UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

YURI RAFAEL GRAJEFE FEITOSA

UMA INVESTIGAÇÃO SOBRE DESENVOLVIMENTO BASEADO EM TESTES NO ENSINO INTRODUTÓRIO DE PROGRAMAÇÃO

DISSERTAÇÃO DE MESTRADO

CORNÉLIO PROCÓPIO
2023

YURI RAFAEL GRAJEFE FEITOSA

UMA INVESTIGAÇÃO SOBRE DESENVOLVIMENTO BASEADO EM TESTES NO ENSINO INTRODUTÓRIO DE PROGRAMAÇÃO

An investigation on the use of test-driven development in introductory programming courses

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Mestre em Informática.

Orientador: Prof. Dr. José Augusto Fabri

Co-orientador: Prof. Dr. Marco Aurélio Graciotto Silva

CORNÉLIO PROCÓPIO
2023

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Cornélio Procópio

YURI RAFAEL GRAJEFE FEITOSA

UMA INVESTIGAÇÃO SOBRE DESENVOLVIMENTO BASEADO EM TESTES COM APLICAÇÃO DE CRITÉRIOS DE TESTE NO ENSINO INTRODUTÓRIO DE PROGRAMAÇÃO.

Trabalho de pesquisa de mestrado apresentado como requisito para obtenção do título de Mestre Em Informática da Universidade Tecnológica Federal do Paraná (UTFPR). Área de concentração: Computação Aplicada.

Data de aprovação: 01 de Fevereiro de 2023

Dr. Jose Augusto Fabri, Doutorado - Universidade Tecnológica Federal do Paraná

Dr. Alexandre L Erario, Doutorado - Universidade Tecnológica Federal do Paraná

Dr. Andre Takeshi Endo, Doutorado - Universidade Federal de São Carlos (Ufscar)

Documento gerado pelo Sistema Acadêmico da UTFPR a partir dos dados da Ata de Defesa em 01/02/2023.

AGRADECIMENTOS

Agradeço a Deus, pela existência e pela força concedida, mesmo com todos os obstáculos, desistir nunca foi uma alternativa.

A UTFPR pela dupla oportunidade, primeiramente como Professor Substituto no Campus de Campo Mourão, no DACOM (Departamento Acadêmico de Computação) e agora como aluno de Mestrado no PPGI (Programa de Pós Graduação em Informática) no Campus de Cornélio Procópio. Ambas as experiências foram muito importantes para o meu crescimento profissional e acadêmico.

Agradeço ao Orientador Prof. Dr. José Augusto Fabri, por sempre me atender rapidamente. Agradeço também ao Co-orientador Prof. Dr. Marco Aurélio Graciotto Silva, que esteve comigo desde o início dessa jornada, foi meu colega de sala quando eu era Professor Substituto na UTFPR, obrigado pela paciência, pela dedicação, pelo conhecimento compartilhado e pelas boas conversas desde de quando trabalhávamos juntos em Campo Mourão.

Aos demais professores do PPGI, membros da banca de qualificação e da defesa, meu muito obrigado, todas as considerações apontadas durante o período acadêmico foram muito relevantes para meu crescimento.

RESUMO

FEITOSA, Yuri. UMA INVESTIGAÇÃO SOBRE DESENVOLVIMENTO BASEADO EM TESTES NO ENSINO INTRODUTÓRIO DE PROGRAMAÇÃO. 105 f. Dissertação – Programa de Pós-Graduação em Informática, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2023.

Contexto: Esta dissertação de mestrado é uma pesquisa sobre o uso de técnicas de teste de software aplicadas na educação em Computação, mais especificamente, no ensino introdutório de programação. Para isso é proposto a utilização do TDD (Test-driven development) no auxílio da aprendizagem baseado em critérios de teste. Esse método busca estruturar melhor o raciocínio lógico do aluno nos primeiros passos com a programação, evitando a tentativa e erro.
Objetivo: O trabalho consiste em propor um novo método de abordagem no ensino das disciplinas introdutórias de programação utilizando casos de teste. Esse novo formato consiste na utilização do *test-first* no desenvolvimento do código. O aluno deve primeiramente elaborar os casos de testes para depois desenvolver o algoritmo pensando melhor na resolução do problema antes de desenvolver o código.
Método: Para esta pesquisa, foi realizado a aplicação de exercícios de programação para os alunos em sala de aula. A amostra foi dividida em dois grupos, o primeiro grupo utilizou o *test-first*, o segundo não. Os exercícios foram monitorados e enviados para análise com o auxilio de *framework* de teste de unidade em C.
Resultados: Há indícios de melhorias na qualidade dos programas desenvolvidos pelos estudantes com o auxílio de critérios de teste.
Conclusões: Considerando as evidências de trabalhos relacionados quanto ao ensino de programação com teste de software e o resultado final da pesquisa, o uso de critérios de teste contribui para um melhor desenvolvimento de problemas computacionais.

Palavras-chave: Educação em Computação, TDD, Teste de Software

ABSTRACT

FEITOSA, Yuri. An investigation on the use of test-driven development in introductory programming courses. 105 f. Dissertação – Programa de Pós-Graduação em Informática, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2023.

Context: This master's thesis is a research on the use of software testing techniques applied in Computer education, more specifically, in introductory programming teaching, for this is purposed the use of TDD (Test-driven development) to help learning based on testing criteria. This method seeks to better structure the student's logical reasoning in the first steps with programming, avoiding trial and error.

Objective: The work consists of proposing a new method of approach to teaching introductory programming disciplines using test criteria. This new format consists of using test case in code development, the student must first develop the test cases and then develop the algorithm, in this sense, thinking better about solving the problem before developing the code.

Method: For this research, programming exercises were applied to students in the classroom. The sample was divided into two groups, the first group used *test-first*, the second did not. The exercises were monitored and sent for analysis with the help of *framework* for unit testing in C.

Results: There are indications of improvements in the quality of programs developed by students with the aid of test criteria.

Conclusions: Considering the evidence from related work on teaching programming with software testing and the final result of the research, the use of test criteria contributes to a better development of computational problems.

Keywords: Computer Education, TDD, Software Testing

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

CT	Computational Thinking
CS1	Computer Science 1
CS2	Computer Science 2
GDU	Grafo de Definição-Uso
GFC	Grafo de Fluxo de Controle
IES	Instituição de Ensino Superior
MDC	Máximo Divisor Comum
PP	Pair Programming
PICO	Population Intervention Control Outcome
POP	Programação Orientada a Problemas
TDD	Test Driven Development
TDL	Test Driven Learning
UML	Unified Modeling Language
XP	Extreme Program

SUMÁRIO

1 INTRODUÇÃO

O ensino de programação é um processo complexo (LUXTON-REILLY et al., 2018). De modo geral, existe uma linha de desenvolvimento que deve ser seguida, começando pela lógica, depois a lógica de programação e por fim os algoritmos (FORBELLONE; EBERSPäCHER, 2005).

A lógica é "correção do pensamento", a "arte de bem pensar", como um silogismo. Entretanto, a lógica de programação é a aplicação desta lógica, de processos de raciocínio e simbolização formais na programação de computadores, objetivando a racionalidade e o desenvolvimento de técnicas que cooperam para a produção de soluções logicamente válidas e coerentes, que resolvam com qualidade os problemas que se deseja programar (FORBELLONE; EBERSPäCHER, 2005).

Diversas pesquisas apontam problemas na aprendizagem que vão desde a dificuldade dos alunos em compreender os conceitos de programação até a falta de motivação em realizar as atividades de programação (SOUZA et al., 2016). Os alunos não aprendem a enfrentar, resolver ou testar problemas complexos nos cursos introdutórios de programação, em grande parte porque estas habilidades não são ensinadas (BELL et al., 1994). Induzir os alunos a pensar de forma mais eficiente, inteligente e sem erros é um desafio para o professor (LUXTON-REILLY et al., 2018).

Um problema consiste de uma tarefa orientada a objetivo que exige a realização de um conjunto de atividades para satisfazer o objetivo, sendo que não se conhece, ao iniciar, a resolução do problema e quais são as atividades requeridas (DUNCKER, 1945, p. 1). De modo geral, a capacidade de resolver problemas é uma competência

importante na formação da maioria dos indivíduos.

No contexto computacional, a resolução de problemas envolve atividades de análise e projeto (design): em um primeiro instante é necessário compreender o problema, analisá-lo e, posteriormente, criar soluções para ele (projeto). Finalmente, essa solução pode ser concretizada pela implementação em uma linguagem de programação. Diversas ferramentas prestam auxílio ao projeto com representação gráficas (modelos em Unified Modeling Language (UML) e fluxogramas) ou textualmente (pseudocódigo), facilitando a definição e avaliação de soluções (LUXTON-REILLY et al., 2018). No entanto, poucos trabalhos exploram os mecanismos de resolução de problema em si, em especial o conhecimento tácito, típico dos programadores mais experientes.

Existe uma relação entre as competências de resolução de problemas de modo geral e de pensamento computacional, tanto em ensino básico (fundamental e médio, ou K-12) (GROVER; PEA, 2013) quanto no ensino superior (SALEHI et al., 2020). No caso do domínio de ciências exatas e engenharias, observa-se que estudantes de Computação parecem ter desempenho superior àqueles de outros cursos quanto à efetividade de resolução de problemas (SALEHI et al., 2020). Também é possível observar que existe uma diferença entre estudantes dos níveis iniciais e finais do curso de Computação (SALEHI et al., 2020). Isso sugere que o desenvolvimento de habilidades de pensamento computacional, compreendendo a formulação de problemas de modo que suas soluções podem ser representadas como passos computacionais e algoritmos, parece ser transferível para resolução de problemas como um todo (AHO, 2011; WING, 2006).

O pensamento computacional envolve a resolução de problemas, projetar sistemas e compreender o comportamento dos humanos com base nos conceitos fundamentais da Ciência da Computação (WING, 2006). De fato, o pensamento computacional apresenta muitos elementos em comum com resolução de problemas, tendo como diferenças a estratégia iterativa e, geralmente, incremental do desenvolvimento da solução (MCCRACKEN et al., 2001) e a natureza dos problemas

tratados, mais abertos e, de certa forma, mais vagamente especificados do que em outras áreas, proporcionando um maior espaço de soluções a serem exploradas (SALEHI, 2018). Dessa forma, os problemas trabalhados em Computação favorecem o desenvolvimento de habilidades de resolução de problemas.

Embora a resolução de problemas seja um aspecto crucial da programação, poucas oportunidades de aprendizagem na Ciência da Computação têm foco em ensino nas habilidades de resolução de problemas, como planejamento (SHI et al., 2019). O foco do processo de resolução de problemas está nas quatro etapas: pensamento algorítmico, implementação, análise e comunicação (SHETH et al., 2016).

Durante a formação acadêmica, a Engenharia de Software é responsável pela especificação, desenvolvimento e manutenção de um software, precisa lidar com a resolução de problemas, para isso, utiliza a lógica de programação e práticas de gerência de projetos incluindo, politicas de organização, produtividade e qualidade de software.

O desenvolvimento de software é um processo realizado em etapas, ao invés da escrita de uma única solução monolítica. Para isso, o programador deve criar as classes, criar os testes, fazer representações alternativas, instanciar os campos para só depois implementar os métodos. Isso é um processo reduzido da Engenharia de Software por completo que pode ajudar os programadores iniciantes (CASPERSEN; KöLLING, 2006). A maioria das correções que os alunos utilizam nos códigos estão relacionadas com a mudança de expressões ou com a reestruturação do código, refletindo dificuldades na lógica e na resolução de problemas (SOUZA et al., 2017),

A disciplina de Engenharia de Software, comumente ofertada no meio da graduação, é exclusiva dos cursos específicos de Computação, não sendo incluída nos demais cursos de Engenharias que têm disciplinas introdutórias de programação. Em Engenharia de Software, são estudadas várias técnicas para desenvolvimento de software. Dentre essas, técnicas de verificação e validação que contribuem para alcançar patamares adequados de qualidade. No ensino de programação, atividades de teste de software são utilizadas para verificar os programas desenvolvidos pelos estudantes. Depois de aprender os princípios e técnicas básicas de teste de software,

os desenvolvedores têm duas vezes mais probabilidade de produzir implementações corretas (LEMOS et al., 2018).

O teste de software pode ser considerado para tratar a resolução de problemas (HSIA; PETRY, 1980; PEARS et al., 2007). Um dos principais pontos do pensamento computacional é a avaliação de solução e o teste de software trata exatamente desta perspectiva. Assim, observam-se indícios de que o teste de software criterioso pode contribuir no processo de aprendizagem e desenvolvimento de código (SCATALON et al., 2019).

Um caso de teste consiste em um conjunto de entradas, condições de execução e a saída esperada para um programa (ISO et al., 2017). A elaboração de um caso de teste na fase introdutória do ensino de programação pode induzir o aluno a refletir melhor sobre a resolução de um determinado problema computacional e, consequentemente, chegar a uma resposta mais eficiente, com menos tentativas e erros no percurso (EDWARDS, 2004). Somado a isso, a aplicação de critérios de testes para o desenvolvimento orientado a testes poderia melhorar a qualidade do Software desenvolvido pelos alunos e fomentar a reflexão na ação.

Existem fortes indícios na literatura sobre os benefícios da utilização do teste de software no ensino de programação (FEITOSA et al., 2021a). No contexto de aprendizagem em computação, resolução de problemas e no desenvolvimento desse trabalho, o Test Driven Development (TDD), um ciclo entre criação de casos de testes, programação de código e refatoração, pode promover mudanças no decorrer das disciplinas introdutórias de programação, transformando comportamentos, despertando habilidades e alterando o ângulo de visão do aluno referente ao problema, consequentemente, forçando-o a resolver o exercício de maneira mais eficiente, tendo uma experiência melhor em programação.

De modo geral, os efeitos do teste de software em disciplinas introdutórias à programação são positivos (FEITOSA et al., 2021b). A reflexão possibilitada pela criação dos casos de teste e execução mental dos casos de teste foram benéficas para a compreensão e resolução de problemas (DENNY et al., 2019; HILTON et al., 2019).

Além disso, observa-se que os estudantes ficaram mais confiantes na resolução de problemas (HILTON et al., 2019; MOURãO, 2017). O uso de critérios de teste, uma condição específica que verifica se um sistema está funcionando, contribuem para a criação de casos de teste, guiando o estudante a criar conjuntos de mais qualidade (CAMARA; SILVA, 2016) e a explorar melhor o problema e as particularidades a serem tratadas nas implementações (BELL et al., 1994).

Embora o esforço ao utilizar teste possa ser ligeiramente maior (e sem diferença estatisticamente significativa) do que em uma abordagem tradicional, os ganhos em qualidade são evidentes e significativos (HSIA; PETRY, 1980). Estratégias que permitem a adoção incremental de teste de software, inclusive com adoção de diversos critérios de teste (BELL et al., 1994) contribuem para o melhor desempenho do estudante, permitindo não apenas notas melhores, mas a realização de atividades de programação mais complexas do que com a abordagem tradicional (BELL et al., 1994). Destaca-se também que a utilização de ferramentas integradas para auxiliar e conduzir a execução da abordagem é considerada importante pelos estudantes, contribuindo para que estudantes com mais dificuldades percebam e tenham benefícios maiores no aprendizado (BELL et al., 1994). Entretanto, mesmo com atividades de teste de software, observam-se dificuldades nas atividades de programação referentes à resolução de problemas, com indícios percebidos pela evolução da estruturação do código durante as atividades (SOUZA et al., 2017).

Com as mudanças geradas pela tecnologia, como o ensino remoto por exemplo, existe a necessidade de renovar esse processo de aprendizagem. Este trabalho busca especificamente preencher essa lacuna melhorando o processo de aprendizagem de programação com o auxílio de critérios de teste, para isto, é proposto uma nova forma no ensino introdutório de computação utilizando o TDD.

O objetivo desta pesquisa consiste em estabelecer uma abordagem de desenvolvimento orientado ao TDD que empregue casos de teste de forma escalonada, começando com exercícios de dificuldade menor, até um mais complexo, associada a objetivos de atividades de aprendizagem de programação. Desta forma, busca-

se aperfeiçoar o desenvolvimento da lógica computacional dos alunos nos primeiros degraus de aprendizagem de programação nos cursos de computação e áreas afins.

Parte-se de um pressuposto em que o aluno não se preocupa apenas com o código a ser desenvolvido em si, mas sim nos resultados que esse algoritmo vai gerar, principalmente se o produto final atende a todos os critérios que devem ser especificados na análise do problema com a maior cobertura de cenários distintos possíveis. O TDD foi escolhido como ferramenta chave para essa pesquisa pois colabora com o desenvolvimento de código testável, visto que, em sua primeira etapa, é necessário que seja proposto um caso de teste que exercite a funcionalidade recém implementada.

Para estruturar melhor este trabalho, foram elaboradas duas questões de pesquisa:

- **QP 1** – O aluno fazer o caso de teste antes de desenvolver o código pode ajudar o estudante a elaborar um algoritmo com a lógica mais apurada e consequentemente, com menos erros?

  Ao definir uma série de casos de teste simulando a resolução para um determinado problema, o aluno está consequentemente pensando em uma forma de chegar no resultado do problema em questão. Entre o contexto de montar os casos de teste e fazer a comparação entre os resultados esperados e os resultados obtidos, o aluno está trabalhando a lógica de programação, pensando mais detalhadamente no problema e analisando os mais diversos parâmetros de entrada possíveis.

  Seguindo a linha de raciocínio do TDD, primeiramente deve ser desenvolvido um caso de teste para o problema proposto, que deverá falhar, para posteriormente ser desenvolvido o código correto para atender a esse caso de teste (BECK, 2002). O aluno deve refletir e entender qual foi o erro no seu primeiro caso de teste, fazer a análise para só depois desenvolver o algoritmo que irá contemplar o caso de teste, nesse processo é preciso raciocinar mais sobre o problema em questão.

- **QP 2** – A utilização de critérios de teste contribui para o melhor aprendizado?

> Levando em consideração que um critério de teste define um conjunto de condições que devem ser utilizadas na atividade de teste, a utilização do mesmo nos primeiros passos da programação força o aluno a pensar em um conjunto de casos de teste mais completo para o exercício em questão.
>
> Ao pensar em ampliar a cobertura dos casos de testes com relação a requisitos de teste associados com objetivos educacionais, o aluno está necessariamente pensando nas possíveis entradas, possíveis saídas, e quais falhas que o seu código poderá apresentar. É justamente nessa etapa que o critério de teste (todos os nós), combinado com TDD, entram como uma ferramenta de apoio ao aluno no sentido de estruturar o seu raciocino lógico com foco em desenvolver um algoritmo mais eficiente e correto.

A aplicação prática deste trabalho foi realizada em duas etapas em uma única turma de alunos, dividida em dois grupos. No primeiro momento foi aplicada a abordagem proposta em um grupo de dez alunos, foram os alunos que utilizaram o TDD. Logo após, os mesmos problemas foram aplicados no segundo grupo, nove alunos, sem usar o TDD.

Ambos estavam iniciando o quarto bimestre do curso de Análise e Desenvolvimento de Sistemas. Nenhum conteúdo abordado neste trabalho, como teste de *software* ou TDD foi detalhado. Foi explicado para os alunos, apenas o que foi considerado suficiente para eles conseguirem resolver os exercícios.

Alguns conceitos importantes relacionados a programação como teste de software, critérios de testes e TDD, são lecionados na disciplina de Engenharia de Software. Esses conteúdos pode ser úteis no processo de aprendizagem de programação inicial do aluno, pois podem mostrar outras perspectivas do desenvolvimento do algoritmo. Entretanto, na fase inicial da graduação, o aluno não deve se preocupar em entender esses conceitos, pois podem ser uma carga a mais de definições que pode sobrecarregar a aprendizagem de programação, que é o foco principal.

Para um programa ser considerado bom, ele precisa interpretar todas as

possíveis entradas e apresentar os resultados esperados, ou seja, independente da informação inicial inserida pelo usuário, em algum lugar, o algoritmo precisar trata-lá. Nesta pesquisa, o aluno deve pensar em como fazer com que seu programa tenha a maior cobertura possível, como tratar as mais diversas entradas.

Somente a explicação de teste de software já iria comprometer mais do que toda a carga horária estipulada para a aplicação da pesquisa e a ideia não é avaliar as habilidades de Engenharia de Software dos alunos, mas sim, a lógica aplicada a resolução de problemas. O aluno não precisa saber de todo o processo de teste de software para ser apto a participar desta pesquisa.

No primeiro contato, foi explicado a lógica de como aplicar técnicas de teste de software na resolução de problemas computacionais utilizando critérios de testes. Essa explicação foi limitada a apenas o necessário para que o aluno seja capaz de desenvolver o trabalho.

Para um melhor entendimento dos alunos que aplicaram o *test-first* na resolução de problemas, foi apresentado um exercício como exemplo (soma entre dois números), realizado pelo professor durante quinze minutos, desta forma, os alunos tiveram um melhor entendimento sobre o que realmente é para ser realizado. O restante do tempo da pesquisa foi dedicada à realização de atividades previamente selecionadas para resolução dos exercícios, aplicando a lógica da método proposto, com o tempo total de uma hora e quarenta minutos (duas horas-aulas), programadas para este projeto, envolvendo treinamento e aplicação.

Perante as dificuldades apresentadas na aprendizagem em programação e nas melhorias encontradas utilizando o teste de software na programação, este trabalho tende a desenvolver o raciocínio lógico do aluno fazendo com que ele evite a tentativa e erro como forma de encontrar a resposta, para isso, é preciso pensar antes de codificar e nesse contexto, é fazer caso de teste antes do código.

Ensinar os alunos a programar é um processo complexo (LUXTON-REILLY et al., 2018), consequentemente, exige preparação de ambos os lados, professor e aluno.

O trabalho busca suavizar o impacto dos alunos ingressantes com a primeira disciplina de programação, a ideia é que o ensino de programação com base em critérios de teste tenha uma melhor curva de aproveitamento na aprendizagem.

Para entender o propósito desta pesquisa, é preciso saber conceitos relacionados a teste de software, TDD, a ligação deles com o ensino introdutório de programação e o que já está publicado na literatura sobre esse assunto, isso é abordado no Capítulo 2.

Este trabalho propõe um novo modelo de ensino para as disciplinas introdutórias de programação, um aprimoramento, baseado no que já foi publicado na literatura, que será abordado na Seção 2.3, envolvendo teste de software no ensino em computação e os resultados obtidos na aplicação prática do projeto.

No Capítulo 3, é apresentado como essa pesquisa pode contribuir para melhorias no processo de aprendizagem de programação dos alunos novatos utilizando critérios de teste. No Capítulo 4 é detalhado a aplicação da pesquisa, o método que foi utilizado e como os alunos foram abordados, também são apresentados os resultados.

O uso de teste de software no ensino instrutório de programação apresenta resultados positivos, conforme descrito no Capítulo 5. Os alunos que utilizaram o método que é proposto nesta pesquisa apresentaram resultados superiores comparados a amostra que não utilizou.

2 REFERENCIAL TEÓRICO

Neste capítulo, primeiramente são apresentados os principais conceitos sobre teste de software e suas características (Seção 2.1). Posteriormente, na Seção 2.2, é abordado o conceito do TDD, seus princípios e aplicações. A Seção 2.3 aborda soluções propostas que consideram teste de software ou TDD no processo de aprendizagem em programação. Na Seção 2.3.1 são apresentadas as contribuições da pesquisa e por fim, na Seção 2.4 são retratadas as considerações finais do levantamento bibliográfico relacionados ao tema proposto por este trabalho.

2.1 TESTE DE SOFTWARE

O teste de software é uma atividade em que um sistema ou um componente é executado sob condições especificadas, os resultados são observados ou registrados e uma avaliação é feita de algum aspecto do sistema ou do componente (IEEE, 2014). Sua aplicação consiste em uma análise dinâmica do produto, uma rotina relevante para a identificação e eliminação de erros que promove experimentos controlados em artefatos que compõem o produto com a finalidade de evidenciar defeitos de software (DELAMARO et al., 2013). Esses defeitos são revelados pelos erros e falhas detectados na execução dos casos de teste. Para entender melhor o que é teste de software, é preciso compreender a diferença entre os seguintes termos (IEEE, 1990):

- defeito (*fault*): passo, processo ou definição de dados incorreto, como por exemplo, uma instrução ou comando incorreto;

- engano (*mistake*): ação humana que produz um defeito, como, por exemplo, uma ação incorreta tomada pelo programador;
- erro (*error*): diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro;
- falha (*failure*): produção de uma saída incorreta com relação a especificação.

O teste de software bem sucedido é aquele que consegue determinar casos de teste para os quais o programa em teste falhe. Um caso de teste é um conjunto de dados de entrada (valor do domínio de entrada) e de condições de execução desenvolvidos a fim de propor assertivas, com o intuito de verificar se o resultado obtido é igual ao resultado esperado (valor do domínio de saída) (MYERS et al., 2011; ISO et al., 2017).

O algoritmo abaixo, em linguagem C, é um programa que deve determinar se um identificador é ou não é válido. Um identificador é valido se começar com uma letra e conter apenas letras ou dígitos e nesse caso, particularmente, ter entre um e cinco caracteres de tamanho.

Algorithm 2.1: Identifier

```
1 #include <stdio.h>
2 #include <locale.h>

3 int valid_s(char ch) {
4     if(((ch >= 'A') && (ch <= 'Z')) || ((ch >= 'a') && (ch <= 'z'))) {
4         return (1);
6         } else {
7                 return (0);
8         }
9     }

10 int valid_f(char ch){
11     if(((ch >= 'A') && (ch <= 'Z')) || ((ch >= 'a') &&
12         (ch <= 'z')) || ((ch >= '0') && (ch <= '9'))){
13         return (1);
14         } else {
15                 return (0);
```

```
16         }
17  }

18 main (){
19 setlocale (LC_ALL,""); //alterar para o idioma local
20 char achar;
21 int length=0, valid_id=1;

22 printf ("Identificador: ");

23 achar = fgetc(stdin);
24 valid_id = valid_s(achar);

25     if(valid_id){
26          length = 1;
27     }

28 achar = fgetc (stdin);

29     while(achar != '\n'){
30         if(!(valid_f(achar))){
31               valid_id = 0;
32         }
33               length++;
34               achar = fgetc (stdin);
35     }

36     if (valid_id && (length >= 1) && (length < 6)) {
37         printf ("Valido\n");
38         } else {
39         printf ("Invalido\n");
40     }
41}
```

Para que os conceitos de defeito, engano, erro e falha fiquem mais claros, a seguir, o exemplo deles aplicados ao *identifier*:

- defeito (*fault*): Seria implementação incorreta, aceitar algum caractere especial no meio ou no final da palavra, por exemplo.
- engano (*mistake*): A ação humana que produz um resultado incorreto, um defeito, ou seja, a lógica da programação incorreta.

- erro (*error*): Trazer o resultado “válido” de uma entrada com caractere especial.
- falha (*failure*): é o resultado errado, que teve a origem em um defeito, neste caso, invalidar um identificador válido, vice-versa.

Em geral, o teste de software se concentra em três fases, unidade, integração e sistema, conforme ilustrado na Fig. 1. O teste de unidade concentra esforços na menor unidade do software, ou seja, procura identificar erros de lógica e de implementação em cada módulo do software, separadamente. O teste de integração é uma atividade sistemática aplicada durante a integração da estrutura do programa, visando a descobrir erros associados às interfaces entre os módulos. O teste de sistema é realizado após a integração do sistema e busca identificar erros de funções e características que não estejam de acordo com a especificação (DELAMARO et al., 2013). Para este trabalho, o foco é o teste de unidade em programação procedimental, abordando os procedimentos, funções e sub-rotinas desenvolvidos na programação.

Testar um software como um todo, com todas as possíveis entradas e saídas esperadas, pode ser muito custoso e muitas vezes impraticável. para mitigar esse problema, é possível utilizar um subconjunto de entradas reduzido, mas que tenha alta probabilidade de revelar a presença de possíveis defeitos, chamados de subdomínios de teste.

Um subdomínio de teste é um subconjunto do domínio de entrada que contém dados de teste “semelhantes”. Por exemplo, é possível argumentar que os casos de teste (2, -1), “Erro” e (2, -2), “Erro” são semelhantes, pois, qualquer que seja a implementação, intuitivamente espera-se que o programa se comporte de maneira semelhante para ambos os casos de teste. Nesse caso, não seria necessário executar o programa em teste com os dois casos de teste. Generalizando, qualquer caso de teste da forma (x, y), “Erro”, neste contexto, em que $x > 0$ e $y < 0$ se comporta do mesmo modo ou, em outras palavras, pertence a um mesmo subdomínio (DELAMARO, 2016).

No teste de subdomínios, procura-se estabelecer quais são os subdomínios a serem utilizados e são selecionados os casos de teste em cada subdomínio. Neste teste,

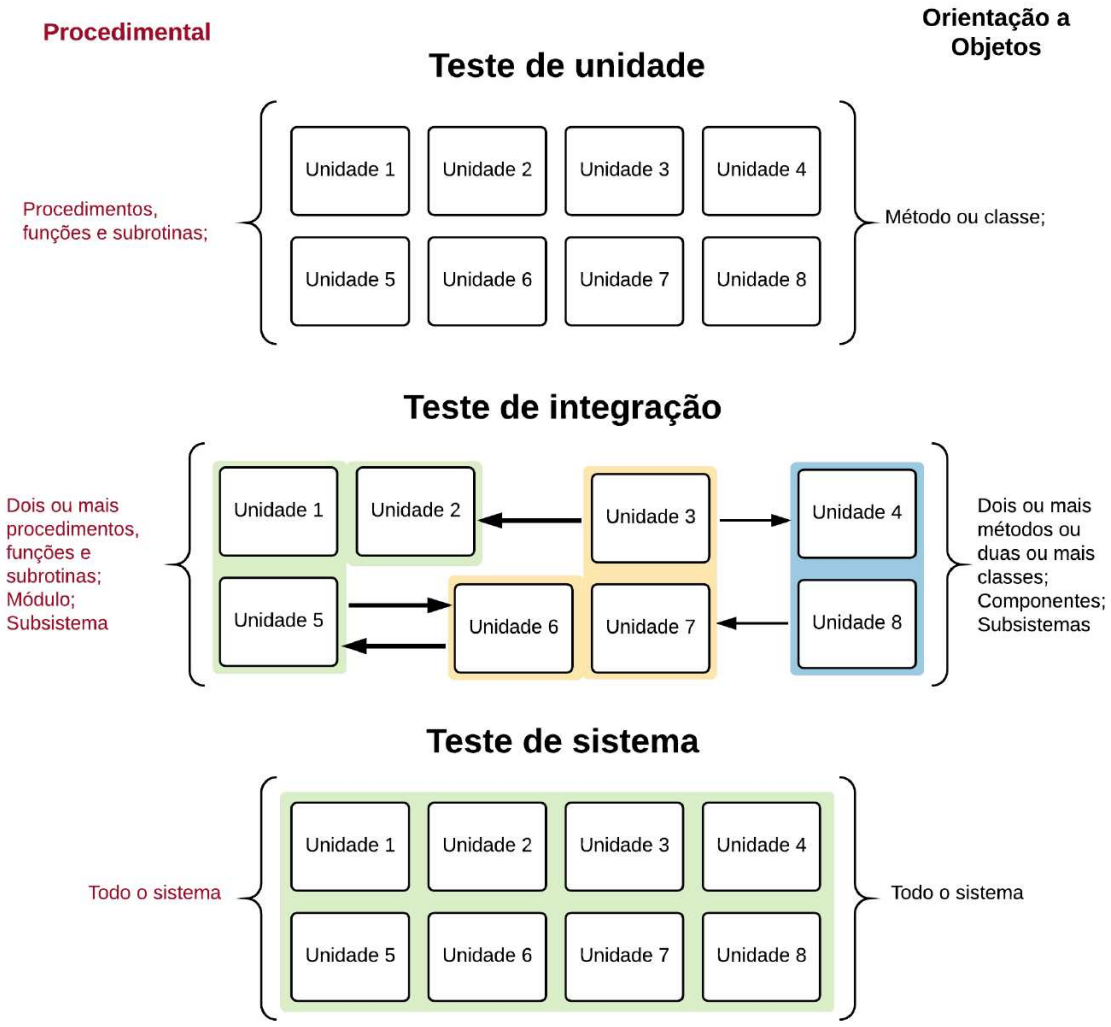

Figura 1: Fase de Teste

Fonte: Adaptado de Binder (2000)

são estabelecidas “regras” para identificar quando os dados de teste devem estar no mesmo subdomínio ou não, essas regras são chamadas “critérios de teste”.

Dependendo do tipo de critério utilizado, são obtidos subdomínios diferentes, o que consequentemente gera conjuntos de teste diferentes. Um único critério de teste ideal é aquele que estabelece que cada subdomínio é unitário, ou seja, cada dado de teste compõe isoladamente o próprio domínio. Sendo assim, todos os elementos do domínio devem ser selecionados para o teste (teste exaustivo).

São definidos “requisitos de teste”, por exemplo, executar determinada estrutura do programa, onde os dados de teste que satisfazem esse requisito pertencem ao mesmo subdomínio (DELAMARO, 2016).

Um critério de teste é um método que serve para direcionar a atividade de teste de software, definindo um conjunto de condições que devem ser utilizados na atividade de teste, ou seja, selecionar os casos de teste de forma a aumentar a probabilidade de revelar a presença de defeitos no código (ROCHA et al., 2001). Critérios de teste podem ser utilizados tanto para auxiliar na geração de conjunto de casos de teste como para auxiliar na avaliação da adequação desses conjuntos. Os critérios de teste de software são estabelecidos, basicamente, a partir de três técnicas: funcional, estrutural e baseada em erros (DELAMARO et al., 2013).

Na técnica funcional, caixa opaca (caixa preta), os critérios e requisitos de teste são estabelecidos a partir da função de especificação do software. As técnicas funcionais mais utilizadas são particionamento em classes de equivalência e análise de valor limite (MYERS et al., 2011).

A técnica funcional de particionamento em classes de equivalência tem como objetivo primário representar o domínio de entrada em pequenos grupos. Desta forma, divide-se determinado domínio em classes, as quais têm por objetivo revelar defeitos para dados de entrada equivalentes. Tais classes são ainda divididas em entradas e saídas tanto válidas quanto inválidas. Essa divisão visa confrontar a especificação com base nestes conjuntos. Para o desenvolvimento dos casos de teste deve ser considerado um

ou mais valores das classes de equivalência, sendo que para as entradas válidas de cada classe deve ser considerada ao menos um caso de teste que satisfaça todos os requisitos das classes válidas. Para as entradas inválidas, devem ser desenvolvidos casos de testes que cubram uma entrada inválida por vez (MYERS et al., 2011).

Na técnica funcional de análise de valor limite, o critério em questão utiliza como base o critério de particionamento em classes de equivalência e ainda determina que, ao identificar as classes de equivalência, deve-se considerar os valores limítrofes de cada classe e os valores próximos ao limite. Ou seja, são considerados valores máximos, mínimos, logo abaixo do máximo e logo acima do mínimo. Os valores limítrofes ou ainda os valores próximos são mais efetivos em revelar defeitos de software se comparados aos outros valores da classe de equivalência (MYERS et al., 2011).

Na técnica estrutural, caixa transparente (caixa branca), os critérios e requisitos são derivados essencialmente a partir das características de uma particular implementação em teste, representadas por estruturas como: Grafo de Fluxo de Controle (GFC), Grafo de Definição-Uso (GDU), dentre outros, conforme o critério de teste estabelecido.

Os critérios baseados em fluxo de controle utilizam apenas características de controle da execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias. Os critérios mais conhecidos dessa classe são (MYERS et al., 2011; PRESSMAN, 2005):

- Todos-Nós: Exige que a execução do programa passe, ao menos uma vez, em cada vértice do GFC; ou seja, que cada comando do programa seja executado pelo menos uma vez;
- Todas-Arestas (ou Todos-Arcos): Requer que cada aresta do grafo, isto é, cada desvio de fluxo de controle do programa, seja exercitada pelo menos uma vez;
- Todos-Caminhos: Requer que todos os caminhos possíveis do programa sejam executados.

Para os critérios baseados em fluxo de controle, utiliza-se o GFC para representação do software sob teste e obtenção de requisitos de teste.

Um programa P pode ser decomposto em um conjunto de blocos disjuntos de comandos. A execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco, na ordem dada. Todos os comandos de um bloco, possivelmente com exceção do primeiro, têm um único predecessor e exatamente um único sucessor, exceto possivelmente o último comando. Usualmente, a representação de um programa P como um GFC (G = (N, E, s)) consiste em estabelecer uma correspondência entre vértices (nós) e blocos e em indicar possíveis fluxos de controle entre blocos por meio das arestas (arcos). Assume-se que um GFC é um grafo orientado, com um único nó de entrada s $\in$ N e um único nó de saída o $\in$ N, no qual cada vértice representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para outro. Quando essas condições não forem satisfeitas, esse fato será indicado explicitamente (DELAMARO, 2016).

A constituição de um GFC não foge à regra de um grafo convencional que é baseado em nós e arestas. Neste contexto, o nó pode representar uma linha ou um conjunto de linhas de código e a aresta é a ligação desse nó com a próxima instrução a ser executada. No caso de uma estrutura de decisão, o grafo apresenta duas ou mais bifurcações (caso seja um *Switch/Case*), já na estrutura de repetição, o grafo apresenta um *loop* caracterizando a repetição em si, independente de qual seja.

Entendendo melhor o código fonte do *identifier*, ilustrado na Fig. 2 juntamente com seus respectivos GFCs, um para cada função. Inicialmente tem a função *valid_s*, que basicamente retorna 1 (nó 13) se a variável *ch* tiver um caractere entre “A” à “Z” ou “a” à “z” (nó 12) e retorna zero caso seja falso (nó 14).

A segunda função, *valid_f*, é definida na linha 12 (nó 16) e tem praticamente o mesmo recurso da função *valid_id*, sendo que a diferença é que ela também válida números. Assim, a verificação é entre “A” à “Z”, “a” à “z” e “0” à “9”: caso a condição seja verdadeira, o retorno é 1 (nó 17), caso a condição seja falsa, o retorno é zero (nó 18) e o nó 19 encerra a função.

Na terceira e última função, o *main*, linha 20, é representada pelo nó 1. Neste primeiro nó, o idioma local é alterado, são criados as variáveis *achar*, *length* e *valid_id*. Em seguida é pedido para o usuário digitar o identificador que é atribuído na variável *valid_f*. Após isso, *valid_id* recebe o resultado da função *valid_s*, que é alimentada com *achar* como parâmetro, neste caso, verificando se o primeiro digito do identificador é um caractere entre a "A" à "Z" ou "a" à "z".

Ainda dentro do *main*, mais especificamente na linha 30, aparece a primeira condição com a variável *valid_id*. Caso a condição da linha 30 for verdadeira (valor 1), o fluxo do algoritmo segue para a linha 31 e a variável *length* (comprimento) recebe o valor 1 (nó 2). Caso a condição seja falsa (valor 0), o algoritmo segue para o nó 3 em que a variável *achar* recebe o comando para efetuar a leitura do próximo caractere informado pelo usuário.

Ambos os caminhos da condição da linha 30 levam para o nó 4 (linha 36), que possui a estrutura de repetição *while* que vai iterar enquanto a variável *achar* for diferente de uma "quebra de linha". Ao entrar no laço de repetição, nó 5, acontece uma condição que chama a negação da função *valid_f*, passando a variável *achar* como parâmetro. Se verdadeira, a variável *valid_id* recebe zero.

Se o laço de repetição for encerrado, a variável *length* é incrementada e a variável *achar* novamente recebe o comando para efetuar a leitura de um carácter a ser digitado, nó 7. Por fim, o algoritmo valida se o conteúdo informado é válido ou não. Na linha 44 tem a condição que verifica se a palavra informada atende os requisitos programados e se tem entre 1 e 6 caracteres (nó 8), se atender as condições, o algoritmo irá apresentar a mensagem de afirmação (nó 9), caso não, será apresentado a mensagem de negação no nó 10. O nó 11 encerra o algoritmo após a apresentação da mensagem.

Com base no GFC, podem ser escolhidos os elementos que devem ser executados, caracterizando, assim, o teste estrutural. Um GFC $G = (N, E, s)$, em que N representa o conjunto de nós, E o conjunto de arcos, e s o nó de entrada. Um "caminho" é uma sequência finita de nós $(n_1, n_2, ..., n_k)$, $k \geq 2$, tal que existe um arco de n_i para n_{i+1} para $i = 1, 2, ..., k-1$. Um caminho é um "caminho simples" se todos os nós que

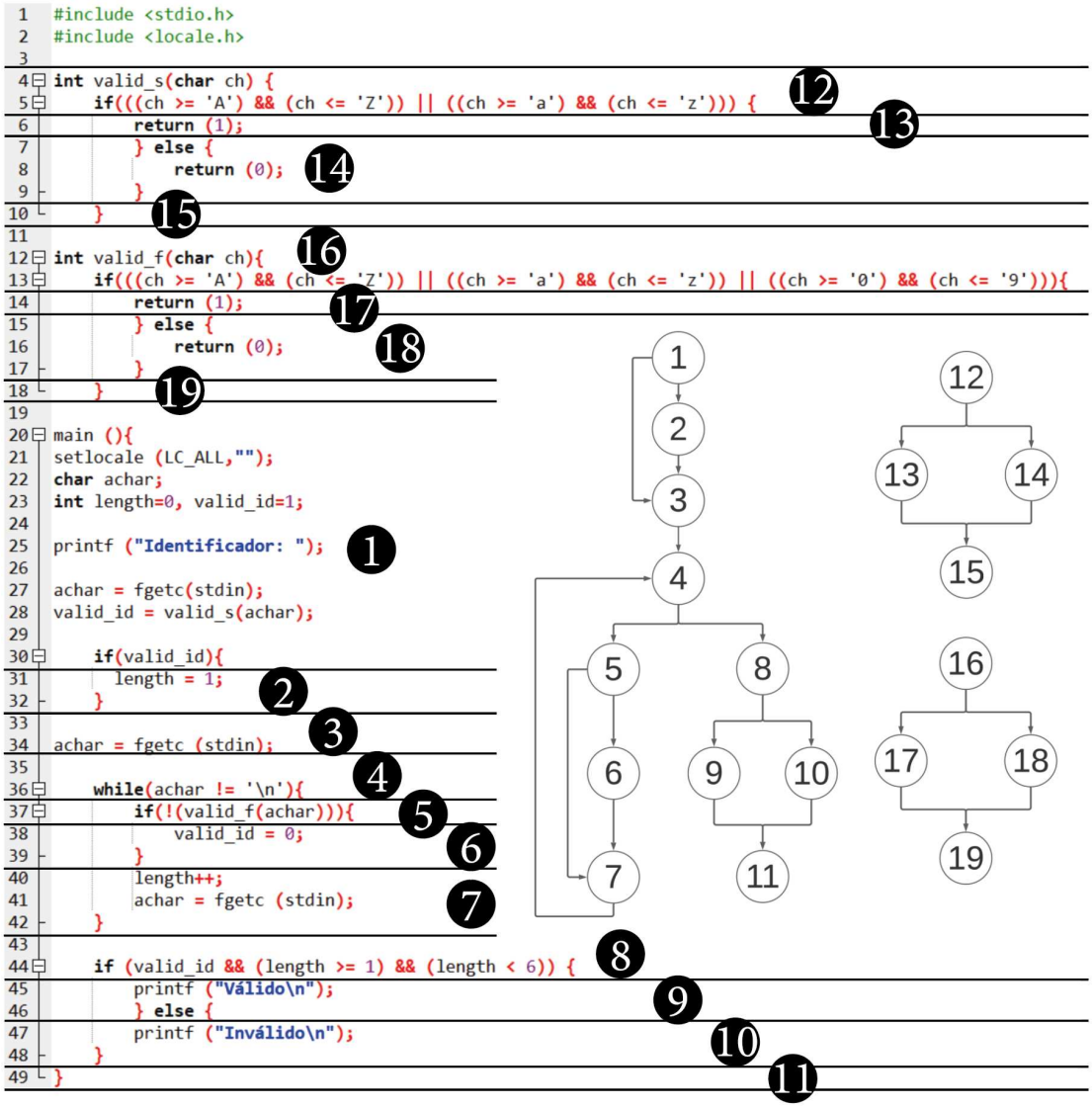

Figura 2: Código do Identifier x GFC

Tabela 1: Requisitos de Teste do Identifier.

Critério	Função	Requisitos de teste
Todos-Nós	main	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
Todos-Nós	valid_s	12, 13, 14, 15
Todos-Nós	valid_f	16, 17, 18, 19
Todos-Arcos	main	(1,2), (1,3), (2,3), (3,4), (4,5), (4,8), (5,6), (5,7), (8,9), (8, 10), (9, 11), (10, 11)
Todos-Arcos	valid_s	(12,13), (12,14), (13,15), (14,15)
Todos-Arcos	valid_f	(16,17), (16,18), (17,19), (18,19)
Todos-Caminhos	main	[(1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (8,9), (8, 10), (9, 11), (10, 11)], [(1,2), (2,3), (3,4), (4,5), (5,7), (7,8), (8,9), (8, 10), (9, 11), (10, 11)], [(1,2), (2,3), (3,4), (4,6), (4,8), (5,6), (5,7), (8,9), (8, 10), (9, 11), (10, 11)], [(1,3), (2,3), (3,4), (4,5), (4,8), (5,6), (5,7), (8,9), (8, 10), (9, 11), (10, 11)], ...
Todos-Caminhos	valid_s	[(12,13),(13,15)], [(12,14), (14,15)]
Todos-Caminhos	valid_f	[(16,17), (17,18)], [(16,18),(18,19)]

compõem esse caminho, exceto possivelmente o primeiro e o último, são distintos; se todos os nós são distintos, diz-se que esse caminho é um "caminho livre de laço". Um "caminho completo" é aquele em que o primeiro nó é o nó de entrada e o último nó é um nó de saída do grafo G (DELAMARO, 2016).

Para o critério todos os nós, os requisitos de teste (ou elementos requeridos) são os nós e para o critério todas as arestas são os arcos/arestas do grafo. No caso das arestas, elas são identificadas pelo nó inicial e nó final (um par ordenado de nós), origem e destino. A Tabela 1 mostra o resultado da aplicação dos critérios todos os nós, todos os arcos e todos os caminhos do *identifer*, relacionando-os diretamente as suas devidas funções e seus requisitos de teste.

Vale observar, ainda na Tabela 1, que o critério Todos os Caminhos para a função *main* gera uma quantidade infinita de requisitos, pois na linha 36 da Fig. 2 existe um *while* que poderá repetir infinitamente enquanto o conteúdo da variável *achar* for diferente de uma quebra de linha. Isto inviabiliza o critério neste programa por ter um alto custo computacional.

A Tabela 2, referente a função *main* do programa *identifier* Fig. 2, faz uma

ligação entre os possíveis dados de entrada (um identificador), o resultado esperado, o critério de teste e o requisito de teste. Cada dado de entrada tem um único resultado mas são apresentados os requisitos de teste para os critérios: Todos-Nós, Todos-Arcos e Todos-Caminhos.

Como a Tabela 2 é baseada em entradas com valores finitos, mesmo envolvendo uma estrutura de repetição em seu corpo, o critério todos os caminhos é "igual ou similar ao critério de todos os arcos, pois o algoritmo segue o mesmo fluxo em seu funcionamento.

O critério estrutural de todos os caminhos é o mais forte dentre os critérios estruturais, pois explora todos os caminhos possíveis de um programa. Entretanto, mesmo para exemplos triviais, é extremamente custoso, tanto para a definição dos casos de teste quanto para a execução. Além deste problema, programas podem conter linhas de código inacessíveis, o que impossibilita a execução dos critérios estruturais (MYERS et al., 2011).

Por fim, existe a técnica baseada em erros, em que os critérios e requisitos de teste são oriundos do conhecimento sobre erros típicos cometidos no processo de desenvolvimento de software. Um exemplo dessa técnica é o critério análise de mutantes, onde assume-se que programadores experientes escrevem programas corretos ou o mais próximo do correto. Os erros são introduzidos nos programas através de pequenos desvios sintáticos que, embora não causem erros sintáticos, alteram a semântica do programa e, consequentemente, conduzem o programa a um comportamento incorreto. Para revelar tais erros, a análise de mutantes identifica os desvios sintáticos mais comuns e, através da aplicação de pequenas transformações sobre o programa em teste, encoraja o testador a construir casos de testes que mostrem que tais transformações levam a um programa incorreto (AGRAWAL et al., 1989).

Ao elaborar conjuntos casos de testes com a maior cobertura possível, consequentemente, a maioria dos nós do GFC são visitados, é possível atender critérios de testes e aos requisitos de testes. Neste sentido, o TDD pode ser considerado como uma estratégia de desenvolvimento, ao projetar os casos de teste antes do código existir

Tabela 2: Casos e critérios de teste para a função *main()* do identifier.

Entradas	**Resultado esperado**	**Critério de teste**	**Requisitos de teste cobertos**
Aaaa	Válido	Todos-Nós	1, 2, 3, 4, 5, 7, 8, 9, 11
		Todos-Arcos	(1,2), (2,3), (3,4), (4,5), (4,7),(4,8), (5,6), (5,7), (8,9), (9, 11)
		Todos-Caminhos	[(1,2), (2,3), (3,4), (4,5), (4,7),(4,8), (5,6), (5,7), (8,9), (9, 11)]
aaaa1	Válido	Todos-Nós	1, 2, 3, 4, 5, 7, 8, 9, 11
		Todos-Arcos	(1,2), (2,3), (3,4), (4,5), (4,7),(4,8), (5,6), (5,7), (8,9), (9, 11)
		Todos-Caminhos	[(1,2), (2,3), (3,4), (4,5), (4,7),(4,8), (5,6), (5,7), (8,9), (9, 11)]
!aaa	Inválido	Todos-Nós	1, 3, 4, 5, 7, 8, 10, 11
		Todos-Arcos	(1,3), (3,4), (4,5), (4,7), (4,8), (5,6), (5,7), (8 10), (10, 11)
		Todos-Caminhos	[(1,3), (3,4), (4,5), (4,7), (4,8), (5,6), (5,7), (8 10), (10, 11)
*aaa2	Inválido	Todos-Nós	1, 3, 4, 5, 7, 8, 10, 11
		Todos-Arcos	1,3), (3,4), (4,5), (4,8), (5,6), (5,7), (8 10), (10, 11)
		Todos-Caminhos	[(1,3), (3,4), (4,5), (4,7), (4,8), (5,6), (5,7), (8 10), (10, 11)]
A*b	Inválido	Todos-Nós	1, 3, 4, 5, 7, 8, 10, 11
		Todos-Arcos	(1,3), (3,4), (4,5), (4,8), (5,6), (5,7), (8 10), (10, 11)
		Todos-Caminhos	[(1,2), (1,3), (3,4), (4,5), (4,7), (4,8), (5,6), (5,7), (8 10), (10, 11)]

(teste funcional), observando-se, na prática, a cobertura de requisitos de teste funcional e de requisitos relacionados ao critério de todos os comandos (teste estrutural). Na próxima seção, o TDD será detalhadamente explicado.

2.2 TDD

O TDD é uma técnica iterativa de desenvolvimento de software que agrega projeto, implementação e teste, auxiliando no processo de entendimento e resolução do problema em questão, direcionado pela criação de soluções testáveis. Ao fazê-lo, indiretamente ele causa a análise do problema em consequência dos casos de teste criados: cada caso de teste exercita características de instâncias do problema e a solução esperada e, nesse processo, permite explorar mais amplamente o domínio do problema. Ao resolver um problema de forma eficiente, é possível questionar se a solução é boa o suficiente.

O TDD, ilustrado na Fig. 3, é uma técnica iterativa de design de software organizada em três etapas (BECK, 2002):

- Primeira etapa: Vermelho, é a definição dos casos de teste em relação a funcionalidade a ser implementada. Todos os casos de teste criados deverão falhar devido à ausência do código na aplicação que contempla a funcionalidade em questão.
- Segunda etapa: Verde, é a implementação do código necessário para passar em todos os casos de teste da aplicação.
- Terceira etapa: É refatoração do código, melhorando o design da solução sem alterar as funcionalidades implementadas.

O TDD, também é conhecido como Desenvolvimento Guiado por Testes, não é considerado uma técnica de teste de software, mas sim uma técnica de design de código, o qual corrobora com código mais coeso e menos acoplado (ANICHE; GEROSA, 2012;

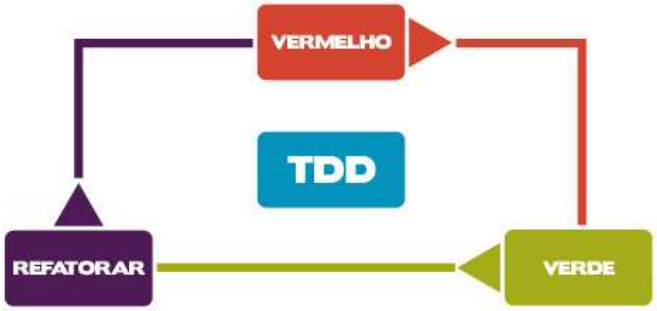

Figura 3: Fluxo do TDD

Fonte: Autoria própria.

BECK, 2002). O TDD estabelece que o código de teste é escrito antes da solução efetiva, o que o classifica também como uma técnica de *test-first*. Cabe destacar que *test-first* não é sinônimo de TDD, pois casos de teste podem ser propostos por uma equipe distinta e encaminhado para a equipe de desenvolvimento, fato que caracteriza que os casos de teste são escritos antes mesmo da implementação, mas não guiam (interferem diretamente) no design das classes e métodos (GASPAR; LANGEVIN, 2007; HEINONEN et al., 2013).

Na prática, ao escrever os casos de teste antes de desenvolver o código, os programadores precisam fazer a diferenciação entre a funcionalidade a ser implementada e a condição básica sob a qual a implementação deve trabalhar (MüLLER; TICHY, 2001). Nessa técnica, os programadores precisam elaborar melhor o design das decisões durante o desenvolvimento. Desta forma, o TDD não garante que o código seja testado de forma completa satisfazendo critérios de cobertura, mas para que o código que tende a resolver o problema proposto seja desenvolvido, é necessário que exista uma série casos de teste criteriosa. Nesse sentido, o TDD é uma ferramenta que pode colaborar para este fim, pois é baseado na construção de testes de unidade.

A aplicação do TDD na indústria, em alguns casos, é vista com um alto custo de desenvolvimento, como por exemplo, na etapa de refatoração. A busca pela melhoria no código pode virar um ciclo de repetição muito grande que consequentemente

tomará muito tempo de desenvolvimento, além de gerar atraso na entrega de novas funcionalidades. Em contrapartida, montar os casos de teste antes do código pode fazer com que logo nas primeiras iterações seja desenvolvido um algoritmo com uma maior corretude, fato que evitará o retrabalho ao longo do desenvolvimento do software.

Na próxima seção será abortado o foco principal deste trabalho, a união de teste de software com critérios de teste, adicionando a técnica de desenvolvimento do TDD, voltados para a o ensino introdutório de programação.

2.3 TESTE DE SOFTWARE E TDD NO ENSINO INTRODUTÓRIO DE PROGRAMAÇÃO

O foco deste trabalho é especificamente na resolução de problemas computacionais no ensino introdutório de programação com a proposta da utilização de critérios de teste de software. Esses temas isolados, trazem grandes quantidades de publicações, mas ao concatenar (Ensino Introdutório de Programação + Teste de Software + TDD), poucos artigos são encontrados, tanto em bases nacionais quanto em bases internacionais.

É comum a confusão entre as definições de defeito, engano, erro e falha entre os alunos que estão iniciando os cursos de Computação. Um engano introduz um defeito no software, um defeito pode produzir um erro que se propagado até a saída do software, constituindo uma falha, conforme demonstra a Fig. 4. Portanto o defeito pode ser considerado uma deficiência mecânica ou algorítmica que, se ativada, pode levar a uma falha.

Em Engenharia de Software, o engano pode ser considerado uma ação humana que produz um resultado incorreto, com por exemplo, uma ação incorreta tomada pelo programador (DELAMARO et al., 2013). Entretanto, conforme visto na Seção 2.1, defeito, engano, erro e falha têm significados diferentes, o que comumente não é explicado para o aluno no primeiro período da graduação e pode causar confusão, principalmente ao trabalhar técnicas de teste de software.

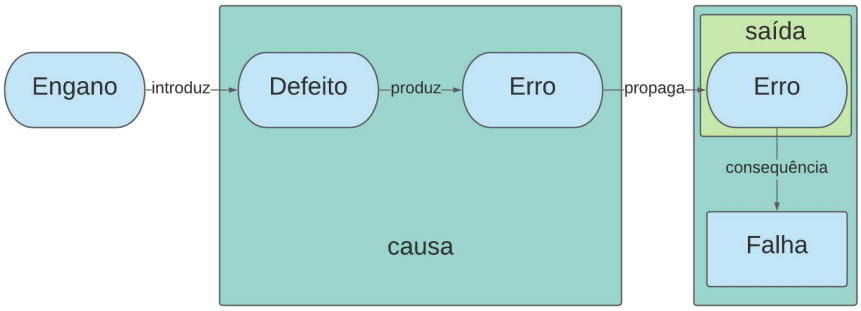

Figura 4: Engano x Defeito x Erro x Falha

Fonte: Autoria própria.

Para analisar melhor o que já foi publicado sobre o tema desta pesquisa, foi realizado uma busca na literatura nos principais veículos nacionais e internacionais condizentes com o uso de teste de software no ensino introdutório de programação.

O resultado dessa busca foi um mapeamento sistemático (FEITOSA et al., 2021b), que teve como o objetivo fazer uma varredura das publicações mais relevantes que tenham algum relacionamento entre teste de software e ensino em programação. Os estudos selecionados apresentam resultados que reforçam os benefícios de considerar atividades de teste de software na educação em Computação e no ensino de programação, com atenção especial à questão de resolução de problemas.

Os artigos classificados no mapeamento sistemático trazem nomenclaturas diferentes referenciando a população utilizada nos estudos. Em uma sintetização realizada, todos se enquadram na população Computer Science 1 (CS1) / Computer Science 2 (CS2), acadêmicos iniciantes dos cursos de computação e afins. Existem referências a estudantes de introdução a programação (KEEFE et al., 2006; BELL et al., 1994; HSIA; PETRY, 1980), estudantes do segundo semestre do curso de graduação (MCKINNEY; DENTON, 2006) e termos como CS1 e CS2, todos como identificação da população das pesquisas (CAMARA; SILVA, 2016).

As avaliações do mapeamento sistemático foram baseadas nas conclusões dos artigos selecionados, principalmente nos critérios de comprovação da eficácia das

técnicas abordadas em seu contexto. As formas de aplicação em que os artigos foram submetidos foram detalhadamente analisadas para verificar quais modos de abordagem e aplicação deram certo e quais não, levando em consideração o público alvo, os estudantes das disciplinas introdutória de programação.

A população do estudo foi restrita ao CS1, alunos ingressantes dos cursos de Ciência da Computação e afins. Justamente por essas especificações, pesquisas em disciplinas mais avançadas como Estrutura de Dados, Programação Orientada a Objetos, a própria Engenharia de Software, entre outras que são lecionadas em períodos mais avançados da grade curricular, foram descartadas.

A classificação dos artigos foi baseada principalmente na resolução de problema e teste de software. Entretanto, outros assuntos como TDD, aprendizado baseado em teste, Test Driven Learning (TDL), *software testing*, *problem resolution*, *problem solving* e Computational Thinking (CT) também foram levados em consideração.

Embora não sejam todos os artigos que tratem especificamente do TDD, de uma forma geral, a maioria aborda o teste de software e em suma, os resultados são positivos quando se correlaciona teste de software a aprendizagem de programação, principalmente nas disciplinas introdutórias.

Ainda que o propósito dessa pesquisa seja explorar o TDD no processo de aprendizagem de programação, vários artigos apresentaram outras formas de abordagem como o LISP-KIE (LISP *Knowledge Integration Environment*), uma ferramentas baseado na prática de resolução de problemas (BELL et al., 1994) e um *plugin* do Eclipse para aplicar o TDD (CAMARA; SILVA, 2016; LAPPALAINEN et al., 2010). Ferramentas que fornecem *feedback* durante a programação, contribui na melhoria de todo o processo de aprendizagem abordado nesse estudo (REIS et al., 2019).

A restrição a estudos voltados para cursos de Computação foi definida para não inserir mais uma variável independente para análise. No entanto, diversos cursos de áreas distintas são atendidos por disciplinas de ensino de Programação e que também

podem ser beneficiados por atividades de teste software para auxiliar na resolução de problemas.

Noções de design *top-down*, refinamento passo a passo, fluxograma estruturado, descrição de caso de teste, análise no contexto do *framework* para o desenvolvimento sistemático, juntamente com a documentação do programas podem ser facilmente aprendido e usado por programadores iniciantes e consecutivamente na produção de bons programas (HSIA; PETRY, 1980).

Outra abordagem analisada foi o Programação Orientada a Problemas (POP), que tem como objetivo desenvolver as habilidades dos alunos novatos em programação em lidar com especificações de problemas e programação. O POP permite os alunos lidarem com problemas mal definidos e trabalhem as diferentes atividades da Engenharia de Software, como elicitação, especificação e testes de requisitos. Os resultados confirmam a eficácia desta abordagem em termos de organização e integridade das especificações dos problemas (MENDONçA et al., 2009).

A única referência nacional utilizada defende a utilização de um projeto de ensino voltado para a disciplina de lógica de programação, utilizando a metodologia ativa de aprendizagem baseada em problemas e aos métodos de programação *Coding* Dojo (MOURãO, 2017).

Outro elemento observado, principalmente nas publicações em veículos nacionais, é a qualidade dos resumos e ausência da palavras-chaves. Devida a limitações de tamanho e termos utilizados, é possível que a busca, normalmente restrita ao título, resumo e palavras-chaves, não recupere artigos que contemplem os critérios de seleção. A baixa utilização de resumos estruturados também impacta nesta busca, dado que muitos resumos tradicionais não são explícitos quanto aos elementos usualmente abordados na estratégia Population Intervention Control Outcome (PICO).

Após ambas as iterações, os autores do mapeamento sistemático realizaram um filtro manual em que todas os artigos que não forem de interesse ao tema da pesquisa fossem descartados. Dessa forma, restaram nove artigos que serviram de base para

criação de um protocolo com a extração de dados dos artigos selecionados. Esse protocolo tem informações como o ano e local de publicação, tipo de estudo, população, intervenção, forma de avaliação da intervenção e resultados obtidos de todos os nove artigos que são apresentados na Tabela 3.

Tabela 3: Relação de artigos selecionados.

ID	Título
1	A Closer Look at Metacognitive Scaffolding: Solving Test Cases Before Programming (DENNY et al., 2019)
2	A Strategy to Combine Test-Driven Development and Test Criteria to Improve Learning of Programming Skills (CAMARA; SILVA, 2016)
3	Translation from Problem to Code in Seven Steps (HILTON et al., 2019)
4	A Framework for Discipline in Programming (HSIA; PETRY, 1980)
5	An approach for problem specification and its application in an Introductory Programming Course (MENDONçA et al., 2009)
6	Most common fixes students use to improve the correctness of their programs (SOUZA et al., 2017)
7	A novice's process of object-oriented programming (CASPERSEN; KöLLING, 2006)
8	Knowledge Integration in Introductory Programming: CodeProbe and Interactive Case Studies (BELL et al., 1994)
9	Uma proposta da eficiência do uso da Metodologia Ativa Baseada em Problemas, utilizando Dojo de Programação, aplicada na disciplina de Lógica de Programação (MOURãO, 2017)

Embora o ensino introdutório de programação, o teste de software e a resolução de problema sejam temas com vasto conteúdo publicado, ao unir os três assuntos é obtido um resultado pequeno, o que por consequência, leva a crer que existem poucos estudos que trabalham especificamente com teste de software no ensino introdutório de programação especificamente na resolução de problemas.

De uma forma geral, todos os artigos selecionados, conforme Tabela 3, trazem resultados positivos relacionados a novas práticas para o ensino introdutório em programação. Fato que reforça a necessidade da comunidade acadêmica a repensar como que essas disciplinas estão sendo lecionadas, quais são as necessidades de mudanças e qual é o melhor caminho a seguir.

A programação é reconhecida como um dos sete grandes desafios na

educação de computação (CASPERSEN; KOLLING, 2009). Para resolver um problema computacional, é requerido uma navegação cuidadosa em múltiplas etapas de aprendizagem (LOKSA et al., 2016). Integrar o TDD com o ensino de Ciência da Computação incorpora a aplicabilidade no mundo real, uma vez que TDD é popular na indústria de software como um elemento da metodologia ágil Extreme Program (XP) (BUFFARDI; EDWARDS, 2014).

Os programadores novatos, muitas vezes, se sacrificam para transformar um problema em um código funcional. Isso acontece devido a necessidade de elaborar um algoritmo para resolver uma classe de problemas, bem como transformar esse algoritmo em um programa que resolva o problema em questão (HILTON et al., 2018). Com a utilização do TDD, os alunos podem especificar os testes antes de implementar o fragmento de programa correspondente. Isso dá aos alunos a chance de considerar o resultado desejado do programa antes de mergulhar nos detalhes de implementação e garante que os alunos possam validar seus novos código escritos imediatamente após sua conclusão (JANZEN; SAIEDIAN, 2005).

A programação utilizando o *test-first* tende a aumentar a confiabilidade do código em termos de cobertura do conjunto de teste (LEMOS et al., 2012), recomendado ser aplicado a cada novo trecho de código (BECK, 2002). O Teste de Software é uma parte indispensável do desenvolvimento de software. Em particular, pode estimular a produção de código mais correto e, portanto, mais confiável (LEMOS et al., 2018).

Os resultados do mapeamento sistemático reforçam a afirmação que há um melhor desempenho dos alunos nas disciplinas introdutórias de programação utilizando técnicas de teste de software, essa evolução inclui: diminuição dos índices de evasão, uma melhor comunicação em pares e em grupos, o desenvolvimento da habilidade de dividir tarefas e a resolução de problemas (MOURãO, 2017).

A comunidade profissional de programação percebeu que projetar testes depois que o programa foi escrito, muitas vezes, leva os testes para uma adaptação ao código escrito, ao invés do problema original. O TDD evoluiu para resolver esse problema (PROULX, 2009) e agora pode ser aplicado na educação em computação.

Determinar quando e como introduzir práticas de TDD em um currículo pode ser difícil. A maioria dos experimentos introduz o TDD no início dos semestres. As apresentações geralmente consistem em (DESAI et al., 2008):

- Explicando o teste de unidade automatizado;
- Descrevendo TDD;
- Fornecimento de documentação;
- Fornecimento de exemplos de como escrever casos de teste, executar casos de teste e interpretar resultados.

A resolução de problemas em computação envolve: entendimento, caracterização e decomposição do problema; coleta, armazenamento e interpretação de dados; reflexão sobre o problema e suas suposições; reflexão sobre o conhecimento necessário, estratégias de solução e sobre a solução propriamente dita, atividades de validação e verificação (SALEHI et al., 2020; BRENNAN; RESNICK, 2012). Alunos que interpretam mal a declaração do problema, provavelmente, formarão um modelo mental inválido do problema e, consequentemente, podem achar difícil de resolve-lo (DENNY et al., 2019).

Embora a resolução de problemas seja um aspecto crucial da programação, poucas oportunidades de aprendizagem na ciência da computação tem foco em ensino nas habilidades de resolução de problemas, como planejamento (SHI et al., 2019). O foco do processo de resolução de problemas está nas quatro etapas: pensamento algorítmico, implementação, análise e comunicação (SHETH et al., 2016).

Recomenda-se que os alunos novatos de programação (CS1) realizem um breve treinamento de resolução de problemas antes do início das sessões de programação. Esta abordagem ajuda os alunos a desenvolver uma compreensão básica de análise, design, conceitos algorítmicos e de resolução de problemas, sem relacioná-los a uma linguagem executável específica. Ter uma base conceitual facilita e acelera a transferência para um

contexto de programação executável dentro de um paradigma funcional, imperativo ou orientado a objetos (KOULOURI et al., 2015).

Para a resolução de problemas no âmbito de programação, observa-se que estudantes com melhor desempenho utilizam estratégias de metacognição e gerenciamento de recursos (BERGIN et al., 2005). Metacognição refere-se ao conhecimento que alguém possui sobre o seu próprio processo cognitivo (FLAVELL, 1976, p. 232)(METCALFE; SHIMAMURA, 1996). A metacognição envolve o planejamento, monitoramento e regulação de cognição (PINTRICH; GROOT, 1990). Gerenciamento de recursos refere-se ao gerenciamento de tempo, esforço, ambiente e pessoas (ZIMMERMAN, 1986). De modo geral, observa-se a aprendizes de programação competentes possuem um desempenho elevado de metacognição (ETELäPELTO, 1993).

Estudos comprovam que os alunos têm dificuldades em lidar com problemas definido como identificar/eliminar ambiguidades no software e formular perguntas para esclarecer os requisitos (MENDONÇA et al., 2009). Tradicionalmente, os cursos de programação se concentram na resolução de problemas bem definidos, no aprendizado da sintaxe da linguagem de programação e não permitem que o aluno desenvolva as habilidades necessárias para superar tais dificuldades (NETO et al., 2013).

A utilização do teste de software como ferramenta de ensino é citada como uma forma de incentivar o raciocínio antes de programar (SHAW, 2000), propor uma situação e verificar sua validade (EDWARDS et al., 2014). Essa técnica possibilita ainda a capacidade de promover o cruzamento de casos de teste entre acadêmicos, a fim de verificar a qualidade de programas desenvolvidos ante os casos de teste de outro programador, professores e assistentes (SHAW, 2000).

Vários estudos relatam que a exigência de técnicas de teste de software em cursos de programação permitem que os alunos produzam software melhor, evidenciado pela maior cobertura do conjunto de casos de teste e pela detecção de defeitos (SPACCO et al., 2006; SOUZA et al., 2015; BUFFARDI; EDWARDS, 2013). Entretanto, o teste de software muitas vezes é utilizado para avaliação, considerando a cobertura do

conjunto de casos de teste (SOUZA et al., 2015; EDUARDS; SHAMS, 2014), não no ensino em programação que é o objetivo desse trabalho.

Na maioria dos casos, os alunos dos cursos introdutórios de programação escrevem programas de baixa qualidade que são superficialmente testados, avaliados e depois descartados. Eles não estão preocupados com a qualidade ou manutenção de seus programas. Desta forma, acredita-se que a forma tradicional de ensino nos cursos introdutórios de programação não são adequados para o desenvolvimento confiável e de alta qualidade de programas (RAHMAN; JUELL, 2006).

O ensino de habilidades de teste de software para os alunos do primeiro ano do curso ciência da Computação pode ser desafiador. Além de aprender as estruturas básicas de programação, os alunos têm que lidar com peculiaridades das técnicas específicas e as ferramentas para teste de software (NETO et al., 2013). Outro fator importante que deve ser levado em consideração na aprimoração da resolução de problemas computacionais é o professor. Desenvolver tarefas de programação eficazes para os alunos é difícil. Os instrutores investem uma grande quantidade de energia para chegar a ideias envolventes que também ilustram os conceitos desejados para melhorar o desempenho do aluno. Embora muitos instrutores estejam dispostos a compartilhar os resultados de seu trabalho, a maioria não tem nenhum tempo ou esforço extra, eles podem se dedicar à publicação ou a documentação para que suas as atribuições possam ser usadas ou adaptadas por outros educadores (EDWARDS et al., 2008), pois existe o desafio de promover aceitação do aluno ao TDD (BUFFARDI; EDWARDS, 2012).

Diversos trabalhos em educação em programação abordam a questão de teste de software (UTTING et al., 2013) e sua integração com ensino de programação (EDWARDS, 2004; JANZEN et al., 2013; CAMARA; SILVA, 2016).

Várias intervenções foram encontradas na literatura, dentre elas, as mais adotadas foram: Pair Programming (PP), TDD, CT e *Test Case*. Essas intervenções não foram todas no mesmo contexto, vale lembrar que cada autor teve sua abordagem aplicada em uma determinada situação.

Assim como na indústria, aplicar o TDD no currículo de formação do aluno também tem um custo, nesse caso, é o de hora-aula. Geralmente o TDD é aplicado na disciplina de Engenharia de Software, dependendo da grade curricular da Instituição de Ensino Superior (IES), lembrando que em alguns casos, o TDD não é incluído na formação acadêmica. Incluir o TDD nas disciplinas introdutórias de programação, mesmo que superficialmente, para depois ter seu aprofundamento em Engenharia de Software, de certa forma gera uma duplicidade na carga horária da grande curricular. Para os alunos que já tiveram alguma experiência com programação no ensino médio, o impacto é menor e o TDD dá a eles algo novo para aprender (WELLINGTON et al., 2007).

O ensino experimental, método que combina o uso de metas e planos, um processo bem definido e uma notação visual, tem o potencial para melhorar significativamente o aprendizado de habilidades de programação. Essa abordagem fornece detalhes que orientam os alunos através do processo de desenvolvimento de um programa e integrada em uma linguagem de programação (visual). Assim que os planos forem definidos e vinculados, o programa resultante pode ser executado (HU et al., 2012). Neste trabalho, especificamente, o planejamento é realizado por TDD e guiado com critérios de teste.

Após aprender os princípios e técnicas básicas de teste de software, os desenvolvedores têm duas vezes mais probabilidade de produzir implementações corretas, embora o código produzido posteriormente seja mais confiável, ele não tende a ser significativamente maior (em termos de linhas de código). Isso indica que a exposição ao conhecimento de teste pode fazer os desenvolvedores produzirem implementações mais confiáveis com aproximadamente a mesma quantidade de código (LEMOS et al., 2018).

Para resolver os problemas de aprendizagem de programação para os alunos que estão iniciando na programação, é preciso ensina-los sobre o processo de desenvolvimento de software, para capacitá-los a seguir etapas organizadas para avançar em direção a um solução para um problema e tratar o desenvolvimento de software

explicitamente como um processo que é realizado em etapas com pequenos passos, ao invés da escrita de uma única solução monolítica (CASPERSEN; KöLLING, 2006).

A aplicação de critérios de teste para o TDD pode melhorar a qualidade do software desenvolvido pelos alunos e fomentar a reflexão na ação. O desenvolvimento de casos de teste, em si, contribui para isso e o uso de critérios de teste melhorou a qualidade do conjunto de teste sem perturbação significativa para o fluxo de trabalho do TDD (CAMARA; SILVA, 2016).

Os principais problemas no ensino e na aprendizagem de programação são as dificuldades em aprender os conceitos de programação, a dificuldade na aplicação desses conceitos durante a construção de programas e a falta de motivação na realização da atividade de programação (SOUZA et al., 2016).

Muitos alunos lutam contra o processo de abstração que está envolvido no projeto e implementação de classes. Ao escrever os programas de teste, os alunos podem ignorar momentaneamente a difícil tarefa de implementar a classe e podem se concentrar na tarefa concreta de usar uma classe. Ao focar no uso da classe, os alunos são mais capazes de raciocinar sobre o comportamento exigido da classe e em seguida, escrever o código que testa esse comportamento necessário (MARRERO; SETTLE, 2005).

As atitudes em relação ao TDD, com a particular atenção ao efeito, atitudes e emoções causados, influenciaram resultados negativos e positivos em alunos no novo método. Compreender a natureza recíproca do afeto em aprendizagem parece ser especialmente importante ao ensinar novos métodos. Métodos como TDD são ensinados, mas não são avaliados diretamente. Consequentemente, é vital avaliar se os alunos estão aderindo aos métodos, bem como entendem suas atitudes que provavelmente informarão seu comportamento (BUFFARDI; EDWARDS, 2012).

A aplicação de critérios de teste para o desenvolvimento orientado a teste poderia melhorar a qualidade do software desenvolvido pelos alunos, e fomentar a reflexão na ação, o desenvolvimento de casos de teste, por em si, contribui para

isso (BRUN, 2010). Os alunos são sensíveis a novas informações, eles parecem acreditar nos benefícios reivindicados do TDD, apesar das dificuldades, é na tarefa mais difícil que os alunos mais precisariam claramente de um design antes de começar TDD (KOLLANUS; ISOMöTTöNEN, 2008).

Para resolver um problema de programação, é requerido uma navegação cuidadosa em múltiplas etapas de aprendizagem (LOKSA et al., 2016), a programação em pares tende a aumentar a confiabilidade em termos de correção, enquanto a programação utilizando o *test-first* tende a aumentar a confiabilidade em termos de tamanho e cobertura do conjunto de teste (BECK, 2002).

2.3.1 CONTRIBUIÇÕES

Pesquisas envolvendo alunos indicam um aumento na compreensão do programa (MÜLLER; HAGNER, 2002) e confiança em fazer alterações e correções no código (MüLLER; TICHY, 2001). Esses resultados tendem a ser mais positivos em cursos mais avançados. Programadores maduros notaram os benefícios do TDD e poderiam conduzir suas práticas corretamente onde os programadores iniciantes lutaram para compreender o propósito do teste (DESAI et al., 2008).

O TDD, na academia, mudou desde a sua fase de concepção, muitos estudos tentaram provar correlações e seus efeitos. Estudos mostram que o TDD expõe os alunos a habilidades analíticas e de compreensão necessárias em teste de software. O TDD é mais uma técnica de programação mais do que um processo de teste e tende a ajudar os alunos com o design de projetos complexos, aumentando sua confiança. O TDD revela habilidades valiosas de teste de software para programadores novatos (DESAI et al., 2008).

Vários artigos discutem problemas na aprendizagem e propõe abordagens de ensino para minimizar as dificuldades na aprendizagem (SOUZA et al., 2016). É preciso ensinar os novatos sobre o processo de desenvolvimento de software a fim de capacitá-los a seguir etapas organizadas para avançar em direção a um solução para um problema,

o desenvolvimento de software deve ser tratado como um processo que é realizado em pequenas etapas ao invés da escrita de uma solução única (CASPERSEN; KOLLING, 2009).

Em um contexto geral, os trabalhos analisados retratam a dificuldade da aprendizagem de programação logo no início da graduação e apresentam soluções diversificadas para amenizar esse problema: (DENNY et al., 2019) propõem resolver casos de teste antes de programar; (CAMARA; SILVA, 2016) já vêm com uma estratégia para combinar o TDD e critérios de teste para melhorar o aprendizado de habilidades de programação; (HILTON et al., 2019) tem uma abordagem que leva o aluno para traduzir o problema em código em sete passos: trabalhar em um exemplo, escrever exatamente o que acabou de ser feito, generalizar, testar o algoritmo, traduzir para o código, testar e depurar.

Baseado na busca realizada na literatura, o TDD ainda é uma técnica pouco explorada especificamente no contexto de aprendizagem de programação, seu foco principal ainda é em teste de software, mas já apresenta resultados positivos em cenários diferentes. O desenvolvimento do caso de testes antes de iniciar a codificação pode auxiliar o aluno a desenvolver a lógica da resolução do problema de uma forma mais eficaz, fazendo com que o aluno pense mais a respeito do problema em questão, reflita, faça os testes, para depois programar.

Foi encontrado uma resistência em adotar técnicas de teste de software no início do desenvolvimento de algoritmos em pessoas com um nível mais alto de conhecimento em programação, por questões culturais, o aluno ou o profissional já aprendeu a programar de uma determinada maneira e tende a ter dificuldades em adotar outro método, mesmo sabendo que poderá ter benefícios.

Os alunos dos cursos de computação e afins podem se beneficiar da incorporação antecipada da aprendizagem colaborativa e do desenvolvimento de habilidades colaborativas, os benefícios incluem um aprendizado mais profundo, desenvolvimento de habilidades desejadas pela indústria, diversão, maior retenção, maior realização, maiores taxas de sucesso no curso e maior interesse (MCKINNEY;

DENTON, 2006).

2.4 CONSIDERAÇÕES FINAIS

O emprego de teste de software no ensino de programação é considerado benéfico para a aprendizagem. No entanto, poucos estudos tratam da questão de resolução de problemas com o auxílio de técnicas de teste. Nesta capítulo, foram apresentados os resultados de um mapeamento sistemático que investiga essa questão.

Comparando-se com a ampla literatura sobre ensino de programação e os estudos sobre integração de teste de software neste ensino introdutório, observa-se que ainda são poucos os trabalhos que tratam das contribuições de teste no desenvolvimento de habilidades para resolução de problemas. Ao mesmo tempo, foi possível identificar abordagens que integram diversos conceitos de teste de software ao ensino de programação, com contribuições que melhoraram a qualidade da educação nesta disciplina introdutória.

Entretanto, cabe a análise de estratégias que conciliem mais técnicas de teste e que permitam ao estudante explorar os problemas a serem resolvidos quanto a cenários mais restritos. Por exemplo, poderiam ser consideradas: classes inválidas considerando critérios funcionais; a particularidades do espaço da solução, relacionados a critérios estruturais; e erros típicos cometidos por estudantes, com auxílio de teste de mutação. No entanto, isso deve ser feito sem exigir o domínio dessas técnicas pelos estudantes, o que sugere a utilização de geração de dados de teste, para interpretação das consequências pelos estudantes, e o fornecimento de informação (*feedback*) que permite a ele entender os princípios relacionados aos critérios envolvidos. Desta forma, será possível abordar a questão de resolução de problema de forma mais completa, contribuindo para uma formação mais sólida e facilitando a ligação com outras disciplinas de Computação e, especialmente, de Engenharia de Software.

3 ABORDAGEM PROPOSTA

A proposta do método deste trabalho consiste em conciliar critérios de teste de software (todos os nós) ao TDD na elaboração da lógica computacional do problema em questão com a finalidade de reduzir curva de aprendizado do aluno nas disciplinas introdutórias de programação.

Neste capítulo, o foco principal é na definição da abordagem. Primeiramente, na Seção 3.1 serão explicadas as etapas que foram percorridas e a forma de como a abordagem foi realizada junto aos alunos, detalhando o objetivo do estudo e como os alunos devem se comportar durante a aplicação da pesquisa.

Será apresentado qual ferramente foi escolhida como apoio nesta pesquisa na Seção 3.2. Posteriormente, na Seção 3.3 será explicado como os alunos realizaram os exercícios. Na Seção 3.4, será detalhado como que a pesquisa foi aplicada com os alunos e e por fim, na Seção 3.5, as conclusões deste capítulo.

3.1 ETAPAS DO MÉTODO PROPOSTO

Para apresentar a aplicação deste trabalho para os alunos, foi criado um passo a passo de como o método deve ser seguido, conforme demonstra a Fig. 5. O fluxo iterativo do método teste de software na resolução de problemas, que deve ser realizada pelos alunos, segue as seguintes etapas:

- Primeira etapa (Problema): Interpretar o problema.

- Segunda etapa (Caso de Teste): Criar casos de testes para resolver o problema.
- Terceira etapa (Programação): Desenvolver o código que atenda aos casos de teste.
- Quarta etapa (Comparação): Comparar resultados obtidos com resultados esperados.
- Quinta etapa (Condição): Verificar o resultado da comparação.

  Sexta etapa (Correção): Correção do código.

  Sétima etapa (Aperfeiçoamento): Melhorar do código.
- Oitava etapa (Condição): Verificação de melhorias.

  Nona etapa (Fim): Encerramento do ciclo.

Para resolver os exercícios que serão tratados nas seções seguintes, o aluno deve primeiramente interpretar o problema (início), refletir as possíveis formas de como chegar no resultado. Posteriormente, é a etapa da elaboração dos casos de teste, o aluno deve montar os casos de testes para que, na próxima etapa, os resultados obtidos sejam comparados com os resultados esperados.

Após a comparação dos resultados, o próximo passo é a primeira condição do ciclo do método. Aqui é preciso verificar, se na comparação, um ou vários casos de teste falharam, ou seja, tiveram falso como resultado. Neste momento existem dois caminhos a seguir: se tem caso de teste que falhou, é preciso ser feito a correção para depois ser comparado novamente; e caso todos os casos de testes não falhem, prossegue-se para a etapa de aperfeiçoamento, onde é preciso verificar se o código ainda pode ser melhorado.

Em uma das últimas etapas, na segunda condição, é verificado se ainda cabem melhorias no código. Caso não, o processo é encerrado e o aluno termina o ciclo do desenvolvimento do programa. Caso sim, é preciso fazer novos casos de teste e o ciclo se repete até não ser mais necessário melhorar o código, ou seja, até ele atender por

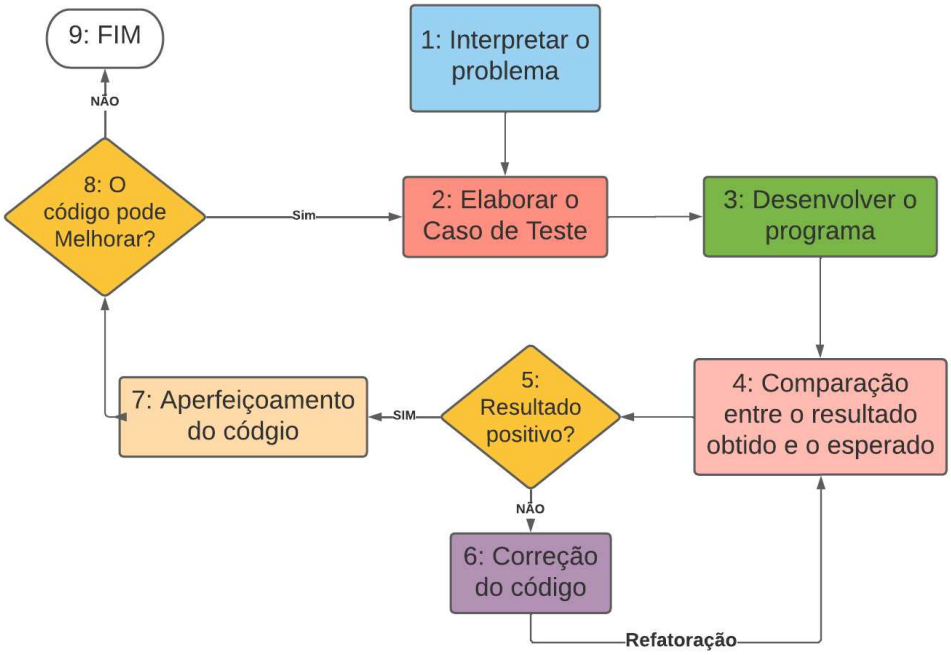

Figura 5: Fluxo da Metodologia Proposta

Fonte: Autoria própria.

completo todos os casos de teste. O aperfeiçoamento pode ser tanto do código da aplicação (o qual não necessitaria de alterações nos casos de teste) quanto no código dos casos de teste.

A abordagem considera um conjunto de conteúdos de Computação, tratados conforme apresentado na Fig. 6. Primeiramente, é explorado o ensino introdutório de computação, o que já delimita o trabalho a um público específico, os alunos ingressantes (CS1).

O segundo quadro trata sobre a resolução de problemas. Nesse contexto, é preciso interpretar o exercício em questão e esboçar as possíveis variáveis para posteriormente, no próximo quadro, desenvolver a lógica para resolver o problema, independente de qual for.

No quarto quadro, existe um ciclo em cima da “Engenharia de Software”, pois é dentro dela que estão o “Teste de Software”, os “Critérios de Testes” e o “TDD”. Dentro do laço que envolve a “Engenharia de Software”, o “Critério de Teste” está ligado com o quadro de “Lógica de Programação”, pois é através da elaboração dos critérios de teste que o aluno deve expandir seu raciocínio pensando na maior cobertura de falhas possíveis. É justamente nesta etapa que entra o objetivo deste trabalho, a aplicação de critérios de teste na resolução de problemas computacionais.

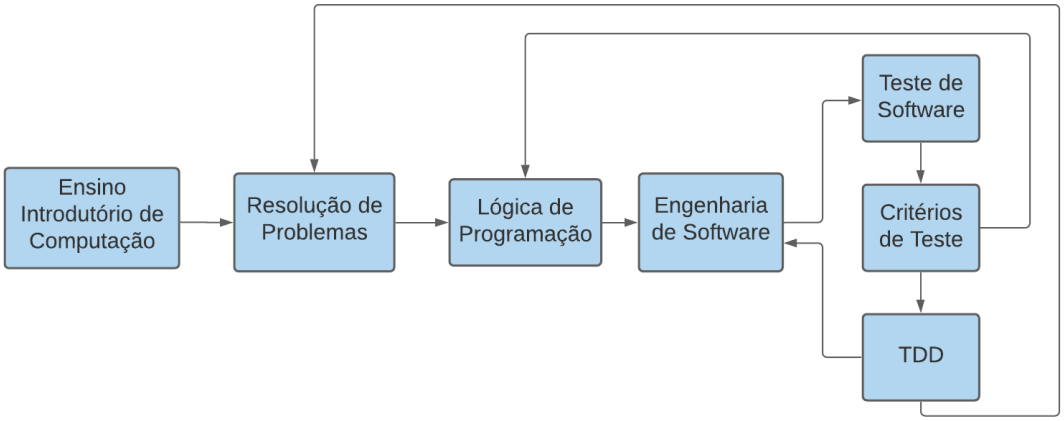

Figura 6: Conteúdos abordados na aplicação da metodologia.

Fonte: Autoria própria.

O último quadro, TDD, tem ligação direta com a “Resolução de Problemas”, pois é através do ciclo do TDD, detalhado no Capítulo 2, que o aluno deve desenvolver melhor a lógica computacional direcionada a resolução do problema. Para isso, o aluno deve fazer as correções e as melhorias de seu código a cada iteração, assim como ampliar a quantidade e variedade dos casos de teste. Neste sentido, o aluno está necessariamente explorando as possíveis falhas do seu código e consequentemente, corrigindo-as.

A relação de critérios de teste com lógica de programação dá-se porque os critérios exploram características (e falhas) de programação. Dessa forma, ao abordar os critérios, é provável que o entendimento sobre lógica de programação seja revisado e equívocos de programação sejam identificados e corrigidos pelo estudante.

Como neste trabalho associa-se TDD a critérios de teste, ocorre um trabalho conjunto para desenvolver tanto competências de resolução de problemas quanto de programação.

3.2 FERRAMENTAS DE AUTOMATIZAÇÃO

Durante a pesquisa, várias ferramentas foram estudadas como forma de aplicar o projeto, como o *CodeWorkout* e o *Comtest*, por exemplo. Entre as ferramentas avaliadas para verificação do porcentual de cobertura dos casos de testes realizadas pelos alunos, foi escolhido a utilização da biblioteca de testes automatizados *Check*.

Em trabalhos anteriores (FEITOSA et al., 2021b), foram identificadas ferramentas para criação de testes automatizados para programas em C. Embora existam soluções que compreendam diversas necessidades de testes automatizados, como *mocks*, *fixtures*, integração com ambientes de desenvolvimento, dentre outras características, no contexto deste trabalho destaca-se a necessidade de facilidade de implementação dos casos de teste, pouca dependência de bibliotecas de terceiros e suporte a medidas de cobertura quanto a critérios baseados em fluxo de controle. Nesse sentido, após reanálise das ferramentas, optou-se pelo *Check*, disponível em: `https://libcheck.github.io/check/`.

3.2.1 CHECK

O *Check* é uma estrutura de teste de unidade para Linguagem de Programação C (Linguagem adotada nesta pesquisa). Os testes são executados em um espaço de endereço separado, portanto, erros de código que causam falhas de segmentação ou outros sinais podem ser detectados. Neste trabalho, o *Check* foi utilizado em uma Máquina Virtual com o Sistema Operacional Ubuntu.

Os alunos não tiveram contato com essa biblioteca, ela foi utilizada posteriormente para verificar os códigos enviados por eles.

START_TEST(test_identifier_01) { ck_assert_str_eq(“pointer”,“Valido”); } END_TEST
START_TEST(test_identifier_02) { ck_assert_str_eq(“*pointer”,“Invalido”); } END_TEST

Tabela 4: Exemplo dos casos de teste do Identifier usando o Check.

Um exemplo da utilização desta biblioteca aplicado ao exemplo do algoritmo *identifier*, exibido na página 20, é demonstrado na Tabela 4. A função *ck_assert_str_eq*, que recebe dois parâmetros, aprova o caso de teste caso a primeira *string* seja idêntica a segunda.

Observe que na Tabela 4 existem dois casos de testes, em caráter de ilustração, foram escolhas *strings* quaisquer, neste caso, “pointer” e “*pointer”. Note que a diferença entre ambas as entradas é que a segunda possui um asterisco no início, fato que a torna um identificador inválido.

Ambos os casos de testes terão resultados positivos, pois estão verificando se o identificador é “Valido” ou “Invalido”, respeitando a lógica do programa.

3.3 EXEMPLO TRABALHADO

O aluno deve encontrar a melhor maneira em como resolver o problema, refletindo a lógica do exercício até esboçar a solução mais adequada. Para isso, são apresentados casos de teste com dados de entrada comparando resultados obtidos com os resultados esperados. O aluno deverá pensar primeiramente nos casos de teste, ou seja, elaborar possíveis entradas para o programa e analisar: “Se a variável ’x’ entrar como parâmetro:”

- Qual será o resultado?
- Será o resultado esperado?

- O resultado será falso ou verdadeiro?
- O algoritmo vai falhar?
- Existe uma forma de refatorar esse código?
- Qual é a melhor maneira de resolver esse problema?
- O que eu posso melhorar nesse meu código?
- É possível reduzir o código sem prejudicar a lógica?
- O código escrito atenderá todas as possíveis entradas?

Depois da análise realizada, o desafio é montar o código que se enquadre perfeitamente no caso de teste que melhor atende a proposta do problema. O principal foco dessa pesquisa é fazer uma análise mais criteriosa do problema antes de começar a desenvolver a solução, para isso é proposto a elaboração de casos de teste.

Na parte prática, os acadêmicos devem ser submetidos a uma série de exercícios previamente selecionados, essas atividades devem ser resolvidos com a finalidade de avaliar o método proposto. Os problemas são apresentados com o nível de dificuldade escalonado, começando do mais simples possível, como a soma de dois números, até exercícios de nível de complexidade maior, que exigem mais dedicação e raciocínio lógico para serem resolvidos, conforme o seguinte modelo:

- Nível 0: Operações Aritméticas Simples
- Nível 1: Operações Aritméticas Avançadas
- Nível 2: Estrutura Condicional
- Nível 3: Estrutura de Repetição
- Nível 4: Estrutura Condicional e de Repetição

Começar com um nível mais baixo de dificuldade de exercício é para que o aluno comece a se familiarizar melhor com o novo método, entender a lógica de seu funcionamento e que não tenha dificuldades na interpretação do problema, tanto é que foram utilizados problemas matemáticos já conhecidos. Posteriormente, assim que a método estiver compreendido por completo, começam os exercícios com níveis de dificuldade maior, escalonadamente.

Para cada exercício, o aluno deve criar casos de teste antes da implementação do programa propriamente dito. Cada caso de teste deve ser especificado de forma simplificada, informando-se apenas os dados de entrada e o resultados esperados como parte da documentação da (futura) implementação (LAPPALAINEN et al., 2010; Python Software Foundation, 1999). Cada caso de teste deve ser processado pela ferramenta associada à abordagem proposta, fornecendo os dados de entrada como parâmetros de entrada da função ou do programa e comparando o resultado obtido com o resultado esperado.

Desenvolver a resolução de um exercício mais detalhadamente, reduz a "tentativa e erro", que faz com que o aluno pense menos na lógica do problema e vá tentando dados de entrada aleatoriamente até conseguir um valor que dê certo, justamente o que deve ser evitado. A pesquisa em questão não foca em fazer com que os alunos inseriram dados na força bruta ou aleatoriamente até chegar ao resultado esperado, mas sim analisar o problema de maneira eficiente, refletir, para resolve-lo da forma mais enxuta possível.

A linguagem de programação adotada nessa pesquisa é o C, imperativa, procedimental, estruturada e tradicionalmente utilizada no início da graduação como os primeiros passos da programação, além de ser considerada uma linguagem base para muitas outras.

Como forma de apresentar o método da utilização de critérios de teste na resolução de problemas computacionais para os alunos de uma maneira simples, a elaboração dos casos de teste, o treinamento segue a linha de raciocínio do *Comtest* (LAPPALAINEN et al., 2010): nele são demonstrados funções relacionadas ao

problema em questão, que são alimentadas com os parâmetros de entrada; os resultados dessas funções são comparados com os resultados obtidos, podendo ser um valor booleano ou até mesmo uma valor matemático, dependendo do problema. Algo como uma função chamado ”soma”, recebe os parâmetros inteiros, dois e três, tendo seu resultado comparado com cinco. A seguir, alguns exemplos:

- soma(x,y) == 5;
- divide(a,b) == 2;
- verificaRaiz(a,b,c) == (x1, x2);
- mdc(10,2) == 2;
- isPalindrome(”aba”) == true;
- isLeapYear(2000) == true;

O *Comtest* (e *doctest*) funcionam desta forma. No entanto, em sua abordagem os casos de teste são informados de forma tabular, sem relação direta com código de programação (que seria o caso do *Comtest/Doctest*).

Em um cenário ideal, a função é alimentada com parâmetros típicos e corretos que retornem o resultado esperado, porém isso pode não acontecer em ambiente de produção, no qual o usuário pode informar valores atípicos e até incorretos. Por isso, é preciso montar os casos de teste com as mais variadas entradas possíveis: números negativos, zero, inclusive resultados inesperados. Isso serve para testar a função e descobrir as possíveis falhas no código do aluno.

A abordagem proposta neste trabalho busca indicar como progredir dentre as diversas possibilidades de entradas possíveis, ou seja, nos tipos de erros que essas entradas possíveis podem apresentar.

Em um primeiro momento, é esperado que o caso de teste falhe em relação ao código do aluno (dada a sua inexistência). Neste instante, ele prossegue para a

implementação de sua solução até que o caso de teste passe. A execução do caso de teste com relação à solução de referência deve ser utilizada para discernir um erro de programação de um erro de compreensão do problema, o que deve ser informado após repetidas tentativas sem perceber o equívoco. Quando a solução do aluno satisfizer os casos de teste criados, os resultados de falha do teste da solução dele com a solução de referência devem ser considerados para guiá-lo a criar novos casos de teste, explorando requisitos de teste e problemas típicos relacionados aos conceitos sendo aprendidos.

O foco principal deste trabalho é como os alunos vão resolver o problema e não entender o problema, por isso, a opção foi por exercícios já conhecidos, alguns até antes mesmo da graduação, esses exercícios são típicos de programação e podem ser utilizados na etapa prática dessa pesquisa, conforme a listagem abaixo:

- Soma de dois números: Deverá entrar dois números
- Verifica se um número é par
- Divisão de números inteiros
- Equação de primeiro grau
- Equação do segundo grau
- MDC (Máximo Divisor Comum)
- Tabuada
- Palíndromo
- Ano bissexto
- Calendário

Os exercícios escolhidos foram a divisão de números inteiros e a fórmula de Bhaskara (equação do segundo grau). Essas atividades foram selecionadas pois

trabalham com os conceitos de matemática básica e estrutura condicional, conteúdos abordados no início das disciplinas introdutórias de programação.

Uma situação que pode ser considerada como simples, é o a divisão entre dois números, porém, é necessário fazer alguns tratamentos como:

- caso o divisor seja igual a zero
- caso o divisor seja maior que o dividendo

Comumente, essas situação não são pensadas pelo aluno e o seu primeiro código, provavelmente não contornará essas situações recém mencionadas.

Algorithm 3.1: div

```
1 int div(int divisor, int dividendo) {
2     int resultado;
3     resultado = dividendo / divisor;
4     return resultado;
5 }
```

Se o divisor for maior que o dividendo, o algoritmo precisa contemplar números reais, pois nesse caso, o resultado será um número fracionado.

Não é possível dividir nenhum número por zero. Essa situação é corriqueira e deve ser tratada, pois caracteriza um erro do programa, situação que poderia ser resolvida com um caso de teste no qual o divisor seja igual a zero.

Levando em consideração o contexto da divisão e os problemas apresentados, é preciso levantar quais os cenários que possivelmente podem acontecer e baseado neles, montar os casos de teste:

Cenário ideal:

- div(10, 2) == 5;
- div(9, 3) == 3;

- div(21, 3) == 7;

Cenário com os parâmetros de entrada invertidos:

- div(2, 10) == “O resultado é um número real”;
- div(3, 9) == “O resultado é um número real”;
- div(3, 21) == “O resultado é um número real”;

Cenário com divisor igual a zero:

- div(7, 0) == “Não é possível dividir por zero”;
- div(6, 0) == “Não é possível dividir por zero”;
- div(14, 0) == “Não é possível dividir por zero”;

Passando pelo segundo grupo de casos de teste, temos o cenário dos parâmetros de entrada da função com valores invertidos, em que certamente trará um resultado fracionado, o aluno deve verificar se seu código é compatível com esse tipo de dado. Tipagem de variáveis é um erro comum quando está aprendendo a programar e pode comprometer o funcionamento do programa.

No terceiro grupo de casos de teste, o cenário é com divisor igual a zero, o aluno deve refletir se o código dele faz esse tratamento, possivelmente, para passar nesse caso de teste será preciso desenvolver uma estrutura de decisão para verificar se o divisor é igual a zero, caso seja, o algoritmo deverá retornar com uma mensagem informando a impossibilidade de continuação na operação matemática e seguir com a lógica do aluno, um possível laço de repetição para digitar o divisor novamente outra outro caminho.

Analisando os casos de teste, principalmente os que falharam (resultado falso), o aluno precisa refatorar seu código para que todos tenham resultado positivo. Nesse contexto, o código a seguir atenderia todos os casos de teste.

Algorithm 3.2: div-final

```
1 float div(float dividendo, float divisor) {
2     float quociente;
3     if (dividor == 0) {
4         printf("Impossivel fazer a divisao por zero");
5     } else {
6        quociente = dividendo / divisor;
7     }
8     return quociente;
9 }
```

Em um exercício de matemática básica, em que o aluno não precisa em entender a lógica da resolução do problema, precisa apenas desenvolver a solução, existem várias situações que devem ser tratadas em nível de programação para que o programa funcione perfeitamente, para isso, é proposto a elaboração dos casos de testes.

Analisando outro problema, o Máximo Divisor Comum (MDC), que consiste em encontrar o máximo divisor comum entre dois números, em um cenário ideal, com valores de entrada 2 e 10 como parâmetros em uma função, o resultado esperado é 2, mas existem várias situações adversas que podem acontecer, como por exemplo: se a ordem dos números de entrada for invertida; se forem números iguais; primos; se entrar 0, -2 (o qualquer outro negativo). Em um problema relativamente simples, existem várias situações que precisam ser pensadas pelo aluno e é nesse sentido que os casos de teste podem auxiliar para desenvolver o código pensando nas diferentes situações que o algoritmo pode se deparar.

Ainda no problema do MDC, o seguinte código em linguagem C pode ser uma possível solução:

Algorithm 3.3: mdc

```
1 int mdc(int m, int n) {
2     int r;
3     while (m % n != 0) {
```

```
4           r = m % n;
5           m = n;
6           n = r;
7       }
8       return n;
9   }
```

Seguindo a linha deste estudo, é preciso montar os casos te testes para verificar se o código acima atende a todos os cenários possíveis, portanto, o aluno deve se preocupar em montar os casos de testes com a maior cobertura possível, explorando os mais diversos cenários e todas as possíveis categorias de parâmetros, conforme a seguir:

Cenário ideal:

- mdc(2, 10) == 2;
- mdc(3, 9) == 3;
- mdc(7, 21) == 7;

Cenário com os parâmetros de entrada invertidos:

- mdc(10, 2) == 2;
- mdc(9, 3) == 3;
- mdc(21, 7) == 7;

Cenário com números primos:

- mdc(5, 2) == 1;
- mdc(3, 7) == 1;

- mdc(31, 47) == 1;

Cenário com zero:

- mdc(0, 2) == 1;
- mdc(3, 0) == 1;

Cenário com números negativos:

- mdc(4, -2) == 2;
- mdc(0, -2) == 1;
- mdc(-3, 0) == 1;

Nesse contexto, a primeira parte da expressão, antes do sinal de igualdade (”==”na Linguagem C) é o resultado obtido, a segunda parte, após o sinal e igualdade, é o resultado esperado. Se o resultado da expressão for verdadeiro, o código do aluno atendeu ao caso de teste, se for negativo, não.

Mediante os mais diversos parâmetros de entrada, o aluno deve refletir se seu código irá atender a todos os casos de testes, caso não, será necessário refatorar o código até que todos os casos de testes sejam verdadeiros na comparação entre resultados obtidos e resultados esperados.

O código que a princípio iria funcionar, deve passar por uma reanalise, envolvendo todos os casos de testes, para posteriormente ter sua versão final implementada (refatoração). A cada ciclo desse, o algoritmo deve ser melhorado, não pela tentativa e erro, mas sim pelos casos de testes e pela análise feita ciclicamente pelo próprio aluno.

O cálculo do MDC é um exemplo de um programa relativamente simples que possui uma ou mais estruturas de repetição. Esse algoritmo se enquadra em um perfil mais avançado, pois durante sua execução, ao visitar todos os caminhos, ele pode entrar em um laço infinito.

Algorithm 3.4: MDC Final

```
1 #include <stdio.h>

2 int MDC(int a, int b){
3  int resto;
4  while(b != 0){
5    resto = a % b;
6    a = b;
7    b = resto;
8  }
9  return a;
10 }

11 int main() {
12   int a=0, b = 0;
13   scanf("%d %d", &a, &b);
14   printf("MDC(%d,%d) = %d\n", a, b, MDC(a,b));
15 }
```

Observando o código acima, é impossível aplicar o critério todos os caminhos na função MDC, pois existem infinitos números que são diferentes de zero. Mesmo que seja testado uma grande quantidade de números (o universo dos números inteiros, por exemplo), este critério de teste seria inviável para esse tipo de programa devido seu alto custo.

A Fig. 7 apresenta os grafos de fluxos de controle da resolução do problema do MDC. Os nós 1 e 2 representam a função *main*, que basicamente é iniciada e logo após faz a impressão e chama a função *MDC* no nó 2. A função *MDC*, representada pelo grafo com os nós 3,4,5 e 6, recebe os parâmetros do *main*, nó 3, depois entra no laço de repetição *while* que irá acontecer enquanto o resto da divisão do valor da variável *b* for diferente de 0, caso seja verdadeiro, o fluxo do algoritmo segue para o nó 5 e prevalecendo o resultado verdadeiro o algoritmo ficará entre os nós 4 e 5. A partir do momento em que a estrutura de repetição tem o resultado como falso, o programa segue

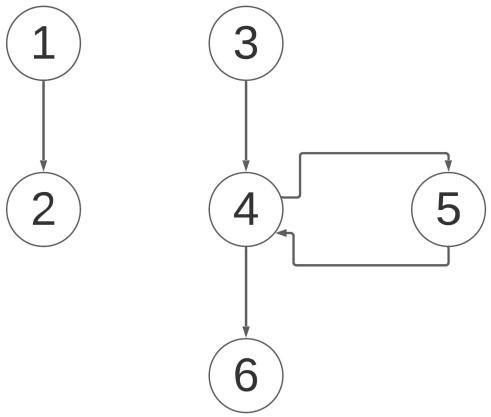

Figura 7: GFCs do MDC

Tabela 5: Requisitos de Teste do MDC.

Critério	**Função**	**Requisitos de teste**
Todos-Nós	main	1, 2
Todos-Nós	mdc	3,4,5,5,6
Todos-Arcos	main	(1,2)
Todos-Arcos	mdc	(3,4), (4,5), (5,4), (4,5)
Todos-Caminhos	main	[(1,2)]
Todos-Caminhos	mdc	[(3,4),(4,5),(5,4),(4,6)], [(3,4),(4,6)], [(3,4),(4,5),(5,4),(4,5),(5,4),(4,6)], ...

para o último nó, o 6 e é finalizado.

A Tabela 5 traz uma relação entre os critérios “todos os nós”, “todos os arcos” e “todos os caminhos” com suas respectivas funções e seus critérios de teste. Observa-se que o critério “todos os caminhos” da função *mdc*, assim como no exemplo do *identifier*, também é infinito, característica típica de um trecho de código que envolve uma estrutura de repetição baseada em uma condição que nunca pode ter o resultado como falso.

A lógica abordada nesses dois problemas estende-se a todos os outros exercícios que fazem parte da aplicação dessa pesquisa. Cada exercício deverá ter seus casos de teste criados para atender os critérios de teste do problema em questão. Após

a criação dos casos de teste, entra a parte da equivalência, onde é preciso confrontar os resultados obtidos com resultados esperados e caso haja necessidade, fazer a refatoração do código até que todos os casos de testes sejam atendidos.

Em todos os exercícios, o aluno deve pensar que se entrar valor “x” e “y” na função, o resultado deve ser “z”, dessa forma montando a maior quantidade e variedade de casos de teste possível para cobrir o maior percentual de margem de erro. Algo similar a função ”*assertEquals*”utilizada no *JUnit*, um *framework open source* utilizado para criar casos de testes automatizados na linguagem de programação Java (WICK et al., 2005).

Trazendo o problema para um cenário real, de forma abstrata, independentemente do nível que for, o professor deve instruir o aluno a pensar na maioria dos cenários possíveis para o exercício em questão, feito esse levantando, considerando os mais diversificados parâmetros possíveis como entrada, devem ser montados os casos de teste, com o objetivo de ter a maior abrangência possível do problema, pois quanto maior for a abrangência, menor será a possibilidade de erros.

3.4 APLICAÇÃO PRÁTICA

O primeiro contato com os alunos foi para apresentação, treinamento e orientação. O professor fez uma explicação superficial sobre Teste de Software, Critérios de Teste, Caso de Teste e TDD, o suficiente para os alunos entender o proposito do exercício. Dependendo do nível em que estiver caminhando a disciplina de programação, é necessário explicar o conceito do funções, o que não foi o caso. Após isso, o aluno deve se dedicar exclusivamente para a realização das atividades, aplicar os conhecimentos recentemente adquiridos nos exercícios dos primeiros degraus de dificuldade, começando do mais simples possível até chegar nos intermediários para, por último, chegar no mais alto grau de dificuldade.

No total, a amostra foi de dezenove alunos, dez para o grupo que utilizou o *test-first* e nove para o grupo que não utilizou o *test-first*. Na aplicação prática do

método proposto por esta pesquisa, para a turma que utilizou teste de software na resolução problemas, foi disponibilizado três folhas impressas, conforme o modelo no Apêndice A.

Foram propostos três problemas para os alunos. O primeiro, que é exposto na primeira página, é um exemplo, um modelo respondido que deve ser seguido para as próximas resposta, referente ao contexto da soma de dois números. O segundo problema, respectivamente na segunda página, é referente a divisão de números inteiros. A terceira e última página é sobre a fórmula de Bhaskara.

As duas folhas dos exercícios possuem três tabelas com três colunas cada (entradas, resultados esperados e resultados obtidos). A ideia é confrontar o que era para o algoritmo retornar e o que retornou. Foi disponibilizado ainda para todos os alunos um formulário online, para que os códigos fontes pudessem serem enviados para análise.

O primeiro exercício aplicado nos alunos foi referente ao nível dificuldade zero do modelo escalonado que foi criado, envolvendo operações aritméticas simples. Já o segundo problema, referente a fórmula de Bhaskara, aborda os níveis um e dois, pois trata de operações aritméticas um pouco mais avançadas e estrutura condicional.

Nesta pesquisa, foi utilizado exercícios com os níveis 0, 1 e 2, respectivamente, operações aritméticas simples, avançadas e estrutura condicional. Os níveis mais avançados de dificuldade de exercícios, que envolvem estruturas de repetição e estruturas de repetição e condição juntas, não foram utilizados nesta pesquisa por questões de tempo e do amadurecimento dos alunos. Os níveis mais avançados podem ser utilizados em trabalhos futuros, assim como a utilização deste método em outras linguagens de programação.

O aluno precisou pensar em duas informações: dados de entrada e resultados esperados. A cada conjunto de casos de testes, o aluno foi estimulado a pensar em entradas diferentes, consequentemente, resultados esperados diferente. Por exemplo, em uma divisão de números inteiros, se o divisor for igual a zero, se algum número

for negativo. A ideia é que seus casos de testes cubram a maior variedade de entradas distintas.

As seguintes orientações foram passadas para os alunos:

- Esta atividade compreende a resolução de três exercícios. Para cada exercício, será apresentada a descrição de um problema a ser resolvido com um programa em C. Para guiar a resolução do problema, serão utilizados e criados casos de teste da seguinte forma:
- Preencha as duas primeiras colunas da tabela “Casos de Testes 01” com as supostas entradas e resultados esperados para o programa em questão.
- Desenvolva um programa em C para resolver o problema.
- Preencha a terceira coluna da tabela “Casos de Testes 01” com os resultados obtidos pelo programa.
- Compare os “Resultados Esperados” e os “Resultados Obtidos”, reflita se seu programa atenderá a todos os possíveis valores de entrada e o corrija se necessário. Ao término desse passo, guarde uma cópia do seu programa para envio da forma como ele está!
- Elabore um conjuntos de diferentes entradas para o seu programa e preencha as duas primeiras colunas da tabela “Casos de Testes 02”
- Melhore seu programa para atender os Casos de Testes da segunda tabela.
- Execute o programa e compare os “Resultados Esperados” e os “Resultados Obtidos”, reflita se seu programa atenderá a todos os possíveis valores de entrada e o corrija se necessário. Ao término desse passo, guarde uma cópia do seu programa para envio da forma como ele está!
- Pense em outros possíveis dados de entrada, diferentes dos anteriores e repita o procedimento com a tabela “Casos de Testes 03”. Seu algoritmo deverá atender

Entradas	Resultados Esperados	Resultados Obtidos
A = 1, B = 3	4	4
A = 0, B = 2	2	2
A = -1, B = 2	1	1

Tabela 6: Exemplo Caso de Testes da Soma de dois números.

aos mais diversos possíveis dados de entrada e, consequentemente, ter menos falhas. Ao término desse passo, guarde uma cópia do seu programa para envio da forma como ele está!

O aluno é provocado a refletir, antes de montar o próximo conjunto de casos de teste, ele é conduzido a pensar:

- Se algum dos "Resultados Esperados" forem diferentes dos "Resultados Obtidos", o que precisa ser alterado no seu algoritmo para ele trazer o resultado correto?
- Os casos de testes realizados no primeiro conjunto abrangem todas as possíveis entradas?
- Quais outros valores de entrada você acha relevante para esse problema?

Em todos os casos, os alunos são induzidos a fazerem um conjunto de três casos de testes em três tentativas, ou seja, nove casos de testes. A seguir, os detalhes de cada página que foram entregues aos alunos:

- Na primeira página, é explicado detalhadamente como que a pesquisa deve ser conduzida e é ilustrado o exemplo da soma entre dois números.

  Um caso de teste consiste em um conjunto de dados de entrada e de resultados esperados. Por exemplo, para o problema da soma de dois números quaisquer (representados pelas variáveis 'a' e 'b'), os dados de entrada consistem de dois números e o resultado de saída é um número (obtido pela soma dos dois indicados como entrada). Veja possíveis casos de teste para esse exemplo:

Entradas	Resultados Esperados	Resultados Obtidos
A = 10, B = 2	5	5
A = 6, B = 3	2	2
A = 4, B = 0	“Não Existe”	“Não Existe”

Tabela 7: Exemplo Caso de Testes da Divisão de Números Inteiros.

- Na segunda página, é apresentado um problema da divisão de números inteiros. A divisão é uma das quatro operações básicas da matemática e é inversa à multiplicação. A divisão de um número inteiro consiste em seu fracionamento, na sua fragmentação, que pode ter como resultado um número inteiro. Foram apresentadas as seguintes instruções para resolução do exercício:

  – Os dados de entrada consistem dos valores para o dividendo e o divisor.

  – O resultado esperado é o resultado da divisão.

  – Caso não seja possível fazer a divisão, retornar com a mensagem “Não é possível realizar a divisão”.

- Na terceira página, é apresentado o segundo problema para o aluno resolver é uma equação polinomial, que é equação que envolve uma variável matemática ‘x’ elevada a uma potência e multiplicada por uma constante. Por exemplo, uma equação polinomial de segundo grau pode ser escrita da seguinte forma: ax2 + bx1 + cx0. ou, de forma mais simples, ax + bx + c. Quando queremos encontrar a raiz de uma equação, queremos resolver o valor da variável matemática ‘x’ de modo que o resultado da equação seja igual a zero. A Fórmula de Bhaskara é uma maneira de calcular os valores de ‘x’ para obter a solução desejada.

  Para resolver esse problema, foi solicitado para o aluno considerar as instruções para resolução do exercício:

  – Os dados de entrada consistem dos valores para ‘a’, ‘b’ e ‘c’.

  – Os resultados esperados são as raízes ‘x1’ e ‘x2’.

  – Caso tenha apenas uma raiz, informar como ’x1’.

Entradas	Resultados Esperados	Resultados Obtidos
A = -1, B = 2, C = 3	X1 = -1, X2 = 3	X1 = -1, X2 = 3
A = 1, B = -4, C = 10	“Delta Negativo”	“Delta Negativo”
A = 0, B = 2, C = 3	“Equação 1º grau”	“Equação 1º grau”

Tabela 8: Exemplo Caso de Testes da Fórmula de Bhaskara.

– Caso não tenha raiz real, informar a mensagem “Erro”.

Material disponível em: `https://github.com/yurifeitosa/dataset-cs1testing`.

3.5 CONSIDERAÇÕES FINAIS

Considerando os resultados do mapeamento sistemático, que será explicado na Seção 4.1, o teste de software no ensino introdutório de programação é uma abordagem que já vem sendo estudada há bastante tempo, desde mil novecentos de oitenta. Em contrapartida, o TDD ainda é pouco explorado no contexto de aprendizagem de programação, mas em seus estudos, apresenta evoluções positivas em cenários distintos. Algumas publicações trazem o *test-first* nesse contexto de ensino, o desenvolvimento de casos de teste contribui para a qualidade do código desenvolvido pelos alunos, mas o uso de critérios de teste melhorou a qualidade do conjunto de testes sem perturbação significativa no fluxo de trabalho do TDD.

Espera-se que o resultado do desenvolvimento de casos de teste, considerando critérios de teste de software antes de iniciar a codificação, possa auxiliar os alunos a desenvolver a lógica da resolução do problema computacional de uma forma mais eficaz, contribuindo para o seu melhor entendimento nas disciplinas introdutórias de programação e no desenvolvimento de sistemas.

Para auxiliar a abordagem com os alunos, foi realizado um exemplo trabalho para que eles tenham certeza do que realmente é para ser feito, desta forma, diminuindo as possibilidades de descarte de amostras.

Nenhuma ferramenta avaliada se enquadrada perfeitamente com o método proposto. Entretanto, utilizar o *Check* para validar os casos de testes foi uma alternativa viável e gratuita encontrada para não comprometer os resultados da pesquisa, levando em consideração que não há tempo hábil para desenvolvimento de uma nova plataforma e que a utilização das ferramentas citadas atendem o proposito dessa pesquisa.

A abordagem da metodologia proposta neste trabalho buscar forçar o aluno a pensar melhor na resolução do seu problema antes mesmo de começar a programar. Para isso foram resgatados conceitos da Engenharia de Software no auxílio do desenvolvimento da lógica computacional.

O aluno deve pensar de forma mais abrangente antes de compilar seu código, primeiramente pensar no caso de testes para depois desenvolver o código, feito isso, deve ser confrontado os resultados esperados com os resultados obtidos. Isso deve ser um ciclo com a maior e mais diversificada quantidade de casos de testes possíveis, até diminuir ao máximo, preferencialmente a zero, a possibilidade de erro de seu programa.

No próximo capítulo, é relatada a aplicação e a avaliação da abordagem proposta, discutindo-se as ameaças e os resultados.

4 AVALIAÇÃO DA ABORDAGEM

Nesse trabalho, é proposto um estudo piloto a fim de verificar a eficácia da técnica em questão. A parte prática ocorreu durante o último bimestre dos alunos que estavam cursando uma disciplina introdutória de programação.

Na Seção 4.1 é apresentado a caracterização das questões de pesquisa, em seguida, na Seção 4.2 é explicado sobre a configuração do estudo.

Os resultados, discutidos na Seção 4.3, foram divididos em: amostragem de alunos, resultados compilados, *feedback* e análise estatística. Logo após, na Seção 4.4, são apresentados as ameaças à validade dessa pesquisa, e por fim, na Seção 4.5 são apresentadas as considerações finais desse capítulo.

O objetivo desse estudo é avaliar o desempenho individual de cada aluno com base nas respostas submetidas e para confrontar os resultados dos alunos, foi utilizado a biblioteca exposta na Seção 3.2.1.

Para verificação dos códigos desenvolvidos pelos alunos, foram utilizadas as funções *ck_assert_int_eq* e *ck_assert_float_eq*, disponíveis na biblioteca *Check*, para comparação dos valores. Ambas as funções comparam dois valores, respectivamente números inteiros e números de ponto flutuante, e exibem uma mensagem predefinida com a condição e os valores de ambos os parâmetros de entrada em caso de falha.

A Tabela 9 traz os casos de testes do problema da divisão de números inteiros, e a Tabela 10 mostra os casos destes utilizados para verificar os algoritmos da formula de Bhaskara.

START_TEST(test_divi_01) {
ck_assert_int_eq(divi(10,2),5);
} END_TEST

START_TEST(test_divi_02) {
ck_assert_int_eq(divi(9,3),3);
} END_TEST

START_TEST(test_divi_03) {
ck_assert_int_eq(divi(10,0),0);
} END_TEST

START_TEST(test_divi_04) {
ck_assert_int_eq(divi(10,4),2);
} END_TEST

START_TEST(test_divi_05) {
ck_assert_int_eq(divi(8,2),4);
} END_TEST

Tabela 9: Casos de testes da divisão de inteiros.

Cada função recebe dois parâmetros, trataremos como 'X' e 'Y'. O caso de teste terá um retorno positivo se ambos os parâmetros forem iguais, ou seja, se 'X' for igual a 'Y'. Neste trabalho, o primeiro parâmetro é a função de cada problema que foi proposto para o aluno resolver, o segundo parâmetro é o resultado obtido.

Entendendo melhor a Tabela 10, o conjunto de caso de teste 01 é referente ao cenário mais simples e intuitivo possível, sendo considerados dados óbvios que dificilmente falharão. No caso de teste 02, o conjunto de valores de entrada similar ao do Caso de Teste 01, entretanto, com valores diferentes. O Caso de Teste 03 verifica se o delta é igual a zero, fato que caracteriza as raízes iguais. O Caso de Teste 04 valida a possibilidade do delta ser negativo e o Caso de Teste 05 valida a possibilidade do parâmetro 'A' ser igual a zero, o que caracteriza uma equação do primeiro grau.

Alguns estudantes foram excluídos da compilação dos dados, para isso, foi adotado o critério cuja cobertura de código não evoluiu ou zerou ao longo do estudo. Os casos de testes e os algoritmos válidos foram avaliados e os resultados são apresentados na Seção 4.3.

```
START_TEST(test_bhaskara_01) {
    float raiz1, raiz2;
    ck_assert_int_eq(bhaskara(-1,2,3,&raiz1,&raiz2),2);
    ck_assert_float_eq(raiz1,-1);
    ck_assert_float_eq(raiz2,3);
} END_TEST
```

```
START_TEST(test_bhaskara_02) {
    float raiz1, raiz2;
    ck_assert_int_eq(bhaskara(2,0,-18,&raiz1,&raiz2),2);
    ck_assert_float_eq(raiz1,3);
    ck_assert_float_eq(raiz2,-3);
} END_TEST
```

```
START_TEST(test_bhaskara_03) {
    float raiz1, raiz2;
    ck_assert_int_eq(bhaskara(-2,20,-50,&raiz1,&raiz2),2);
    ck_assert_float_eq(raiz1,5);
    ck_assert_float_eq(raiz2,5);
    ck_assert_float_eq(raiz1,raiz2);
} END_TEST
```

```
START_TEST(test_bhaskara_04) {
    ck_assert_int_eq(bhaskara(1,-4,10,0,0),-1);
} END_TEST
```

```
START_TEST(test_bhaskara_05) {
    ck_assert_int_eq(bhaskara(0,2,3,0,0),1);
} END_TEST
```

Tabela 10: Casos de testes de Bhaskara.

4.1 CARACTERIZAÇÃO DAS QUESTÕES DE PESQUISA

Para a comprovação dessa proposta, as seguintes hipóteses foram definidas:

- Hipótese nula: A adoção do TDD no ensino de programação não traz melhorias no processo de aprendizagem e desenvolvimento.
- Hipótese alternativa: A qualidade do código do programa produzido pode não ser melhorada com a adoção do TDD no desenvolvimento de programas.

O estudo proposto consiste em validar a hipótese apresentada seguindo o planejamento de treinamento, execução com e sem *test-first*, analise e resultados.

O propósito deste estudo é avaliar os efeitos da adoção do TDD durante os primeiros passos com a programação nas disciplinas introdutórias de algoritmos, no ponto de vista acadêmico.

- Objeto de estudo: TDD;
- Propósito: Avaliar o TDD no ensino em programação;
- Foco: Melhoria na aprendizagem em programação;
- Interessados: Acadêmicos e professores;
- Contexto: Na acadêmia.

Métodos como TDD são ensinados, mas não são necessariamente avaliados. Consequentemente, é vital avaliar se os alunos estão aderindo ao método corretamente, bem como entender suas atitudes que provavelmente informarão seu comportamento (BUFFARDI; EDWARDS, 2012). Como forma de verificação da eficácia da abordagem proposta, foi realizada a comparação entre a abordagem tradicional de ensino em programação e a abordagem com uso do método proposto neste trabalho.

Para comprovar a validade do método proposto neste trabalho, foi medido:

- A evolução do porcentual de cobertura de casos de testes dos alunos do grupo de intervenção, foram três conjuntos de casos de teste para cada exercício.
- A diferença entre a terceira versão do código do grupo de intervenção com os códigos do grupo de controle, que fizeram apenas uma tentativa sem os casos de teste.

Para fazer a verificação da cobertura dos casos de testes dos alunos, de ambos os grupos, foi utilizado a ferramenta *Check*, citada na Seção 3.2.1.

4.2 CONFIGURAÇÃO DO ESTUDO

A amostra escolhida pelo estudo são os alunos ingressantes de cursos de computação (CS1) do Centro Universitário Ingá. Do total da amostra de alunos, independentemente do nível de conhecimento de cada estudante, foram divididos em dois grupos aleatoriamente, sendo:

- Grupo A (10 Alunos): Resolveram os problemas utilizando o *test-first*;
- Grupo B (9 Alunos): Resolveram os problemas utilizando o método convencional que está sendo aplicado no percurso da disciplina, sem o *test-first*.

Os alunos do grupo A (grupo de intervenção) receberam um treinamento de aproximadamente vinte minutos sobre o método proposto, tempo suficiente para explicar o método e fazer um exemplo. Nesta capacitação, foi abordado somente o conteúdo necessário para o cumprimento das tarefas, a ideia de fazer o caso de teste antes de desenvolver o código e como fazê-lo. O foco desse treinamento é em como resolver os problemas computacionais aplicando técnicas de teste de software.

O treinamento teve uma apresentação rápida dos conceitos do TDD e um exercício prático com grau de dificuldade baixo, a soma de dois números. Não foi abordado profundamente os conceitos sobre Teste de Software e funções, foi

apresentado apenas os conceitos necessários para o entendimento da proposta do trabalho

Feito a capacitação, os estudantes receberam os exercícios com níveis de dificuldade equânime. Neles foram aplicados os conhecimentos adquiridos em teste de software no desenvolvimento de problemas computacionais previamente elaborados para validar a hipótese proposta deste trabalho.

Os alunos do grupo B receberam apenas as orientações para resolver os problemas da mesma maneira que vinham fazendo no decorrer da disciplina de programação. Foi orientado para postagem a resposta no formulário de respostas, conforme o Apêndice A.

A metodologia foi aplicada com os alunos e os resultados foram analisados e comparados. Da avaliação proposta, foi verificado quais as melhorias em questões de qualidade de software que foram implementadas no código desenvolvido, conforme será demonstrado na Seção 4.3

4.3 RESULTADOS

Como resultado da utilização de técnicas de teste de *software* nos primeiros passos na programação, o aluno é forçado a raciocinar de uma maneira mais eficaz antes de desenvolver o código e o produto final tende a ter menos falhas. Nesta seção, são apresentados os resultados da pesquisa assim como detalhes de como tudo ocorreu.

4.3.1 AMOSTRAGEM DE ALUNOS

A aplicação deste trabalho foi realizada com os alunos do quarto bimestre do primeiro ano do Curso de Análise e Desenvolvimento de Sistemas. Um total de dezenove alunos participaram da pesquisa, sendo que dez aplicaram o método que está sendo proposto e nove seguiram da mesma forma que a disciplina estava sendo conduzida, o ensino tradicional de programação, um ciclo entre teoria, exemplos e

exercícios.

Para compilação dos resultados, dos alunos que utilizaram os casos de testes para resolução dos exercícios, no primeiro problema, a divisão de números inteiros, todos as amostras foram aproveitadas. Entretanto, no segundo problema, o de Bhaskara, quatro amostras foram descartadas pois a cobertura de código não evoluiu ou zerou ao longo do estudo.

Tabela 11: Amostra de Alunos do Exercício da Divisão de Números Inteiros.

Elemento	Amostras	Amostras Válidas
Alunos que utilizaram Caso de Teste	10	10
Alunos que não utilizaram Caso de Teste	9	9
TOTAL	19	19

É possível observar na Tabela 11 que todas as respostas foram aproveitadas, já na Tabela 12 quatro amostras foram descartadas por não atenderem os critérios da pesquisa.

4.3.2 RESULTADOS

Para ambos os exercícios que os alunos fizeram, a divisão de números inteiros e o cálculo de Bhaskara, são apresentados três resultados sobre cada um deles. Primeiramente um gráfico com o desempenho individual de cada aluno sobre o percentual da cobertura de casos de testes. Depois, são apresentados dois *boxplots*, o primeiro comparando a evolução dos três conjuntos dos casos de testes e o segundo comparando o conjunto do resultado do terceiro casos de testes com o grupo que não utilizou o *test-first*. Primeiramente, será tratado o problema da divisão de números inteiros.

Tabela 12: Amostra de Alunos do Exercício de Bhaskara.

Elemento	Amostras	Amostras Válidas
Alunos que utilizaram Caso de Teste	10	6
Alunos que não utilizaram Caso de Teste	9	9
TOTAL	19	15

É possível notar na Fig. 8 que no problema da divisão de dois números inteiros houve uma melhoria na cobertura dos casos de testes entre o primeiro e o segundo conjunto de casos de testes, sendo que do segundo para o terceiro não houve mudanças. No primeiro conjunto de casos de testes, todos os dez alunos conseguiram 80% de cobertura, desse total, metade desses alunos conseguiram evoluir para 100% quando fizeram o segundo conjunto de casos de testes. Portanto, houve uma evolução.

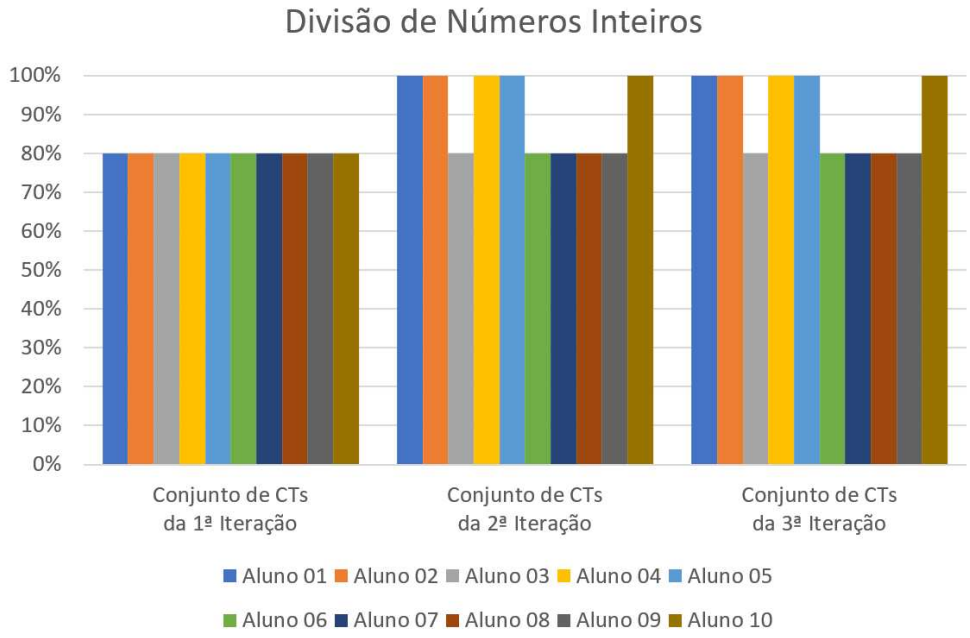

Figura 8: Porcentagem de cobertura dos casos de testes da Divisão de Números Inteiros
Fonte: Autoria própria.

A Fig. 9 demonstra a evolução dos três conjuntos de casos de testes (1, 2 e 3) do problema da divisão de inteiros. Os resultados do segundo e terceiro blocos são idênticos, ou seja, todos os alunos repetiram o que foi desenvolvido em ambas as etapas. A mediana da cobertura do primeiro conjunto de caso de testes ficou em 80%, enquanto do segundo e terceiro evoluiu para 90%.

A Fig. 9 demonstra que houve uma evolução entre o primeiro conjunto de casos de testes comparados com o segundo e terceiro conjunto. Entretanto, isso não acontece

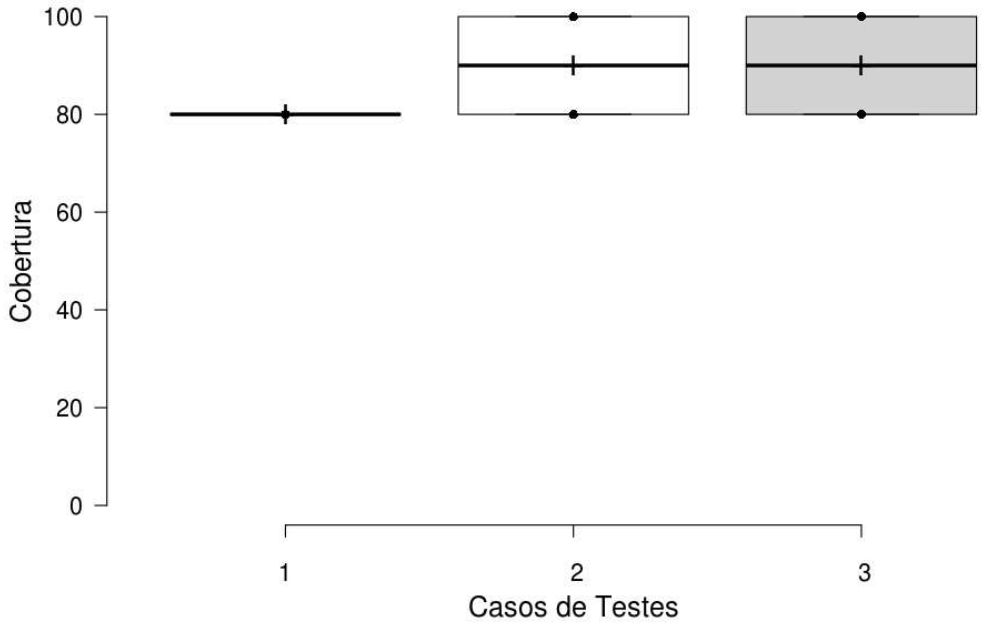

Figura 9: Bloxplot da Divisão de Números Inteiros: Versões 1, 2 e 3 dos Casos de Testes
Fonte: Autoria própria

quando comparamos do segundo para o terceiro conjunto de casos de testes, os números se mantiveram idênticos.

Como forma de confirmar a evolução dos alunos nos casos de testes, a Fig. 10 traz a comparação entre o grupo de intervenção na etapa terceira etapa dos casos de testes, contra o grupo de controle, que não utilizou o *test-first*.

Observa-se que na comparação entre o melhor resultado apresentado do grupo de alunos que usou o método que está sendo proposto neste trabalho e o grupo que não o utilizou, a mediana da cobertura apresentada é aproximadamente 90% para o grupo da intervenção contra 80% para o grupo de controle. Ou seja, há um crescimento na cobertura dos casos de testes entre ambos os grupos.

Da mesma forma que foram ilustradas as evoluções dos alunos com o primeiro problema que foi proposto para eles resolverem, a divisão de números inteiros, a seguir

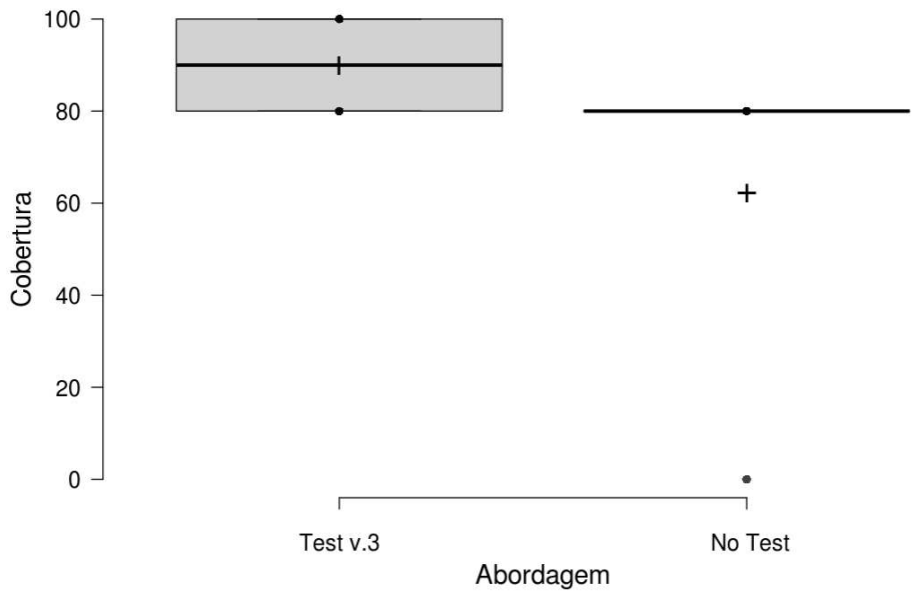

Figura 10: Bloxplot da Divisão de Números Inteiros: Grupo de intervenção (abordagem) na etapa 3 vs Grupo de controle (sem abordagem)

Fonte: Autoria própria.

os resultados referente ao segundo problema, o de Bhaskara.

Seguindo na mesma linha de raciocínio, para apresentar os resultados coletados do segundo exemplo, primeiramente um gráfico ilustrando o percentual de cobertura dos casos de testes de testes de todos os alunos, depois um *boxplots* comparando os resultados da evolução dos três conjuntos de casos de testes e, por fim, um *boxplots* comparando o resultado da terceira etapa dos casos de testes com o grupo que não utilizou o *test-first*.

Vale lembrar que o número de amostra deste problema foi menor comparada ao exercício anterior, pois alguns alunos não evoluíram durante os casos de testes ou simplesmente ignoraram o que era para ser feito.

É possível verificar na Fig. 11 que existe uma evolução na porcentagem de

cobertura dos casos de testes em relação aos casos de testes 1, 2 e 3. Todos os alunos conseguiram evoluir conforme iam fazendo os novos casos de testes. Os alunos 02 e 05 tiveram o mesmo desempenho, por isso são representados por apenas um traço. Sobre todos, é possível concluir que conforme os casos de testes vão passando de versões, a porcentagem da cobertura de dados de testes aumenta.

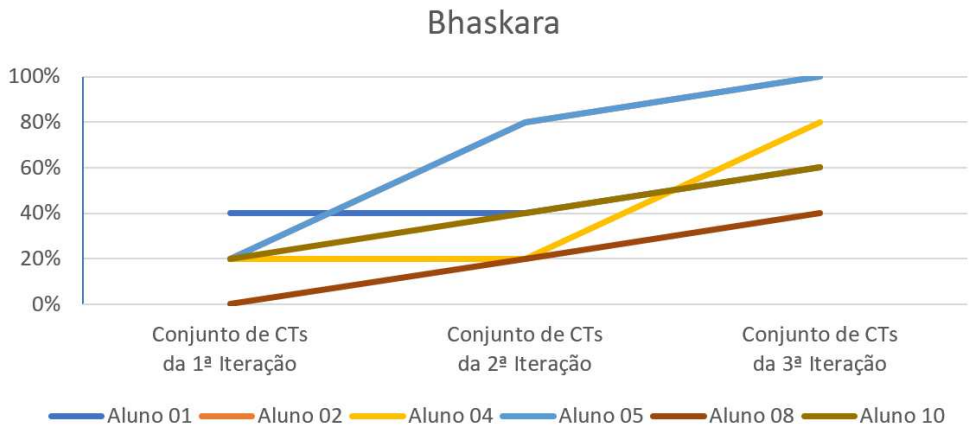

Figura 11: Porcentagem de cobertura dos casos de testes de Bhaskara

Fonte: Autoria própria

Na Fig. 12, a cobertura dos casos de testes aumentam conforme vão passando pelos conjuntos de casos de testes 1, 2, e 3. O conjunto de casos de testes 1 teve a cobertura em 20%, já o segundo conjunto, 40%, o terceiro e último chegou a 75% de cobertura dos casos de testes, ou seja, houve uma evolução na cobertura de casos de testes conforme os alunos iam amadurecendo seus próprios casos de testes.

Da mesma forma que foi realizado o problema anterior, como forma de confirmar a evolução dos alunos nos casos de testes, agora no problema de Bhaskara, a Fig. 13 traz a comparação entre o grupo de intervenção (abordagem) na etapa terceira etapa dos casos de testes, contra o grupo de controle (sem abordagem), que não utilizou o *test first*.

Fica nítido na Fig. 13 que o percentual da cobertura dos casos de teste da última

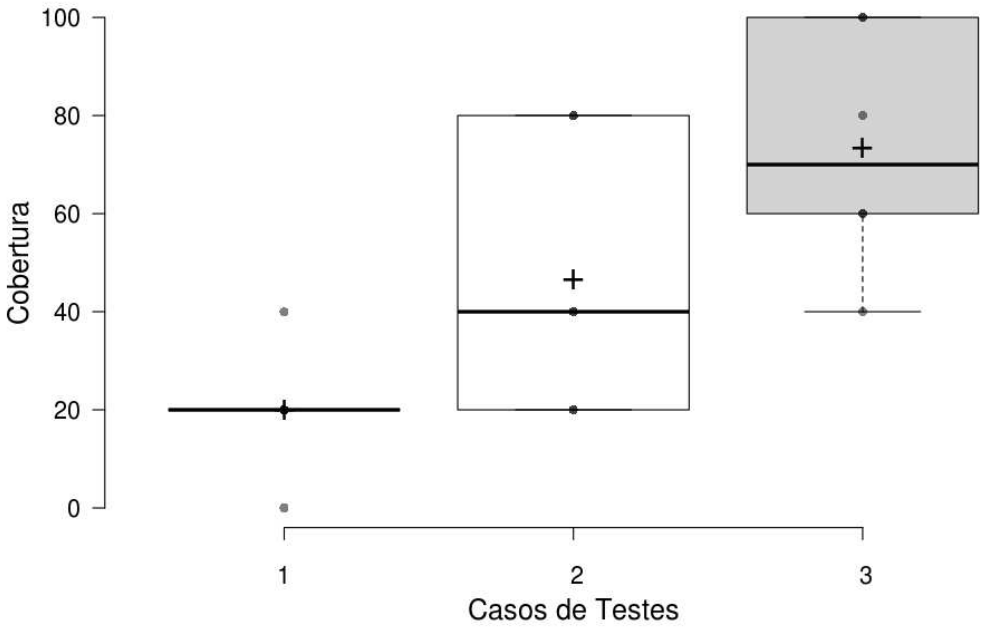

Figura 12: Bloxplot de Bhaskara: Versões 1, 2 e 3 dos Casos de Testes
Fonte: Autoria própria.

versão do grupo de alunos que utilizaram o *test first* é maior comparada aos alunos que fizeram as atividades da mesma forma que estavam fazendo, conforme foi ensinado durante as aulas.

4.3.3 FEEDBACK

Para validar o método proposto neste trabalho com os alunos, foi realizado um levantamento com os participantes que utilizaram casos de testes no desenvolvimento dos problemas. As seguintes questões foram levantadas:

- A criação de casos de teste com a abordagem proposta facilitou o desenvolvimento do programa/resolução do problema?
- O que você achou das progressões da criação dos casos de teste?

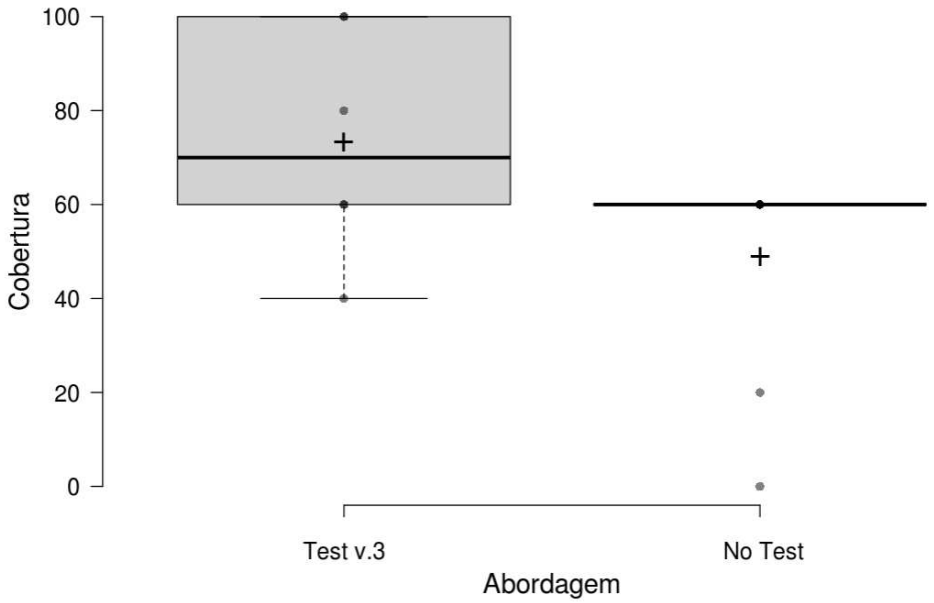

Figura 13: Bloxplot de Bhaskara: Grupo de intervenção (abordagem) na etapa 3 vs Grupo de controle (sem abordagem)

Fonte: Autoria própria.

- Você utilizaria casos de teste para o desenvolvimento de programas no futuro?
- Quais dificuldades você encontrou ao criar os casos de teste?
- O que você faria de diferente em relação a essa abordagem?

Dos dez estudantes que receberam o questionário, apenas duas alunas responderam à pesquisa. Com relação à primeira pergunta, todas concordaram que a abordagem facilitou a resolução do problema. Um aspecto destacado foi a definição prévia à programação dos casos de teste, que facilitou a programação: “Sim, facilitou já saber o que o programa precisava fazer”. Outro ponto destacado pelos respondentes foi quanto a pensar nos erros e como isso contribui para desenvolver melhores programas: “Assim, erros que poderiam ter, foram evitados” e “eu quanto aluna penso no óbvio do erro, não temos a ideia do que pode dar errado pensar fora da caixa”.

Na segunda pergunta, referente as progressões nos casos de testes, as respostas foram positivas. Uma aluna relatou que achou a abordagem interessante e que não tinha pensado nessa estratégia antes. Outro relato favorável foi que, resolvendo os problemas desta maneira, é possível pensar em várias soluções diferentes.

A respeito da terceira pergunta, ambas as alunas disseram que utilizariam casos de teste para o desenvolvimento de programas no futuro. Sobre as dificuldades encontradas ao criar casos de testes, quarta pergunta, uma aluna relatou que não teve dificuldade, enquanto a outra mencionou: "A maior dificuldade foi pensar no que o usuário poderia fazer, de diferente do que eu pensei que era pra ser feito, exemplo declarar uma variável de número inteiro e o usuário escrever um número negativo ou com vírgula".

Quanto à quinta e última pergunta, em que a questão era sobre mudanças e melhorias no método recém aplicado, não houveram sugestões. Uma aluna relatou que a experiência da aplicação desta proposta foi muito boa. De forma geral, o objetivo da coleta deste *feedback* é de validar a praticidade e a efetividade da técnica que foi abordada, o uso de teste de software no desenvolvimento de algoritmos no ensino introdutório de programação,

4.3.4 ANALISE ESTATÍSTICA

Para confrontar os resultados produzidos pelos dois grupos, foi utilizado o teste estatístico *Mann-Whitney* para validar a hipótese apresentada. Observa-se na Tabela 13 o "p-value" do cruzamento de dados entre as amostras de ambos os problemas resolvidos pelos alunos, a divisão de números inteiros e o cálculo de Bhaskara. Este valor é a probabilidade de se obter uma estatística de teste igual ou mais extrema que aquela observada em uma amostra, sob a hipótese nula.

No primeiro grupo, Grupo de intervenção (abordagem) na etapa 3 vs Grupo de controle (sem abordagem), no problema da divisão de números inteiros, exite um número relevante do "p-value", 0.0278. Já no Bhaskara, o resultado não é tão

expressivo, 0.14156.

Tabela 13: Resultados Estatísticos.

Grupos	**Divisão**	**Bhaskara**
Grupo de intervenção (abordagem) na etapa 3 vs Grupo de controle (sem abordagem)	0.0278	0.14156
Grupo de intervenção (abordagem) na etapa 1 vs Grupo de intervenção (abordagem) na etapa 2	0.06432	0.09296
Grupo de intervenção (abordagem) na etapa 2 vs Grupo de intervenção (abordagem) na etapa 3	0.9681	0.12852
Grupo de intervenção (abordagem) na etapa 1 vs Grupo de intervenção (abordagem) na etapa 3	0.06432	0.00652

Outra situação que traz um resultado interessante é o "p-value" da comparação entre o grupo de intervenção (abordagem) na etapa 1 vs o grupo de intervenção (abordagem) na etapa 3, levando em consideração amos os problemas, o da divisão apresenta o resultado 0.06432 e o de Bhaskara 0.00652. Levando em consideração que o valor limite é 0.05, ambos os resultados são positivos, mesmo o da divisão apresentando um resultado ligeiramente acima.

Através da análise estatística, é possível analisar o grau de conformidade do TDD aplicado no processo de ensino na programação.

Os dados foram analisados e com base nos resultados, existe a possibilidade de fazer a correlação do TDD com o ensino em programação. Esses mesmos dados foram confrontados para validar a hipótese apresentada. Todos os dados são baseado no percentual de cobertura dos casos de testes.

4.4 AMEAÇAS À VALIDADE

O trabalho limita-se ao ensino introdutório de programação, conceitos mais aprofundados de Engenharia de Software não serão abordados, assim como conceitos inicias de Estrutura de Dados, Programação Orientada a Objetos e outras disciplinas voltadas a programação que compõem a grade curricular dos cursos de Computação que geralmente ocorrem após o primeiro ano da graduação.

Outro delimitador do trabalho, devido seu foco ser no ensino introdutório de programação, está relacionado a quantidade da amostra, nesse caso, são os alunos do primeiro ano da graduação (ingressantes), podendo ser do primeiro ou segundo semestre, preferencialmente sem experiencia profissional com desenvolvimento de *software*.

Uma ameaça que pode comprometer o resultado da pesquisa é o plágio, podendo ser interno (entre os alunos) ou externo (pesquisando respostas prontas na *Internet*. Como forma de mitigar essa ameaça, o desenvolvimento dos exercícios pelos alunos foi monitorado pelo professor aplicador, caso haja alguma amostra suspeita, esta deve ser descartada para não influenciar no resultado final, não houve.

Devido a amostragem da pesquisa ser restrita aos acadêmicos ingressante, esse perfil (geralmente) tem pouca ou nenhuma experiência com pesquisa, o que por consequência pode acontecer de terem pouca maturidade para levar a pesquisa a sério.

Alunos que estejam mais avançados no curso, que possivelmente podem estar cursando as disciplinas introdutórias de programação em regime de dependência, foram excluídos. O propósito é que a amostra seja uniforme, embora os alunos possam ter níveis de aprendizagem diferente nessa etapa da graduação.

O nível de conhecimento e a facilidade com a disciplina de programação em diferentes escalas entre os acadêmicos pode influenciar no resultado da pesquisa, visto que alguns já podem ter experiências passadas com lógica de programação e desenvolvimento de software. Para minimizar essa possível discrepância, todos os alunos receberam questões de nível equânime, seguindo a ordem de dificuldade escalonada dividida em níveis.

Existe a possibilidade do(s) aluno(s) não ter entendido a lógica de toda a metodologia, desde o raciocínio e a reflexão sobre o problema até a elaboração dos casos de teste, envolvendo a forma de estruturar a lógica para resolução do problema, o que consequentemente levara o aluno a não resolver o exercício conforme proposto, fato que fará com que a amostra em questão seja descartada.

Uma investigação sobre as percepções dos alunos relacionados a prática de programação XP, encontrou uma correlação positiva entre a idade dos alunos e uma atitude em relação ao TDD. Foi considerado que isso poderia ser explicado por um maior nível de disciplina de estudantes mais maduros. Possivelmente, isso ocorre porque TDD não é sobre testes, mas sobre design. Os alunos mais velhos tendem a ser melhores em design, o que por sua vez os leva a ter uma melhor opinião sobre TDD comparados aos alunos iniciantes(MELNIK; MAURER, 2005). Portanto, a maturidade do aluno pode ser um fator de ameaça a validade deste trabalho, visto que o público alvo são estudantes novatos em programação que podem não ter a maturidade desejada para conduzir o experimento do projeto.

4.5 CONSIDERAÇÕES FINAIS

Considerando os resultados do mapeamento sistemático, o teste de software no ensino introdutório de programação é uma abordagem que já vem sendo estudada há bastante tempo, desde mil novecentos de oitenta.

O TDD ainda é pouco explorado no contexto de aprendizagem de programação, mas em seus estudos, apresenta evoluções positivas em cenários distintos. Algumas publicações trazem o *test-first* nesse contexto de ensino, o desenvolvimento de casos de teste contribui para a qualidade do código desenvolvido pelos alunos, mas o uso de critérios de teste melhorou a qualidade do conjunto de testes sem perturbação significativa no fluxo de trabalho do TDD.

O desenvolvimento de casos de teste, considerando critérios de teste de software antes de iniciar a codificação, pode auxiliar os alunos a desenvolver a lógica da resolução do problema computacional de uma forma mais eficaz, contribuindo para o seu melhor entendimento nas disciplinas introdutórias de programação.

5 CONCLUSÕES

Neste capítulo é apresentado a conclusão sobre a utilização teste de software no ensino introdutório de programação. Existem fortes indícios na literatura sobre os benefícios da utilização do teste de software no ensino de programação (FEITOSA et al., 2021a). Baseado nisto, o método proposto oferece possibilidades de contribuir positivamente com o ensino introdutório de programação, visto que esse assunto já é estudado desde a década de oitenta e desde então se mostra promissora.

Mediante as dificuldades de aprendizagem apresentadas, baseado nas informações do Mapeamento Sistemático, a proposta do TDD como ferramenta de ensino na programação tende a melhorar o nível de aprendizagem dos alunos, principalmente no desenvolvimento da lógica computacional.

Não é de agora que outros métodos de ensino em programação são estudados nas disciplinas introdutórias de programação, inclusive o próprio teste de software (HSIA; PETRY, 1980). Nesse contexto, o TDD vem contribuir com as pesquisas no ensino de programação.

Os dados compilados do mapeamento sistemático indicam que o teste de software criterioso pode contribuir no processo de aprendizagem e desenvolvendo de código, com melhorias em qualidade e quantidade de linhas de códigos (FEITOSA et al., 2021a).

A avaliação da abordagem demonstra os benefícios que o *test-first* pode trazer no ensino introdutório de programação, visto que, em sua maioria, os alunos que utilizaram a metodologia proposta neste trabalho, tiveram um resultado superior

comparado ao grupo que não a utilizou.

O uso do TDD pode ser uma alternativa inteligente como metodologia de ensino nas disciplinas de programação. Para isso, cabe ao professor se capacitar e treinar os alunos para sua adoção, um desafio de ambos os lados que trabalhado em equipe, pode melhorar todo o processo de ensino e aprendizagem das disciplinas introdutórias de programação (LEMOS et al., 2018).

O método proposto pode simplificar a didática do ensino das disciplinas introdutórias de programação, usando caso de teste, quebrar paradigmas e resultar em uma melhor percepção dos acadêmicos sobre o desenvolvimento de programas, principalmente, por ser uma das principais disciplinas do curso de Ciência da Computação e afins.

Outro fator importante que justifica as melhorias nessa etapa da graduação, é a absorção do conhecimento técnico, fator relevante que influência positivamente na hora de enfrentar o competitivo mercado de trabalho, nos mais diversos cargos que envolvem todo o processo de desenvolvimento de software.

Espera-se que esse trabalho seja um ponto de partida para novos métodos no ensino e aprendizagem nas disciplinas introdutórias de programação. A tecnologia se renova, o perfil dos alunos muda e o método de ensino também precisa evoluir, principalmente com o perfil autodidata dos alunos dos cursos de tecnologia.

Essa pesquisa foi aplicação na linguagem de programação C, mas não existe impedimento ou restrição para que o mesmo estudo seja aplicado em outras linguagens, em disciplinas mais avançadas dos cursos de graduação de áreas correlatas, independentemente de ser em uma plataforma *mobile* ou *web*. Essa proposta não está vinculada a uma linguagem especifica, mas sim em forçar o aluno a raciocinar melhor antes de desenvolver o código utilizando critérios de teste.

O TDD é uma das várias ferramentas utilizadas na indústria, utilizada para testar sistema ou partes deles já produzidos, que podem ser utilizadas na academia como forma de melhoria em todo o processo de aprendizagem e desenvolvimento dos

primeiros algoritmos.

Como melhoria, é sugerido o desenvolvimento de um software didático que combina vários fatores que foram levantados como positivo no ensino de aprendizagem, como exemplo, uma ferramenta que podem contemplar teste de software, *feedback* e uma interface gráfica intuitiva.

Ao considerar os critérios de teste não para a avaliação, mas para auxiliar a construção de programas, a presente proposta de TDD com critérios de teste como ferramenta de ensino na programação busca induzir a melhora na aprendizagem dos alunos, principalmente no desenvolvimento da lógica computacional. Portanto, espera-se que os alunos pensem mais antes de resolver um problema e que consequentemente a solução desse problema seja feita da melhor forma possível.

O TDD demonstra aumentar a produtividade das equipes de programação e a melhoria da qualidade do código que eles produzem. No entanto, a maior parte dos currículos introdutórios não fornecem nenhuma introdução ao design de teste, nenhum suporte para definir os testes e não insistem em uma abrangente de cobertura de teste, que é a força motriz do TDD (PROULX, 2009).

Várias ferramentas podem auxiliar a aprendizagem de programação desde passos a serem seguidos, ferramentas com *feedback* de forma gráfica (REIS et al., 2019) e até mesmo o próprio teste de software. O TDD, aplicado de forma correta, uma associação entre o ensino de lógica, linguagem e técnicas de teste de software, tende a propiciar uma melhor aprendizagem em programação para os alunos dos cursos introdutórios de computação a áreas afins.

Em ambos os exercícios aplicados nos alunos, houve evolução conforme eles iam desenvolvendo novos conjuntos de casos de testes, pensando em aumentar a cobertura do programa e em diminuir as possíveis falhas. A aplicação de critérios de teste no ensino introdutório de programação pode influenciar positivamente em uma melhor desempenho do aluno no desenvolvimento dos primeiros programas no início de sua graduação.

REFERÊNCIAS

AGRAWAL, H.; DEMILLO, R.; HATHAWAY, R.; HSU, W.; HSU, W.; KRAUSER, E. W.; MARTIN, R. J.; MATHUR, A. P. **Design of mutant operators for the C programming language**. [S.l.], 1989.

AHO, A. V. Ubiquity symposium: Computation and computational thinking. **Ubiquity**, ACM, New York, NY, EUA, v. 2011, jan. 2011. ISSN 1530-2180.

ANICHE, M. F.; GEROSA, M. A. How the practice of tdd influences class design in object-oriented systems: Patterns of unit tests feedback. In: **2012 26th Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2012. p. 1–10.

BECK, K. **Test-Driven Development: By Example**. 1. ed. EUA: Addison-Wesley Professional, 2002. 240 p. ISBN 978-0321146533.

BELL, J.; LINN, M.; CLANCY, M. Knowledge Integration in Introductory Programming: CodeProbe and Interactive Case Studies. **Interactive Learning Environments**, v. 4, n. 1, p. 75–95, 1994. ISSN 10494820.

BERGIN, S.; REILLY, R.; TRAYNOR, D. Examining the role of self-regulated learning on introductory programming performance. In: **1st International Workshop on Computing Education Research**. [S.l.]: ACM, 2005. p. 81–86. ISBN 1595930434.

BINDER, R. **Testing object-oriented systems: models, patterns, and tools**. [S.l.]: Addison-Wesley Professional, 2000.

BRENNAN, K.; RESNICK, M. New frameworks for studying and assessing the development of computational thinking. In: **Annual Meetting 2012 – American Educational Research Association**. [S.l.: s.n.], 2012. p. 1–25.

BRUN, Y. Improving impact of self-adaptation and self-management research through evaluation methodology. In: **Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems**. Cape Town, South Africa: Association for Computing Machinery, 2010. (SEAMS '10), p. 1–9. ISBN 978-1-60558-971-8.

BUFFARDI, K.; EDWARDS, S. H. Exploring influences on student adherence to test-driven development. In: **Proceedings of the 17th ACM annual conference on**

Innovation and technology in computer science education. Haifa, Israel: Association for Computing Machinery, 2012. (ITiCSE '12), p. 105–110. ISBN 978-1-4503-1246-2.

BUFFARDI, K.; EDWARDS, S. H. Effective and ineffective software testing behaviors by novice programmers. In: **Proceedings of the ninth annual international ACM conference on International computing education research**. San Diego, San California, USA: Association for Computing Machinery, 2013. (ICER '13), p. 83–90. ISBN 978-1-4503-2243-0.

BUFFARDI, K.; EDWARDS, S. H. A formative study of influences on student testing behaviors. In: **Proceedings of the 45th ACM technical symposium on Computer science education**. Atlanta, Georgia, USA: Association for Computing Machinery, 2014. (SIGCSE '14), p. 597–602. ISBN 978-1-4503-2605-6.

CAMARA, B. H. P.; SILVA, M. A. G. A Strategy to Combine Test-Driven Development and Test Criteria to Improve Learning of Programming Skills. In: **Proceedings of the 47th ACM Technical Symposium on Computing Science Education**. Memphis, Tennessee, USA: Association for Computing Machinery, 2016. (SIGCSE '16), p. 443–448. ISBN 978-1-4503-3685-7.

CASPERSEN, M.; KöLLING, M. A novice's process of object-oriented programming. In: **Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA**. Portland, OR: [s.n.], 2006. v. 2006, p. 892–900. ISBN 1-59593-491-X 978-1-59593-491-8.

CASPERSEN, M. E.; KOLLING, M. Stream: A first programming process. **ACM Trans. Comput. Educ.**, Association for Computing Machinery, New York, NY, USA, v. 9, n. 1, mar. 2009.

DELAMARO, M. **Introdução ao Teste de Software**. [S.l.]: ELSEVIER, 2016. ISBN 9788535283525.

DELAMARO, M.; JINO, M.; MALDONADO, J. **Introdução ao teste de software**. [S.l.]: Elsevier Brasil, 2013.

DENNY, P.; PRATHER, J.; BECKER, B. A.; ALBRECHT, Z.; LOKSA, D.; PETTIT, R. A Closer Look at Metacognitive Scaffolding: Solving Test Cases Before Programming. In: **Proceedings of the 19th Koli Calling International Conference on Computing Education Research**. Koli, Finland: Association for Computing Machinery, 2019. (Koli Calling '19), p. 1–10. ISBN 978-1-4503-7715-7.

DESAI, C.; JANZEN, D.; SAVAGE, K. A survey of evidence for test-driven development in academia. **ACM SIGCSE Bulletin**, v. 40, n. 2, p. 97–101, jun. 2008. ISSN 0097-8418.

DUNCKER, K. On problem-solving. **Psychological Monographs**, The American Psychological Association (APA), Washington, DC, EUA, v. 58, n. 5, p. i–113, 1945. ISSN 1896-1966.

EDUARDS, S. H.; SHAMS, Z. Do student programmers all tend to write the same software tests? In: **Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: Association for Computing Machinery, 2014. (ITiCSE '14), p. 171–176. ISBN 9781450328333.

EDWARDS, S. H. Using software testing to move students from trial-and-error to reflection-in-action. In: **35th SIGCSE Technical Symposium on Computer Science Education**. New York, NY, EUA: ACM, 2004. p. 26–30. ISBN 1-58113-798-2.

EDWARDS, S. H.; BöRSTLER, J.; CASSEL, L. N.; HALL, M. S.; HOLLINGSWORTH, J. Developing a common format for sharing programming assignments. **ACM SIGCSE Bulletin**, v. 40, n. 4, p. 167–182, nov. 2008. ISSN 0097-8418.

EDWARDS, S. H.; SHAMS, Z.; ESTEP, C. Adaptively identifying non-terminating code when testing student programs. In: **Proceedings of the 45th ACM technical symposium on Computer science education - SIGCSE '14**. Atlanta, Georgia, USA: ACM Press, 2014. p. 15–20. ISBN 978-1-4503-2605-6.

ETELäPELTO, A. Metacognition and the expertise of computer program comprehension. **Scandinavian Journal of Educational Research**, Routledge, v. 37, n. 3, p. 243–254, 1993. ISSN 0031-3831.

FEITOSA, Y. R. G.; SILVA, M. A. G.; FABRI, J. A. Desenvolvimento baseado em testes com aplicação de critérios de teste no ensino introdutório de computação. In: **Anais Estendidos do Simpósio Brasileiro de Educação em Computação**. Porto Alegre, RS, Brasil: SBC, 2021. p. 18–19. ISSN 0000-0000.

FEITOSA, Y. R. G.; SILVA, M. A. G.; FABRI, J. A. Mapeamento sistemático sobre resolução de problemas em disciplina introdutória de programação com teste de software. In: **Anais do XXIX Workshop sobre Educação em Computação**. Porto Alegre, RS, Brasil: SBC, 2021. p. 358–367. ISSN 2595-6175.

FLAVELL, J. H. Metacognitive aspects of problem solving. In: RESNICK, L. B. (Ed.). 1. ed. New York, NY, EUA: Lawrence Erlbaum, 1976. p. 364. ISBN 978-0470013847.

FORBELLONE, A. L. V.; EBERSPäCHER, H. F. **Lógica de programação: a construção de algoritmos e estruturas de dados**. 3. ed. São Paulo, SP, Brasil: Pearson, 2005. 218 p. ISBN 978-85-7605-024-7.

GASPAR, A.; LANGEVIN, S. Restoring ”coding with intention”in introductory programming courses. In: **Proceedings of the 8th ACM SIGITE conference on Information technology education**. Destin, Florida, USA: Association for Computing Machinery, 2007. (SIGITE '07), p. 91–98. ISBN 978-1-59593-920-3.

GROVER, S.; PEA, R. Computational thinking in K-12: A review of the state of the field. **Educational Researcher**, SAGE, v. 42, n. 1, p. 38–43, jan. 2013. ISSN 0013-189X.

HEINONEN, K.; HIRVIKOSKI, K.; LUUKKAINEN, M.; VIHAVAINEN, A. Learning agile software engineering practices using coding dojo. In: **Proceedings of the 13th annual ACM SIGITE conference on Information technology education - SIGITE '13**. Orlando, Florida, USA: ACM Press, 2013. p. 97. ISBN 978-1-4503-2239-3.

HILTON, A. D.; LIPP, G. M.; RODGER, S. H. A technique for translation from problem to code. In: **Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education**. Larnaca, Cyprus: Association for Computing Machinery, 2018. (ITiCSE 2018), p. 375. ISBN 978-1-4503-5707-4.

HILTON, A. D.; LIPP, G. M.; RODGER, S. H. Translation from Problem to Code in Seven Steps. In: **Proceedings of the ACM Conference on Global Computing Education**. Chengdu,Sichuan, China: Association for Computing Machinery, 2019. (CompEd '19), p. 78–84. ISBN 978-1-4503-6259-7.

HSIA, P.; PETRY, F. A Framework for Discipline in Programming. **IEEE Transactions on Software Engineering**, SE-6, n. 2, p. 226–232, 1980. ISSN 00985589.

HU, M.; WINIKOFF, M.; CRANEFIELD, S. Teaching novice programming using goals and plans in a visual notation. In: **Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123**. [S.l.: s.n.], 2012. p. 43–52.

IEEE. **IEEE Standard Glossary of Software Engineering Terminology**. 1990. 1-84 p.

IEEE. **IEEE Standard for Software Quality Assurance Processes**. 2014. 1-138 p.

ISO; IEC; IEEE. **ISO/IEC/IEEE 24765:2017 – Systems and software engineering – Vocabulary**. 2017. Padrão Internacional.

JANZEN, D.; SAIEDIAN, H. Test-driven development concepts, taxonomy, and future direction. **Computer**, v. 38, n. 9, p. 43–50, 2005.

JANZEN, D. S.; CLEMENTS, J.; HILTON, M. An evaluation of interactive test-driven labs with WebIDE in CS0. In: **35th International Conference on Software Engineering**. Piscataway, NJ, EUA: IEEE, 2013. p. 1090–1098. ISBN 978-1-4673-3076-3.

KEEFE, K.; SHEARD, J.; DICK, M. Adopting XP practices for teaching object oriented programming. In: **Proceedings of the 8th Australasian Conference on Computing Education - Volume 52**. Hobart, Australia: Australian Computer Society, Inc., 2006. (ACE '06), p. 91–100. ISBN 978-1-920682-34-7.

KOLLANUS, S.; ISOMöTTöNEN, V. Understanding TDD in academic environment: experiences from two experiments. In: **Proceedings of the 8th International Conference on Computing Education Research**. Koli, Finland: Association for Computing Machinery, 2008. (Koli '08), p. 25–31. ISBN 978-1-60558-385-3.

KOULOURI, T.; LAURIA, S.; MACREDIE, R. D. Teaching introductory programming: A quantitative evaluation of different approaches. **Transactions on Computing Education**, ACM, New York, NY, EUA, v. 14, n. 4, p. 26:1–26:28, fev. 2015. ISSN 1946-6226.

LAPPALAINEN, V.; ITKONEN, J.; ISOMöTTöNEN, V.; KOLLANUS, S. ComTest: a tool to impart TDD and unit testing to introductory level programming. In: **Proceedings of the fifteenth annual conference on Innovation and technology in computer science education**. Bilkent, Ankara, Turkey: Association for Computing Machinery, 2010. (ITiCSE '10), p. 63–67. ISBN 978-1-60558-820-9.

LEMOS, O. A. L.; FERRARI, F. C.; SILVEIRA, F. F.; GARCIA, A. Development of auxiliary functions: should you be agile? an empirical assessment of pair programming and test-first programming. In: **Proceedings of the 34th International Conference on Software Engineering**. Zurich, Switzerland: IEEE Press, 2012. (ICSE '12), p. 529–539. ISBN 978-1-4673-1067-3.

LEMOS, O. L.; SILVEIRA, F. F.; FERRARI, F. C.; GARCIA, A. The impact of Software Testing education on code reliability: An empirical assessment. **Journal of Systems and Software**, v. 137, p. 497–511, 2018. ISSN 01641212. Publisher: Elsevier Inc.

LOKSA, D.; KO, A. J.; JERNIGAN, W.; OLESON, A.; MENDEZ, C. J.; BURNETT, M. M. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In: **Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems**. San Jose, California, USA: Association for Computing Machinery, 2016. (CHI '16), p. 1449–1461. ISBN 978-1-4503-3362-7.

LUXTON-REILLY, A.; Simon; ALBLUWI, I.; BECKER, B. A.; GIANNAKOS, M.; KUMAR, A. N.; OTT, L.; PATERSON, J.; SCOTT, M. J.; SHEARD, J.; SZABO, C. Introductory programming: a systematic literature review. In: **Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer**

Science Education. Larnaca, Cyprus: Association for Computing Machinery, 2018. (ITiCSE 2018 Companion), p. 55–106. ISBN 978-1-4503-6223-8.

MARRERO, W.; SETTLE, A. Testing first: emphasizing testing in early programming courses. In: **Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education**. [S.l.: s.n.], 2005. p. 4–8.

MCCRACKEN, M.; ALMSTRUM, V.; DIAZ, D.; GUZDIAL, M.; HAGAN, D.; KOLIKANT, Y. B.-D.; LAXER, C.; THOMAS, L.; UTTING, I.; WILUSZ, T. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. **SIGCSE Bulletin**, ACM, New York, NY, EUA, v. 33, n. 4, p. 125–180, dez. 2001. ISSN 0097-8418.

MCKINNEY, D.; DENTON, L. F. Developing collaborative skills early in the CS curriculum in a laboratory environment. In: **Proceedings of the 37th SIGCSE technical symposium on Computer science education - SIGCSE '06**. Houston, Texas, USA: ACM Press, 2006. p. 138. ISBN 978-1-59593-259-4.

MELNIK, G.; MAURER, F. A cross-program investigation of students' perceptions of agile methods. In: IEEE. **Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.** [S.l.], 2005. p. 481–488.

MENDONÇA, A.; OLIVEIRA, C. de; GUERRERO, D.; COSTA, E. Difficulties in solving ill-defined problems: A case study with introductory computer programming students. In: IEEE. **2009 39th IEEE Frontiers in Education Conference**. [S.l.], 2009. p. 1–6.

MENDONçA, A.; GUERRERO, D.; COSTA, E. An approach for problem specification and its application in an Introductory Programming Course. In: **2009 39th IEEE Frontiers in Education Conference**. [S.l.: s.n.], 2009. p. 1–6. ISSN: 2377-634X.

METCALFE, J.; SHIMAMURA, A. P. **Metacognition: Knowing about Knowing**. 1. ed. [S.l.]: A Bradford Book, 1996. 350 p. ISBN 978-0262631693.

MOURãO, A. B. Uma proposta da eficiência do uso da metodologia ativa baseada em problemas, utilizando dojo de programação, aplicada na disciplina de lógica de programação. XXIII Workshop de Informática na Escola (WIE 2017), n. 10, p. 667–676, 2017.

MÜLLER, M.; HAGNER, O. Experiment about test-first programming. **IEE Proc. Softw.**, v. 149, p. 131–136, 2002.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3rd. ed. [S.l.]: Wiley Publishing, 2011. ISBN 1118031962.

MüLLER, M. M.; TICHY, W. F. Case study: extreme programming in a university environment. In: **Proceedings of the 23rd International Conference on Software Engineering**. Toronto, Ontario, Canada: IEEE Computer Society, 2001. (ICSE '01), p. 537–544. ISBN 978-0-7695-1050-7.

NETO, V. L.; COELHO, R.; LEITE, L.; GUERRERO, D. S.; MENDONçA, A. P. POPT: a problem-oriented programming and testing approach for novice students. In: **Proceedings of the 2013 International Conference on Software Engineering**. San Francisco, CA, USA: IEEE Press, 2013. (ICSE '13), p. 1099–1108. ISBN 978-1-4673-3076-3.

PEARS, A.; SEIDMAN, S.; MALMI, L.; MANNILA, L.; ADAMS, E.; BENNEDSEN, J.; DEVLIN, M.; PATERSON, J. A survey of literature on the teaching of introductory programming. In: **Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR 2007)**. New York, NY, EUA: ACM, 2007. p. 204–223.

PINTRICH, P. R.; GROOT, E. V. de. Motivational and self-regulated learning components of classroom academic performance. **Journal of Educational Psychology,**, American Psychological Association (APA), EUA, v. 82, n. 1, p. 33–40, mar. 1990. ISSN 0022-0663.

PRESSMAN, R. S. **Software engineering: a practitioner's approach**. [S.l.]: Palgrave macmillan, 2005.

PROULX, V. K. Test-driven design for introductory OO programming. In: **Proceedings of the 40th ACM technical symposium on Computer science education**. Chattanooga, TN, USA: Association for Computing Machinery, 2009. (SIGCSE '09), p. 138–142. ISBN 978-1-60558-183-5.

Python Software Foundation. **doctest – Test interactive Python examples**. 1999. Software.

RAHMAN, S. M.; JUELL, P. L. Applying software development lifecycles in teaching introductory programming courses. In: IEEE. **19th Conference on Software Engineering Education & Training (CSEET'06)**. [S.l.], 2006. p. 17–24.

REIS, R.; SOARES, G.; MONGIOVI, M.; ANDRADE, W. Evaluating Feedback Tools in Introductory Programming Classes. In: **Proceedings - Frontiers in Education Conference, FIE**. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2019. v. 2019-October. ISBN 978-1-72811-746-1. ISSN: 15394565.

ROCHA, A. da; MALDONADO, J.; WEBER, K. **Qualidade de software: teoria e prática**. [S.l.]: Prentice Hall, 2001. ISBN 9788587918543.

SALEHI, S. **Improving problem-solving through reflection**. Tese (phdthesis) — Stanford University, Stanford, CA, EUA, ago. 2018.

SALEHI, S.; WANG, K. D.; TOORAWA, R.; WIEMAN, C. Can majoring in computer science improve general problem-solving skills? In: **Proceedings of the 51st ACM Technical Symposium on Computer Science Education**. New York, NY, EUA: ACM, 2020. p. 156–161. ISBN 9781450367936.

SCATALON, L. P.; CARVER, J. C.; GARCIA, R. E.; BARBOSA, E. F. Software testing in introductory programming courses: A systematic mapping study. In: **Proceedings of the 50th ACM Technical Symposium on Computer Science Education**. New York, NY, EUA: ACM, 2019. p. 421–427. ISBN 978-1-4503-5890-3.

SHAW, M. Software engineering education: a roadmap. In: **Proceedings of the Conference on The Future of Software Engineering**. Limerick, Ireland: Association for Computing Machinery, 2000. (ICSE '00), p. 371–380. ISBN 978-1-58113-253-3.

SHETH, S.; MURPHY, C.; ROSS, K. A.; SHASHA, D. A Course on Programming and Problem Solving. In: **Proceedings of the 47th ACM Technical Symposium on Computing Science Education - SIGCSE '16**. Memphis, Tennessee, USA: ACM Press, 2016. p. 323–328. ISBN 978-1-4503-3685-7.

SHI, J.; SHAH, A.; HEDMAN, G.; O'ROURKE, E. Pyrus: Designing A Collaborative Programming Game to Promote Problem Solving Behaviors. In: **Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems**. Glasgow, Scotland Uk: Association for Computing Machinery, 2019. (CHI '19), p. 1–12. ISBN 978-1-4503-5970-2.

SOUZA, D. D.; KöLLING, M.; BARBOSA, E. Most common fixes students use to improve the correctness of their programs. In: **Proceedings - Frontiers in Education Conference, FIE**. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2017. v. 2017-October, p. 1–9. ISBN 978-1-5090-5919-5. ISSN: 15394565.

SOUZA, D. M.; BATISTA, M. H. da S.; BARBOSA, E. F. Problemas e dificuldades no ensino de programaçao: Um mapeamento sistemático. **Revista Brasileira de Informática na Educação**, v. 24, n. 1, p. 39, 2016.

SOUZA, D. M. de; ISOTANI, S.; BARBOSA, E. F. Teaching novice programmers using progtest. **Int. J. Knowl. Learn.**, v. 10, n. 1, p. 60–77, 2015.

SPACCO, J.; HOVEMEYER, D.; PUGH, W.; EMAD, F.; HOLLINGSWORTH, J. K.; PADUA-PEREZ, N. Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses. In: **Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer**

Science Education. New York, NY, USA: Association for Computing Machinery, 2006. (ITICSE '06), p. 13–17. ISBN 1595930558.

UTTING, I.; TEW, A. E.; MCCRACKEN, M.; THOMAS, L.; BOUVIER, D.; FRYE, R.; PATERSON, J.; CASPERSEN, M.; KOLIKANT, Y. B.-D.; SORVA, J.; WILUSZ, T. A fresh look at novice programmers' performance and their teachers' expectations. In: **18th Annual Conference on Innovation and Technology in Computer Science Education**. New York, NY, EUA: ACM, 2013. p. 15–32. ISBN 978-1-4503-2665-0.

WELLINGTON, C. A.; BRIGGS, T. H.; GIRARD, C. D. Experiences using automated 4ests and 4est driven development in computer 9cience i. In: IEEE. **Agile 2007 (AGILE 2007)**. [S.l.], 2007. p. 106–112.

WICK, M.; STEVENSON, D.; WAGNER, P. Using testing and junit across the curriculum. **SIGCSE Bull.**, Association for Computing Machinery, New York, NY, USA, v. 37, n. 1, p. 236–240, fev. 2005. ISSN 0097-8418.

WING, J. M. Computational thinking. **Communications of the ACM**, ACM, New York, NY, EUA, v. 49, n. 3, p. 33–35, mar. 2006. ISSN 0001-0782.

ZIMMERMAN, B. J. Becoming a self-regulated learner: Which are the key subprocesses? **Contemporary Educational Psychology**, Elsevier, v. 11, n. 4, p. 307–313, out. 1986. ISSN 0361-476X.

APÊNDICE A – FORMULÁRIO DE RESPOSTAS

Estudante: _____

Instituição de Ensino:_____ Data: _____/_____/_____

Um caso de teste consiste em um conjunto de dados de entrada e de resultados esperados. Por exemplo, para o problema da soma de dois números quaisquer (representados pelas variáveis ‘a’ e ‘b’), os dados de entrada consistem de dois números e o resultado de saída é um número (obtido pela soma dos dois indicados como entrada). Veja possíveis casos de teste para esse exemplo:

Entradas		**Resultados esperados**
a = 1	b = 3	4
a = 0	b = 2	2
a = -1	b = 2	1

Ao desenvolver a solução para esse problema, os dados apresentados em cada caso de teste podem ser utilizados para verificar se o programa funciona corretamente. Às vezes nós cometemos enganos e erramos algo na implementação. Quando isso acontece, a implementação falha, apresentando um resultado diferente do esperado. Na coluna “Resultados obtidos” temos o resultado obtido ao executar nosso programa com as entradas de cada caso de teste. Observe que, nos exemplos abaixo, o resultado obtido para o último caso de teste foi diferente do resultado esperado. Isso indica que nosso programa falhou! Ao identificar uma falha, devemos analisar o código da implementação e tentar encontrar qual foi o erro que causou essa falha.

Entradas		**Resultados esperados**	**Resultados obtidos**
a = 1	b = 3	4	4
a = 0	b = 2	2	2
a = -1	b = 2	1	3

Esta atividade compreende a resolução de três exercícios. Para cada exercício, será apresentada a descrição de um problema a ser resolvido com um programa em C. Para guiar a resolução do problema, serão utilizados e criados casos de teste da seguinte forma:

1. Preencha as duas primeiras colunas da tabela “Casos de Testes 01” com as supostas entradas e resultados esperados para o programa em questão.
2. Desenvolva um programa em C para resolver o problema.
3. Preencha a terceira coluna da tabela “Casos de Testes 01” com os resultados obtidos pelo programa.
4. Compare os “Resultados Esperados” e os “Resultados Obtidos”, reflita se seu programa atenderá a todos os possíveis valores de entrada e o corrija se necessário. **Ao término desse passo, guarde uma cópia do seu programa para envio da forma como ele está!**
5. Elabore um conjuntos de diferentes entradas para o seu programa e preencha as duas primeiras colunas da tabela “Casos de Testes 02”
6. Melhore seu programa para atender os Casos de Testes da segunda tabela.
7. Execute o programa e compare os “Resultados Esperados” e os “Resultados Obtidos”, reflita se seu programa atenderá a todos os possíveis valores de entrada e o corrija se necessário. **Ao término desse passo, guarde uma cópia do seu programa para envio da forma como ele está!**
8. Pense em outros possíveis dados de entrada, diferentes dos anteriores e repita o procedimento com a tabela “Casos de Testes 03”. Seu algoritmo deverá atender aos mais diversos possíveis dados de entrada e, consequentemente, ter menos falhas. **Ao término desse passo, guarde uma cópia do seu programa para envio da forma como ele está!**

Exercício 1: Divisão de dois números inteiros.

Descrição: A divisão é uma das quatro operações básicas da matemática e é inversa à multiplicação. A divisão de um número inteiro consiste em seu fracionamento, na sua fragmentação, que pode ter como resultado um número inteiro.

Considere as instruções para resolução dos exercícios e resolva os itens a seguir.
Os dados de entrada consistem dos valores para o dividendo e o divisor.
O resultado esperado é o resultado da divisão.
Caso não seja possível fazer a divisão, retornar com a mensagem "Não é possível realizar a divisão".

Casos de Testes 01

Entradas		**Resultados Esperados**	**Resultados Obtidos**
dividendo = 10	divisor = 2		
dividendo = 100	divisor = 10		
dividendo = 16	divisor = 4		

Antes de montar o próximo conjunto de casos de teste, reflita:

- Se algum dos "Resultados Esperados" forem diferentes dos "Resultados Obtidos", o que precisa ser alterado no seu algoritmo para ele trazer o resultado correto?
- Os casos de testes realizados no primeiro conjunto abrangem todas as possíveis entradas?
- Quais outros valores de entrada você acha relevante para esse problema?

Por exemplo: E se na estrada dos dados, tivesse o valor do dividendo como zero? Fato que matematicamente impossibilita a divisão a ser realizada.

Casos de Testes 02

Entradas		**Resultados Esperados**	**Resultados Obtidos**
dividendo =	divisor =		
dividendo =	divisor =		
dividendo =	divisor =		

Para o último conjunto de casos de testes, o ideal é tentar chegar onde os primeiros não alcançaram, neste contexto da divisão por números inteiros, os casos de testes ainda podem explorar uma combinação de entradas em que o resultado seja um número real, o que impossibilita a atribuição da resposta em uma variável inteira

Casos de Testes 03

Entradas		**Resultados Esperados**	**Resultados Obtidos**
dividendo =	divisor =		
dividendo =	divisor =		
dividendo =	divisor =		

Exercício 2: Resolução de equação polinomial com uma variável de segundo grau.

Descrição: Uma equação polinomial é equação envolvendo uma variável matemática 'x' elevada a uma potência e multiplicada por uma constante. Por exemplo, uma equação polinomial de segundo grau pode ser escrita da seguinte forma: $ax^2 + bx^1 + cx^0$. ou, de forma mais simples, ax + bx + c. Quando queremos encontrar a raiz de uma equação, queremos resolver o valor da variável matemática 'x' de modo que o resultado da equação seja igual a zero. A Fórmula de Bhaskara é uma maneira de calcular os valores de 'x' para obter a solução desejada.

Considere as instruções para resolução dos exercícios e resolva os itens a seguir.
Os dados de entrada consistem dos valores para 'a', 'b', e 'c'.
Os resultados esperados são as raízes 'x1' e 'x2'.
Caso tenha apenas uma raiz, informar como 'x1'. Caso não tenha raiz real, informar a mensagem "Erro".

Casos de Testes 01

Entradas			**Resultados Esperados**		**Resultados Obtidos**	
a = -1	b = 2	c = 3	x1 = -1	x2 = 3	x1 =	x2 =
a = 2	b = 0	c = -18	x1 = 10	x2 = 20	x1 =	x2 =
a = -2	b = 20	c = -50	x1 =	x2 =	x1 =	x2 =

Antes de montar o próximo conjunto de casos de teste, reflita:

- Se algum dos "Resultados Esperados" forem diferentes dos "Resultados Obtidos", o que precisa ser alterado no seu algoritmo para ele trazer o resultado correto?
- Os casos de testes realizados no primeiro conjunto abrangem todas as possíveis entradas?
- Quais outros valores de entrada você acha relevante para esse problema?

Por exemplo: Se na estrada dos dados, tivesse o valor de 'a' como zero (fato que caracteriza uma equação do primeiro grau)?

Casos de Testes 02

Entradas			**Resultados Esperados**		**Resultados Obtidos**	
a =	b =	c =	x1 =	x2 =	x1 =	x2 =
a =	b =	c =	x1 =	x2 =	x1 =	x2 =
a =	b =	c =	x1 =	x2 =	x1 =	x2 =

Para o último conjunto de casos de testes, o ideal é explorar valores e situações não previstas nos casos de teste feitos até então. Neste contexto da Fórmula de Bhaskara, os casos de testes ainda podem explorar uma combinação de entradas em que o delta seja negativo, o que impossibilita o cálculo das raízes, neste caso, o algoritmo deverá retornar com uma mensagem informação a impossibilidade do cálculo.

Casos de Testes 03

Entradas			**Resultados Esperados**		**Resultados Obtidos**	
a =	b =	c =	x1 =	x2 =	x1 =	x2 =
a =	b =	c =	x1 =	x2 =	x1 =	x2 =
a =	b =	c =	x1 =	x2 =	x1 =	x2 =