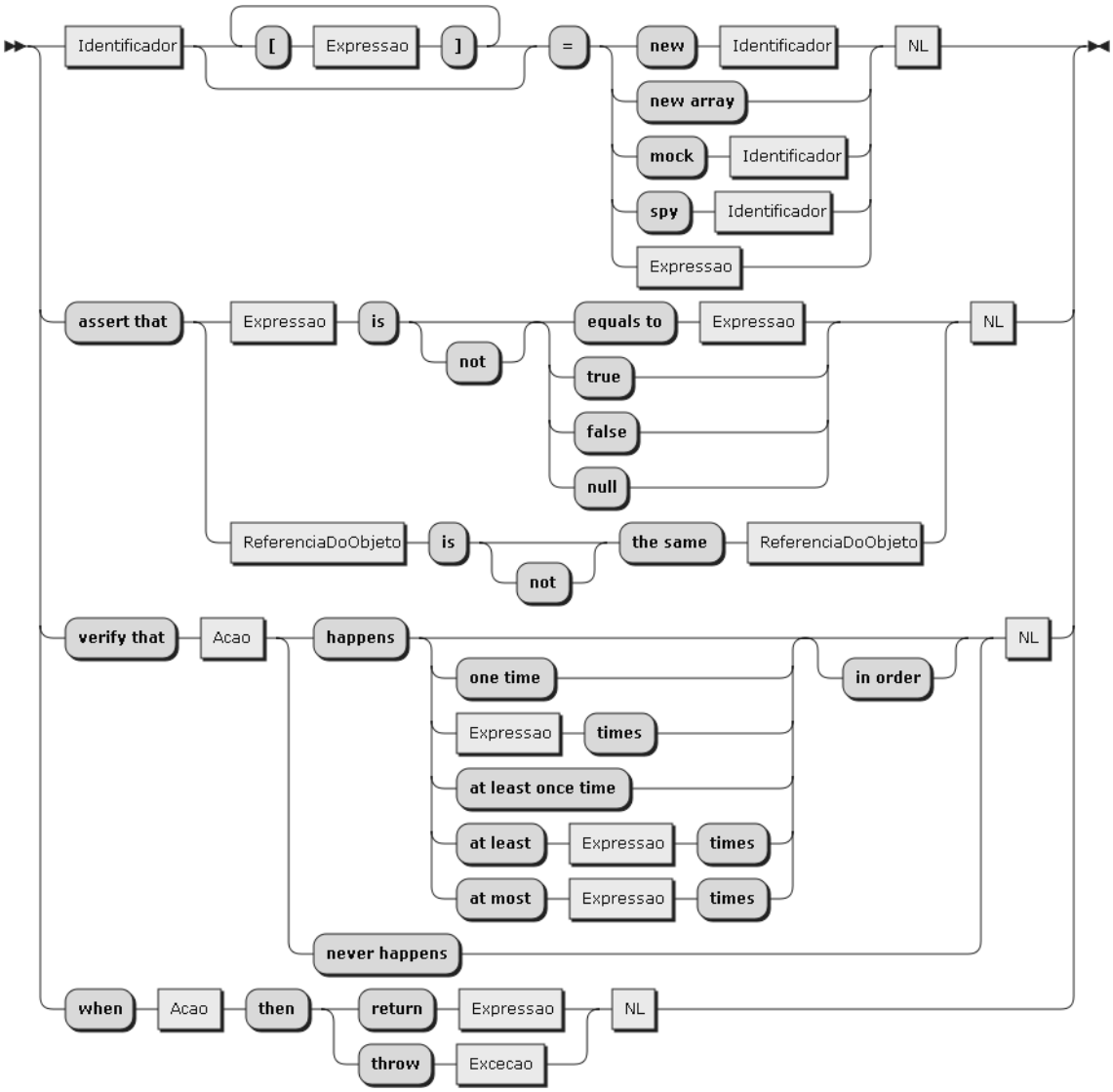

Figura 3. Diagrama de sintaxe dos comandos do domínio específico. Fonte: Elaboração do autor, 2017.

Comando assert

O comando assert é baseado nos métodos de asserção do JUnit e oferece as mesmas funcionalidades, ou seja, para cada método da classe Assert do JUnit, é possível expressar a asserção equivalente com este comando.

Comando mock

O comando mock cria um objeto mock a partir de uma classe ou interface.

Comando spy

O comando spy cria um objeto spy a partir de uma classe ou interface.

Lembra-se que, segundo MOCKITO (2013), um objeto spy é aquele que chama as implementações reais do objeto, a menos que alguma seja sobrescrita.

Comando verify

O comando verify tem por objetivo verificar o comportamento de um objeto mock ou spy.

Comando when

Por padrão, um objeto mock retorna um valor apropriado para cada método da classe ou interface. Se o método retorna um objeto, então será retornado um valor null, se retorna uma coleção, então será retornada uma coleção vazia, se retorna um inteiro, então será retornado zero e assim por diante. Este comando também se aplica aos objetos spy. O comando when tem por objetivo definir o que será retornado quando um método de um objeto mock ou spy é chamado.

Comandos de propósito geral

O diagrama de sintaxe a seguir mostra os comandos de propósito geral baseados na linguagem Pascal com algumas modificações na sintaxe, tais como: cada comando termina com uma nova linha em vez de ponto-e-vírgula, não sendo possível ter mais de um comando por linha, e não se faz uso da palavra begin, exigindo que cada bloco sempre termine com a palavra end.

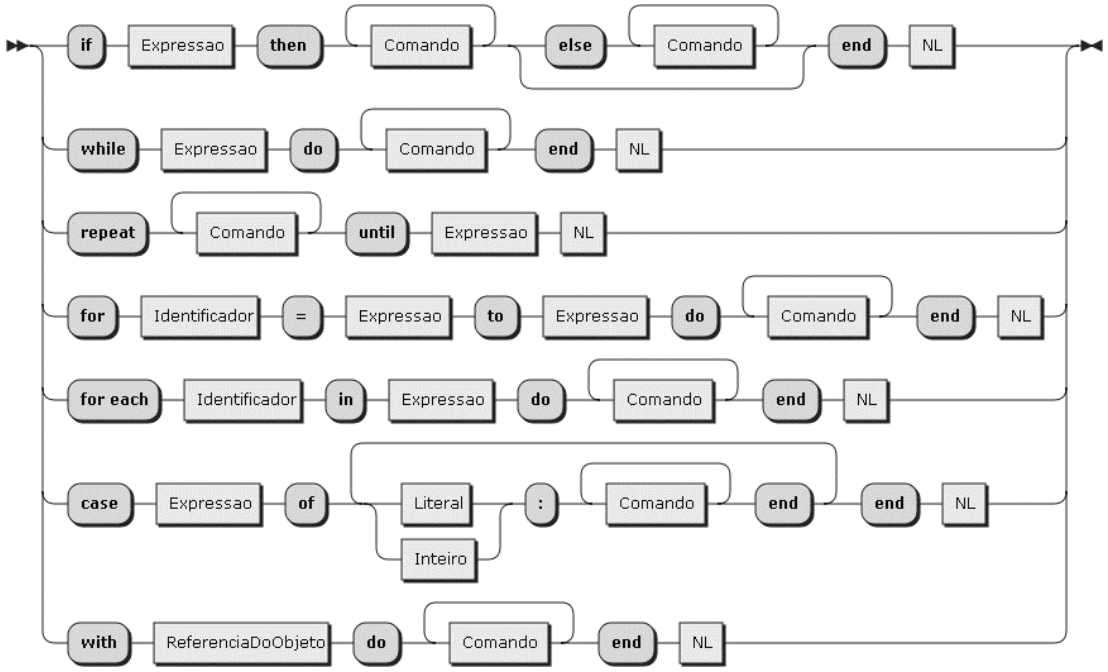

Figura 4. Diagrama de sintaxe dos comandos de propósito geral. Fonte: Elaboração do autor, 2017.

Comando with

A utilização desse comando contribui para tornar o código menos verboso.

Com o comando with é possível acessar um objeto sem a necessidade de fazer referência ao mesmo a cada vez. Os atributos e métodos do objeto referenciado por esse comando se tornam acessíveis no escopo do bloco, tendo a mesma semântica do comando de mesmo nome existente na linguagem Pascal. Como diferencial, caso a classe

implemente as propriedades de acordo com a convenção JavaBeans, essas poderão ser acessadas como atributos (campos).

Implementação de referência

Como forma de validar a linguagem proposta e servir como referência para os aspectos semânticos da linguagem, um compilador para a JVM foi implementando.

O compilador foi implementado na linguagem Java fazendo uso do gerador de parser ANTLR e o processo de tradução foi feito diretamente para bytecode da JVM sem a necessidade de um tradutor intermediário.

Para o efetivo uso em produção ainda é necessário desenvolver plug-ins para os principais ambientes de desenvolvimento integrados (IDEs).

Resultados

Quantitativo

Foram escolhidas como métricas a quantidade de símbolos e de linhas para comparar testes escritos entre a linguagem proposta e a linguagem Java fazendo uso do framework JUnit e da biblioteca Mockito. Para tal, objetivando imparcialidade, a amostra foi composta por 7 exemplos de testes contidos na obra do Aniche (2002) e dos respectivos testes reescritos na linguagem proposta.

Tabela 1. Média de símbolos entre a linguagem proposta (DSL) e a linguagem Java (GPL). Fonte: Elaboração do autor, 2017.

	DSL	GPL	Diferença
Média	60,00	88,00	46,66%
Desvio Padrão	26,39	32,32	

Conforme apresentado na tabela 1, observa-se uma diferença média de 46,66% favorável à linguagem proposta com relação à quantidade de símbolos. Tal resultado foi atingido devido às simplificações que foram possíveis de serem realizadas ao se adotar uma linguagem de domínio específico externa.

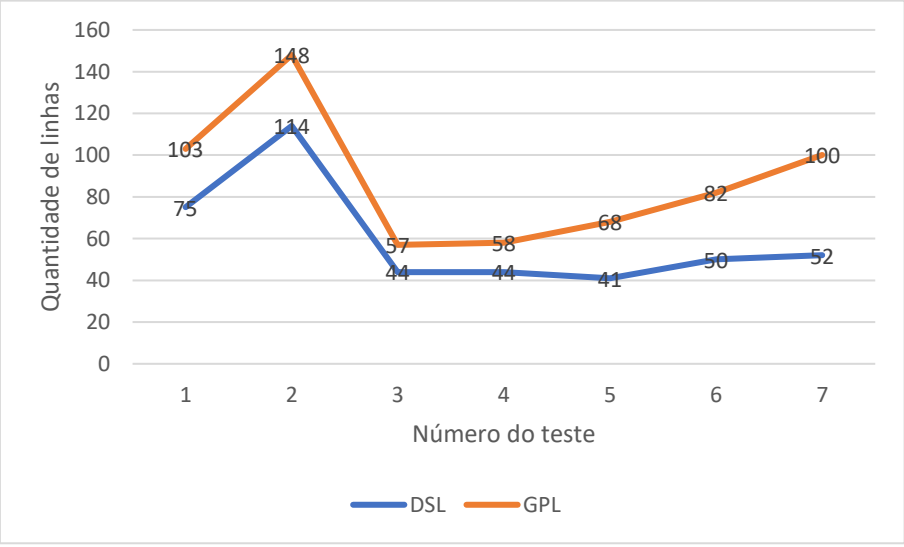

Figura 5. Gráfico comparativo da quantidade de símbolos entre a linguagem proposta (DSL) e a linguagem Java (GPL). Fonte: Elaboração do autor, 2017.

Tabela 2. Média de linhas entre a linguagem proposta (DSL) e a linguagem Java (GPL). Fonte: Elaboração do autor, 2017.

	DSL	GPL	Diferença
Média	12,43	15,00	20,67%
Desvio Padrão	6,19	6,35	

Em relação à quantidade de linhas, observa-se uma diferença média de 20,67% favorável à linguagem proposta, conforme apresentado na tabela 2. Esse resultado pode ser atribuído às simplificações obtidas pela forma como a sintaxe da linguagem foi construída.

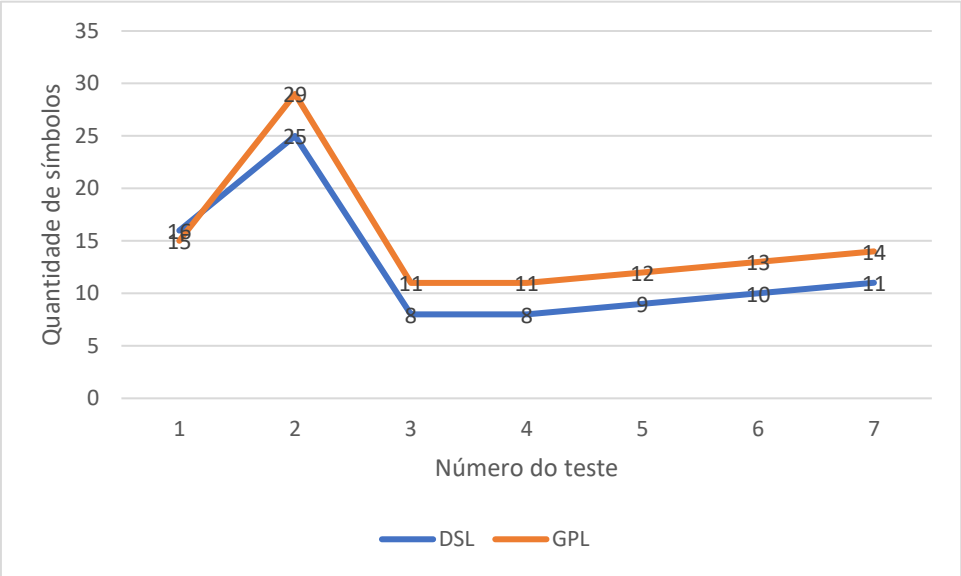

Figura 6. Gráfico comparativo da quantidade de linhas entre a linguagem proposta (DSL) e a linguagem Java (GPL). Fonte: Elaboração do autor, 2017.

Nos gráficos apresentados nas figuras 5 e 6 é possível observar que as curvas das duas linguagens não apresentam discrepância, sendo que a tendência é favorável à linguagem proposta. A linha DSL representa a linguagem proposta e a linha GPL representa a linguagem Java. Em ambos os gráficos a abscissa corresponde ao número do teste. No gráfico da figura 5 a ordenada corresponde à quantidade de símbolos e no gráfico da figura 6 corresponde ao número de linhas.

Qualitativo

Ainda que se tenha obtido bons resultados com relação à quantidade de símbolos e linhas, a verbosidade e legibilidade do código não podem ser avaliados somente por esses aspectos.

O principal apelo da linguagem é propor uma forma inovadora de escrever testes de unidade e de integração por meio de uma linguagem de domínio específico externa. Nesse contexto, a linguagem oferece:

- Um maior nível de abstração em relação ao uso de uma GPL;
- Padronização tendo uma forma restrita para implementação dos testes.

A incorporação das funcionalidades do JUnit e do Mockito como comandos da linguagem contribuiu para um maior nível de abstração em relação ao uso de uma GPL,

que é o esperado para uma DSL, sendo que essas funcionalidades estão representadas sintaticamente na linguagem. Tal nível de abstração é observado pelos comandos *assert*, *mock*, *spy*, *verify* e *when*. Ou seja, o usuário da linguagem precisa apenas conhecer a sintaxe e a semântica desses poucos comandos.

Tendo como comandos da linguagem as funcionalidades para a implementação de testes, se obtém uma forma padronizada, em comparação às diversas formas de implementar testes numa GPL.

Considerações finais

Antes de iniciar o desenvolvimento da linguagem, imaginava-se que o conjunto de comandos do domínio específico da linguagem seria maior. Porém, foi possível representar todas as funcionalidades do framework JUnit e da biblioteca Mockito numa sintaxe enxuta. Somente o comando *assert* da linguagem proposta cobre todas as funcionalidades de asserção oferecidas pelo JUnit. Em relação ao Mockito, todas as funcionalidades estão contempladas em 4 comandos definidos na linguagem: *mock*, *spy*, *verify* e *when*.

Assim, a linguagem possui um conjunto pequeno de comandos. Esse resultado está de acordo com o esperado para uma linguagem de domínio específico, que é possuir uma sintaxe e gramática pequenas, ter um alto nível de abstração e ser voltada para o especialista do domínio.

Referências

ANICHE, Maurício. Test-driven development: teste e design no mundo real. São Paulo: Casa do Código, 2012.

FOWLER, Martin. Domain-Specific Languages. [S.l.]: Addison-Wesley Professional, 2010.

INTHURN, Cândida. Qualidade & teste de software. Florianópolis: Visual Books, 2001.

MASSOL, Vicente; HUSTED, Ted. JUnit em Ação. Rio de Janeiro: Ciência Moderna, 2005.

OTTO, Leonardo. DSL: quebre a barreira entre desenvolvimento e negócios. São Paulo: Casa do Código, 2017.

PARR, Terence. Language implementation patterns: create your own domain-specific and general programming languages. [S.l.]: Pragmatic Bookshelf, 2010.

SOFTEXRECIFE. MPT.Br: melhoria do processo de teste brasileiro: guia de referência do modelo. [S.l.]: SoftexRefice, 2011. Disponível em: <https://goo.gl/Z37FNU>. Acesso em: 6 nov. 2017.

FREEMAN, Steve; PRYCE, Nat. Evolving an embedded domain-specific language in Java. [S.l.]: ACM: New York, 2006. Disponível em: <https://goo.gl/AkV6Xj>. Acesso em: 6 nov. 2017.