UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RUAN CARLOS ALVES DA SILVA

DSL PARA CRIAÇÃO DE TESTES UNITÁRIOS EM JAVA: Uma abordagem inspirada na linguagem Racket

Recife
2025

RUAN CARLOS ALVES DA SILVA

DSL PARA CRIAÇÃO DE TESTES UNITÁRIOS EM JAVA: Uma abordagem inspirada na linguagem racket

Tese/Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de mestre(a) em Ciência da Computação.

Área de Concentração: Engenharia de Software

Orientador (a): Prof. Dr. Henrique Emanuel Mostaert Rebêlo

Recife

2025

Catalogação de Publicação na Fonte. UFPE - Biblioteca Central

Silva, Ruan Carlos Alves da.
DSL para criação de testes unitários em Java: uma abordagem inspirada na linguagem Racket / Ruan Carlos Alves da Silva. - Recife, 2025.
72f.: il.

Dissertação (Mestrado)- Universidade Federal de Pernambuco, Centro de Informática, Programa de Pós-Graduação em Ciência da Computação, 2025.
Orientação: Dr. Henrique Emanuel Mostaert Rebêlo.

1. DSL; 2. Racket; 3. TDD; 4. JUnit 5; 5. ANTLR4. I. Rebêlo, Henrique Emanuel Mostaert. II. Título.

UFPE-Biblioteca Central

Ruan Carlos Alves da Silva

“DSL PARA CRIAÇÃO DE TESTES UNITÁRIOS EM JAVA: Uma abordagem inspirada na linguagem Racket”

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação.

Aprovado em: 29/08/2025.

BANCA EXAMINADORA

_____

Profa. Dra. Paola Rodrigues de Godoy Accioly
Centro de Informática / UFPE

_____

Prof. Dr. Márcio de Medeiros Ribeiro
Instituto de Computação / UFAL

_____

Prof. Dr. Henrique Emanuel Mostaert Rebêlo
Centro de Informática / UFPE
(**orientador**)

RESUMO

Desenvolver software de qualidade continua sendo um desafio, devido à complexidade dos sistemas, à velocidade de entrega, a equipes desalinhadas, a mudanças frequentes de escopo e outros fatores. Nesse cenário, práticas como o *Test-Driven Development* (TDD) podem facilitar a implementação de testes e contribuir para a melhoria da qualidade do software. Contudo, sua adoção ainda apresenta limitações, como a complexidade na elaboração e na manutenção manual dos testes à medida que a aplicação cresce, o que exige tempo e esforço consideráveis para sustentar a prática. Para enfrentar essas dificuldades, esta dissertação propõe uma abordagem que simplifica a implementação e manutenção de testes unitários em Java, incentivando a prática do TDD. A solução consiste em uma *Domain-Specific Language* (DSL) chamada JCheck, inspirada na linguagem Racket e integrada ao framework JUnit 5. O parser da DSL foi gerado com *ANother Tool for Language Recognition*, versão 4 (ANTLR4) e a DSL foi incorporada em uma anotação Java aplicada a métodos, simplificando a especificação dos casos de teste e integrando-os de forma natural ao processo de desenvolvimento. A abordagem foi validada por meio de uma prova de conceito em um cenário realista, executando testes com diferentes instruções da DSL e avaliando o esforço de escrita da DSL por meio da contagem de linhas de código. Como resultado da prova de conceito, a DSL apresentada atingiu os objetivos da pesquisa, realizando com sucesso testes unitários em Java de acordo com as instruções fornecidas, além de demonstrar, por meio da análise da contagem de linhas de código, que sua utilização pode reduzir o esforço na escrita dos testes unitários. Dessa forma, podemos concluir que a JCheck é uma ferramenta promissora para facilitar a criação de testes unitários em Java. Além disso, por adotar uma abordagem que aproxima a definição dos testes dos próprios métodos, a DSL torna mais fácil a prática do TDD.

.

Palavras-chave: DSL; Racket; TDD; JUnit 5; ANTLR4.

ABSTRACT

Developing high-quality software remains a challenge due to system complexity, delivery speed, misaligned teams, frequent scope changes, and other factors. In this scenario, practices such as Test-Driven Development (TDD) can facilitate test implementation and contribute to improving software quality. However, its adoption still presents limitations, such as the complexity in the creation and manual maintenance of tests as the application grows, requiring considerable time and effort to sustain the practice. To address these difficulties, this dissertation proposes an approach that simplifies the implementation and maintenance of unit tests in Java, encouraging the practice of TDD. The solution consists of a Domain-Specific Language (DSL) called JCheck, inspired by the Racket language and integrated with the JUnit 5 framework. The DSL parser was generated with ANother Tool for Language Recognition, version 4 (ANTLR4), and the DSL was incorporated into a Java annotation applied to methods, simplifying the specification of test cases and integrating them naturally into the development process. The approach was validated through a proof of concept in a realistic scenario, executing tests with different DSL instructions and evaluating the effort required to write the DSL by counting lines of code. As a result of the proof of concept, the presented DSL achieved the research objectives, successfully performing unit tests in Java according to the provided instructions, and demonstrating, through the analysis of the line count, that its use can reduce the effort in writing unit tests. Therefore, we can conclude that JCheck is a promising tool to facilitate the creation of unit tests in Java. Furthermore, by adopting an approach that brings the definition of tests closer to the methods themselves, the DSL makes the practice of TDD easier.

Keywords: DSL; Racket; TDD; JUnit 5; ANTLR4.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

ANTLR4	*ANother Tool for Language Recognition*, versão 4
CIO	*Chief Information Officer*
DbC	*Design by Contract*
DSL	*Domain-Specific Language*
JML	*Java Modeling Language*
JSON	*JavaScript Object Notation*
LL(k)	*Left-to-right, Leftmost derivation with k lookahead symbols*
LOC	*Lines of Code*
MBT	*Model-Based Testing*
NASA	*National Aeronautics and Space Administration*
TDD	*Test-Driven Development*
XP	*Extreme Programming*

SUMÁRIO

1 INTRODUÇÃO

Segundo Pressman (2005), na década de 1990, grandes empresas perceberam que bilhões de dólares eram desperdiçados anualmente em softwares que não atendiam às funcionalidades prometidas. Além disso, tanto setores governamentais quanto a indústria expressavam preocupação com falhas críticas em sistemas que poderiam comprometer infraestruturas essenciais, gerando prejuízos ainda maiores. Ainda segundo Pressman (2005), a situação foi noticiada por veículos de comunicação, como a revista *Chief Information Officer* (CIO), que alertavam sobre os altos custos desperdiçados com softwares ineficazes, evidenciando a necessidade de melhores práticas de engenharia de software. Esse contexto histórico destaca a importância de concentrar esforços na produção de software confiável e eficiente, uma preocupação que permanece central na indústria tecnológica atual. Nesse sentido, o processo de produção de software de alta qualidade tornou-se uma incessante busca na indústria desde que a computação passou a desempenhar papel fundamental nos mais diversos tipos de domínios de aplicação (ANDRADE, 2023).

Contudo, controlar a qualidade de sistemas de software é um grande desafio devido à alta complexidade dos produtos e às inúmeras dificuldades relacionadas ao processo de desenvolvimento, que envolve questões humanas, técnicas, burocráticas, de negócio e políticas (BERNARDO; KON, 2008). Nesse cenário, Umar e Zhanfang (2019) afirmam que a escolha de métodos de teste adequados, bem como de ferramentas apropriadas, representa um fator fundamental para o sucesso de projetos com teste de software.

Diante desses métodos, destacam-se algumas práticas que têm o propósito de otimizar a implementação de testes. Entre essas práticas, observa-se a metodologia TDD, que, segundo Maximilien e Williams (2003), consiste em uma prática de desenvolvimento de software em que os testes unitários são escritos antes do código de implementação, seguindo um ciclo iterativo no qual o desenvolvedor cria os testes, implementa o código necessário para satisfazê-los e refina a solução de forma incremental.

Com base nessas informações, é possível afirmar que o TDD não apenas pode organizar o processo de desenvolvimento, mas também pode produzir impactos concretos na qualidade do produto e na experiência dos desenvolvedores.

Para apoiar essa afirmação, Agha et al. (2023) argumenta que o TDD contribui para a melhoria da qualidade do software, pois possibilita a detecção precoce de defeitos e favorece a criação de um código consistente. Os autores também destacam que a prática aumenta a produtividade ao reduzir o tempo de depuração, impactando positivamente os desenvolvedores ao elevar a confiança e a satisfação no trabalho.

Contudo, para que a utilização do TDD seja feita de uma forma ainda mais otimizada, o uso de ferramentas que facilitem a implementação de testes é essencial. Entre essas ferramentas, destaca-se o JUnit. Segundo García et al. (2022), o JUnit é um dos frameworks de teste mais populares e influentes para a linguagem Java. Além disso, segundo Gorla et al. (2025), mesmo com o JUnit consolidado como uma das principais estruturas para testes unitários em Java, a elaboração e a manutenção manuais desses testes ainda representam um desafio, sobretudo em projetos de grande porte, o que tem impulsionado o desenvolvimento de soluções voltadas à criação automática de testes.

Para solucionar o desafio citado anteriormente, uma alternativa eficiente seria a utilização de linguagens específicas de domínio, conhecidas como DSL. Uma DSL é um meio de descrever e gerar membros de uma família de programas dentro de um determinado domínio, sem a necessidade de conhecimento sobre programação geral (KOSAR et al., 2008, p. 390). Além disso, de acordo com Jahić; Guelfi e Ries (2023), as linguagens específicas de domínio favorecem a criação de um ambiente adequado para que especialistas fiquem mais concentrados na resolução de problemas específicos de sua área de conhecimento. Nesse contexto, a utilização de DSL é uma solução estratégica para potencializar o uso do TDD. Segundo Gundlach; Jung e Hasselbring (2023), o uso de linguagens específicas de domínio pode facilitar a aplicação do TDD, permitindo maior abstração e redução do esforço necessário para escrever os testes.

1.1 OBJETIVO

O objetivo principal desta dissertação é apresentar uma solução que facilite a implementação de testes unitários em Java e incentive a prática do TDD. Dito isso, a proposta consiste em uma DSL para a criação de testes unitários, denominada JCheck, baseada em instruções inspiradas na linguagem Racket, utilizando o

framework JUnit 5 como mecanismo de execução desses testes e aplicada no código por meio da anotação `@Check`, restrita apenas a métodos. A escolha do Racket como inspiração para a DSL se justifica especificamente pela presença da expressão `check-expect`, que se encaixa perfeitamente no contexto de instruções para testes.

De acordo com Felleisen et al. (2018), Racket foi originalmente desenvolvida como uma linguagem de ensino, voltada para a construção de outras linguagens de programação, embora atualmente também seja utilizada em aplicações práticas, como jogos e controle de sistemas complexos. Além disso, de acordo com Culpepper et al. (2019), a linguagem Racket adota a programação orientada a linguagens, abordagem defendida em seu manifesto.

Sob o contexto do objetivo, este trabalho também propõe a geração do parser da DSL apresentada, utilizando a gramática desenvolvida com a ferramenta ANTLR4. De acordo com Ladeinde (2023), o ANTLR4 é amplamente reconhecido por possibilitar a criação de gramáticas que vão desde linguagens de programação consolidadas até linguagens personalizadas para contextos específicos.

Outro objetivo é criar uma prova de conceito para validar o funcionamento da DSL JCheck. Essa prova de conceito busca demonstrar, em um cenário mais realista, que é possível executar testes em Java a partir de instruções escritas em uma DSL, além de realizar uma análise do esforço de uso da DSL por meio da contagem de linhas de código. Dessa forma, também será possível verificar se, de fato, essa solução apoia a prática do TDD.

1.2 CONTRIBUIÇÕES DA DISSERTAÇÃO

A principal contribuição desta dissertação é a criação de uma DSL para a escrita de testes unitários em Java, inspirada na linguagem Racket e integrada ao framework JUnit 5. Essa contribuição se estende ao uso da ferramenta ANTLR4 para a geração do parser da DSL.

1.3 ESBOÇO DA DISSERTAÇÃO

A estrutura da dissertação está organizada nas seguintes seções. A Seção 2 fornece os conceitos relacionados ao desenvolvimento da solução proposta. A Seção 3 descreve o desenvolvimento dessa solução. A Seção 4 apresenta a prova de conceito criada para validar a solução. A Seção 5 discute os trabalhos relacionados. A Seção 6 apresenta as conclusões, e a Seção 7 aborda os trabalhos futuros.

2 CONCEITOS

Esta seção tem o objetivo de apresentar os principais elementos utilizados para propor a solução do problema destacado neste trabalho, são eles: Testes de Software, Racket, ANTLR4 e JUnit 5.

2.1 UMA VISÃO GERAL SOBRE TESTES DE SOFTWARE

De acordo com Crespo et al. (2004), o teste de software pode ser entendido como um processo organizado e controlado, cujo objetivo é verificar se um sistema funciona conforme as especificações previamente estabelecidas. Em outras palavras, trata-se de uma prática fundamental para garantir que o que foi planejado seja realmente implementado e se comporte adequadamente, atuando como um mecanismo de validação da qualidade do produto final. Com o intuito de aprofundar a discussão sobre a utilização dos testes de software nesta pesquisa, a seguir serão apresentados conceitos que deram suporte a este trabalho.

2.1.1 Testes Unitários

Dentre os processos de teste de software existentes, destacam-se os testes unitários, que desempenham um papel especialmente importante. Segundo Gomes (2020), esse tipo de teste verifica a lógica interna de uma pequena parte do software, conhecida como unidade, entendida como o menor componente funcional. Com os testes unitários, é possível avaliar de forma precisa a confiabilidade de métodos ou funções isoladamente, o que contribui para detectar problemas desde as fases iniciais e aumentar a robustez do sistema como um todo.

Com base nos conceitos apresentados, fica evidente que os testes de software, especialmente os unitários, são fundamentais para assegurar a qualidade e a confiabilidade dos sistemas. Ao possibilitar a verificação de cada componente e a validação das funcionalidades, esses testes permitem identificar falhas precocemente, reduzir custos de manutenção e elevar a segurança durante o desenvolvimento. Dessa forma, compreender e aplicar corretamente os testes

unitários torna-se essencial para uma abordagem de desenvolvimento orientada à qualidade.

2.1.2 *Test-Driven Development*

Embora os testes unitários sejam uma prática consolidada no desenvolvimento de software, seu potencial pode ser ainda mais bem aproveitado com a utilização do Test-Driven Development (TDD). Essa abordagem consiste em escrever os testes antes da implementação do código. Segundo Abushama, Alassam e Elhaj (2021), o TDD inverte a lógica tradicional do ciclo de vida do software, no qual o código precede os testes. Nesse modelo, o processo se inicia pela elaboração e execução de casos de teste, que orientam tanto a implementação quanto a validação das funcionalidades do sistema.

Além disso, de acordo com Mylsamy (2025), o TDD é uma prática fundamentada na escrita de testes antes da implementação da lógica da aplicação, estruturando-se em um ciclo disciplinado conhecido como Red-Green-Refactor. Nesse processo, o desenvolvedor escreve inicialmente um teste que falha, em seguida implementa o código mínimo necessário para fazê-lo passar e, por fim, realiza a refatoração para melhorar a qualidade interna do código sem alterar seu comportamento externo.

Com base nessa natureza de implementação do TDD, é possível afirmar que a adoção dessa abordagem traz impactos positivos durante o desenvolvimento do software. Segundo Calais e Franzini (2023), embora o TDD possa parecer inicialmente contraintuitivo, seus praticantes defendem que começar pelo teste permite maior foco nos requisitos, favorece a escrita de códigos mais simples e de melhor qualidade e, como consequência, garante que o sistema esteja sempre coberto por testes unitários.

2.2 UMA VISÃO GERAL SOBRE RACKET

De acordo com Felleisen et al. (2015), Racket pode ser compreendida não apenas como uma linguagem de programação, mas como uma família de

linguagens. A ferramenta vai desde uma versão funcional e não tipada, baseada em valores, até variações que incorporam recursos adicionais, como pilhas internas e suporte à tipagem estática.

Ainda mais, de acordo com Felleisen e Flatt (2020), a linguagem Racket teve seu início em 1995. Desde o começo, ela foi criada para ajudar no ensino de matemática e programação nas escolas de ensino fundamental e médio, adotando uma abordagem voltada para a programação funcional. Inicialmente, a linguagem foi nomeada como Jam, uma linguagem simples baseada no Scheme, uma linguagem minimalista da família Lisp, criada na década de 1970. Em 2001, após algumas melhorias, a linguagem mudou de nome e passou a se chamar PLT Scheme, dessa vez utilizando diretamente o Scheme. Ainda os autores, afirmam que no mesmo período, foi desenvolvido um ambiente gráfico nomeado como DrScheme, que mais tarde passou a se chamar DrRacket. Esse ambiente foi projetado para uso educacional. E somente em 2010 a linguagem foi rebatizada como Racket, nome pelo qual é conhecida atualmente.

O tempo passou e hoje, segundo Racket (2025), a linguagem pode ser interpretada de três formas diferentes: como uma linguagem de programação, uma família de linguagens ou como um conjunto de ferramentas voltadas para o uso e desenvolvimento de linguagens. Além disso, Racket (2025) é amplamente utilizada para fins educacionais e a sua estrutura fornece apoio a iniciativas pedagógicas, e também dispõe de ambiente para realização de experimentos e implementação de linguagem de programação. Para acrescentar mais detalhes sobre a estrutura da linguagem apresentada, a seguir serão explicados os mecanismos e conceitos que deram suporte a este trabalho.

2.2.1 Sistema De Macros

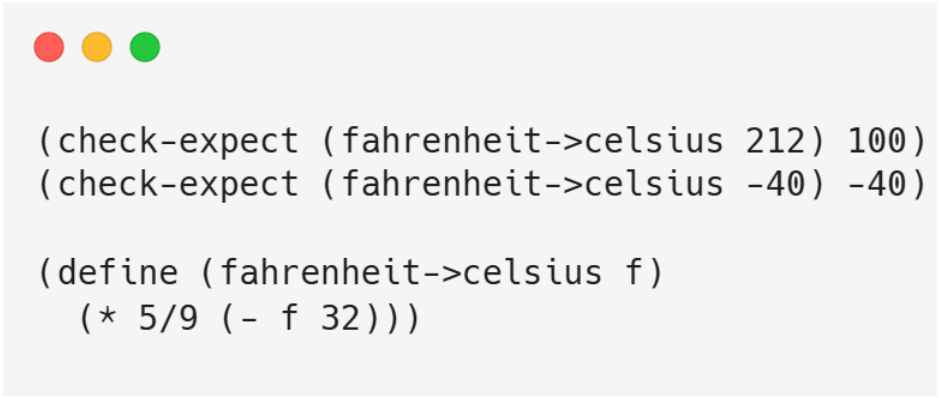
De acordo com o guia oficial do Racket (2025), as macros são estruturas sintáticas que têm um transformador ligado a elas, permitindo expandir a forma original. Elas funcionam como uma extensão do compilador, ajudando na manipulação do código. Além disso, o guia também aponta que muitas das formas sintáticas do Racket (2025) são implementadas como macros, e que o sistema oferece suporte tanto a macros baseadas em padrões quanto a transformadores mais complexos. Além disso, ele fornece facilidades como ferramentas de

depuração e mensagens de erro específicas no nível da macro. Dito isso, é importante destacar que a expressão `check-expect`, abordada neste trabalho, implementa o sistema de macros.

2.2.2 Expressão Check-Expect

Essa expressão funciona fazendo a comparação entre o resultado de uma função ou de um valor fornecido, com o resultado esperado informado ao final da expressão. Ela foi desenvolvida para ser utilizada no contexto educacional, normalmente testando funções definidas pelo aluno. Para mais detalhes, a imagem da Figura 1 apresenta um exemplo de uso da expressão, extraído da documentação oficial.

Figura 1 – Exemplo de uso do *check-expect*



Fonte: RACKET. The Racket Language Levels: Advanced Student. 2025.

A Figura 1 mostra um trecho de código onde é usada a expressão `check-expect`. Nesse trecho, há uma função nomeada `fahrenheit->celsius`, que tem como objetivo receber um valor em *Fahrenheit* e convertê-lo para *Celsius*. Na função, são feitas duas verificações usando `check-expect`: a primeira testa se, ao converter 212ºF, o resultado é 100ºC. A segunda verifica se, ao converter -40ºF, o resultado é -40ºC. Se os valores obtidos estiverem corretos, o programa continua normalmente, sem mostrar nenhuma mensagem. Mas, se algum resultado não for o esperado, o ambiente sinaliza um erro. Ao analisar este exemplo, é possível afirmar que a expressão apresentada dispõe de uma sintaxe simples e prática. Com isso, ela demonstra eficiência e clareza na criação de instruções para execução de testes de código.

2.3 UMA VISÃO GERAL SOBRE O ANTLR4

Nesta seção, serão apresentados conceitos relevantes para este trabalho sobre o gerador de parsers ANTLR4. Os autores Parr e Quong (1995) explicam que o ANTLR foi criado para atender a certas necessidades que surgiram na utilização das ferramentas de geração de analisadores sintáticos disponíveis na época. Embora essas ferramentas oferecessem uma boa capacidade de análise, a maioria dos desenvolvedores preferia escrever seus analisadores manualmente. Essa abordagem buscava maior flexibilidade, um tratamento de erros mais preciso e uma depuração mais fácil. Foi nesse cenário que Parr e Quong (1995) criaram o ANTLR, pensando em oferecer uma alternativa que atendesse às necessidades dos desenvolvedores.

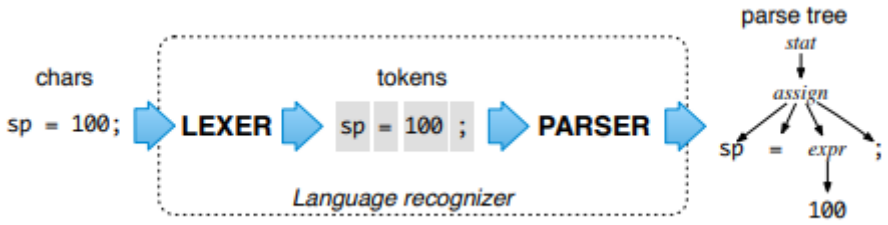
Segundo Parr e Quong (1995), a ferramenta trouxe ideias inovadoras, como o uso de predicados sintáticos e semânticos para orientar a análise, suporte a gramáticas LL(k) (*Left-to-right, Leftmost derivation with k lookahead symbols*), integração entre análise léxica e sintática, além da geração de árvores de sintaxe.
Daí em diante, o ANTLR passou por várias melhorias, chegando na versão atual, o ANTLR4. Essa versão mantém os princípios originais da ferramenta, mas trouxe avanços importantes, facilidade de uso e flexibilidade. Com essas mudanças, o ANTLR4 ficou ainda mais popular, tornando-se uma ferramenta bastante utilizada em diferentes projetos.

De acordo com a documentação oficial ANTLR (2025), o ANTLR4 é um gerador de *parsers* que pode ser utilizado para ler, processar, executar e traduzir textos estruturados. Além disso, o guia oficial também destaca que essa ferramenta está presente na criação de uma grande variedade de linguagens e frameworks. E ainda mais, segundo Tarazona Bernal (2021), algumas funcionalidade de aplicações já consolidadas no mercado usam o ANTLR4, nelas estão incluídas, a análise sintática de consultas no Twitter, a extração de informações jurídicas com a Lex Machina, ferramentas da Oracle, análise de código C++ no NetBeans IDE e a construção da linguagem HQL no framework Hibernate. Diante disso, é evidente que o gerador de parsers apresentado é robusto, possui maturidade consolidada e é utilizado por ferramentas amplamente reconhecidas no mercado.

2.3.1 Funcionamento

O ANTLR4 tem como função ler uma gramática e, com base nela, gerar um código capaz de interpretar instruções de uma DSL. A Figura 2, a seguir, apresenta uma ilustração do funcionamento da ferramenta em questão.

Figura 2 – Reconhecedor de linguagem



Fonte: PARR, 2013. The Definitive ANTLR 4 Reference. Dallas: Pragmatic Bookshelf, 2013. Disponível em: https://media.pragprog.com/titles/tpantlr2/picture.pdf. Acesso em: 10 jul. 2025.

Na imagem, é possível observar que, para reconhecer uma linguagem, o processo é dividido em etapas. A primeira consiste em receber os caracteres e transformá-los em tokens. Na segunda etapa, esses tokens são organizados em estruturas que fazem sentido para a gramática definida, formando o resultado do parser, uma estrutura de dados construída a partir da derivação especificada na gramática. Para reforçar essa explicação, Tarazona Bernal (2021) destaca que a geração de um parser com o ANTLR é dividida em duas etapas: a análise lexical e a análise sintática. A análise lexical corresponde à tokenização, em que os caracteres são agrupados em palavras ou símbolos e classificados em diferentes tipos, como identificadores, inteiros ou números de ponto flutuante. Já a análise sintática tem como objetivo reconhecer a estrutura da frase a partir da sequência de tokens produzida pelo analisador léxico, resultando em uma representação hierárquica denominada árvore sintática.

2.3.2 Gramática

Segundo o ANTLR (2025), a gramática utilizada pelo ANTLR4 corresponde a uma versão ampliada das gramáticas LL(k), o que lhe confere maior poder de análise e flexibilidade na definição das regras. Essa característica permite que a

ferramenta trate estruturas de linguagem mais complexas de maneira eficiente, possibilitando a criação de parsers robustos e ao mesmo tempo adaptáveis às necessidades do desenvolvedor.

Código 1 – Exemplo de gramática ANTLR4.

```
grammar Expr;

prog:   (expr NEWLINE)* ;

expr:   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   INT
    |   '(' expr ')'
    ;

NEWLINE : [\r\n]+ ;
INT     : [0-9]+ ;
```

Fonte: ANTLR (2025). ANTLR4 Documentation. Disponível em: https://www.antlr.org/. Acesso em: 18 jul. 2025.

O Código 1 ilustra um exemplo de gramática utilizada pela ferramenta ANTLR4, que define uma linguagem para expressões aritméticas. É possível observar que ela permite processar uma sequência de expressões, separadas por quebras de linha. As expressões contemplam todas as operações aritméticas para números inteiros e também permitem o agrupamento de sub-expressões. As regras léxicas descrevem os tokens básicos, como números inteiros e quebras de linha. Diante disso, é possível afirmar que essa gramática possibilita a leitura de cálculos matemáticos simples.

2.3.3 Importância da Ferramenta

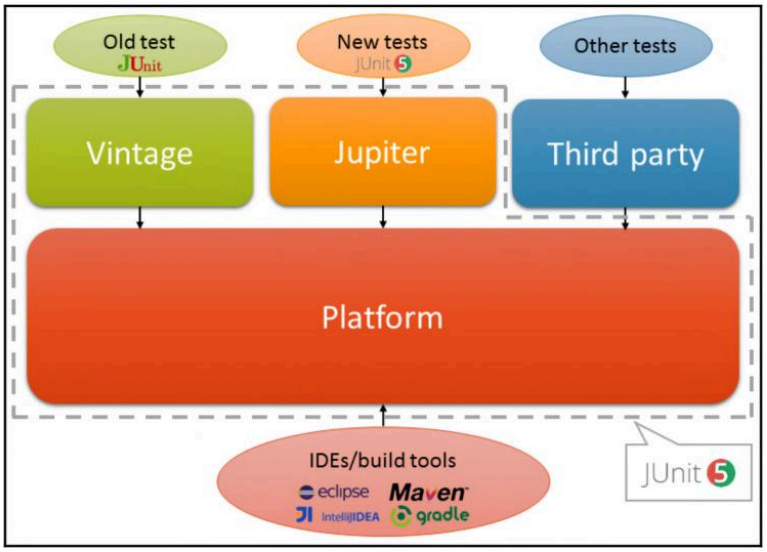
Com base nas informações apresentadas, é possível afirmar que o ANTLR4 é uma ferramenta com bom nível de maturidade e, por isso, é utilizada na construção de funcionalidades em linguagens consolidadas no mercado. Sua capacidade de lidar com gramáticas mais complexas torna a ferramenta útil para projetos que envolvem análise de texto. Diante disso, neste trabalho, o ANTLR4 foi utilizado na construção de uma DSL, desenvolvida para ser utilizada na linguagem Java. Essa DSL é capaz de interpretar instruções como `check-expect 1 1`, as quais têm a função de comparar valores esperados com resultados obtidos. Esta DSL tem o

propósito de simplificar a criação de testes unitários na linguagem Java, conforme o processo será detalhado nas próximas seções.

2.4 UMA VISÃO GERAL SOBRE O JUNIT 5

Nesta seção, são apresentados os principais conceitos sobre o framework JUnit 5. O JUnit trata-se de uma ferramenta amplamente utilizada no desenvolvimento e na execução de testes unitários em código Java, sendo uma das soluções mais consolidadas nesse contexto. Segundo a documentação oficial do JUnit (2025), a versão atual da ferramenta é organizada em três subprojetos principais: Platform, Jupiter e Vintage, cada um responsável por diferentes aspectos da execução e compatibilidade dos testes. A Figura 3 ilustra em maior detalhe os componentes que integram a arquitetura do JUnit 5.

Figura 3 – Arquitetura JUnit 5: Componente de Alto Nível



Fonte: GARCIA, Boni. Mastering Software Testing with JUnit 5. Packt Publishing Ltd, 2017, p. 56.

Ainda segundo a documentação oficial JUnit (2025), o Platform serve como base para *frameworks* de testes na Java Virtual Machine (JVM), facilitando a integração com ferramentas como IDEs e sistemas de *build*. O Jupiter funciona como um núcleo que reúne os componentes essenciais para criar os testes. Já o Vintage atua como uma ponte, garantindo compatibilidade com versões mais antigas do JUnit.

2.4.1 Execução e Organização de Testes

De acordo com o que foi apresentado, é possível afirmar que o JUnit 5 dispõe de uma arquitetura moderna, focada em modularidade e flexibilidade, o que facilita tanto a execução quanto a organização dos testes unitários.

Convém acrescentar que a estrutura dos testes possui um ciclo de vida bem definido, com funcionalidades que ajudam a organizar e facilitar o processo. Para oferecer um panorama mais detalhado sobre essas estruturas mencionadas, o Quadro 1 apresenta as principais anotações fornecidas pela versão atual da ferramenta, conforme descrito na documentação oficial JUnit (2025).

Quadro 1 – Anotações do JUnit Jupiter

Anotação	Descrição
`@Test`	Marca um método como caso de teste que será executado.
`@BeforeEach`	Executa o método antes de cada teste, preparando o ambiente necessário.
`@AfterEach`	Executa o método após cada teste, para limpeza ou reset de estados.
`@BeforeAll`	Executa o método uma única vez antes de todos os testes da classe.
`@AfterAll`	Executa o método uma única vez após todos os testes da classe.
`@Nested`	Permite agrupar testes em classes internas, organizando-os hierarquicamente.
`@DisplayName`	Atribui um nome descritivo ao teste ou à classe para facilitar a leitura dos relatórios.
`@Tag`	Permite categorizar testes para execução seletiva, como por grupos ou características comuns.

Fonte: Elaborado pelo autor a partir de JUNIT (2025).

2.4.2 Vantagens Do Junit 5 Sobre Versões Anteriores

Em comparação com as versões anteriores, o JUnit 5 dispõe de uma arquitetura mais modular e extensível. A versão é dividida em três componentes principais, e essa fragmentação facilita a execução de testes em versões antigas,

como o JUnit 4 e o JUnit 3, e também a integração com ferramentas de apoio, como IDEs, ferramentas de build, como Maven e Gradle, e outras bibliotecas.

O uso de anotações intuitivas, como `@BeforeEach`, `@AfterEach`, `@DisplayName` e `@Disabled`, permite escrever testes mais legíveis, organizados e com menos código desnecessário. Além disso, a ferramenta passou a oferecer suporte à criação de testes unitários dinâmicos por meio da anotação `@TesFactory`, essa funcionalidade é especialmente útil para a execução de múltiplos testes.

2.4.3 JUnit 5 na prática

A seguir, serão apresentados exemplos simples de como criar testes unitários em Java, com o objetivo de mostrar na prática como usar o JUnit. Essa abordagem ajuda a entender como a ferramenta pode ser usada para estruturar e executar testes de forma organizada, facilitando a verificação do funcionamento do código e promovendo boas práticas de desenvolvimento.

Para utilizar o JUnit 5, é necessário configurar a dependência da ferramenta no projeto. Isso pode ser feito por meio de ferramentas de *build*, como Maven e Gradle. A biblioteca utilizada para a versão atual é a `junit-jupiter`, que oferece suporte à engine e à API do JUnit 5. Para mais informações, o Código 2 ilustra um exemplo de como realizar a configuração das dependências.

Código 2 – Exemplo da configuração das dependências do JUnit 5.

```
plugins {
   id 'java'
}

group 'org.example'
version '1.0-SNAPSHOT'

repositories {
   mavenCentral()
}

dependencies {
   testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'
   testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.1'
}

test {
```

```
    useJUnitPlatform()
}
```

Fonte: O autor (2025).

A seguir, o Código 3 demonstra a estrutura da classe usada nos testes com JUnit 5. Essa classe, chamada `User`, tem dois atributos: `name` e `email`. Ela também possui métodos `getters` e `setters` para acessar e modificar esses valores. Essa é uma versão bem simples, criada apenas para ilustrar como implementar os testes unitários.

Código 3 – Exemplo da classe User em java.

```
package org.example.user;

public class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

Fonte: O autor (2025).

Para dar continuidade, o Código 4 apresenta a classe `UserTest`, utilizada para escrever os testes unitários da classe `User`. Nela, é possível observar o uso da API JUnit Jupiter, com testes que cobrem os métodos `getters` e `setters` da classe mencionada anteriormente.

Código 4 – Exemplo de testes unitários com JUnit 5.

```
import org.example.user.User;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNull;

class UserTest {

   private User user;

   @BeforeEach
   void setUp() {
       user = new User("User01", "user01@example.com");
   }

   @Test
   void testGetName() {
       assertEquals("User01", user.getName());
   }

   @Test
   void testGetEmail() {
       assertEquals("user01@example.com", user.getEmail());
   }

   @Test
   void testSetName() {
       user.setName("User02");
       assertEquals("User02", user.getName());
   }

   @Test
   void testSetEmail() {
       user.setEmail("user02@example.com");
       assertEquals("user02@example.com", user.getEmail());
   }

   @Test
   void testConstructorWithNulls() {
       User newUser = new User(null, null);
       assertNull(newUser.getName());
       assertNull(newUser.getEmail());
   }
}
```

Fonte: O autor (2025).

A classe de teste `UserTest` utiliza a anotação `@BeforeEach` para criar uma nova instância da classe `User` antes de cada teste, garantindo que todos os testes sejam executados com valores consistentes e aumentando a confiabilidade dos resultados.

Entre os testes implementados, estão a verificação de que os valores de `name` e `email` são corretamente retornados após a criação do objeto, a atualização desses atributos por meio dos métodos `setName` e `setEmail`, e a criação de objetos com valores nulos, assegurando que a classe mantenha seu comportamento esperado. Para validar os resultados, foram utilizadas funções de asserção do JUnit 5, como `assertEquals` e `assertNull`, que comparam os valores obtidos com os esperados. Esse exemplo evidencia como o JUnit 5 facilita a implementação de testes unitários claros, robustos e eficientes, graças às suas anotações intuitivas e à API bem estruturada.

2.4.4 Importância da Ferramenta

Com base nos benefícios destacados, podemos afirmar que o JUnit 5 é uma ferramenta indispensável para quem trabalha com testes em Java. Sua estrutura organizada, uma API forte, suporte a testes mais dinâmicos e fácil integração com outras ferramentas de desenvolvimento fazem do JUnit 5 uma escolha assertiva para projetos de todos os tamanhos e objetivos. Além disso, por tornar a escrita de testes mais simples e acessível, essa ferramenta ajuda a fortalecer a prática de implementação dos testes unitários, que é fundamental para o desenvolvimento do software.

Diante disso, justifica-se a escolha do JUnit 5 para compor a solução proposta neste trabalho. A ferramenta citada é utilizada como base para a validação dos testes definidos por meio da DSL.

A linguagem específica de domínio desenvolvida nesta pesquisa tem como objetivo simplificar a criação de testes unitários, tornando-os mais legíveis e alinhados ao fluxo natural de desenvolvimento. Para atingir esse propósito de forma eficaz, é fundamental dispor de uma base de execução confiável e consolidada, e ao adotar o JUnit 5 como suporte, a DSL passa a contar com uma estrutura moderna, segura e amplamente utilizada, o que facilita sua aceitação e favorece sua adoção pela comunidade.

3 DESENVOLVIMENTO DA SOLUÇÃO PROPOSTA

Nesta seção, será apresentado como a solução proposta neste trabalho foi desenvolvida. Trata-se de uma DSL criada especialmente para facilitar a definição de testes unitários em código Java e incentivar a prática do TDD. A ideia surgiu da necessidade de tornar a escrita desses testes mais simples e alinhada à forma de pensar dos desenvolvedores. Com esta abordagem, espera-se superar obstáculos que muitas vezes levam os programadores a evitar a criação de testes unitários utilizando ferramentas tradicionais, como o JUnit. Entre as principais limitações do JUnit, destacam-se a elaboração manual de testes, que pode ser repetitiva e trabalhosa, e a necessidade de lidar com uma API detalhada, que requer conhecimento prévio sobre anotações, métodos de asserção e ciclo de vida dos testes. Esses fatores frequentemente desestimulam a produção de testes consistentes e abrangentes.

A DSL JCheck, proposta neste trabalho, foi criada com o objetivo de superar essas dificuldades. Ela permite que os testes sejam definidos de forma mais natural e direta, usando instruções inspiradas na expressão `check-expect`, amplamente utilizada na linguagem Racket. Essa expressão é comum em ambientes educativos, pois auxilia na validação do comportamento de funções durante o desenvolvimento do código. Baseando-se no funcionamento dessa expressão, a DSL surge como uma solução que simplifica a criação e manutenção de testes. A DSL JCheck oferece aos desenvolvedores Java uma maneira mais fácil de escrever testes unitários próximos da linguagem natural, tornando o processo de criação e revisão mais ágil e eficiente.

Para facilitar a construção da DSL, foi utilizada a ferramenta ANTLR4. Ela é responsável pela geração do *parser*, utilizado para interpretar as instruções fornecidas na linguagem de especificação de domínio. A escolha dessa ferramenta foi fundamentada em sua ampla adoção, compatibilidade com o ecossistema Java e capacidade de gerar *parser* robusto a partir de uma gramática descomplicada.

Para executar os testes unitários definidos na DSL, foi utilizado o *framework* JUnit 5. Essa ferramenta é bastante popular por oferecer um suporte completo na execução de testes em código Java, além de ser confiável e fácil de integrar com outras ferramentas do ecossistema. A escolha do JUnit 5 se deu principalmente pelos relatórios detalhados que são gerados após a execução dos testes unitários, o que ajuda bastante a entender o funcionamento da aplicação. Outro ponto importante é seu funcionamento intuitivo, que torna a escrita e organização dos

testes mais simples e prática. Esses benefícios fazem do *framework* uma das opções mais utilizadas pela comunidade, sendo uma escolha sólida e eficiente para desenvolver e executar testes de software.

Dessa forma, após a apresentação dos critérios que justificam a escolha das ferramentas, é importante destacar que os conceitos por trás das ferramentas Racket, ANTLR4 e JUnit 5 já foram abordados na seção “2. Conceitos” deste trabalho. Nesta seção, o objetivo é demonstrar de que maneira essas tecnologias foram efetivamente integradas à estrutura da solução, destacando o papel de cada uma no funcionamento da DSL. Além disso, pode-se afirmar que a escolha destas tecnologias foram essenciais para garantir que a implementação estivesse alinhada ao propósito do trabalho, fortalecendo a conexão entre a pesquisa e sua aplicação técnica.

Com base nesses mecanismos, foi possível construir uma solução que combina a simplicidade da expressão inspirada na linguagem Racket, o poder de análise do ANTLR4 e a robustez do JUnit 5, essa solução oferece uma abordagem prática, acessível e eficiente para a implementação de testes unitários em Java. A seguir, serão apresentados os detalhes da implementação da solução, no qual será descrita a forma como ela foi criada, interpretada e integrada à execução dos testes unitários.

3.1 QUESTÕES DE PESQUISA

Com o objetivo de orientar o desenvolvimento da solução proposta, foram elaboradas as seguintes questões de pesquisa:

- Como construir uma expressão similar ao `check-expect` do Racket em Java, utilizando uma DSL?
- Como integrar a DSL com ferramentas de execução de teste existentes na linguagem Java?
- Como a DSL desenvolvida se comporta em um ambiente que simula o desenvolvimento de software real?

Essas questões têm como objetivo direcionar o estudo, orientando o design da DSL e a avaliação de sua aplicabilidade e eficácia no contexto do desenvolvimento de software.

3.2 METODOLOGIA

Para atingir o objetivo geral da pesquisa, que é desenvolver uma DSL voltada à criação de testes unitários em Java, com o intuito de facilitar a criação dos testes e estimular a prática do TDD, foi necessário dividir o trabalho em três etapas principais. A primeira etapa foi a investigação, que envolveu estudos sobre testes de software, a linguagem Racket, o gerador de parsers ANTLR4 e o framework JUnit 5.

A segunda etapa correspondeu ao desenvolvimento da solução proposta, na qual foi criada a DSL denominada JCheck, inspirada na expressão `check-expect`. Nessa fase, também foram definidos o funcionamento da anotação `@Check` e a gramática da linguagem, elaborada com o uso da ferramenta ANTLR4.

Por fim, a terceira etapa consistiu na prova de conceito, na qual a DSL foi aplicada a uma variedade de métodos em Java e foi realizada uma análise de esforço baseada na contagem de linhas de código, com o objetivo de validar seu funcionamento e demonstrar sua aplicabilidade prática.

3.3 CONSTRUÇÃO DA GRAMÁTICA

A criação da gramática é uma etapa fundamental na implementação da DSL, pois é a partir dela que a estrutura da linguagem é formalmente definida. Para esta solução, a estrutura foi desenvolvida utilizando a ferramenta ANTLR4, que facilita a criação de gramáticas e produz um código consistente na geração do parser. Com base nessa ferramenta, o código da gramática foi escrito seguindo o padrão recomendado, em um arquivo com extensão .g4, no qual são definidas as regras sintáticas necessárias para interpretar as instruções da DSL. O Código 5 apresenta a gramática utilizada como base para gerar o parser da linguagem proposta.

Código 5 – Gramática da DSL Proposta.

```
grammar CheckGrammar;

prog: stmt* EOF;

stmt
   : 'check-expect' expr expr                    # checkExpect
   | 'check-effect' expr expr                    # checkEffect
   ;

expr
   : FLOAT                                        # floatExpr
   | INT                                          # intExpr
   | STRING                                       # stringExpr
   | ID                                           # idExpr
   | '(' expr ')'                                 # parenExpr
   ;

FLOAT: '-'? [0-9]+ '.' [0-9]+;
INT: '-'? [0-9]+;
STRING:
   '"'  (~["\\] | '\\' .)*  '"'
 | '\'' (~['\\] | '\\' .)* '\'';

ID: [a-zA-Z_] [a-zA-Z_0-9]*;

WS: [ \t\r\n]+ -> skip;
```

Fonte: O autor (2025).

A gramática apresentada no Código 5 define a estrutura sintática da DSL proposta neste trabalho. A linguagem gerada por estas definições permite a escrita declarativa de testes que comparam expressões ou efeitos resultantes da execução do código. A seguir, o texto apresenta com mais detalhes as definições ilustradas no código.

A regra `prog` define um programa como uma sequência de instruções encerradas obrigatoriamente por `EOF`, que marca o fim do arquivo. Isso garante que toda a entrada seja consumida. Já a regra `stmt` define as instruções válidas, como os comandos `check-expect` e `check-effect`.

A regra `expr` define as expressões válidas da DSL. Nela, são incorporados tipos literais como inteiros, pontos flutuantes e strings. Além disso, espaços em branco e quebras de linha não afetam a sintaxe da linguagem. Quem cuida disso é a regra `WS`, que usa a diretiva `skip` para ignorar esses elementos na hora da análise.

Com base nisso, é possível afirmar que a gramática apresentada serve como ponto de partida para a criação do *parser* da linguagem proposta. O código gerado a

partir dela permite que as instruções definidas na DSL sejam interpretadas e acessadas por meio da estrutura fornecida pelo ANTLR4.

3.4 IMPLEMENTAÇÃO DA INFRAESTRUTURA

A implementação da DSL proposta é realizada por meio da combinação de duas tecnologias: ANTLR4 e JUnit 5. Essa integração permite que os testes sejam processados e executados no ambiente Java, com base nas instruções escritas pelo desenvolvedor na própria DSL.

Para possibilitar essa integração, o primeiro passo foi a construção da gramática da linguagem. Após o desenvolvimento da gramática, conforme explicado na “Seção 3.3 Construção da Gramática”, o passo seguinte foi utilizar o ANTLR4 para gerar o *parser* da linguagem. O código gerado nesse processo é capaz de reconhecer e interpretar as instruções da DSL. Além disso, o *parser* apresenta uma estrutura que reproduz a árvore sintática da linguagem, o que possibilita a separação dos valores e o correto reconhecimento das instruções no ambiente Java.

Para facilitar a comunicação entre o parser gerado e o JUnit 5, foi adotado o padrão *visitor*, fornecido pela ferramenta. Esse padrão permite associar cada regra sintática a um comportamento específico na aplicação. Com isso, a infraestrutura fornecida percorre a árvore sintática e executa as ações correspondentes de maneira eficiente. Dessa forma, as expressões reconhecidas pela DSL são transformadas em operações lógicas concretas no ambiente Java, tornando o processo mais fluido e organizado.

Além disso, a infraestrutura conta com uma camada de organização responsável por carregar as instruções da DSL a partir da anotação `@Check`. Para realizar esse processo, é utilizada a API de reflexão do Java, acessível pelo pacote `java.lang.reflect`, que faz parte da *Java Standard Library* e está disponível em qualquer ambiente Java. Esse mecanismo permite representar classes e interfaces no momento da execução, fornecendo acesso a estruturas como métodos e atributos.

Para ajudar a entender melhor, os Códigos 6, 7, 8, 9 e 10 dispõem de trechos da implementação que mostram, de forma mais detalhada, como funciona a infraestrutura. Esses trechos de código ilustram como as instruções da DSL são

reconhecidas, processadas e transformadas em testes unitários, por meio da anotação `@Check` e da API de reflexão do Java.

Código 6 – Implementação da Anotação `@Check`.

```
package com.example.ecommerce.annotation;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(Checks.class)
public @interface Check {
   String validation();
   String[] args() default {};
   String mockValues() default "{}";
}
```

Fonte: O autor (2025).

A anotação `@Check`, apresentada no Código 6, foi criada com o objetivo de permitir a definição de testes unitários diretamente associados aos métodos da aplicação, utilizando a DSL proposta como linguagem de validação. Essa anotação é marcada com `@Retention(RetentionPolicy.RUNTIME)`, o que garante que suas informações estejam disponíveis em tempo de execução, possibilitando o acesso por meio da API de reflexão.

Na mesma interface, utiliza-se `@Target(ElementType.METHOD)`, que garante que a anotação seja aplicada exclusivamente a métodos. Além disso, o código emprega `@Repeatable(Checks.class)`, permitindo a aplicação múltipla da mesma anotação sobre um único método. Dessa forma, o desenvolvedor pode criar diferentes testes para um mesmo método, cobrindo variados cenários de validação.

Ainda em relação a interface, observa-se que ela possui três elementos principais. O primeiro é o `validations()`, responsável por indicar a instrução que será interpretada pelo mecanismo da DSL, como nos exemplos `check-expect %s 4.0` e `check-effect currentValue 0`. O segundo é o `args()`, que define os argumentos a serem passados para o método testado, possibilitando sua utilização durante a execução do teste. Um exemplo seria `args = {"2.0", "2.0"}`. Por fim, o terceiro elemento é o `mockValues()`, de uso opcional, cuja finalidade é configurar valores simulados a serem atribuídos a determinados elementos no

momento da execução, permitindo maior controle sobre o ambiente de teste. Esses três elementos funcionam de forma integrada, contribuindo para a correta definição e execução dos testes dentro do ambiente Java.

Código 7 – Definição da Anotação Contêiner `@Checks` para Múltiplos Testes.

```
package com.example.ecommerce.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Checks {
   Check[] value();
}
```

Fonte: O autor (2025).

O Código 7 apresenta um trecho que define a anotação `@Checks`, a qual atua como contêiner para múltiplas anotações `@Check`. No Java, a partir da versão 8, para que uma anotação possa ser aplicada várias vezes sobre o mesmo elemento, ela deve ser marcada com `@Repeatable`, indicando qual anotação contêiner a agrupa. Nesse contexto, a anotação `@Check` é definida com `@Repeatable(Checks.class)`, o que permite que múltiplas instâncias de `@Check` sejam aplicadas a um único método, enquanto `@Checks` funciona como o recipiente dessas anotações.

Código 8 – Implementação do Executor de Testes Baseado na Anotação `@Check`.

```
package com.example.ecommerce.check;

import com.example.ecommerce.annotation.Check;
import com.example.ecommerce.check.utils.JsonUtil;
import com.fasterxml.jackson.core.JsonProcessingException;
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

import java.lang.reflect.Method;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.stream.Stream;
```

```
import static org.junit.jupiter.api.DynamicTest.dynamicTest;

class CheckTest {

   @TestFactory
   Stream<DynamicTest> dynamicCheckTests() {
       Set<Method> allMethods =
TestHelper.getMethodsForTesting();

       return allMethods.stream()
               .flatMap(method -> {
                   Check[] checks =
method.getAnnotationsByType(Check.class);
                   return Arrays.stream(checks).map(check -> {
                       String displayName = String.format(
                               "Test: %s(%s)", method.getName(),
                               String.join(", ", check.args()));
                       return dynamicTest(displayName, () -> {
                           try {
                               String[] args = check.args();
                               Object[] parsedArgs =
TestHelper.parseArguments(args,

method.getParameterTypes());

                               Map<String, Object> mock = new
HashMap<>();
                               if
(!check.mockValues().equals("{}")) {
                                   mock =
JsonUtil.readJsonString(check.mockValues());
                               }

                               TestHelper.methodExecute(method,
check,
                                       parsedArgs, mock);
                           } catch (JsonProcessingException |
NoSuchFieldException
                                    | IllegalAccessException e) {
                               throw new RuntimeException(e);
                           }
                       });
                   });
               });
   }
}
```

Fonte: O autor (2025).

E o Código 8 ilustra como os métodos são organizados para que seus testes unitários sejam executados corretamente. A estratégia emprega a anotação `@TestFactory`, que faz parte da API do JUnit 5 e permite a criação de testes a partir de uma coleção de dados. No código apresentado, essa coleção é

representada por um array que reúne os métodos da aplicação anotados com `@Check`.

A execução dos testes tem início com a recuperação dos métodos que estão marcados com a anotação `@Check`. Em seguida, são extraídas as instâncias dessa anotação para cada método identificado. Dessa forma, torna-se possível executar múltiplos cenários de teste para um mesmo método.

Ainda em relação à execução, cada instância da anotação é convertida em um teste dinâmico e, para facilitar a identificação de cada execução, o nome do teste é gerado com base no nome do método e nos argumentos declarados na anotação. Essa estratégia contribui para um acompanhamento mais eficiente dos casos durante a execução dos testes.

Ademais, como o Java é uma linguagem fortemente tipada, torna-se necessário converter os argumentos para os tipos esperados pelos parâmetros do método antes de sua execução.

Além disso, caso a instância disponha de valores simulados no momento da execução, esses são convertidos de uma string em formato *JavaScript Object Notation* (JSON) para um objeto do tipo Map.

Após todos os parâmetros necessários para a execução estarem prontos, é chamado o método `TestHelper.methodExecute(method, check, parsedArgs, mock)`. Esse método é responsável por invocar o método testado por meio do mecanismo de reflexão da linguagem Java e encaminhar a instrução da DSL para o processamento pelo *parser*. O Código 9 ilustra esse método com mais detalhes.

Código 9 – Execução reflexiva de métodos anotados com `@Check`.

```
public static void methodExecute(Method method, Check check,
Object[] parsedArgs, Map<String,
       Object> mock) throws IllegalAccessException,
NoSuchFieldException {

   Class<?> clazz = method.getDeclaringClass();

   Object classInstance = mockValues(mock, clazz);

   try {
       if(method.getReturnType().equals(Void.TYPE)) {
           method.invoke(classInstance, parsedArgs);
```

```
            DSLRunner.execute(check.validation(), method,
classInstance);
        } else {
            Object methodResponse =
ConverterUtil.convertNonNumberToQuotedString(method.invoke(classI
nstance, parsedArgs));
            DSLRunner.execute(
                    String.format(check.validation(),
JsonUtil.isJsonString(methodResponse)
                            ?
JsonUtil.wrapWithSingleQuotes(String.valueOf(methodResponse)) :
methodResponse),
                    method,
                    classInstance);
        }
    } catch (IllegalAccessException | InvocationTargetException |
JsonProcessingException e) {
        throw new RuntimeException(e);
    }
}
```

Fonte: O autor (2025).

O método apresentado no Código 9, por meio de reflexão, invoca outro método da aplicação e passa para ele os argumentos necessários. Em seguida, faz a validação de acordo com as instruções definidas na DSL. Para isso, ele precisa de alguns insumos: o próprio método a ser invocado, a anotação associada a ele, os argumentos já convertidos para os tipos corretos e, se existirem, os valores simulados reunidos em um Map.

Após obter os parâmetros, o método inicia buscando a classe na qual o método testado está contido. Em seguida, insere os valores simulados nos atributos dessa classe, conforme as definições presentes no `Map<String, Object> mock`. Depois, um bloco `try-catch` é iniciado, contendo uma verificação do tipo do método: se é void ou se possui valor de retorno. Após essa verificação, o método é invocado. Caso seja void, a próxima etapa é a execução da DSL. Caso contrário, o valor de retorno é armazenado em uma variável e utilizado para substituir o marcador de retorno dentro da string da DSL, permitindo a comparação entre o valor esperado e o obtido.

Código 10 – Interpretador da DSL.

```
package com.example.ecommerce.check.antlr4;

import
com.example.ecommerce.check.antlr4.gen.CheckGrammarBaseVisitor;
```

```
import com.example.ecommerce.check.antlr4.gen.CheckGrammarParser;
import com.example.ecommerce.check.utils.CheckHelper;
import com.example.ecommerce.check.utils.ConverterUtil;

import java.lang.reflect.Field;

public class Interpreter extends CheckGrammarBaseVisitor<Object>
{

   private final ExecutionContext executionContext;

   public Interpreter(ExecutionContext executionContext) {
       this.executionContext = executionContext;
   }

   @Override
   public Object
visitCheckExpect(CheckGrammarParser.CheckExpectContext ctx) {

       Object actual = visit(ctx.expr(0));
       Object expected = visit(ctx.expr(1));
       executeCheckExpect(actual, expected);

       return actual;
   }

   @Override
   public Object
visitCheckEffect(CheckGrammarParser.CheckEffectContext ctx) {
       Object attributeName = visit(ctx.expr(0));
       String expected = String.valueOf(visit(ctx.expr(1)));
       try {
           Field field =
executionContext.methodClass.getDeclaredField((String)
attributeName);
           field.setAccessible(true);
           Object value = field.get(executionContext.instance);
           executeCheckExpect(value,
ConverterUtil.convertToSameType(expected, value));
       }catch (NoSuchFieldException | IllegalAccessException
exception) {
           throw new RuntimeException(exception);
       }
       return null;
   }

   @Override
   public Object visitIntExpr(CheckGrammarParser.IntExprContext
ctx) {
       return Integer.valueOf(ctx.INT().getText());
   }

   @Override
   public Object
visitFloatExpr(CheckGrammarParser.FloatExprContext ctx) {
```

```
        return Double.valueOf(ctx.FLOAT().getText());
    }

    @Override
    public Object
visitStringExpr(CheckGrammarParser.StringExprContext ctx) {
        String raw = ctx.STRING().getText();
        if ((raw.startsWith("\"") && raw.endsWith("\"")) ||
(raw.startsWith("'") && raw.endsWith("'"))) {
            return raw.substring(1, raw.length() - 1);
        }
        return raw;
    }

    @Override
    public Object visitIdExpr(CheckGrammarParser.IdExprContext
ctx) {
        return ctx.ID().getText();
    }

    private void executeCheckExpect(Object actual, Object
expected) {
        CheckHelper.checkExpect(
                actual,
                expected,
                """
                Test to method %s failed
                Actual: %s
                Expected: %s
                Path: %s
                """.formatted(executionContext.methodName,
                        actual,
                        expected,
                        executionContext.methodPath)
        );
    }
}
```

Fonte: O autor (2025).

E o Código 10 ilustra a classe Interpreter, responsável por percorrer e interpretar a estrutura sintática da DSL. Essa classe estende `CheckGrammarBaseVisitor`, o que permite implementar o padrão *visitor* sobre os nós da árvore sintática.

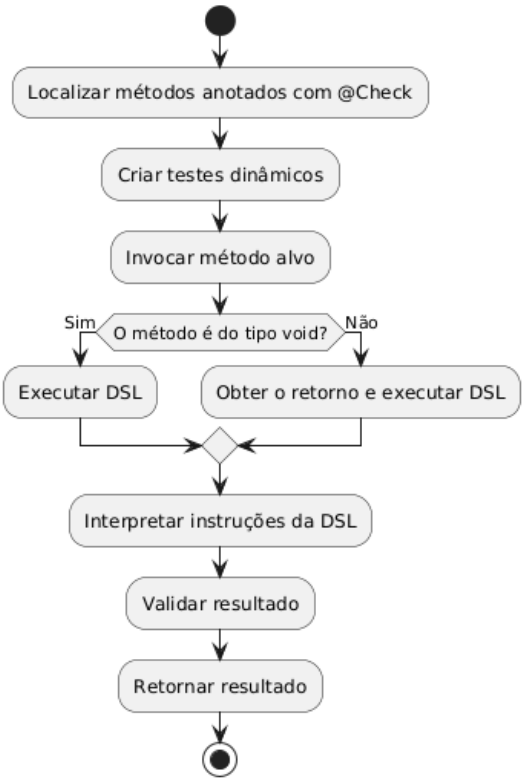
A classe recebe informações do contexto da execução da DSL, essas informações são cruciais para acessar estruturas dos métodos através da reflexão. Essas estruturas contém dados como a classe e a instância do método no teste corrente, além do nome e do caminho do mesmo.

É nessa etapa do processo que são executados os métodos responsáveis pelas instruções `check-expect` e `check-effect`. O método referente à instrução `check-expect` acessa dois atributos do contexto: o primeiro corresponde ao valor retornado pelo método invocado, e o segundo, ao valor esperado. Após obter esses atributos, ambos são passados a um método auxiliar denominado `executeCheckExpect`, responsável por verificar a igualdade entre os valores fornecidos e retornar uma resposta coerente com o resultado da validação.

De forma análoga, o método associado à instrução `check-effect` também acessa dois atributos do contexto. O primeiro indica o nome do atributo da classe onde o método está declarado, e o segundo, o valor esperado que esse atributo deve conter ao final da execução. Com essas informações, o fluxo cria uma instância do campo com base no nome do atributo e, em seguida, recupera o valor associado. Por fim, assim como ocorre na instrução `check-expect`, o método `executeCheckExpect` é acionado para realizar a comparação entre os valores e retornar uma resposta coerente com o resultado da validação.

Com base nas informações apresentadas nesta seção, é possível afirmar que a infraestrutura desenvolvida permite a execução de testes definidos por meio da DSL proposta. A seguir, o diagrama apresenta o fluxo completo da execução da solução.

Figura 4 – Fluxo de execução da solução



Fonte: O autor (2025).

Os componentes apresentados foram desenvolvidos para que as instruções escritas na DSL sejam corretamente interpretadas e aplicadas. Este mecanismo é potencializado pelo uso das tecnologias mencionadas na seção “2. Conceitos”, somadas à anotação `@Check`, criada especialmente para este trabalho, e ao uso de reflexão em tempo de execução, resultando em uma conexão eficiente entre a linguagem de domínio específico e os testes unitários. Com isso, obteve-se uma prática de criação de testes mais acessível, organizada e com menor necessidade de escrita de código. Além disso, devido à proximidade entre os métodos e as instruções de teste, é facilitada a adoção da metodologia TDD, na escrita dos testes utilizando a DSL proposta.

4 PROVA DE CONCEITO

Com o objetivo geral de validar a solução proposta neste trabalho, foi elaborada uma prova de conceito. Segundo Neto et al. (2023), uma prova de conceito é compreendida como uma atividade exploratória e experimental que visa produzir e difundir conhecimento sobre artefatos tecnológicos e seus comportamentos, contribuindo para o avanço de determinada área do saber. Dessa forma, a motivação desta seção é demonstrar, na prática, que a solução proposta pode ser utilizada em um cenário mais realista, validando, assim, a aplicabilidade dos conceitos teóricos discutidos nas seções anteriores.

Para isso, esta prova de conceito tem como objetivo específico comprovar que é possível executar testes unitários em Java utilizando instruções definidas em uma DSL e que o uso da DSL pode demandar menor esforço na construção dos testes unitários do que a utilização exclusiva do JUnit. Para alcançar isso, foram aplicados na prática os recursos apresentados na seção ‘3.2 Implementação da Infraestrutura’ e foi desenvolvida uma análise de esforço baseada na contagem de linhas de código. Além disso, por meio dessa validação, busca-se também evidenciar que a DSL proposta pode simplificar a escrita dos testes e promover a adoção da metodologia TDD.

4.1 DESCRIÇÃO DA IMPLEMENTAÇÃO

A prova de conceito foi desenvolvida utilizando métodos escritos em Java, com foco na diversidade de tipos de dados e de comportamentos. Para isso, os métodos foram marcados com a anotação `@Check`, previamente mencionada na seção "3.2 Implementação da Infraestrutura". Cada uma dessas anotações define uma instrução diferente, escrita por meio da DSL proposta, juntamente com os argumentos para os métodos e os valores simulados. O processamento dessa DSL ocorre no momento da execução dos testes unitários, por meio do comando `mvn clean test -Dtest=CheckTest#dynamicCheckTests`.

Para realizar este estudo, foi utilizada uma API desenvolvida com o *framework Spring Boot*, projetada para simular algumas funcionalidades comuns em um ambiente de *e-commerce*. Esse sistema forneceu um cenário mais realista,

permitindo utilizar a DSL em diferentes camadas da aplicação e verificar sua eficácia em casos variados de uso, como manipulação de dados e validações. O repositório com os códigos executados na prova de conceito está disponível no GitHub de Silva (2025).

Os trechos de código a seguir apresentam os métodos utilizados no estudo, evidenciando como a DSL deve ser utilizada. Essa abordagem reforça que ao utilizar a solução proposta nesta pesquisa, o processo de escrita dos testes se torna mais simples, fluido e alinhado aos princípios da metodologia TDD.

O código 11 apresenta um método de acesso e modificação de atributos do tipo `Long`. Nesse caso, foi utilizado o atributo `id` para demonstrar tanto a validação de retorno, por meio da instrução `check-expect`, quanto a verificação de efeitos colaterais, utilizando a instrução `check-effect`. No método `getId`, a DSL define que o valor retornado deve ser igual a 1, simulando esse valor por meio do parâmetro `mockValues`. Já no método `setId`, a anotação estabelece que, ao receber o argumento 1, o atributo `id` deve refletir corretamente essa alteração, validando o comportamento esperado. O código correspondente é apresentado a seguir:

Código 11 – Testes em métodos `get/set` com tipo `Long` por meio da DSL.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
Long id;
private String name;

@Check(validation = "check-expect %s 1", mockValues = "{ \"id\":
1 }")
public Long getId() {
  return id;
}

@Check(validation = "check-effect id 1", args = { "1" })
public void setId(Long id) {
  this.id = id;
}
```

Fonte: O autor (2025).

Outro exemplo é apresentado no Código 12. Neste caso, o método é responsável por acessar e alterar atributos do tipo `String`, sendo o atributo em questão o `name`. No método `getName`, a DSL testa se o valor retornado é igual a

"name", utilizando o parâmetro mockValues para simular o valor esperado durante o teste. Já no método setName, a DSL verifica se, ao passar o argumento "Category", o atributo name reflete essa mudança corretamente. A seguir, apresenta-se o código correspondente:

Código 12 – Testes em métodos get/set com tipo String por meio da DSL.

```
private String name;

@Check(validation = "check-expect %s 'name'", mockValues = "{
\"name\": \"name\" }")
public String getName() {
  return name;
}

@Check(validation = "check-effect name 'category'", args = {
"category" })
public void setName(String name) {
  this.name = name;
}
```

Fonte: O autor (2025).

Já o Código 13 demonstra que a DSL também dá suporte a métodos que lidam com tipos temporais, evidenciando, assim, sua flexibilidade. O exemplo abaixo apresenta o método addDaysToInstant, responsável por adicionar um número específico de dias a uma data, lançando uma exceção caso a data fornecida seja nula. Para testá-lo, através da DSL foi definido o uso do comando check-expect, especificando que, ao receber como argumentos a data "2025-08-05T14:30:00Z" e o número 15, o resultado esperado deve ser "2025-08-20T14:30:00Z". O código correspondente é apresentado a seguir:

Código 13 – Uso da anotação @Check em método com manipulação de datas do tipo Instant.

```
@Check(validation = "check-expect %s '2025-08-20T14:30:00Z'",
args = {"2025-08-05T14:30:00Z", "15"})
public static Instant addDaysToInstant(Instant baseDate, int
daysToAdd) {
   if (baseDate == null) {
       throw new IllegalArgumentException("The base date cannot
be null.");
   }

   return baseDate
           .atZone(ZoneOffset.UTC)
           .plusDays(daysToAdd)
```

```
            .toInstant();
}
```

Fonte: O autor (2025).

E o Código 14 demonstra como a DSL pode ser utilizada para validar múltiplas entradas em um método que verifica o formato de endereços de e-mail. Por meio da linguagem de domínio específico, foram definidas duas instruções do tipo `check-expect`, uma para validar se o endereço `"user@gmail.com"` é válido e outra para verificar uma entrada inválida, como `"user@gmail"`. Esse tipo de teste comprova que a DSL permite cobrir diferentes casos de teste para o mesmo método. O código correspondente é apresentado a seguir:

Código 14 – Testando formatos de e-mail com diferentes entradas por meio da anotação `@Check`.

```
@Check(validation = "check-expect %s true", args =
{"user@gmail.com"})
@Check(validation = "check-expect %s false", args =
{"user@gmail"})
public static boolean isValidEmail(String email) {
   if (email == null || email.isBlank()) {
       return false;
   }
   return EMAIL_PATTERN.matcher(email).matches();
}
```

Fonte: O autor (2025).

O trecho do Código 15 ilustra como a DSL pode ser aplicada em métodos de cálculo com retorno numérico. Para isso, a instrução definida utiliza o comando `check-expect` para verificar se, ao multiplicar o valor 19,99 por 3, o resultado retornado será 59,97. Esse tipo de teste é especialmente útil em operações do domínio financeiro, em que a precisão e a consistência dos cálculos são essenciais.

O código correspondente é apresentado a seguir:

Código 15 – Testando valor total com suporte da anotação `@Check`.

```
@Check(validation = "check-expect %s 59.97", args = {"19.99",
"3"})
public static BigDecimal calculateTotal(BigDecimal unitPrice, int
quantity) {
   if (unitPrice == null || unitPrice.compareTo(BigDecimal.ZERO)
< 0) {
       throw new IllegalArgumentException("Unit price invalid.");
   }
```

```
    if (quantity < 0) {
        throw new IllegalArgumentException("Quantity invalid.");
    }

    return unitPrice.multiply(BigDecimal.valueOf(quantity));
}
```

Fonte: O autor (2025).

O Código 16 ilustra o uso da DSL em um método responsável pela anonimização de números de CPF, ocultando os seis primeiros dígitos. Para isso, o método recebe uma `String` como parâmetro e, inicialmente, remove todos os caracteres que não são dígitos. Em seguida, verifica se o valor resultante contém exatamente 11 dígitos. Caso essa condição seja atendida, ele retorna o CPF formatado com os seis primeiros dígitos mascarados por asteriscos. Para validar esse método, é definida, por meio da DSL, uma instrução que utiliza o comando `check-expect` para verificar se a saída do CPF `"123.456.789-01"`, passado como parâmetro, é de fato `"***.***.789-01"`. A seguir, está o código correspondente:

Código 16 – Anonimização de CPF com suporte a testes via DSL.

```
@Check(validation = "check-expect %s '***.***.789-01'", args =
{"123.456.789-01"})
public static String maskCPF(String cpf) {
    if (cpf == null) return null;

    cpf = cpf.replaceAll("\\D", "");

    if (cpf.length() != 11) return cpf;

    return "***.***." + cpf.substring(6, 9) + "-" +
cpf.substring(9);
}
```

Fonte: O autor (2025).

O Código 17 apresenta um método utilitário responsável por validar valores monetários, assegurando que estes não sejam nulos nem negativos. Para testar esse cenário, múltiplas instruções em DSL descrevem os comportamentos esperados em diferentes situações, como o retorno de `true` para valores positivos e zero, e `false` para valores negativos. A seguir, está o código correspondente:

Código 17 – Validador de preços com anotação embutida para testes unitários.

```
@Check(validation = "check-expect %s true", args = {"19.99"})
@Check(validation = "check-expect %s false", args = {"-19.99"})
@Check(validation = "check-expect %s true", args = {"0"})
public static boolean isValidPrice(BigDecimal price) {
   return price != null && price.compareTo(BigDecimal.ZERO) >= 0;
}
```

Fonte: O autor (2025).

O Código 18 apresenta um método responsável por formatar valores do tipo `BigDecimal` no padrão monetário brasileiro. Para realizar o teste unitário desse método, é utilizada a linguagem de domínio específico. Neste caso, a expressão da DSL indica que, ao receber o argumento 9.00, o método deve retornar exatamente a string formatada R$ 9,00. A seguir, está o código correspondente:

Código 18 – Conversão de valores monetários para o formato brasileiro com suporte a teste via DSL.

```
@Check(validation = "check-expect %s 'R$ 9,00'", args = {"9.00"})
public static String formatPrice(BigDecimal price) {
   if (price == null) return null;

   NumberFormat format = NumberFormat.getCurrencyInstance(new
Locale("pt", "BR"));
   return format.format(price);
}
```

Fonte: O autor (2025).

4.2 RESULTADOS DA EXECUÇÃO

A primeira etapa da prova de conceito teve como resultado o sucesso da execução da DSL para a execução dos testes unitários dos métodos, abrangendo casos em que o método retorna um valor e aqueles em que o método é do tipo void. Além disso, o mecanismo desenvolvido demonstrou capacidade de lidar com diferentes cenários de execução, retornando respostas consistentes em todos eles. O Quadro 2 apresenta o resultado das execuções dos testes durante a prova de conceito.

Quadro 2 – Resultado da execução dos códigos apresentados na seção “4.1 - Descrição da implementação”.

Método	DSL	Argumento(s)	Mock(s)	Resultado
`getId`	`check-expect %s`	-	`id: 1`	Sucesso

	1			
	check-expect %s 1	-	id: 2	Falha
setId	check-effect id 1	1	-	Sucesso
	check-effect id 1	2	-	Falha
getName	check-expect %s ‘name’	-	name: name	Sucesso
	check-expect %s ‘name’	-	name: usuário	Falha
setName	check-effect name ‘category’	category	-	Sucesso
	check-effect name ‘category’	categoria	-	Falha
addDaysToInstant	check-expect %s ‘2025-08-20T14:30:00Z’	2025-08-05T14:30:00Z	-	Sucesso
		15		
	check-expect %s ‘2025-08-20T14:30:00Z’	2025-08-05T14:30:00Z	-	Falha
		16		
isValidEmail	check-expect %s true	user@gmail.com	-	Sucesso
	check-expect %s false	user@gmail	-	Sucesso
	check-expect %s true	user@gmail	-	Falha
calculateTotal	check-expect %s 59.97	19.99	-	Sucesso
		3		
	check-expect %s 59.97	19.99	-	Falha
		4		
maskCPF	check-expect %s '***.***.789-01'	123.456.789-01	-	Sucesso
	check-expect %s '***.**6.789-01	123.456.789-01	-	Falha

	'			
isValidPrice	check-expect %s true	19.99	-	Sucesso
	check-expect %s false	-19.99	-	Sucesso
	check-expect %s true	0	-	Sucesso
	check-expect %s true	-1	-	Falha
formatPrice	check-expect %s 'R$ 9,00'	9.00	-	Sucesso
	check-expect %s '$ 9,00'	9.00	-	Falha

Fonte: O autor (2025).

Com base nos resultados ilustrados no Quadro 2, foi possível confirmar que esta solução demonstrou capacidade de validar os métodos com base nas instruções fornecidas pela DSL e nas propriedades passadas para a anotação `@Check`.

Além disso, também evidenciou a capacidade da DSL de lidar com diferentes tipos de métodos, desde aqueles que retornam valores até os que produzem efeitos colaterais. Para mais, a integração com a anotação `@Check` mostrou-se eficiente para parametrizar os cenários de teste, permitindo a definição clara e objetiva das condições esperadas para cada método testado.

Além de validar a viabilidade técnica, a prova de conceito também evidenciou os benefícios práticos da abordagem para o desenvolvimento de testes unitários em Java. A DSL proposta demonstrou ser uma ferramenta expressiva e flexível, capaz de simplificar a definição e execução dos testes, promovendo maior organização e reduzindo a complexidade no processo de validação do código.

Além de tudo, os resultados obtidos também reforçam a contribuição deste trabalho para o aprimoramento das práticas de teste de software, especialmente no contexto do desenvolvimento orientado por testes, além de abrir caminho para futuras investigações e melhorias na linguagem e em sua infraestrutura de suporte.

4.3 ANÁLISE DE ESFORÇO POR LINHAS DE CÓDIGO

Com o objetivo de destacar o ganho em termos de esforço no desenvolvimento de testes unitários utilizando a DSL JCheck, em comparação com a escrita de testes unitários usando apenas JUnit, foi realizada uma análise de esforço baseada na quantidade de linhas de código. Para a contagem, foi utilizada a medida *Lines of Code* (LOC), que, segundo Ochodek et al. (2023), é amplamente empregada para estimar o esforço no desenvolvimento de software.

Ainda segundo Ochodek et al. (2023), esse tipo de contagem deve considerar apenas as linhas que efetivamente contribuem para a execução da tarefa em análise, neste caso, a implementação da lógica de testes unitários. Foram contabilizadas linhas referentes à criação de instâncias, chamadas de métodos e `asserts`, enquanto linhas em branco foram excluídas da contagem. Anotações como `@Test` foram incluídas para refletir o esforço completo de implementação. Essa abordagem garante que a métrica LOC esteja mais fortemente correlacionada com o esforço real de desenvolvimento e manutenção dos testes, evitando distorções comuns em contagens de LOC padrão. Para tornar mais clara a forma como foi feita a contagem das linhas de código e destacar o ganho em agilidade na implementação dos testes unitários, a seguir são apresentadas comparações entre o código desenvolvido com a DSL JCheck e a abordagem usando apenas JUnit.

Código 19 – Primeiro exemplo de uso da DSL extraído do Código 11.

```
@Check(validation = "check-expect %s 1", mockValues = "{ \"id\": 1 }")
```

Fonte: O autor (2025).

.

Código 20 – Implementação do exemplo do Código 19 apenas com JUnit.

```
@Test
void shouldReturnIdWhenIdIsOne() {
   Category category = new Category(1L, "");
   assertEquals(1L, category.getId());
}
```

Fonte: O autor (2025).

Quadro 3 – Comparativo de LOC entre a DSL JCheck e o JUnit 5 no método `getId`, ilustrado no Código 11.

Tipo de Implementação	LOC Totais	Redução (%)

DSL JCheck	1	80%
JUnit 5	5	

Fonte: O autor (2025).

No Código 19, a DSL JCheck é utilizada para testar o método `getId`. Por outro lado, o Código 20 mostra como o teste seria feito manualmente com JUnit, exigindo a criação da instância da classe `Category`, a chamada do método e a verificação com `assertEquals`. Esse método manual ocupa mais linhas e exige mais esforço no desenvolvimento. Como evidencia o Quadro 3, a versão com a DSL requer apenas uma linha, enquanto o teste com JUnit puro precisa de cinco, o que representa uma redução de cerca de 80% do total de linhas de código.

Código 21 – Segundo exemplo de uso da DSL extraído do Código 11.

```
@Check(validation = "check-effect id 1", args = { "1" })
```

Fonte: O autor (2025).

Código 22 – Implementação do exemplo do Código 21 apenas com JUnit.

```
@Test
void shouldUpdateIdWhenValueIsOne() {
   Category category = new Category();
   category.setId(1L);
   assertEquals(1L, category.getId());
}
```

Fonte: O autor (2025).

Quadro 4 – Comparativo de LOC entre a DSL JCheck e o JUnit 5 no método `setId`, ilustrado no Código 11.

Tipo de Implementação	LOC Totais	Redução (%)
DSL JCheck	1	83,33%
JUnit 5	6	

Fonte: O autor (2025).

No Código 21, a DSL JCheck é empregada para testar o método `setId`. Com o intuito de fazer a comparação, o Código 22 mostra a implementação manual do mesmo teste com JUnit puro, que exige criar uma instância da classe `Category`, invocar o método setId e verificar o resultado com `assertEquals`. Essa abordagem manual consome mais linhas e requer maior esforço de implementação. Como demonstra o Quadro 4, a versão com a DSL utiliza apenas uma linha, enquanto o

teste em JUnit ocupa seis, representando uma redução de cerca de 83,33% no total de linhas de código.

Código 23 – Primeiro exemplo de uso da DSL extraído do Código 12.

```
@Check(validation = "check-expect %s 'name'", mockValues = "{
\"name\": \"name\" }")
```

Fonte: O autor (2025).

Código 24 – Implementação do exemplo do Código 23 apenas com JUnit.

```
   @Test
   void shouldReturnNameWhenNameIsSet() {
        Category category = new Category(1L, "name");
        assertEquals("name", category.getName());
    }
```

Fonte: O autor (2025).

Quadro 5 – Comparativo de LOC entre a DSL JCheck e o JUnit 5 no método `getName`, ilustrado no Código 12.

Tipo de Implementação	LOC Totais	Redução (%)
DSL JCheck	1	80%
JUnit 5	5	

Fonte: O autor (2025).

No Código 23, a DSL JCheck é utilizada para validar o retorno do método `getName`. Em contraste, o Código 24 mostra a mesma verificação implementada manualmente com JUnit, onde é preciso instanciar a classe `Category`, inicializar seus atributos e realizar a verificação com `assertEquals`. Essa abordagem tradicional demanda mais linhas e maior detalhamento da lógica do teste. Conforme apresentado no Quadro 5, a versão com a DSL requer apenas uma linha de código, enquanto a implementação com JUnit puro utiliza cinco, resultando em uma redução de cerca de 80% no total de linhas.

Código 25 – Segundo exemplo de uso da DSL extraído do Código 12.

```
@Check(validation = "check-effect name 'category'", args = {
"category" })
```

Fonte: O autor (2025).

Código 26 – Implementação do exemplo do Código 25 apenas com JUnit.

```
   @Test
   void shouldSetNameWhenCategoryIsGiven() {
        Category category = new Category();
        category.setName("category");
        assertEquals("category", category.getName());
```

```
    }
```

Fonte: O autor (2025).

Quadro 6 – Comparativo de LOC entre a DSL JCheck e o JUnit 5 no método `setName`, ilustrado no Código 12.

Tipo de Implementação	LOC Totais	Redução (%)
DSL JCheck	1	83,33%
JUnit 5	6	

Fonte: O autor (2025).

No Código 25, a DSL JCheck é empregada para validar o comportamento do método `setName`. Já o Código 26 apresenta a mesma verificação implementada manualmente com JUnit, exigindo a criação de uma instância da classe `Category`, a chamada explícita do método `setName` e a posterior validação com `assertEquals`. Essa forma tradicional demanda mais etapas e maior detalhamento na escrita do teste. Como evidenciado no Quadro 6, a versão com a DSL utiliza apenas uma linha de código, enquanto a implementação com JUnit puro requer seis, resultando em uma redução de cerca de 83,33% no total de linhas de código.

Código 27 – Exemplo de uso da DSL extraído do Código 13.

```
@Check(validation = "check-expect %s '2025-08-20T14:30:00Z'",
args = {"2025-08-05T14:30:00Z", "15"})
```

Fonte: O autor (2025).

Código 28 – Implementação do exemplo do Código 13 apenas com JUnit.

```
@Test
void shouldReturnInstantPlus15Days() {
   Instant baseDate = Instant.parse("2025-08-05T14:30:00Z");
   Instant result = DateUtil.addDaysToInstant(baseDate, 15);
   assertEquals(Instant.parse("2025-08-20T14:30:00Z"), result);
}
```

Fonte: O autor (2025).

Quadro 7 – Comparativo de LOC entre a DSL JCheck e o JUnit 5 no método `addDaysToInstant`, ilustrado no Código 13.

Tipo de Implementação	LOC Totais	Redução (%)
DSL JCheck	1	83,33%
JUnit 5	6	

Fonte: O autor (2025).

No Código 27, a DSL JCheck é utilizada para verificar o comportamento do método `addDaysToInstant`. Já o Código 28 mostra a mesma validação feita manualmente com JUnit, exigindo a criação da data base, a chamada explícita do método e a comparação do resultado com `assertEquals`. Essa implementação tradicional demanda mais instruções e detalhamento da lógica do teste. Conforme apresentado no Quadro 7, a versão com a DSL requer apenas uma linha, enquanto o teste em JUnit ocupa seis, representando uma redução de cerca de 83,33% no total de linhas de código necessárias para realizar o teste.

Código 29 – Exemplo de uso da DSL extraído do Código 14.

```
@Check(validation = "check-expect %s true", args =
{"user@gmail.com"})
@Check(validation = "check-expect %s false", args =
{"user@gmail"})
```

Fonte: O autor (2025).

Código 30 – Implementação do exemplo do Código 14 apenas com JUnit.

```
   @Test
   void shouldReturnTrueWhenEmailIsValid() {
      boolean result = EmailUtil.isValidEmail("user@gmail.com");
      assertTrue(result);
   }

   @Test
    void shouldValidateEmailCorrectly() {
        boolean resultForCorrectEmail =
EmailUtil.isValidEmail("user@gmail.com");
        assertTrue(resultForCorrectEmail);
        boolean resultForInCorrectEmail =
EmailUtil.isValidEmail("user@gmail");
        assertFalse(resultForInCorrectEmail);
    }
```

Fonte: O autor (2025).

Quadro 8 – Comparativo de LOC entre a DSL JCheck e o JUnit 5 no método `isValidEmail`, ilustrado no Código 14.

Tipo de Implementação	LOC Totais	Redução (%)
DSL JCheck	2	71,42%
JUnit 5	7	

Fonte: O autor (2025).

No Código 29, a DSL JCheck é empregada para validar o comportamento do método `isValidEmail`. Já o Código 30 apresenta a mesma verificação implementada manualmente com JUnit, onde é necessário criar dois testes distintos,

invocar explicitamente o método e realizar asserções separadas com `assertTrue` e `assertFalse`. Essa abordagem demanda mais instruções e maior detalhamento da lógica de verificação. Conforme evidenciado no Quadro 8, a implementação com a DSL utiliza apenas duas linhas, enquanto a versão com JUnit requer sete, representando uma redução de cerca de 71,42% no total de linhas de código.

Código 31 – Exemplo de uso da DSL extraído do Código 15.

```
@Check(validation = "check-expect %s 59.97", args = {"19.99",
"3"})
```

Fonte: O autor (2025).

Código 32 – Implementação do exemplo do Código 15 apenas com JUnit.

```
   @Test
    void shouldCalculateTotalCorrectly() {
        BigDecimal unitPrice = new BigDecimal("19.99");
        BigDecimal result = PriceUtil.calculateTotal(unitPrice,
3);
        assertEquals(new BigDecimal("59.97"), result);
    }
```

Fonte: O autor (2025).

Quadro 9 – Comparativo de LOC entre a DSL JCheck e o JUnit 5 no método `calculateTotal`, ilustrado no Código 15.

Tipo de Implementação	LOC Totais	Redução (%)
DSL JCheck	1	83,33%
JUnit 5	6	

Fonte: O autor (2025).

No Código 31, a DSL JCheck é utilizada para validar o resultado do método `calculateTotal`. Já no Código 32 mostra a mesma validação implementada manualmente com JUnit, exigindo a criação de uma instância de `BigDecimal`, a chamada explícita do método e a comparação do resultado com `assertEquals`. Essa abordagem precisa de mais etapas e maior detalhamento na configuração do teste. Como mostra o Quadro 9, a versão com a DSL requer apenas 1 linha de código, enquanto a implementação com JUnit necessita de 6, o que representa uma redução de cerca de 83,33% no total de linhas de código.

Código 33 – Exemplo de uso da DSL extraído do Código 16.

```
@Check(validation = "check-expect %s '***.***.789-01'", args =
{"123.456.789-01"})
```

Fonte: O autor (2025).

Código 34 – Implementação do exemplo do Código 16 apenas com JUnit.

```
   @Test
    void shouldMaskCpfCorrectly() {
        String result = CpfUtil.maskCPF("123.456.789-01");
        assertEquals("***.***.789-01", result);
    }
```

Fonte: O autor (2025).

Quadro 10 – Comparativo de LOC entre a DSL JCheck e o JUnit 5 no método `maskCPF`, ilustrado no Código 16.

Tipo de Implementação	LOC Totais	Redução (%)
DSL JCheck	1	80%
JUnit 5	5	

Fonte: O autor (2025).

No Código 33, a DSL JCheck é empregada para validar o comportamento do método `maskCPF`. Já o Código 34 apresenta o mesmo teste implementado manualmente com JUnit, realizando a chamada explícita do método `maskCPF`, o armazenamento do resultado em uma variável e a verificação por meio do `assertEquals`. Essa implementação requer mais instruções e detalhamento do fluxo de teste. Conforme demonstrado no Quadro 10, a versão com a DSL utiliza apenas 1 linha de código, enquanto a abordagem com JUnit demanda 5, resultando em uma redução de cerca de 80% no total de linhas de código.

Código 35 – Exemplo de uso da DSL extraído do Código 17.

```
@Check(validation = "check-expect %s true", args = {"19.99"})
@Check(validation = "check-expect %s false", args = {"-19.99"})
@Check(validation = "check-expect %s true", args = {"0"})
```

Fonte: O autor (2025).

Código 36 – Implementação do exemplo do Código 17 apenas com JUnit.

```
   @Test
    void shouldValidatePriceCorrectly() {
        boolean resultPositive = PriceUtil.isValidPrice(new
BigDecimal("19.99"));
        assertTrue(resultPositive);
        boolean resultNegative = PriceUtil.isValidPrice(new
BigDecimal("-19.99"));
        assertFalse(resultNegative);
        boolean resultZero =
PriceUtil.isValidPrice(BigDecimal.ZERO);
        assertTrue(resultZero);
    }
```

Fonte: O autor (2025).

Quadro 11 – Comparativo de LOC entre a DSL JCheck e o JUnit 5 no método `isValidPrice`, ilustrado no Código 17.

Tipo de Implementação	LOC Totais	Redução (%)
DSL JCheck	3	66,66%
JUnit 5	9	

Fonte: O autor (2025).

No Código 35, a DSL JCheck é usada para validar o método `isValidPrice`, cobrindo três situações diferentes: quando o preço é positivo, negativo e igual a zero. Essa forma de escrita torna possível representar vários cenários de teste de maneira simples e compacta. Já no Código 36, a mesma verificação é feita manualmente com JUnit, o que exige criar três chamadas explícitas ao método `isValidPrice` e usar diferentes asserções para cada caso. Essa abordagem tradicional acaba gerando mais linhas e repetição de código. Como mostra o Quadro 11, a versão feita com a DSL precisa de apenas três linhas, enquanto a implementação com JUnit utiliza nove, o que representa uma redução de aproximadamente 66,66% no total de código.

Código 37 – Exemplo de uso da DSL extraído do Código 18.

```
@Check(validation = "check-expect %s 'R$ 9,00'", args = {"9.00"})
```

Fonte: O autor (2025).

Código 38 – Implementação do exemplo do Código 18 apenas com JUnit.

```
   @Test
    void shouldReturnFormattedPriceWhenValueIsNine() {
        String result = PriceUtil.formatPrice(new
BigDecimal("9.00"));
        result = result.replace('\u00A0', ' ');
        assertEquals("R$ 9,00", result);
    }
```

Fonte: O autor (2025).

Quadro 12 – Comparativo de LOC entre a DSL JCheck e o JUnit 5 no método `formatPrice`, ilustrado no Código 18.

Tipo de Implementação	LOC Totais	Redução (%)
DSL JCheck	1	83,33%
JUnit 5	6	

Fonte: O autor (2025).

No Código 37, a DSL JCheck é empregada para validar o formato retornado pelo método `formatPrice`, verificando se o valor numérico é corretamente convertido para o padrão monetário esperado. Já no Código 38, a mesma verificação é realizada com JUnit, exigindo a criação explícita de um objeto `BigDecimal`, a chamada ao método, o tratamento do caractere de espaço não separável e a asserção do resultado. Essa abordagem resulta em maior detalhamento e número de instruções. Conforme evidenciado no Quadro 12, a implementação com a DSL requer apenas uma linha, enquanto a versão com JUnit utiliza seis, representando uma redução de cerca de 83,33% no total de linhas de código.

4.4 AMEAÇAS À VALIDADE

De acordo com Wohlin et al. (2012), a análise das ameaças à validade é fundamental para garantir a confiabilidade dos resultados obtidos em estudos empíricos, pois permite compreender as limitações relacionadas ao desenho experimental adotado. A seguir, são apresentadas as principais ameaças à validade associadas à prova de conceito desenvolvida neste trabalho.

4.4.1 Validade Interna

A principal ameaça à validade interna está relacionada à possibilidade de que fatores não controlados tenham afetado os resultados observados. Como a execução dos testes foi realizada em um ambiente controlado, utilizando uma aplicação adaptada para esta pesquisa, há o risco de que particularidades do código, tenham favorecido o desempenho da DSL JCheck. Para mitigar esse risco, foram incluídos métodos com diferentes tipos de dados e comportamentos, abrangendo tanto métodos com retorno quanto métodos void, a fim de diversificar os cenários avaliados.

4.4.2 Validade Externa

A validade externa diz respeito à capacidade de generalizar os resultados obtidos para outros contextos de software. A prova de conceito foi aplicada em um sistema de pequeno porte, com escopo limitado a uma aplicação simulada. Dessa forma, os resultados não podem ser generalizados diretamente para sistemas com estruturas mais complexas. No entanto, como o objetivo principal foi demonstrar a aplicabilidade da DSL em um cenário específico, considera-se que os resultados fornecem uma base consistente para futuras avaliações em contextos mais amplos.

4.4.3 Validade de Construção

A validade de construção está ligada à coerência entre os conceitos teóricos adotados e as medições realizadas. Nesta pesquisa, a métrica LOC foi utilizada para estimar o esforço necessário na implementação dos testes unitários. Embora conforme apontam Ochodek et al. (2023), essa métrica seja amplamente reconhecida e utilizada na área, ela não reflete aspectos qualitativos importantes. Assim, a redução observada em LOC deve ser entendida como um indicador quantitativo de simplificação, e não como uma medida absoluta de produtividade.

4.4.4 Validade de Conclusão

A validade de conclusão refere-se à robustez das inferências realizadas a partir dos dados obtidos. Como a análise foi baseada em uma amostra limitada de métodos, há um risco de que as conclusões sobre a redução de esforço possam variar em outros contextos. Ainda assim, a consistência observada entre diferentes exemplos e tipos de dados sugere que a tendência de redução no número de linhas de código é plausível e sustentada pelos resultados obtidos.

De forma geral, mesmo que existam fatores que possam limitar a ampliação dos resultados, as ações adotadas para lidar com essas limitações, como a escolha de casos de teste variados, o uso de métricas reconhecidas e a execução cuidadosa dos experimentos, ajudam a fortalecer a prova de conceito, tornando suas conclusões mais seguras e confiáveis.

5 TRABALHOS RELACIONADOS

Nesta seção serão apresentados os principais trabalhos relacionados para esta pesquisa.

5.1 *ASPECT-ORIENTED PROGRAMMING RELOADED*

Segundo Rebelo e Leavens (2017), o AspectJML, é uma extensão do AspectJ que integra a Java Modeling Language (JML) ao paradigma de programação orientada a aspectos, permitindo a definição de pré-condições, pós-condições e invariantes como contratos formais que podem ser verificados em pontos de junção durante a execução do programa.

A ferramenta AspectJML, e a DSL proposta neste estudo dividem o objetivo central de reforçar a especificação do comportamento do software por meio da incorporação de anotações no próprio código, utilizando linguagens específicas de domínio para executar instruções.

Contudo, enquanto o AspectJML utiliza a programação orientada a aspectos para verificar contratos *Design by Contract* (DbC), a DSL apresentada nesta pesquisa foca na especificação de testes unitários por meio de anotações que incorporam instruções de verificação, como `check-expect` e `check-effect`. Além disso, a DSL também desempenha um papel importante na promoção da adoção da metodologia TDD, que, segundo Beck (2003) e Astels (2003), assegura a correta escrita e facilita a manutenção dos testes unitários nas fases iniciais do desenvolvimento, promovendo maior qualidade no processo de construção do software. Diante disso, é possível afirmar que a relação entre os trabalhos existe no fato de que os mecanismos citados promovem a expressividade do código ao dar suporte à verificação de comportamentos de forma declarativa.

5.2 *TEASY FRAMEWORK*: UMA SOLUÇÃO PARA TESTES AUTOMATIZADOS EM APLICAÇÕES WEB

Segundo Lima (2021), o *Teasy Framework* funciona como uma ferramenta para a execução de testes funcionais em aplicações web. O autor também destaca

que a ferramenta inclui uma linguagem específica de domínio, chamada *Teasy Language*, que facilita a escrita dos testes.

O *Teasy Framework*, apresentado por Lima (2021), e a DSL desenvolvida neste estudo compartilham o objetivo de simplificar a criação de testes por meio de uma linguagem específica de domínio. Segundo Lima, o *Teasy* foi concebido para automatizar testes funcionais de aplicações web, utilizando a metodologia *Model-Based Testing* (MBT), o que possibilita descrever cenários de teste de forma intuitiva e executá-los automaticamente.

Contudo, enquanto o *Teasy* funciona com foco nos testes funcionais, executando scripts que interagem com a aplicação web, a DSL apresentada nesta pesquisa é focada em especificar testes unitários diretamente no código Java. Para isso, utiliza a anotação personalizada `@Check`, que permite incorporar comandos textuais de verificação, como `check-expect` e `check-effect`.

A relação entre esses trabalhos acontece pelo uso de DSLs para tornar a definição de testes mais expressiva. No entanto, eles diferem em relação ao escopo, pois o *Teasy* é focado na automação de testes funcionais de alto nível, enquanto a proposta deste estudo é voltada para integrar testes unitários ao ciclo de desenvolvimento. Contudo, apesar de funcionarem em contextos distintos, as soluções se alinham no propósito de tornar a escrita de testes uma tarefa mais simples e rápida.

5.3 TESTE BASEADO EM MODELOS EM PROJETOS ÁGEIS, UMA ABORDAGEM BASEADA EM LINGUAGEM DE DOMÍNIO ESPECÍFICO

Segundo Zanin (2019), a abordagem proposta no seu trabalho é baseada em Model-Based Testing (MBT) para equipes de desenvolvimento ágil, para isso é utilizada uma DSL chamada Aquila, que permite a especificação de cenários de testes através de palavras chaves genéricas. A partir desses cenários, são gerados automaticamente modelos comportamentais e scripts de teste, reduzindo o esforço manual de criação e manutenção dos testes, além de facilitar a adaptação às mudanças frequentes típicas de ambientes ágeis.

A ferramenta Aquila e a DSL apresentada neste estudo têm como objetivo facilitar a definição e a execução de verificações de comportamento do software.

Para isso, elas permitem que instruções sejam inseridas diretamente no código, usando linguagens específicas de domínio que tornam esse processo mais automático e intuitivo.

No entanto, enquanto a Aquila foca na geração de modelos e na automação de testes funcionais, a DSL JCheck se dedica à especificação de testes unitários, incorporando instruções como `check-expect` e `check-effect` por meio de anotações Java aplicadas diretamente aos métodos.

Diante disso, os dois trabalhos se conectam porque compartilham o objetivo de diminuir o esforço manual e tornar o código mais expressivo, aproximando a definição do comportamento do próprio processo de desenvolvimento. Apesar disso, cada um atua em um nível diferente, com a Aquila focada em testes funcionais por meio de modelos e a JCheck concentrada em testes unitários diretamente no código Java.

5.4 *A TESTING TOOL FOR WEB APPLICATIONS USING A DOMAIN-SPECIFIC MODELLING LANGUAGE AND THE NUSMV MODEL CHECKER*

Segundo Törsel (2013), o trabalho propõe uma abordagem para automatizar o teste de aplicações web por meio do MBT. Para isso, foi desenvolvida uma DSL que descreve o comportamento do sistema, considerando as páginas e as transições que o usuário pode acionar. Esse modelo, apresentado de forma textual e abstrata, é transformado em uma entrada para o verificador formal NuSMV, que cria sequências de teste com base em critérios de cobertura definidos pelo usuário. Em seguida, essas sequências são transformadas em testes executáveis para ferramentas de automação, como o Selenium, permitindo que os testes sejam rapidamente ajustados sempre que houver alterações nos requisitos.

A DSL apresentada neste estudo e a solução proposta por Törsel (2013) compartilham a meta de reduzir o esforço manual na criação de testes utilizando DSL para esse fim. No entanto, enquanto a abordagem baseada no NuSMV é voltada para testes de integração e fluxo em aplicações web, a DSL JCheck concentra-se em testes unitários em Java, incorporando instruções de verificação diretamente nos métodos por meio de anotações.

Dessa forma, ambos os trabalhos evidenciam a relevância das linguagens específicas de domínio como mecanismos para aproximar a especificação de comportamento do processo de desenvolvimento, ainda que atuem em diferentes níveis de teste e tipos de aplicação.

5.5 *ACCELERATING TEST AUTOMATION THROUGH A DOMAIN SPECIFIC LANGUAGE*

Segundo Dwarakanath et al. (2017), testar o software é uma das etapas mais caras do processo de desenvolvimento, podendo representar entre 30% e 90% do esforço total de um projeto. Apesar da existência de diversas ferramentas de automação, o uso delas ainda pode ser uma tarefa complexa e trabalhosa. Isso ocorre, principalmente, porque a criação dos scripts de automação exige conhecimentos em programação e, muitas vezes, demanda um esforço considerável para manter esses scripts, que costumam ser complexos e difíceis de compreender para testadores que não possuem um conhecimento mais aprofundado em programação.

Buscando contornar essas dificuldades, Dwarakanath et al. (2017) desenvolveram a ferramenta Automation Test Acceleration Platform (ATAP). A proposta se baseia em uma DSL que permite criar scripts de automação de forma simples e próxima da linguagem natural. Com isso, testadores manuais, mesmo sem experiência em programação, conseguem descrever cenários de teste de maneira clara e intuitiva. As instruções escritas nessa linguagem são automaticamente convertidas em código Java executável com o uso do Selenium WebDriver, por meio da linguagem Xtend. O resultado é um processo de automação menos verboso, mais acessível e que reduz a dependência de conhecimentos técnicos avançados.

Assim como o ATAP, a DSL JCheck desenvolvida nesta pesquisa também busca facilitar o trabalho com testes de software, especialmente no que diz respeito à sua criação e manutenção. No entanto, cada uma atua em um contexto diferente. Enquanto o ATAP é voltado para testes de integração e interface web, a JCheck foi pensada para o universo dos testes unitários em Java. Nessa abordagem, as verificações são escritas diretamente no código por meio da anotação `@Check`,

tornando o processo mais natural para o desenvolvedor e reduzindo a distância entre o código da aplicação e seus testes.

Dessa forma, tanto a solução proposta por Dwarakanath et al. (2017) quanto a DSL JCheck evidenciam que o uso de linguagens específicas de domínio pode tornar a criação de testes mais próxima da forma como os desenvolvedores pensam. Ao aproximar a especificação de testes do raciocínio humano, essas abordagens ajudam a reduzir o esforço necessário para escrever e manter os testes, ao mesmo tempo em que contribuem para aumentar a qualidade e a confiabilidade do processo de verificação de software.

5.6 UMA METODOLOGIA PARA A GERAÇÃO DE TESTES UNITÁRIOS BASEADA EM EXTRAÇÃO DE MODELOS

De acordo com Brito (2021), gerar testes para sistemas legados é uma tarefa que exige bastante cuidado, especialmente quando não há documentação atualizada ou quando o sistema não possui testes anteriores. Para lidar com esse desafio, o autor propôs uma metodologia semiautomática que combina técnicas de teste de software com métodos formais de verificação de modelos.

A metodologia consiste na criação automática de um modelo que descreve o comportamento do sistema, construído a partir de informações coletadas durante a execução do código. Esse modelo, denominado *Labelled Transition System* (LTS), é então analisado por uma ferramenta que busca identificar possíveis inconsistências. Com base nas inconsistências identificadas, novos testes unitários são gerados, estabelecendo um ciclo contínuo em que os testes são criados, executados e aprimorados constantemente. Esse processo amplia a cobertura de verificação e, consequentemente, reforça a confiabilidade do sistema.

A proposta apresentada por Brito (2021) e a DSL JCheck discutida nesta dissertação compartilham o objetivo de tornar o processo de criação e manutenção de testes unitários mais simples e eficiente. Apesar de buscarem o mesmo resultado, cada solução utiliza abordagens distintas. A metodologia de Brito (2021) consiste na extração de modelos e na aplicação de técnicas formais de verificação. Por sua vez, a DSL JCheck segue um percurso mais direto e intuitivo, permitindo que o desenvolvedor escreva as verificações diretamente no código. Dessa forma, embora suas estratégias sejam diferentes, ambas as soluções convergem para o

propósito comum de aumentar a produtividade do desenvolvedor e reforçar a confiabilidade do software por meio de testes bem construídos e consistentes.

6 CONCLUSÃO

Neste trabalho, apresentamos a implementação de uma nova DSL, baseada na sintaxe da linguagem Racket, denominada JCheck. Essa DSL tem como objetivo promover o uso da metodologia TDD durante o desenvolvimento de código Java. Para isso, foi criada uma linguagem específica de domínio, utilizada por meio da anotação `@Check`, restrita apenas a métodos. A sintaxe da DSL é baseada na expressão `check-expect` da linguagem Racket. Contudo, nesse contexto, a DSL é empregada para validar métodos por meio de testes unitários em Java, executados com JUnit 5. Para esse efeito, a DSL foi desenvolvida utilizando a gramática do gerador de analisador sintático ANTLR4, a partir da qual foi gerado o *parser* da linguagem.

Realizamos uma prova de conceito para validar a implementação da DSL JCheck utilizando uma API Spring Boot. Segundo Webb et al. (2013), o Spring Boot é uma ferramenta que simplifica a criação de aplicações independentes, prontas para produção, e baseadas no framework Spring.

A prova de conceito envolveu dez métodos escolhidos por suas características distintas, com o objetivo de demonstrar a capacidade da solução JCheck de lidar com comportamentos variados e de evidenciar que o uso da DSL pode demandar menor esforço na implementação dos testes unitários. Para isso, foram considerados métodos que retornam diferentes tipos de dados, assim como métodos que, quando executados, resultam em efeitos nos atributos da classe. Os resultados mostraram que a DSL foi executada com sucesso em todos os cenários, evidenciando sua aplicabilidade em contextos diversos. Além disso, foi possível executar múltiplos casos de teste para o mesmo método, comprovando a escalabilidade na cobertura dos cenários. Para mais, a integração com a anotação `@Check` facilitou a incorporação dos testes diretamente aos métodos, permitindo a definição clara das instruções da DSL e promovendo a adoção de práticas como a metodologia TDD.

Com base neste estudo, concluímos que a solução proposta mostrou-se eficaz para a criação de testes unitários em Java, oferecendo expressividade, clareza e integração direta com o código. A prova de conceito confirmou que a DSL é capaz de lidar com diferentes cenários de testes e que sua utilização pode

demandar menor esforço na implementação dos testes unitários, reforçando seu potencial como ferramenta de apoio à prática da metodologia TDD. Os resultados indicam que a JCheck pode ser expandida e aprimorada em trabalhos futuros, possibilitando a inclusão de novas instruções, maior automação na geração de testes e integração com outras ferramentas de análise e validação de software, contribuindo assim para o desenvolvimento de aplicações mais consistentes.

7 TRABALHOS FUTUROS

Embora a implementação da DSL JCheck tenha atingido os objetivos propostos, diversas direções podem ser exploradas em trabalhos futuros. Uma possibilidade é a ampliação do conjunto de instruções da linguagem, incorporando novas expressões além do `check-expect` e `check-effect`, como instruções específicas para validação de exceções esperadas, comparações aproximadas de valores numéricos ou suporte para estruturas de dados mais complexas, como coleções, objetos aninhados e expressões aritméticas. Essa expansão permitiria lidar com cenários de teste mais ricos e próximos da realidade do mercado.

Outro mecanismo que também pode ser acrescentado à solução é uma ferramenta de apoio à depuração e análise de falhas, para auxiliar os desenvolvedores na identificação de inconsistências durante a execução da DSL. Essa ferramenta pode oferecer relatórios mais detalhados, mensagens de erro mais explicativas e integração com *Integrated Development Environments* (IDE), tornando a experiência de uso mais intuitiva.

Para finalizar, outro trabalho significativo seria a realização de um estudo de caso, aplicando a solução proposta neste trabalho em projetos de maior porte, envolvendo aplicações com maior complexidade arquitetural. Essa avaliação teria o propósito de verificar o impacto da DSL JCheck em termos de produtividade e facilidade de manutenção ao longo do ciclo de desenvolvimento do software.

REFERÊNCIAS

ABUSHAMA, Hisham M.; ALASSAM, Hanaa Altigani; ELHAJ, Fatin A. The effect of test-driven development and behavior-driven development on project success factors: A systematic literature review based study. In: **2020 International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE).** IEEE, 2021.

AGHA, Dua et al. Test Driven Development and Its Impact on Program Design and Software Quality: A Systematic Literature Review. **VAWKUM Transactions on Computer Sciences**, v. 11, n. 1, p. 268-280, 2023.

ANDRADE, Stevão et al. **Uma abordagem de teste de software para aplicações de realidade virtual utilizando testes metamórficos.** Tese de Doutorado. Universidade de São Paulo. 2023.

ANTLR. **What is ANTLR?** In: ANother Tool for Language Recognition**.** 2025. Disponível em: <https://www.antlr.org/>. Acesso em: 10 jul. 2025.

ASTELS, Dave. **Test driven development: A practical guide.** Prentice Hall Professional Technical Reference, 2003.

BECK, Kent. **Test-driven Development: By Example.** Addison-Wesley Professional, 2003.

BERNARDO, Paulo Cheque; KON, Fabio. A importância dos testes automatizados. **Engenharia de Software Magazine**, v. 1, n. 3, p. 54-57, 2008.

BRITO, Iara Ramos. Uma metodologia para a geração de testes unitários baseada em extração de modelos. 2021.

CALAIS, Pedro; FRANZINI, Lissa. Test-driven development benefits beyond design quality: Flow state and developer experience. In: **2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)**. IEEE, 2023. p. 106-111.

CRESPO, Adalberto Nobiato et al. Uma metodologia para teste de Software no Contexto da Melhoria de Processo. In: **Simpósio Brasileiro de Qualidade de Software (SBQS)**. SBC, 2004. p. 204-218.

CULPEPPER, Ryan et al. From macros to dsls: The evolution of racket. In: **3rd summit on advances in programming languages (snapl 2019).** Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019. p. 5: 1-5: 19.

DWARAKANATH, Anurag et al. Accelerating test automation through a domain specific language. In: **2017 IEEE International Conference on Software Testing, Verification and Validation (ICST).** IEEE, 2017. p. 460-467.

FELLEISEN, Matthias et al. **How to design programs: an introduction to programming and computing.** Mit Press, 2018.

FELLEISEN, Matthias et al. The racket manifesto. In: **1st Summit on Advances in Programming Languages (SNAPL 2015).** Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015. p. 113-128.

FELLEISEN, Matthias; FLATT, Matthew. **Racket is 25.** Blog oficial do Racket, 4 maio de 2020. Disponível em: <https://blog.racket-lang.org/2020/05/racket-is-25.html>. Acesso em: 8 jul. 2025.

GARCIA, Boni. **Mastering Software Testing with JUnit 5: Comprehensive guide to develop high quality Java applications.** Packt Publishing Ltd, 2017.

GARCÍA, Boni et al. Selenium-jupiter: A junit 5 extension for selenium webdriver. **Journal of Systems and Software**, v. 189, p. 111298, 2022.

GOMES, Renata Faria. Uma metodologia remota gamificada para o ensino de testes unitários. 2020.

GORLA, Daniele et al. CubeTesterAI: Automated JUnit Test Generation Using the LLaMA Model. In: **2025 IEEE Conference on Software Testing, Verification and Validation (ICST)**. IEEE, 2025. p. 565-576.

GUNDLACH, Sven; JUNG, Reiner; HASSELBRING, Wilhelm. **Facilitating test-driven development via domain-specific languages in computational science software engineering.** Research Square preprint deposition service, 2023.

JAHIĆ, Benjamin; GUELFI, Nicolas; RIES, Benoît. SEMKIS-DSL: A domain-specific language to support requirements engineering of datasets and neural network recognition. **Information**, v. 14, n. 4, p. 213, 2023.

JUNIT. **User Guide – JUnit 5.** Disponível em: <https://junit.org/junit5/docs/current/user-guide/>. Acesso em: 19 jul. 2025.

KOSAR, Tomaž et al. A preliminary study on various implementation approaches of domain-specific language. **Information and software technology**, v. 50, n. 5, p. 390-405, 2008.

LADEINDE, Opeoluwa Joseph. An adaptive feedback framework for a language-independent intelligent programming tutor (IPT) using ANTLR. 2023.

LIMA, Yury Alencar. Teasy framework: uma solução para testes automatizados em aplicações web. 2021.

MAXIMILIEN, E. Michael; WILLIAMS, Laurie. Assessing test-driven development at IBM. In: **25th International Conference on Software Engineering, 2003. Proceedings.** IEEE, 2003. p. 564-569.

MYLSAMY, Sekar. Test-Driven Development (TDD) and Code Quality. 2025.

NETO, Antônio José Rodrigues et al. **Um Modelo de Contexto das Práticas e Redes de Conhecimento nas Provas de Conceito.** 2023. Tese de Doutorado. Universidade de Coimbra.

OCHODEK, Miroslaw et al. Mining Task-Specific Lines of Code Counters. **IEEE Access**, v. 11, p. 100218-100233, 2023.

PARR, Terence. **The Definitive ANTLR 4 Reference.** Dallas: Pragmatic Bookshelf, 2013. Disponível em: <https://media.pragprog.com/titles/tpantlr2/picture.pdf>. Acesso em: 10 jul. 2025.

PARR, Terence J.. ; QUONG, Russell W. ANTLR: A predicated‐LL (k) parser generator. **Software: Practice and Experience**, v. 25, n. 7, 1995.

PRESSMAN, Roger S. **Software engineering: a practitioner's approach.** Palgrave Macmillan, 2005.

RACKET. **The Racket Guide.** [S.l.]: PLT, 2025. Disponível em: <https://docs.racket-lang.org/guide/index.html>. Acesso em: 1 jul. 2025.

REBÊLO, Henrique; LEAVENS, Gary T. Aspect-Oriented Programming Reloaded. In: **Proceedings of the 21st Brazilian Symposium on Programming Languages**. Association for Computing Machinery, New York, NY, USA, Article 10, 1–8. https://doi.org/10.1145/3125374.3125383, 2017.

SILVA, Ruan Carlos Alves da. **poc-dsl-check-springboot.** Repositório GitHub, 2025. Disponível em: <https://github.com/ruaanc/poc-dsl-check-springboot>. Acesso em: 15 ago. 2025.

TARAZONA BERNAL, Yeisson Steven. ANTLR 4 grammar of the Swift 5 programing language. 2021.

TÖRSEL, Arne-Michael. A testing tool for web applications using a domain-specific modelling language and the nusmv model checker. In: **2013 IEEE Sixth International Conference on Software Testing, Verification and Validation.** IEEE, 2013. p. 383-390.

UMAR, Mubarak Albarka; ZHANFANG, Chen. A study of automated software testing: Automation tools and frameworks. **International Journal of Computer Science Engineering (IJCSE)**, v. 6, n. 217-225, p. 47-48, 2019.

WEBB, Phillip et al. Spring boot reference guide. **Part IV. Spring Boot features**, v. 24, 2013.

WOHLIN, Claes et al. **Experimentation in software engineering.** Berlin: Springer, 2012.

ZANIN, Aline. Teste baseado em modelos em projetos ágeis, uma abordagem baseada em linguagem de domínio específico. 2019.