Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Investigando o Uso de Testes para Apoiar a Resolução de Problemas de Programação

André Almeida

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Computação e Educação

Profa. Dra. Eliane Cristina de Araújo (Orientadora)

Prof. Dr. Jorge César Abrantes de Figueredo (Orientador)

Campina Grande, Paraíba, Brasil

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Investigando o Uso de Testes para Apoiar a Resolução de Problemas de Programação

André Almeida

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Computação e Educação

Profa. Dra. Eliane Cristina de Araújo (Orientadora)
Prof. Dr. Jorge César Abrantes de Figueiredo (Orientador)

Campina Grande, Paraíba, Brasil

A447i Almeida, André.
Investigando o uso de testes para apoiar a resolução de problemas de programação / André Almeida. – Campina Grande, 2023.
72 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2023.
"Orientação: Profa. Dra. Eliane Cristina de Araújo, Prof. Dr. Jorge César Abrantes de Figueiredo".
Referências.

1. Programação. 2. Resolução de Problemas. 3. Feedback. 4. Testes. 5. Avaliação Automatizada. 6. Computação e Educação. I. Araújo, Eliane Cristina de. II. Figueiredo, Jorge César Abrantes de. III. Título.

CDU 004.42(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO EM CIENCIA DA COMPUTACAO
Rua Aprígio Veloso, 882, Edifício Telmo Silva de Araújo, Bloco CG1, - Bairro Universitário, Campina Grande/PB, CEP 58429-900
Telefone: 2101-1122 - (83) 2101-1123 - (83) 2101-1124
Site: http://computacao.ufcg.edu.br - E-mail: secretaria-copin@computacao.ufcg.edu.br / copin@copin.ufcg.edu.br

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

ANDRÉ ALMEIDA

INVESTIGANDO O USO DE TESTES PARA APOIAR A RESOLUÇÃO DE PROBLEMAS DE PROGRAMAÇÃO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Mestre em Ciência da Computação.

Aprovada em: 06/04/2023

Profa. Dra. ELIANE CRISTINA DE ARAÚJO, UFCG, Orientadora

Prof. Dr. JORGE CESAR ABRANTES DE FIGUEIREDO, UFCG, Orientador

Profa. Dra. MELINA MONGIOVI BRITO LIRA, UFCG, Examinadora Interna

Profa. Dra. PASQUELINE DANTAS SCAICO, UFPB, Examinadora Externa

Documento assinado eletronicamente por **ELIANE CRISTINA DE ARAUJO**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 11/04/2023, às 12:04, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **MELINA MONGIOVI CUNHA LIMA SABINO**, **COORDENADOR DE POS-GRADUACAO**, em 11/04/2023, às 15:00, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **JORGE CESAR ABRANTES DE FIGUEIREDO**, **PROFESSOR 3 GRAU**, em 12/04/2023, às 15:04, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **3275953** e o código CRC **DB16F493**.

Referência: Processo nº 23096.024928/2023-68 SEI nº 3275953

Resumo

Os cursos introdutórios de programação desenvolvem esta habilidade através da resolução de diversos exercícios relacionados aos conceitos apresentados, os quais atuam também como forma de avaliar o desempenho dos alunos. No entanto, antes de produzir o programa o aluno precisa compreender o que é requerido pelo enunciado do exercício. É necessário entender a especificação como primeiro passo para a efetiva resolução de problemas de programação. Neste contexto, o objetivo desta pesquisa de mestrado é investigar a efetividade de uma estratégia baseada em testes para melhor esclarecer a especificação do problema e, com isso, favorecer a sua correta resolução. Iniciamos a pesquisa realizando um mapeamento dos tipos de erro identificados em exercícios de programação, com o objetivo de investigar se os testes automatizados associados capturam falhas na compreensão dos problemas. Com a análise dos dados foi possível perceber que cerca de 80% dos erros cometidos estão relacionados a dificuldade no entendimento ou falsas suposições levantadas com base nos enunciados dos exercícios. Desta forma, percebendo que os testes automatizados buscam capturar minuciosamente determinados cenários especificados nos enunciados dos exercícios, de maneira a verificar se o aluno os compreendeu em sua totalidade, este estudo preliminar denota a estratégia baseada em testes como sendo promissora. Sendo assim, criamos o oráculo para aplicar a abordagem e permitir que os alunos interajam com a solução de referência através de testes, perguntando se a entrada de um problema corresponde com a saída. No estudo principal, realizamos a análise comparativa entre o oráculo e outras estratégias de resolução de problemas com base em métricas associadas a sistemas de avaliação automatizada. Os resultados evidenciam uma melhora significativa no desempenho dos alunos, reduzindo o tempo para traçar a solução correta, que foi menor em 65% dos casos; e também o número de submissões ao sistema utilizado, que foi menor em 68% dos casos. Com estes efeitos somados ao *feedback* positivo dos alunos participantes, apontamos a estratégia como sendo eficaz na resolução de problemas de programação.

Palavras-chave: resolução de problemas, especificação de problemas, testes, avaliação automatizada, *feedback*, programação.

Abstract

The introductory programming courses develop this skill through the resolution of several exercises related to the presented concepts, which also act as a way of evaluating the students' performance. However, before producing the program, the student needs to understand what is required by the statement of the exercise. It is necessary to understand the specification as a first step for the effective resolution of programming problems. In this context, the objective of this master's research is to investigate the effectiveness of a test-based strategy to better clarify the specification of the problem and, therefore, favor its correct resolution. We started the research by mapping the types of errors identified in programming exercises, with the aim of investigating whether the associated automated tests capture failures in understanding the problems. With the analysis of the data, it was possible to perceive that about 80% of the errors committed are related to difficulty in understanding or false assumptions raised based on the statements of the exercises. In this way, realizing that the automated tests seek to capture in detail certain scenarios specified in the statements of the exercises, in order to verify if the student understood them in their entirety, this preliminary study denotes the strategy based on tests as being promising. Therefore, we created the oracle to apply the approach and allow students to interact with the reference solution through tests, asking if the input of a problem corresponds with the output. In the main study, we performed a comparative analysis between the oracle and other problem solving strategies based on metrics associated with automated evaluation systems. The results show a significant improvement in the students' performance, reducing the time to draw the correct solution, which was shorter in 65% of the cases; and also the number of submissions to the system used, which was lower in 68% of cases. With these effects added to the positive *feedback* of the participating students, we point out the strategy as being effective in solving programming problems.

Keywords: problem solving, problem specification, testing, automated assessment, *feedback*, programming.

Agradecimentos

À minha mãe Rosinere Almeida, uma mulher exemplar, repleta de força e coragem para enfrentar os desafios.

Ao meu "filho", talvez o melhor acontecimento que a pós-graduação pôde me proporcionar.

À minha companheira Sanara Gomes, por sua presença e pelo apoio durante toda esta trajetória.

Aos orientadores, Eliane Araújo e Jorge Figueiredo, pela tranquilidade e por todos os encaminhamentos desde os primeiros passos deste caminho, agradeço fortemente. Sempre os terei como referência.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo apoio financeiro para a execução desta pesquisa.

Aos professores e profissionais que contribuíram para a realização deste trabalho, fornecendo informações e recursos valiosos.

Conteúdo

Lista de Símbolos

TDD - *Test Driven Development*

AA - *Avaliação Automatizada*

POPT - *Problem-Oriented Programming and Testing*

MOOCs - *Massive Open Online Courses*

IPO - *Input-Process-Output*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

No ensino de programação para iniciantes, a habilidade de refletir sobre problemas e desenvolver programas para alcançar soluções é tida como um dos requisitos principais para o sucesso [49]. Os cursos introdutórios enfatizam o desenvolvimento desta habilidade através da resolução de diversos exercícios relacionados aos conceitos apresentados, os quais atuam também como forma de mensurar o desempenho dos alunos. Desta forma, o desenvolvimento de ferramentas de avaliação automatizada para lidar com a correção de muitos exercícios de programação está sempre em avanço, uma vez que estas se empenham em prover *feedback* de maneira rápida e constante, além de reduzir a carga de trabalho sobre os instrutores [21].

No entanto, antes mesmo de produzir o programa, se faz necessário que os alunos compreendam a especificação do problema, ou seja, entendam do que se trata o assunto e quais são os requisitos necessários para desenvolver a solução. Neste sentido, os sistemas de avaliação automatizada não fornecem *feedback* sobre o que eles precisam saber para planejar uma solução. O estudo de Cruz et al. [44] argumenta que aprender a programar não se trata apenas de dominar a sintaxe e a semântica de cada construção de uma linguagem de programação, mas que é um processo que se inicia desde a construção dos modelos mentais à respeito do programa, ou mais especificamente da representação mental de um problema.

Nesse sentido, Bennedsen e Caspersen discutem a ideia de "revelar o processo de programação"como fundamental para ensinar alunos novatos em programação [13]. De acordo com os autores, este processo engloba um conjunto de atividades executadas interativamente e que podem ser revisitadas, diferentemente de um processo linear. Em uma perspectiva si-

milar, a metodologia de Polya sobre como resolver problemas matemáticos [59] foi adaptada ao cenário de resolução de problemas de programação. Conforme já mencionado, programar consiste na escrita de código para resolver problemas e é justamente essa habilidade que se busca desenvolver durante os cursos de programação. Devido a grande proximidade entre exercícios matemáticos e de programação, a metodologia de Polya pode ser vista como um caminho para favorecer a correta resolução dos problemas. A Figura 1.1 resume as etapas de resolução de problemas de programação, adaptação da metodologia de Polya.

Figura 1.1: Etapas do processo de resolução de problemas em programação.

De maneira geral, resolver problemas de programação envolve quatro etapas, com algumas variações encontradas na literatura. Primeiro, o programador precisa entender o problema (1) e desenvolver um plano para encontrar a solução deste (2). Depois, o programador precisa traduzir a solução encontrada para uma linguagem que o computador seja capaz de entender e executar (3); e por fim, avaliar sua solução por meio de testes e refatorá-la caso seja necessário (4). Portanto, percebe-se que o esforço de entender o problema de forma correta é algo crucial para o desenvolvimento eficaz de programas.

Este processo pode ser visto como um caminho natural e eficaz para os alunos realizarem suas tarefas de programação. No entanto, todo o processo pode ser muito complexo para os novatos enquanto enfrentam suas primeiras experiências de programação, aumentando sua necessidade de orientação e assistência, em especial na primeira fase “entender o problema”, onde os alunos devem identificar os aspectos relevantes da definição do problema e entender

o cenário apresentado.

Para atender esta demanda de orientação e assistência aos alunos, além da correção de um conjunto de exercícios, geralmente os instrutores utilizam sistemas de avaliação automatizada com o intuito de fornecer *feedback* rápido e frequente. Ademais, o uso desses sistemas em cursos de programação permite avaliar de maneira mais eficiente o avanço dos alunos nas atividades propostas, possibilitando a melhoria no *design* destes cursos. No entanto, os alunos ainda precisam de um suporte maior para lidar com a complexidade que gira em torno da programação. É importante destacar que dificuldades no processo podem impedir que os alunos possam progredir de maneira autônoma ou levar à implementação incorreta da solução do problema [76].

Certos alunos não conseguem prosseguir para além da primeira fase sem pedir esclarecimentos aos professores ou colegas. Esta situação pode ser dificultada quando os alunos se sentem constrangidos em pedir ajuda e acabam buscando informações em outras fontes, isto quando não optam por seguir para as demais fases do processo de programação sem entender o problema em sua totalidade. Tais problemas tendem a impactar no cumprimento dos exercícios de programação e consequentemente no aprendizado pretendido. Alunos de programação devem ser treinados para pensar de forma crítica e criativa para resolver problemas. Devem se tornar adeptos da formulação, análise, decomposição e solução de problemas computacionalmente e devem internalizar os processos de *design* e desenvolvimento de software para que se tornem "hábitos mentais" aplicáveis a novos problemas [57].

1.1 Motivação

A atividade de programar, transformar um problema inicial em um conjunto de etapas bem definidas que quando executadas produzem um resultado, é uma das competências básicas que alunos com formação em ciência da computação, engenharia e cursos associados à área de tecnologia devem apresentar [55]. Aprender programação pode ser visto como um meio para desenvolver o pensamento computacional e a habilidade de resolução de problemas [14]. No entanto, educadores e pesquisadores em programação têm relatado bastante sobre as dificuldades dos alunos nos primeiros passos deste processo.

Em estudos sobre dificuldades enfrentadas por programadores iniciantes, um argumento

unânime é o de que aprender a programar não é uma tarefa fácil. Os alunos precisam lidar com vários obstáculos devido a complexidade do assunto, uma vez que é necessário entender corretamente conceitos que não estão diretamente associados ao seu cotidiano e que alguns alunos ainda não são estimulados a praticar até iniciarem um curso de graduação [48; 34]. Estes precisam entender o problema, formular a solução utilizando algumas técnicas e escrevê-la utilizando uma linguagem de programação que se aproxima da linguagem que o computador consegue entender para seguir as instruções. Na verdade, estudos têm indicado que trabalhar a habilidade de resolução de problemas enquanto estuda uma linguagem de programação traz uma série de dificuldades [1; 47], isto porque além de todos os aspectos sobre as especificações dos problemas, os alunos ainda precisam entender a sintaxe da linguagem. Com dificuldades nestas práticas, os alunos iniciantes podem se sentir frustrados e inseguros, pois não conseguem ter sucesso em suas atividades de programação, problema este que está entre os motivos das altas taxas de evasão de cursos de programação.

De acordo com Kadar et al. [47] outro fato que influencia nas dificuldades dos alunos no aprendizado de programação é a forma como aprendem. Alguns alunos consideram mais interessante estudar em grupo, uma vez que a troca de informações pode favorecer o surgimento de alguns *insights* que um estudo mais independente não o faria. Além disso, em uma pesquisa sobre dificuldades no ensino e aprendizado em programação Oroma et al. [54] aponta que os alunos que já têm contato com tecnologia há algum tempo são capazes de evoluir no curso com mais facilidade se comparados com aquele que não têm um contato prévio. Isto vai de encontro ao conceito da aprendizagem significativa, em que a construção do conhecimento progride a partir de conhecimentos prévios [5]. E mais, como alguns alunos não "vivem programação"(no sentido de terem contato desde as fases iniciais de ensino e ainda correlacionando com o dia a dia), estes enxergam como um conteúdo novo e muito complexo. Estes desafios acabam contribuindo para a prática do plágio, em que os alunos recorrem a internet ou colegas na busca por trechos de código que resolvem o problema, mas que pouco contribuem para o desenvolvimento das habilidades necessárias para desenvolver soluções [64].

No estudo conduzido por Gomes et al. [35] é apontada a correlação entre programação a motivação dos alunos. Ocasionalmente, quando os alunos se deparam com a complexidade dos problemas de programação e com a dificuldade de encontrar soluções que correspondem

com o que está sendo especificado, muitos se sentem desmotivados e tendem a colocar menos esforço em suas atividades. Esta atitude, associada ao pouco conhecimento em resolução de problemas contribui para o baixo rendimento em disciplinas de graduação e, posteriormente ao alto índice de evasão dos cursos [65].

Até então apontamos os desafios que são oriundos da natureza dos alunos, mas na literatura é apontada uma série de questões que competem aos instrutores dos cursos, corroborando com a vivência em sala de aula. Inicialmente podemos mencionar os profissionais pouco qualificados que, por vezes, não têm o hábito de correlacionar o que está sendo transmitido em sala de aula com trabalhos e atividades futuras que os alunos poderão desempenhar futuramente; e isto contribui para o ruir do interesse nas aulas de programação [54]. Além disso, cabe mencionar o quão importante é propor situações que se aproximam do cotidiano, fazendo uso de analogias que permitem que os alunos associem o que está sendo apresentado a alguma tarefa que desempenham com frequência e que não se atentaram que se trata de programação, ou resolução de problemas.

Algumas das dificuldades também podem estar associadas a habilidade dos instrutores fornecerem suporte adequado aos alunos iniciantes [78]. Os instrutores podem não estar aptos a verem pela perspectiva dos alunos e, por consequência, não compreendem os obstáculos com os quais estes se deparam. Neste sentido, convém a melhoria do *design* instrucional, buscando materiais complementares e recursos que possam agregar na aprendizagem dos alunos, reduzir os equívocos cometidos e melhorar o processo avaliativo. Os instrutores devem dar respostas e comentários suficientes sobre as avaliações e demonstrar soluções adequada, pois isto ajudará os alunos a entender como cada programa funciona [47]. E ainda, cabe mencionar que uma turma de iniciantes em programação dificilmente será igual às demais (em termos de habilidades, motivações etc), então é importante que o *design* instrucional seja um ciclo que sempre busca a melhoria para atender as necessidades observadas pelos instrutores, mesmo sendo um desafio devido a heterogeneidade das turmas (diferentes níveis de conhecimento).

Em um estudo sobre as habilidades de programação de alunos do primeiro ano, McCracken defende que os alunos precisam de trabalho adicional na primeira etapa do processo de programação: quando abstraem o problema de uma determinada descrição [49]. Os alunos muitas vezes interpretam mal ou não entendem completamente os problemas de programa-

ção, o que os impede de encontrar uma solução para o problema. Outro estudo argumenta que poucos programadores iniciantes podem identificar ambiguidades nas especificações das atividades e alguns deles contam com suas próprias suposições quando não conseguem entender o que é necessário [16]. Outro comportamento típico dos alunos nessa situação é "escrever o programa por meio de tentativa e erro"ou até mesmo chegar à solução pedindo dicas para instrutores ou outros colegas [38].

Na verdade, responder a perguntas de esclarecimento sobre a especificação de problemas é uma atividade rotineira de professores, assistentes ou instrutores de laboratório em um curso de programação. Infelizmente, essa interação pessoal pode ser muito custosa, quando há muitos alunos para um número reduzido de instrutores em uma aula de laboratório; ou quase impossível, quando se trata de cursos de educação a distância de grande porte, como os MOOCs (*Massive Open Online Courses*) [9].

1.2 Objetivos e Relevância

Diante do exposto, esta pesquisa visa lidar com o problema de fornecer suporte no esclarecimento da especificação do problema inspirado na tutoria individual. Nesse caso, é enfatizada a primeira etapa do processo de programação, quando os alunos devem tentar identificar o que é "o desconhecido". Como reunir as entradas (ou argumentos) para produzir saídas? Quais são as condições especiais? Eles devem tentar reformular o problema em diferentes perspectivas. Quais são as outras entradas possíveis e suas respectivas saídas? Quais são as conexões entre entradas e saídas? Em outras palavras, os alunos devem ser capazes de formular novos pares de exemplos de entrada / saída (ou testes) para o programa.

Este trabalho de mestrado gira em torno da investigação da efetividade de uma estratégia baseada em testes para melhor esclarecer a especificação do problema e, com isso, favorecer a sua correta resolução. Pretendendo alcançar este objetivo, os seguintes objetivos específicos podem ser traçados:

- Avaliar os erros cometidos por alunos iniciantes em programação e como estes estão relacionados com os testes os quais as soluções são submetidas;
- Empregar a abordagem baseada em testes (definida como "oráculo") em cursos intro-

dutórios de programação, como a primeira etapa na resolução de exercícios de programação propostos pelos instrutores;

- Comparar a abordagem "oráculo"com outros métodos de resolução de exercícios;
- Coletar o *feedback* dos alunos sobre a eficácia do oráculo.

Por meio da solução proposta, vislumbra-se que seja possível treinar os alunos para ler criticamente a especificação de um problema e extrair dados e condições úteis para entender o que é necessário para criar um programa com eficácia. Expor os alunos a um processo sistemático de solução de problemas, iniciando com a confirmação do entendimento do problema, pode impedir a prática usual de ir "imediatamente ao teclado", pulando os estágios iniciais de compreensão do problema e reflexões sobre como resolvê-lo [19].

Nossa conjectura é que se os alunos forem capazes de cobrir, com pares de entrada / saída, cada teste fornecido pelos instrutores para a tarefa, talvez eles tenham alcançado o entendimento conceitual desejado da especificação do problema. Consequentemente, eles podem estar prontos para prosseguir pelas demais etapas do processo de programação.

1.3 Questões de Pesquisa e Hipóteses

As questões de pesquisa e as respectivas hipóteses que nortearam este trabalho foram as seguintes:

- QP1: Os alunos que tentam compreender as especificações dos problemas utilizando o oráculo, criando e executando testes, produzem programas mais funcionalmente corretos?
  - H1.0: Os alunos que resolvem as questões utilizando o oráculo como apoio apresentam corretude funcional maior em suas soluções.
  - H1.1: Os alunos que resolvem as questões utilizando o oráculo como apoio não apresentam melhoras significativas na corretude funcional de suas soluções.
- QP2: Os alunos que tentam compreender as especificações dos problemas utilizando o oráculo, criando e executando testes, convergem para a solução correta mais rapidamente?

- H2.0: Os alunos que resolvem as questões utilizando o oráculo apresentam tempo de resolução e número de submissões menor.
- H2.1: Os alunos que resolvem as questões utilizando o oráculo como apoio não apresentam melhoras significativas no tempo de resolução e número de submissões.

1.4 Organização do Trabalho

O presente documento está organizado da seguinte forma: no Capítulo 2 são apresentados alguns conceitos que estão alinhados com pesquisa e que fornecem a base para a compreensão do estudo. O Capítulo 3 apresenta os trabalhos relacionados e que compreendem estratégias semelhantes a esta pesquisa. O Capítulo 4 esclarece o *design* dos dois estudos conduzidos, bem como a metodologia adotada. O Capítulo 5 descreve os resultados obtidos de maneira quantitativa e qualitativa. E, por fim, discorremos no Capítulo 6 sobre as conclusões da pesquisa e sobre os trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo são apresentados conceitos e argumentos importantes para favorecer a compreensão desta pesquisa. Neste trabalho argumentamos sobre a importância de fornecer assistência na primeira fase do processo de resolução de problemas - compreensão do problema - possibilitando melhoria no desempenho dos alunos em cursos introdutórios de programação. A este respeito, iniciamos com a discussão sobre o papel do *feedback* no processo de aprendizagem, seguida pela definição de avaliação automatizada de programas, teste de software e erros em exercícios de programação.

2.1 O *Feeback* no Processo de Aprendizagem

Existem várias definições para *feedback* presentes na literatura. Neste trabalho adotamos a definição de Boud e Molloy [18] que define como "o processo pelo qual os alunos obtêm informações sobre seu trabalho para apreciar as semelhanças e diferenças entre os padrões apropriados para qualquer trabalho e as qualidades do próprio trabalho, para gerar um trabalho melhorado". Em cursos de programação, o *feedback* pode ser visto como a consequência do processo. Uma vez que o professor propõe um conjunto de atividades para que os alunos pratiquem os conceitos apresentados em sala de aula, posteriormente os alunos elaboram programas de acordo com o que é especificado e recebem parecer sobre o que foi desenvolvido, seja em termos de código ou da compreensão do que é requisitado.

Na educação o *feedback* está associado a duas formas de avaliação, sendo elas a) somativa, onde um resultado é derivado por meio de critérios e padrões e b) formativa, em que

o resultado é dado aos alunos para ajudá-los a estar cientes das lacunas existentes em seus objetivos de aprendizagem [6]. Simplificando, o *feedback* somativo é visto como uma forma de reportar o progresso da aprendizagem, geralmente na forma de notas associadas a avaliações; enquanto que o *feedback* formativo é mais informal e tem o objetivo de guiar melhorias nas atividades de aprendizagem.

O *feedback* formativo tem sido constantemente considerado mais eficaz do que o *feedback* somativo em diferentes campos, visto que este funciona melhor por ser capaz de auxiliar os alunos a entenderem em que ponto estão no aprendizado, o que precisam alcançar e como proceder para atingir os objetivos [53; 37]. Percebendo o potencial deste tipo de *feedback*, diversos estudos na área de computação e educação têm se debruçado na ideia de produzi-lo e automatizá-lo para o processo de correção dos exercícios de programação. Sistemas de avaliação automatizada têm sido concebidos com o intuito de permitir que os alunos submetam seus exercícios e verifiquem a correção destes, como por exemplo, através da verificação com base nos casos de teste criados pelos instrutores. É nítida a importância do *feedback*, mas como produzi-los com qualidade?

Narciss e Huth [51] propuseram uma classificação de *feedback* para atividades de aprendizagem. Os autores definem basicamente três níveis de *feedback*, iniciando por 1) conhecimento do resultado, no qual é informado se uma resposta é correta ou incorreta; 2) conhecimento da resposta correta, o qual apresenta a solução correta para o problema especificado; e 3) *feedback* elaborado, algo mais detalhado e que identifica os problemas encontrados, ainda recomendando dicas de como a solução pode ser melhorada. Diversos trabalhos têm adotado esta classificação como objeto de estudo de maneira a analisar o quão cada categoria contribui para o aprendizado dos alunos [73; 50]. Em estudo, Van der Kleij et al. [71] expõem que o *feedback* elaborado permite melhores resultados em termos de aprendizagem que as demais categorias e que, em decorrência disto, os profissionais de educação podem conduzir melhores escolhas com relação às ferramentas digitais de aprendizagem e o desenvolvimento de *softwares* educacionais.

O fato é que, idealmente, o *feedback* precisa ser capaz de nortear os alunos para que o processo de submissão de atividades (em termos de sistemas de avaliação automatizada) não se torne um simples ciclo de envio, verificação dos resultados e ressubmissão de maneira exaustiva carente da capacidade de reflexão sobre erros e acertos. Para que o *feedback* seja

eficaz, quatro condições precisam ser atendidas: 1) os alunos devem precisar do *feedback*, 2) os alunos devem ter tempo suficiente para processar o *feedback*, 3) os alunos precisam estar dispostos a utilizar o *feedback* e 4) os alunos precisam ser capazes de entender o *feedback* [37]. O *feedback* desprovido de detalhamento tem maior probabilidade de resultar em frustração e perda de motivação [72].

2.2 Avaliação Automatizada no Ensino de Programação

As ferramentas de avaliação automatizada (AA) ganharam papel de destaque nos desenhos didáticos dos cursos de programação introdutória. Elas viabilizam a produção automatizada de *feedbacks* frequentes, rápidos, padronizados e a um baixo custo, permitindo que os alunos possam fazer muitos exercícios, necessários para o domínio da programação, à medida que reduzem a carga de trabalho empregada na avaliação manual por parte dos instrutores [4].

O primeiro exemplo de teste automatizado de atividades de programação é datado de 1960 [40] onde, em vez de usar compiladores e editores de texto, os alunos enviaram programas escritos em linguagem *assembly* em cartões perfurados. Uma das principais vantagens do método foi a utilização de maneira eficientes dos recursos de informática, o que possibilitou que um maior número de alunos aprendesse programação. Partindo deste ponto, os sistemas de avaliação foram evoluindo gradativamente, passando por sistemas orientados a ferramenta, frequentemente caracterizados por interfaces de linha de comando e operação manual de *scripts*; e chegando aos orientados a *web*, os quais fazem uso de abordagens de teste cada vez mais sofisticadas [29].

Diversas abordagens para avaliação automatizada de programas podem ser encontradas na literatura, bem como em outras fontes [3; 29; 43; 58]. O requisito básico para a análise é que algum tipo de valor de medição possa ser extraído do programa e que os valores possam ser comparados a determinados requisitos ou a uma solução de modelo [3]. A seguir listamos os principais recursos encontrados e trabalhados nos sistemas de avaliação automatizada relatados na literatura.

- Funcionalidade: a forma mais comum de avaliação, na qual consiste verificar se o programa funciona de acordo com os requisitos estabelecidos nos enunciados dos problemas. A funcionalidade é testada através da execução do programa sobre um conjunto

de casos de teste e a correção é feita através da comparação da saída impressa ou pelos valores de retorno.

- Eficiência: geralmente é baseada na execução do programa em diferentes casos de teste, medindo o comportamento durante a execução em termos de tempo de uso de CPU. Os resultados são frequentemente comparados com a solução modelo existente e este recurso é bastante encontrado em juízes online, como o *beecrowd* [15].

- Escrita de casos de teste: o objetivo de desenvolver a abordagem de testes automáticos é propor que os alunos desenvolvam dirigido a testes enquanto trilha seus primeiros passos em programação, projetando casos de testes para verificar minuciosamente os programas antes de submetê-los. Os professores notaram que tão importante quanto oferecer ferramentas de avaliação automática para os alunos, está a necessidade de garantir que os alunos projetem suas soluções com base em casos de teste criados para testar seus programas por conta própria [22; 30].

- Erros de programação: embora a abordagem mais comum gire em torno dos erros funcionais, detectados com base nos casos de teste pré-definidos, alguns trechos de código com erro podem ser reconhecidos estaticamente. Este tipo de análise de erro também pode estar relacionada à avaliação do estilo de programação, por exemplo, aos aspectos de confiabilidade do código em termos de redundância [77].

- Métricas de software: são consideradas como importantes para caracterizar os programas e, sendo assim, as métricas podem oferecer uma base para comparar e avaliar programas. No entanto, é importante definir métricas que agreguem algum valor no contexto educacional, no cenário em que os alunos estão inseridos [3]. Algumas métricas avaliadas na literatura são número de linhas de código [42]; e métricas relacionadas a programação orientada a objetos [62].

- Avaliação somativa e formativa: diversos sistemas de avaliação automatizada dispõem da avaliação somativa, a qual podemos resumir como "se todos os casos de teste resultarem em um comportamento de programa bem-sucedido, o aluno foi aprovado."[68]. No entanto, há tempos que a ideia de trazer à tona uma avaliação formativa, que se propõe a fornecer *feedback* sobre os trabalhos e permitir que os alunos melhorem seus

programas, tem sido trabalhada. Entre os diversos recursos associados a avaliação somativa estão a possibilidade ressubmissão [22]; e os programas de tutoria, um tipo especial de ferramenta de avaliação automática desenvolvida para suporte ao aprendizado [24].

É perceptível que os sistemas de avaliação automatizada facilitam o trabalho dos professores em termos de compilação de programas e execução de casos de teste. Em contrapartida, os recursos presentes nestes ambientes devem ser bem planejados e fornecer valor significativo para a aprendizagem. Os alunos precisam de um suporte que vai além de *feedback* e notas. "Se avaliações automáticas são incorporadas aos cursos e afetam os alunos, a relevância dessas questões para a programação deve ser claramente justificada para os alunos."[2]

2.3 Teste de *Software*

De acordo com Jamil et al. [45], teste pode ser definido como o processo de verificação e validação sobre se um sistema específico atende aos requisitos originalmente especificados ou não. Teste de software refere-se a encontrar *bugs*, erros ou requisitos ausentes no sistema ou software desenvolvido.

Tradicionalmente, testes de software são ensinados a partir da metade dos cursos de Ciência da Computação [66], muito embora o processo de testar esteja presente desde o primeiro contato com programação. No geral, em cursos introdutórios de programação a abordagem tradicional consiste em apresentar alguns conceitos sobre ciência da computação e logo em seguida praticar os fundamentos de programação utilizando uma linguagem (como Python, C ou Java) [39]. No entanto, na maioria dos casos o foco está na sintaxe da linguagem, quando na verdade deveria ser em resolver os problemas através da escrita de algoritmos. Como consequência, os alunos desenvolvem a prática de programar por "tentativa e erro", desprovidos do desenvolvimento das habilidades adequadas de compreensão e análise [31].

Edwards [31] destaca que incluir testes de software em disciplinas introdutórias de programação podem promover o desenvolvimento de programas de maneira cuidadosa, em que os alunos buscam compreender melhor a elaboração dos algoritmos. Experiências recentes têm sugerido que esta atividade pode ser ensinada o mais cedo possível no processo de aprendizagem, pois permite a melhora do raciocínio sobre o programa (e sua solução), le-

vando a produtos de melhor qualidade; além de induzir e facilitar o uso de testes ao longo do processo de desenvolvimento de software [7]. No entanto, o ensino de testes em cursos introdutórios não é uma tarefa trivial. Várias razões sobre este assunto podem ser apontadas [30; 56]:

- Teste de software requer que os alunos tenham conhecimento bem concebido em programação;
- Alunos precisam de *feedback* constante sobre como melhorar seu desempenho nos testes em muitos momentos ao longo do processo de desenvolvimento da solução, não apenas ao final da tarefa;
- Alunos veem os testes como uma atividade cansativa, em que muito esforço é empregado na execução e elaboração dos planos de teste, aumentando assim a carga de trabalho.

Apesar dessas limitações, alguns trabalhos têm mostrado que o ensino de testes mais cedo pode melhorar a qualidade do código implementado e pode facilitar o processo de aprendizagem, tanto de teste quanto de programação [31; 7; 8].

Inicialmente, o primeiro passo do processo de teste consiste em gerar os casos de teste. Os casos de teste são desenvolvidos usando várias técnicas de teste, as quais visam o desenvolvimento eficaz e preciso. Uma técnica bastante conhecida é o *Test Driven Development* (TDD), que faz uso de testes automatizados com o intuito de guiar o *design* do software. Em um processo de teste tradicional, os testes são utilizados como forma de encontrar erros associados ao desenvolvimento, mas o TDD fornece um guia referente ao sucesso da implementação, aumentando o nível de confiança sobre o sistema atender ao que foi especificado [45].

Em um estudo sobre a aplicação do método TDD em cursos introdutórios, Desai et al. [28] argumentam que embora as cargas de trabalho dos alunos possam aumentar com a incorporação do método, os alunos são capazes de desenvolver testes com sucesso enquanto aprendem a programar. Incorporar o teste desde o início da experiência de programação de um aluno é fundamentalmente importante para ensinar as habilidades analíticas e de compreensão necessárias no teste de software. Se os cursos podem deixar os alunos “infectados por

testes" desde o início, os autores sugerem que eles provavelmente perceberão que os testes são parte integrante da programação, beneficiando-os ao longo de suas carreiras acadêmicas e profissionais.

2.4 Os Erros em Exercícios de Programação

Esforços para melhorar o ensino em programação estão em constante avanço, especialmente na proposição de estratégias para auxiliar os professores a ajudarem os alunos a desenvolverem sua compreensão da programação e da área como um todo. Isto é algo bastante desafiador em cursos introdutórios de programação, onde os alunos iniciantes costumam apresentar concepções equivocadas e outras dificuldades no contexto de aprender a programar que inibem sua capacidade de aprender e progredir. Desta forma, a capacidade dos professores identificarem tais equívocos e avaliarem os alunos, revelando estes equívocos, é considerado importante para o ensino eficaz [63]. Compreender a conceituação imprecisa de um aluno é um pré-requisito necessário para ajudá-lo a avançar em uma conceituação precisa [46].

De acordo com Sutcliffe e Rugg [70], a resolução de problemas pode ser dividida em três níveis, com seus respectivos esclarecimentos sobre os tipos de erro que podem acontecer.

- Baseado em conhecimento: existe pouco ou nenhum conhecimento sobre como resolver um problema. Nesse caso, as pessoas consideram difícil resolver problemas porque precisam formular um modelo mental do problema e depois buscar soluções. Os erros neste nível estão mais intimamente ligados às habilidades gerais dos solucionadores de problemas do que à sua experiência.

- Baseado em regra: nesse nível, existe algum conhecimento, de modo que a resolução de problemas é alcançada pela aplicação de regras na ordem correta para obter um efeito. Os erros podem ser causados por problemas de memória, muitas vezes não reconhecendo o contexto para a aplicação de uma regra. As regras também podem ser formadas inadequadamente.

- Habilidade: Nesse caso, os problemas já foram resolvidos anteriormente e usamos uma solução pré-formada (uma habilidade) para alcançar a solução desejada. Erros neste nível normalmente dizem respeito à atenção e à memória. Se tivermos um programa

ruim armazenado na memória, ele não será apropriado; alternativamente, se a atenção se desviar, podemos perder etapas em um programa bem formado.

Ao longo dos anos, pesquisadores utilizaram diferentes termos para descrever o entendimento impreciso ou incompleto de conceitos de programação por parte dos alunos, tais como "dificuldades", "equívocos", "erros", *"bugs"* dentre outros [63]. Por existir esta variação nos termos, não existe uma definição universal destes problemas no contexto de programação. No entanto, uma definição frequentemente citada nos trabalhos é a de Sorva [67], que define equívocos como sendo "entendimentos que são deficientes ou inadequados para muitos contextos de programação prática", os quais podem incluir erros de sintaxe, mal-entendidos relacionados à semântica e outras dificuldades referentes ao planejamento e à escrita de código para resolução de problemas.

Neste sentido, os equívocos dos alunos sobre programação e outros temas são alvo de pesquisa. Tomando como base a linguagem BASIC, Bayman e Mayer [10] examinaram os equívocos relacionados a instruções de programação do ponto de vista de formulação de modelos mentais, onde puderam constatar que muito alunos tinham entendimentos incorretos ou concepções bastante inconsistentes de até mesmo comandos mais simples. Posteriormente, Bonar e Soloway verificam de forma mais geral como os alunos descrevem seu entendimento sobre enunciados de programação utilizando linguagem natural [17]. Os autores conjecturam e analisam que quando alunos se deparam com "impasses"que impedem seu progresso em exercícios, estes acabam recorrendo a declarações feitas utilizando linguagem natural que destoam de como de fato o programa deve se comportar, uma vez que determinados conectivos são abstraídos em linguagens de programação formais. Com isto, alguns *bugs* são introduzidos.

Spohrer e Soloway [69] analisaram a fonte de erros de programação para verificar se eles resultam de “concepções erradas sobre a semântica das construções de linguagem de programação”. Os autores discorrem que os problemas são mais prováveis de surgirem de erros de alunos na leitura e análise de especificações, além de falhas em perceber a falta de coerência no código. Vários trabalhos na literatura vão de encontro a esta e outras constatações [12; 27; 41], além de serem úteis para alcançar uma compreensão rica dos problemas comuns que os alunos podem enfrentar e assim ajudar os professores no *design* dos cursos de programação, ou fornecer suporte adicional para erros incomuns [20].

Edwards [31] defende a ideia de trabalhar testes de software desde os primeiros passos dos iniciantes no mundo na programação, de maneira a evitar a estratégia mais usual adotada pelos alunos, a de "tentativa e erro", levando assim a uma maior taxa de sucesso nos exercícios. Tentativa e erro é uma técnica bastante utilizada que, de acordo com o autor, se baseia nas seguintes visões:

- "Se o compilador aceita meu código sem "reclamações", todos os erros foram removidos";
- "Se meu código produz a saída esperada para um ou dois casos de teste, ele funcionará bem para todos";
- "Meu código parece correto. Se produz a saída incorreta em alguns casos, algo foi ignorado. Vou tentar trocar algumas estruturas para verificar se o problema desaparece."

Esta estratégia pode parecer útil e até resolver alguns problemas, mas as habilidades principais que vêm a tona no momento de escrever código ficam em segundo plano. Edwards [31] menciona que, embora esta estratégia se torne ineficaz em algum ponto, os exercícios práticos aplicados nas disciplinas de programação objetivam que os alunos apliquem habilidades básicas de compreensão e análise de problemas. Sem essas habilidades, os alunos estão carentes do conhecimento necessário para qualquer outra estratégia, exceto a de tentativa e erro.

O fato é que o erro precisa ser visto como uma forma de verificar se o nível de entendimento de um problema foi alcançado em sua totalidade, gerando reflexão sobre a especificação, aquele elemento que compõe o primeiro passo do processo de resolução de problemas. E mais, para que exista avanço neste método de reflexão, os alunos precisam de mais do que prever como as mudanças no código resultarão na mudança no comportamento do programa como um todo. Eles precisam praticar a formulação de hipóteses sobre o comportamento de seus programas - Algo como "se o meu programa deve receber um número par, o que deve acontecer se for digitado um número ímpar?- para em seguida verificar estas hipóteses [31]. Também precisam de *feedback* frequente, útil e imediato sobre seu desempenho, tanto para formular hipóteses quanto para testá-las experimentalmente. Entender os erros mais comuns em programação é importante, assim como fomentar como o erro pode ajudar na percepção dos pormenores dos exercícios de programação e promover melhorias no código.

Capítulo 3

Trabalhos Relacionados

É possível encontrar na literatura uma variedade de trabalhos que propõem estratégias para nortear a resolução de problemas, tendo a etapa de entendimento do problema como principal foco de pesquisa. Neste capítulo sintetizamos os trabalhos com ênfase no detalhamento da aplicação de cada estratégia e nas constatações dos autores.

Vários trabalhos encontrados na bibliografia apresentam alguma estratégia ou mecanismo que busca evidenciar que oferecer um auxílio para que o aluno reflita sobre os diversos cenários que um programa deve representar traz benefícios no desempenho, uma vez que um dos principais problemas percebidos nos alunos é a dificuldade de construir um modelo conceitual correto do problema [61]. Sendo assim, direcionamos a análise do estado da arte para os trabalhos que tomam como base os testes como principal meio para se atingir o objetivo.

Neto et al. [52] apresentam o método de ensino de Programação e Teste Orientado a Problemas (POPT - *Problem-Oriented Programming and Testing*) para cursos de Introdução à Programação, com o objetivo de melhorar as habilidades dos alunos novatos em testes e em lidar com problemas mal definidos (aqueles que não apresentam um objetivo claro, dicas de resolução ou a saída esperada). Suportado por uma ferramenta chamada *TestBoot*, que utiliza planilha eletrônica para permitir que alunos iniciantes definam casos de teste de entrada-saída simples no estilo de tabela. Desta forma, o método é apresentado em função de um estudo de caso no qual o POPT é comparado com uma abordagem de teste cego que é frequentemente usada para avaliar exercícios de alunos em cursos de programação.

Basu et al. [9] apresentam um método aplicado no contexto de MOOCs (*Massive Open Online Courses*). Os autores propõem uma ferramenta, o Sistema OK, que realiza a ava-

liação automática de exercícios de programação com base em suítes de testes criadas para verificar as soluções fornecidas nos mais diversos cenários. No entanto, além desta proposta, os autores incluem um recurso para que os alunos busquem validar o entendimento dos exercícios. Através de um processo interativo, o aluno precisa lidar com algumas questões de respostas curtas sobre o potencial comportamento que o código deve esboçar; e cada uma dessas questões corresponde a um caso de teste "bloqueado" (aqui, bloqueado pode ser definido como aquele caso de teste que o aluno não tem acesso). Respondendo corretamente a uma dessas questões, o aluno pode desbloquear o caso de teste e aplicar em sua solução antes de realizar sua submissão.

Em outra abordagem, Denny et al. [26] observam que provisionar suporte de maneira a evitar interpretações incorretas e a aperfeiçoar a consciência metacognitiva mostrou ser benéfica em alguns casos [23; 60]. Estendendo esses trabalhos, os autores conduzem um amplo experimento examinando o impacto da intervenção em escala e em um ambiente de trabalho mais típico para alunos, onde estes devem reinterpretar sua leitura inicial da especificação do problema respondendo uma questão sobre sua compreensão e um único caso de teste, para garantir que formem um modelo mental apropriado da tarefa antes de escrever qualquer código.

A estratégia de Denny et al. [26] apresenta forte embasamento no *Test Driven Development* [11] e na inclusão do método no processo de desenvolvimento dos alunos, pois um dos benefícios do *Test Driven Development* é impor ao programador a pensar em casos de entrada especiais que podem ocorrer. No entanto, solicitar que os alunos resolvam os casos de teste primeiro também pode ser usado para ajudá-los a entender melhor o problema determinado. Uma das ferramentas de avaliação automatizada mais populares, o WebCAT [32], foi criada justamente com o intuito de integrar o TDD no contexto dos alunos; assim como o ProgTest [25], o Kattis [33] entre outros.

Tomando como base o *IPO Model* [36], uma abordagem amplamente utilizada para modelar estruturas de programação que descreve o processo de resolver problemas em três tarefas - identificar as entradas, realizar o processamento para transformar as entradas e obter as saídas como resultado deste processamento - Cabo [19] busca quantificar tanto o entendimento de problemas, quanto a performance utilizando a linguagem de programação Python de um conjunto de alunos de um curso introdutório de *Problem Solving*. De maneira con-

cisa, no estudo os alunos recebem a especificação de um problema de programação e são solicitados a sugerir, no mínimo, três possíveis entradas, para em seguida encontrarem as saídas correspondentes para o problema de maneira manual (ou utilizando uma calculadora, por exemplo); posteriormente os alunos podem partir para a codificação de suas soluções. A ideia principal é verificar se a dificuldade de escrever programas funcionalmente corretos é identificada antes mesmo da escrita de qualquer código, além de propor a extensão do entendimento dos problemas através de definição de entradas e saídas.

Wrenn e Krishnamurthi [76] propõem um mecanismo que fornece aos alunos um *feedback* instantâneo sobre se eles exploraram correta e completamente um problema, independentemente de seu progresso de implementação. Nesta abordagem os exemplos de entrada-saída, como casos de teste, podem ser articulados como afirmações do comportamento de entrada-saída das funções, ou seja, os alunos podem fornecer tanto a entrada quanto a saída que esperam do(s) problema(s) em questão. Com esta estratégia, os autores objetivam fornecer uma maneira de verificar a compreensão dos problemas e prevenir a implementação sem o conhecimento suficiente das particularidades dos mesmos.

Posteriormente, estes mesmos autores trazem à tona e extensão desse estudo, agora sobre a vertente de investigar quais dúvidas ainda persistem sobre os problemas e qual a influência do *feedback* automático e sob demanda promovido pelo mecanismo [75]. Eles argumentam que o mecanismo apresenta benefícios, mas foi observado alguns comportamentos por parte dos alunos, como teste exaustivo de todas as entradas e saídas possíveis sem o conhecimento necessário para testar o comportamento não especificado das tarefas. Além disso, os alunos que compreenderam o modelo de avaliação do mecanismo, foi prejudicado pelo seu *feedback* "opaco". Por fim, destacam um aspecto marcante sobre a compreensão de problemas ser algo que parece exigir novas habilidades que não são muito bem abordadas nas bases curriculares atuais; e que a utilização de *feedback* automatizado não impede que os alunos façam perguntas sobre o comportamento esperado dos problemas.

A análise do estado da arte em relação aos trabalhos que denotam aproximação com nossa estratégia revela que a inclusão de testes na etapa de compreensão do problema é um objeto de estudo com diversas variações. De recursos mais simples como planilhas, até ferramentas de automatização de *feedback*, a atividade de escrever testes capazes de capturar comportamentos que devem ser representados em código se mostra deveras benéfica quando

comparada apenas a aplicação de testes cegos que pouco agregam na construção do modelo mental de um problema.

Observamos que o trabalho de Wrenn e Krishnamurthi [76], dentre os que foram apreciados, é o que mais se aproxima de nossa estratégia - propondo que o aluno interaja livremente com o oráculo através de testes compostos por entradas e saídas - e que investiga em profundidade os proveitos da mesma. No entanto, não deixamos de perceber as argumentações referentes à inclusão da programação direcionada e amparada por testes no contexto de alunos iniciantes em programação; e que em suas investigações apresentam diferenças sutis com relação a nossa estratégia. Desse modo se torna pertinente, além da aplicação e avaliação da nossa estratégia, um estudo comparativo com as abordagens mais cotidianas de resolução de problemas de programação.

Capítulo 4

Metodologia

Neste capítulo será descrita a metodologia utilizada para alcançar os objetivos propostos que foram mencionados anteriormente. Em geral, o processo de investigação da estratégia foi realizado de acordo com as seguintes etapas:

- **Analisando erros comuns em exercícios de programação**: conduzida com o objetivo de investigar o uso de casos de teste como forma de identificar falhas na compreensão de problemas de programação.
- **Empregando o oráculo em cursos introdutórios de programação**: de maneira a averiguar a estratégia do foco deste trabalho, realizando a comparação com outros métodos de resolução de exercícios de programação. Esta etapa pode ser segmentada nas seguintes:
  - **Aplicação do estudo e variáveis coletadas**: para definir o *design* do estudo em termos de forma de resolução de cada exercício e as variáveis de interesse para a análise da efetividade.
  - **Seleção dos exercícios**: com o objetivo de determinar os exercícios a serem propostos para os alunos.
  - **Definição das ferramentas**: englobando atividades como a escolha da plataforma de submissão dos exercícios e a concepção da estratégia, o oráculo.
  - **Elaboração de questionários**: com objetivo de coletar informações sobre o nível de conhecimento em programação dos alunos e seus *feedbacks* sobre a estratégia

estudada.

A pesquisa foi aplicada no período de JUL a SET de 2022 e desenvolvida no contexto de disciplinas introdutórias de programação da UFCG, tendo sido submetida ao Comitê de Ética em Pesquisa com CAAE nº 46272421.7.0000.5182 .

4.1 Analisando erros comuns em exercícios de programação

Este estudo foi conduzido com o objetivo de investigar o uso de casos de teste como forma de identificar falhas na compreensão de problemas de programação, de maneira a evidenciar esta estratégia como sendo promissora e eficaz tanto para este cenário, quanto para o esclarecimento das respectivas especificações. Para isso, inicialmente objetivamos classificar as respostas dos problemas com algum erro identificado, tomando como base a taxonomia apresentada em Sutcliffe e Rugg [70] e aplicada em Veerasamy et al. [74]. Ou seja, esta análise se trata de uma reprodução do estudo que foi conduzido por Veerasamy et al. [74], o qual toma como base a taxonomia de erros proposta em Sutcliffe e Rugg [70], a qual é brevemente apresentada na Seção 2.4.

Como definido em [74], os erros costumam ser cometidos devido à falta de conhecimento, descuido e/ou erro de julgamento. Com isso, a literatura define três categorias apresentadas a seguir. Ver Tabela 4.1.

Em seguida, conhecendo os erros cometidos pelos alunos e classificados de acordo com a taxonomia de erros proposta pela literatura, realizamos o mapeamento destes para os casos de teste que captam exatamente o mal-entendido identificado.

4.2 Empregando o oráculo em cursos introdutórios de programação

Após realizarmos a análise do impacto dos erros de compreensão nas soluções dos exercícios, partimos para a organização do estudo que busca verificar o potencial da abordagem baseada em testes para melhor esclarecer a especificação do problema e, com isso, favorecer

Tabela 4.1: Tipos de erro, de acordo com Sutcliffe e Rugg [70]

Tipo	**Descrição**
Erro baseado em conhecimento	Ocorrem quando um aluno não consegue realizar a tarefa devido à falta de conhecimento, sobre como resolver o problema. Neste nível, os erros estão mais relacionados a habilidades gerais do que a experiência. Ex.: Não utilizar corretamente um laço. Ex.: Não conseguir implementar um comportamento especificado no enunciado.
Erro de habilidade	Ou "escorregões e lapsos", ocorrem devido a fortes invasões de hábitos. Normalmente se refere a atenção e memória. Ex.: Utilizar o print quando a especificação do problema pede que utilize o return.
Erro baseado em regras	Aqui os erros ocorrem devido a incorreta implementação e/ou inadequação de um método ou regra; Ex.: A aplicação de funções de manipulação de string a valores numéricos no programa. Ex.: A implementação de algoritmos de ordenação quando o enunciado do problema exigia a não utilização.

a sua correta resolução. Em resumo, o estudo consiste em propor alguns exercícios de programação aos participantes e sugerir que cada um deles seja resolvido utilizando diferentes estratégias de desenvolvimento de código. Iniciamos com a definição das variáveis a serem coletadas para investigar a estratégia proposta; em seguida partimos depois para seleção dos exercícios a serem aplicados; posteriormente, definimos as ferramentas para fornecer suporte aos alunos; e por fim a elaboração dos questionários para conhecer o nível de conhecimento prévio dos participantes, bem como o *feeback* após o estudo.

4.2.1 Aplicação do estudo e variáveis coletadas

Os participantes do estudo foram convidados a uma atividade extra-classe que ocorreu durante o horário de aula de disciplinas de programação introdutória da UFCG. Inicialmente houve uma apresentação de cerca de vinte minutos explicando como seria realizado o estudo

e quais plataformas os alunos precisariam acessar para realizar o planejado. Posteriormente, não houve divisão em grupos e todos os alunos resolveram os mesmos exercícios utilizando a linguagem Python englobando os conteúdos de estruturas de controle e de repetição. Para cada exercício proposto, uma estratégia de resolução foi aplicada, sendo elas:

- Método usual: representado na Figura 4.1, no qual geralmente o aluno vai imediatamente ao teclado após ler a especificação do problema, tentando delinear a solução.
- Método baseado no TDD (*Test-Driven Development*): representado na Figura 4.2, neste o aluno, inicialmente, escreve um caso de teste, para em seguida produzir o código que pode ser validado pelo teste desenvolvido.
- Método proposto pelo estudo (interação com o oráculo): representado na Figura 4.3, no qual o aluno deve criar testes de entrada/saída e executá-los sobre uma solução de referência (que faz parte do oráculo), observando em seguida o *feedback* recebido. Posteriormente pode seguir para a implementação, ou utilizar alternadamente enquanto desenvolve.

Foram selecionados quatro exercícios diferentes com o objetivo de realizar uma análise entre pares de estratégias, ou seja, Usual-Oráculo e TDD-Oráculo. O estudo foi realizado de maneira presencial e em dois dias diferentes devido a limitações de tempo durante a disciplina do curso, o que favoreceu a comparação mencionada anteriormente.

Referente às variáveis, os aspectos que nos propusemos a avaliar nas soluções fornecidas para mensurar o desempenho dos alunos nos três métodos foram:

- Número de submissões (**NSub**): com o intuito de observar se a criação/execução de testes de entrada e saída antes da fase de codificação (oráculo) resulta em um número menor de tentativas de submissão, com a resposta se aproximando da solução de referência em menos tentativas.
- Tempo (**T**): tempo gasto entre o aluno recebendo uma especificação e o aluno submetendo a versão final do programa ao sistema.
- Número de testes aceitos (**AT**): para observar o quão funcionalmente corretas estão as respostas fornecidas pelos alunos.

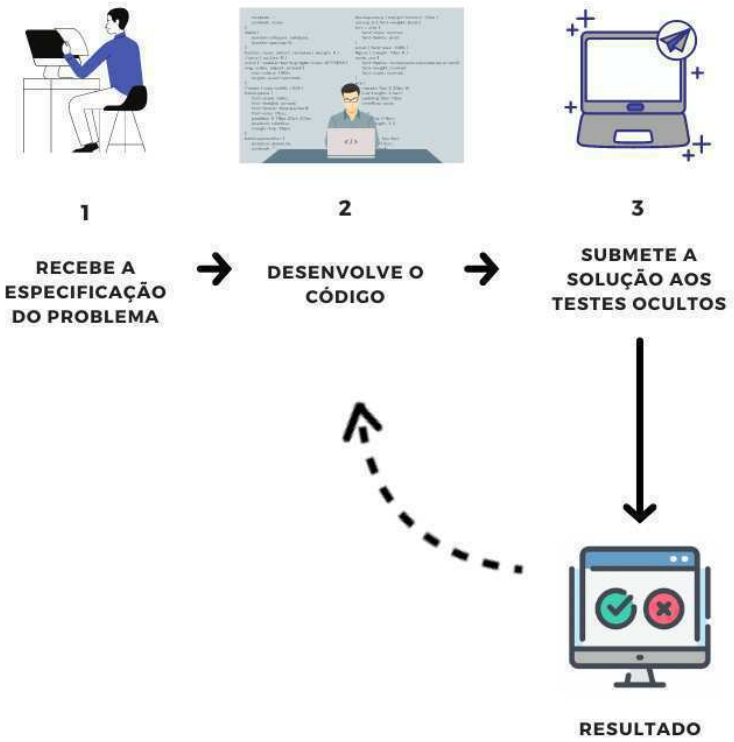


Figura 4.1: Processo usual de resolução.

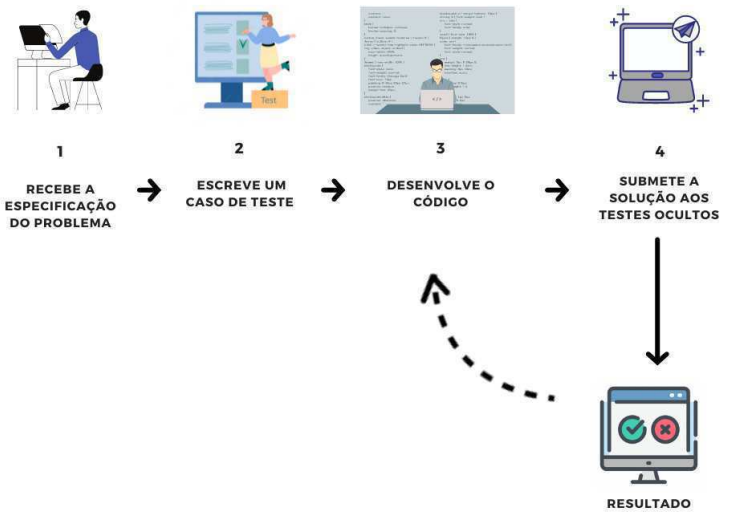


Figura 4.2: Processo de resolução baseado no TDD.

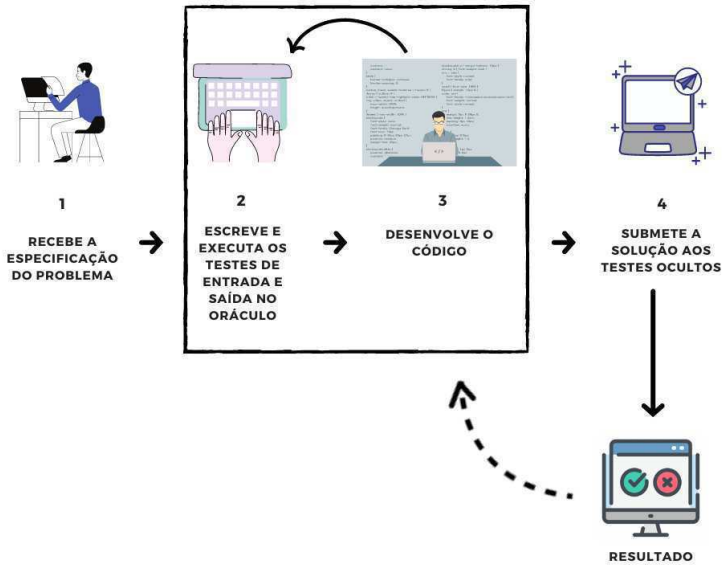


Figura 4.3: Processo de resolução com a inclusão do oráculo.

4.2.2 Seleção dos exercícios

Conforme mencionado anteriormente, o estudo consiste em avaliar o desempenho dos alunos com base na resolução de exercícios de programação introdutória e de acordo com as estratégias citadas na subseção anterior. Desta forma, se fez necessário realizar uma triagem dos exercícios e tomando as devidas precauções para o estudo, sendo elas:

- Exercícios que não excedam o conteúdo programático já abordado na disciplina de graduação;
- Exercícios que permitam a resolução em tempo hábil, dentro do tempo limite estabelecido (30 minutos para cada proposta);
- Exercícios que possuam o mesmo nível de dificuldade.

Tendo estas preocupações em mente, utilizamos como uma das referência os exercícios do *URI Online Judge - beecrowd*[1], uma vez que estão organizados por categorias e identificados por nível de dificuldade. Selecionamos quatro exercícios (consultar apêndice B), dois deles de origem da plataforma - 'Idade em dias' e 'Soma de ímpares consecutivos' - e

[1] https://www.beecrowd.com.br/judge/en/login

os outros dois propostos em livros de programação - 'Calculando desconto' e 'Calculando IMC'.

Vale evidenciar que selecionamos quatro exercícios para aplicar três estratégias seguindo esta organização: 2 exercícios para aplicação utilizando o oráculo, 1 exercício para o método usual e 1 exercício para o método baseado no TDD. Desta forma objetivamos realizar comparações em dupla: Usual vs. Oráculo e TDD vs. Oráculo.

4.2.3 Definição das ferramentas

Com os exercícios selecionados, partimos para a definição e preparação dos ambientes que os alunos deveriam utilizar para aplicar os procedimentos propostos pelo estudo.

Inicialmente analisamos plataformas de submissão de exercícios de programação em busca daquela que apresentasse maior flexibilidade de uso e uma interface simples que não prejudicasse o fluxo de trabalho dos alunos, optando ao final pelo *MIMIR Classroom* [2]. Com proposta bem semelhante a plataforma do Google, porém voltado para cursos de ciência da computação, o *MIMIR Classroom* tem o objetivo de melhorar a experiência do aluno, economizando tempo dos professores através de recursos como *feedback* individual, tarefas mais claramente definidas e tempos de avaliação mais rápidos. A figura a seguir apresenta uma visão geral da plataforma.

Uma vez com o cadastro aprovado na plataforma, os exercícios escolhidos foram cadastrados juntamente com um conjunto de testes ocultos criados para verificar os cenários possíveis para os códigos devolvidos como solução. Com relação ao processo de submissão, a plataforma fornece um ambiente de desenvolvimento próprio ou a possibilidade de enviar um arquivo do tipo Python (.py), sendo este último o procedimento adotado para a realização do estudo. Ademais, não foi estabelecido um limite no número de submissões, uma vez que isto impactaria diretamente na coleta da variável NSub.

No que se refere a abordagem alvo desta pesquisa, o oráculo, várias ferramentas similares podem ser encontradas como objeto de estudo nos trabalhos relacionados, no entanto algumas não estão disponíveis para uso ou exigem uma carga considerável de conhecimento para poder operá-las. Mediante a dificuldade, decidimos desenvolver uma prova de conceito capaz de realizar o comportamento esperado pelo oráculo: munida das soluções de referên-

[2] https://www.mimirhq.com/

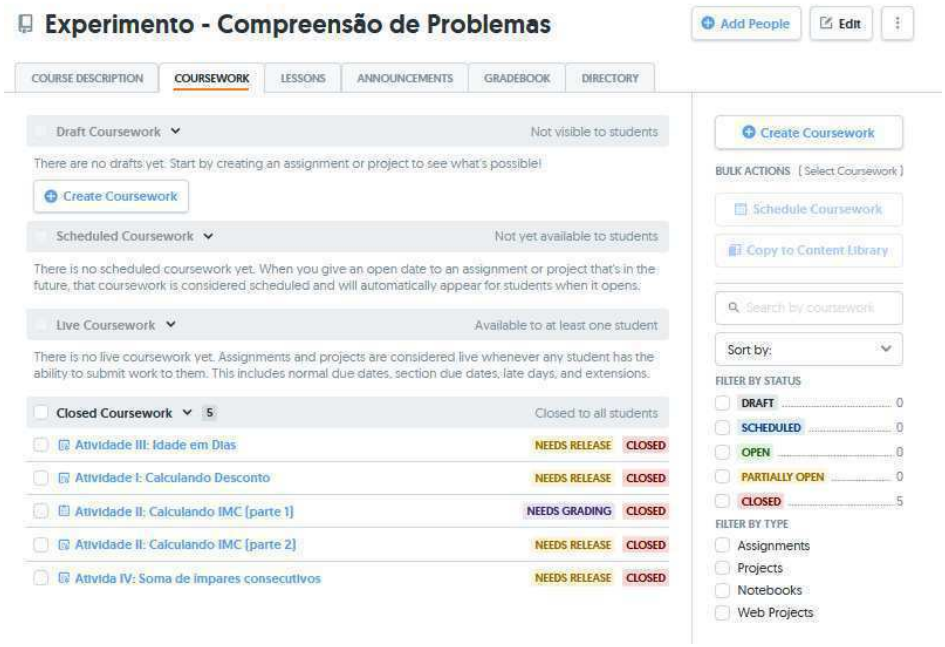


Figura 4.4: Visão da sala de aula criada para utilização no estudo na plataforma *Mimir Classroom*.

cia dos exercícios, a ferramenta oferece uma interface que permite que o usuário forneça uma entrada para determinado problema e a saída que ele imagina que o programa irá gerar; de posse desse par de informações, a ferramenta oferece *feedback* sobre a correspondência entre esses parâmetros - correto ou incorreto. A figura a seguir apresenta o menu de acesso aos exercícios cadastrados no oráculo[3] (à esquerda) e o menu de interação (à direita).

4.2.4 Elaboração dos questionários

Como última etapa do design do estudo, partimos para a elaboração dos questionários para registrar dados referentes à identificação e ao *feedback* dos participantes.

Inicialmente, antes da aplicação do estudo, objetivamos coletar informações dos alunos que participaram do com relação às suas aptidões frente a resolução dos problemas de programação. Desejamos observar se os alunos levam em consideração o *feedback* dado pelos testes automáticos no desenvolvimento de suas questões. Assim sendo, o questionário foi dividido em duas seções, sendo elas: (1) seção de informações pessoais e (2) seção sobre aptidões em programação.

[3] https://github.com/almdanddre/oraculo

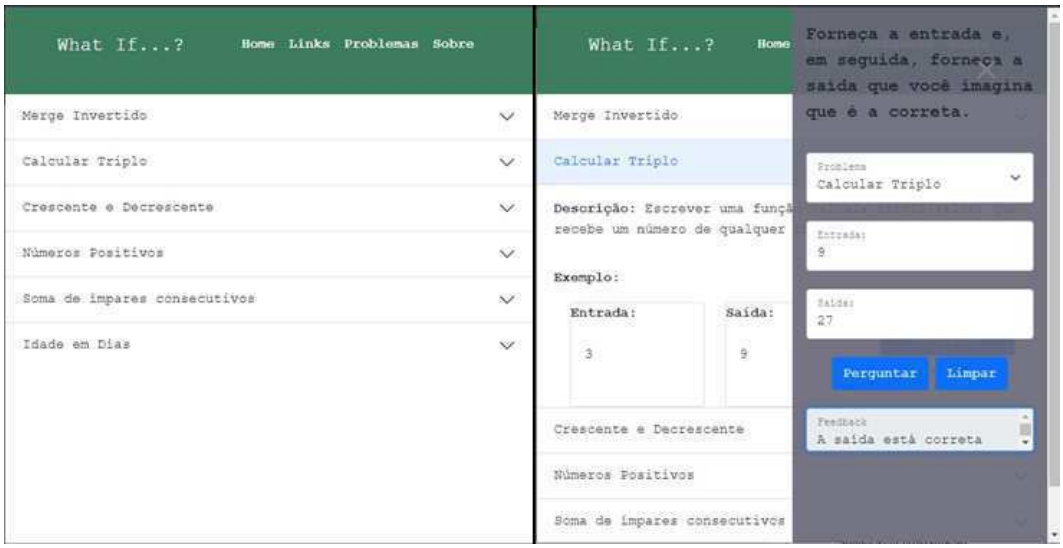


Figura 4.5: Telas de interação do oráculo.

Na seção 1 do questionário, buscou-se coletar informações dos participantes como cidade e faixa etária. Já na seção 2, buscou-se identificar o nível de concordância com algumas afirmações utilizando a escala Likert, além de uma múltipla escolha sobre o quanto o aluno trabalha sobre um exercício. As afirmações são apresentadas na tabela C.1.

Após a aplicação do estudo, pretendemos verificar a opinião dos alunos sobre a efetividade do suporte oferecido pela estratégia alvo do trabalho: a interação com o oráculo através de pares de testes de entrada e saída. Portanto, possuindo apenas três questões e um campo para comentários livres, o questionário foi desenhado de acordo com o apresentado na tabela D.1.

Capítulo 5

Resultados

Neste capítulo são apontados os resultados referentes a esta pesquisa. Na Seção 5.1 apresentamos os resultados do estudo preliminar que consistiu na classificação dos erros mais comuns em exercícios de programação, em busca de evidências de que a possível falta de compreensão dos enunciados impacta na corretude dos exercícios. Na Seção 5.2 exprimimos os resultados concernentes a aplicação do oráculo em turmas iniciantes em programação.

5.1 Mapeamento de erros sobre os exercícios de programação

Para classificar as respostas dos alunos de acordo com a taxonomia de erros, foi realizado um estudo de caso retrospectivo considerando quatro questões de mini testes aplicados na disciplina de Programação I da UFCG, as quais foram apresentadas aos alunos na forma de especificações bem definidas, ou seja, enunciados bem descritos de maneira a evitar ambiguidades; e cujas respostas foram enviadas por meio do sistema de submissão automática de exercícios de programação. Conhecendo os resultados dos testes funcionais executados pelo sistema e ainda, a nota dada pelos professores para cada questão, o foco do estudo foi nas submissões com erro. Os enunciados podem ser consultados no Apêndice A.

Desta forma, primeiramente as respostas foram corrigidas funcionalmente, de maneira a classificar cada uma delas de acordo com a taxonomia de erros citada anteriormente. Na segunda etapa, foi levantado o modelo de erros, ou seja, o mapeamento dos erros que os

alunos cometeram para a falta de entendimento da especificação, identificando os casos de teste que captaram esse comportamento.

Nesta subseção são ilustrados os tipos de erros propostos na literatura que foram identificados nos códigos analisados, bem como uma análise quantitativa da ocorrência destes.

5.1.1 Análise Qualitativa

Tendo em mãos as respostas dos alunos aos problemas, cada uma delas foi verificada em relação a uma solução estabelecida, um conjunto de casos de teste definido e os detalhes dados na especificação para identificar se o aluno teve algum equívoco ou cometeu algum erro. A seguir são ilustrados cada tipo de erro definido na literatura.

Erro de conhecimento: o aluno utiliza *loops* aninhados na tentativa de fazer a operação solicitada, porém não é feita da maneira correta. Além disso, o enunciado demanda uma nova lista para armazenar o resultado, o que não é feito. Ver Figura 5.1.

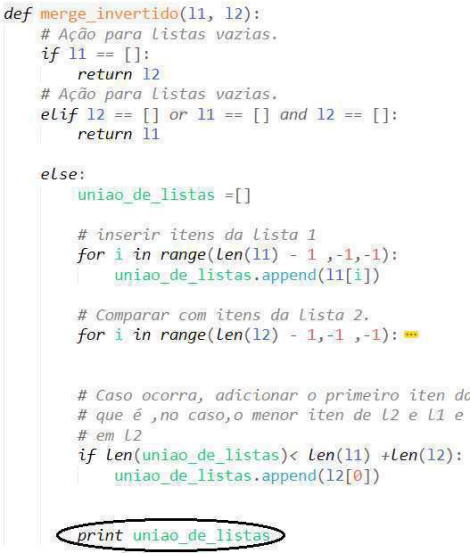
```
def merge_invertido(l1, l2):
    for i in range(len(l2)):
        for j in range(len(l1)): # @com
            k = j + 1
            if l2[i] < l1[j]:
                l1.append(l2[i])
                l1[j] = l1[k]
                k += 1

    return l1
```

Figura 5.1: Exemplo de erro de conhecimento.

Erro baseado em habilidade: aqui observamos que o aluno utilizou o *print* para mostrar os valores da lista final quando o problema pedia que fosse retornado. Ver Figura 5.2.

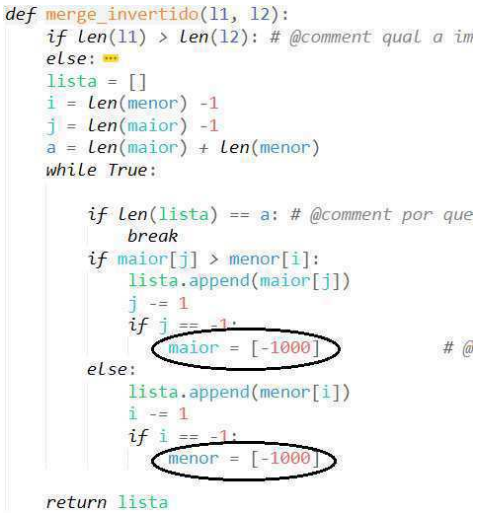
Erro baseado em regra #1: neste ponto, o aluno altera manualmente o valor da copia de uma das listas para sinalizar que não há mais elementos (da lista original) a serem comparados. Ver Figura 5.3.

Erro baseado em regra #2: O aluno utiliza um método de ordenação próprio da linguagem,



```
def merge_invertido(l1, l2):
    # Ação para listas vazias.
    if l1 == []:
        return l2
    # Ação para listas vazias.
    elif l2 == [] or l1 == [] and l2 == []:
        return l1

    else:
        uniao_de_listas =[]

        # inserir itens da lista 1
        for i in range(len(l1) - 1 ,-1,-1):
            uniao_de_listas.append(l1[i])

        # Comparar com itens da lista 2.
        for i in range(len(l2) - 1,-1 ,-1): ...


        # Caso ocorra, adicionar o primeiro iten da
        # que é ,no caso,o menor iten de l2 e l1 e
        # em l2
        if len(uniao_de_listas)< len(l1) +len(l2):
            uniao_de_listas.append(l2[0])

        print uniao_de_listas
```

Figura 5.2: Exemplo de erro baseado em habilidade.



```
def merge_invertido(l1, l2):
    if len(l1) > len(l2): # @comment qual a im
    else: ...
    lista = []
    i = len(menor) -1
    j = len(maior) -1
    a = len(maior) + len(menor)
    while True:

        if len(lista) == a: # @comment por que
            break
        if maior[j] > menor[i]:
            lista.append(maior[j])
            j -= 1
            if j == -1:
                maior = [-1000]              # @
        else:
            lista.append(menor[i])
            i -= 1
            if i == -1:
                menor = [-1000]

    return lista
```

Figura 5.3: Exemplo de erro baseado em regra #1.

porém o enunciado descreve a não aceitação. Ver Figura 5.4.

5.1.2 Análise Quantitativa

No total, foram avaliadas 83 submissões, das quais 60% destas apresentaram algum erro. A distribuição das respostas dos alunos de acordo com a classificação dos erros pode ser observada na Tabela 5.1. É importante destacar que mais de um tipo de erro foi detectado em algumas submissões, uma vez que podem estar associados dependendo do cenário em

```
def merge_invertido(l1,l2):
    l3 = []
    while l1 and l2:
        if l1[0] == l2[0]:
            l3.append(l1.pop(0))
        elif l1[0] < l2[0]:
            l3.append(l1.pop(0))
        else:
            l3.append(l2.pop(0))
    if l1:
        l3.extend(l1)
    if l2:
        l3.extend(l2)
    l3 = l3[::-1]  # @comment não
    return l3
print merge_invertido(l1,l2)
```

Figura 5.4: Exemplo de erro baseado em regra #2.

questão e do código produzido.

Tabela 5.1: Classificação dos erros encontrados nas submissões.

Tipo de Erro	**Frequência Absoluta**	**Frequência Relativa**
Conhecimento	6	0,09
Habilidade	8	0,12
Regra	54	0,79

É possível perceber que o erro baseado em regra é o mais frequente nas submissões, em torno de 79%. Esta mesma análise pode ser feita para os exercícios isoladamente, onde é possível analisar a ocorrência dos erros em cada cenário e perceber que erros baseados em regras são consideravelmente frequentes (Figura 5.5).

5.1.3 Mapeando os erros para os casos de teste

Neste ponto, o objetivo foi analisar as respostas e indicar os casos de teste que correspondem às respectivas falhas no entendimento da especificação ou outros tipos de erro, ainda fornecendo uma descrição detalhada do erro. Nas tabelas (de E.1 a E.4), **ID** representa a identificação do aluno; **TR/TE** representa a razão entre o número de testes rejeitados e o

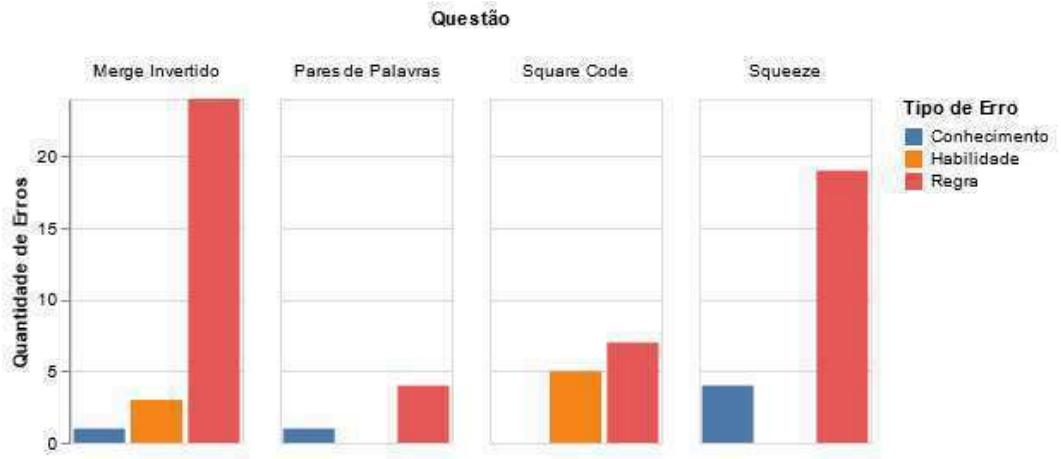


Figura 5.5: Quantidades e classificação dos erros encontrados nos exercícios.

número de testes executados; e **Classificação** especifica o(s) tipo(s) de erro(s) de acordo com a descrição.

Como observado nas tabelas, é possível isolar os casos de teste que conseguem capturar os erros que possivelmente se referem a dificuldades na compreensão dos enunciados; e presumir que se os alunos propuserem entradas e saídas corretas para tais tipos de testes, antes mesmo de escrever o código, dominarão esses pontos críticos para a compreensão do problema e possivelmente não cometerão esses erros. Esta hipótese embasa a realização dos trabalhos seguintes, os quais buscaram analisar se alunos que tentam entender as especificações dos problemas por meio da definição e execução de testes apresentam melhor desempenho em suas atividades.

5.2 A utilização do oráculo na resolução de problemas

Nesta seção descrevemos a condução dos resultados referentes ao *design* do estudo apresentação na subseção 4.2.

Iniciamos com a caracterização do alunos, obtida por meio do questionário aplicado em um primeiro momento; em seguida apresentamos os resultados relacionados às métricas coletadas, do ponto de vista quantitativo e qualitativo; por fim, retratamos o *feedback* dos alunos, coletado pelo questionário aplicado ao fim do estudo.

O estudo foi realizado com duas turmas iniciantes em programação em sessões diferentes, onde participaram efetivamente um total de 45 alunos. Em ambas as turmas o processo

de resolução dos exercícios foi o mesmo:

- Na primeira sessão, os alunos tiveram que resolver o primeiro exercício da forma usual e o segundo exercício com auxílio do oráculo.
- Na segunda sessão, os alunos tiveram que resolver o primeiro exercício através da estratégia TDD e o segundo exercício com auxílio do oráculo.

5.2.1 Caracterização dos participantes

Conforme descrito na subseção 4.2.4, construímos um questionário para traçar um perfil dos alunos que aceitaram participar do estudo, totalizando cinco afirmações.

Os alunos foram classificados por faixa etária. Na faixa de 18-24 anos estavam cerca de 85% dos alunos, enquanto 10% e 5% estavam entre 25-34 anos e 14-17 anos, respectivamente.

Sobre a primeira afirmação "Eu já tinha conhecimento em programação antes do curso", a Figura 5.6 apresenta os resultados. Relembrando, e isto é valido para até a quarta afirmação, os alunos marcaram em uma escala de 1 a 5 (na qual 1 indica total discordância e 5 indica total concordância) o nível de concordância. Aproximadamente 50% dos alunos afirmaram não ter conhecimento em programação antes do curso.

Figura 5.6: Quantificando os alunos que já tinham conhecimento prévio em programação.

Figura 5.7: Quantificando os alunos que dominam o conteúdo visto até o momento no curso.

Referente a segunda afirmação "Eu domino visto até o momento"(figura 5.7 , não houveram respostas nos pontos 1 e 2 da escala, o que revela que os alunos possuem certo nível de conhecimento sobre o que foi apresentado até o momento de realização deste estudo.

Nas respostas sobre terceira afirmação "Eu estudo programação apenas resolvendo as questões de listas de exercícios"(figura 5.8), observamos que os resultados estão mais bem distribuídos, com 36% dos alunos em total concordância com a afirmação.

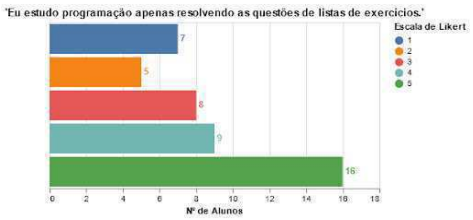


Figura 5.8: Quantificando os alunos que estudam programação apenas resolvendo as listas de exercícios.

Figura 5.9: Quantificando os alunos que consideram úteis os resultados dos testes.

No penúltimo item "Considero o resultado dos testes úteis para fazer corretamente a questão"(figura 5.9), quase que a totalidade dos alunos concordaram que o resultado dos testes tem utilidade no processo de resolução de exercícios.

Por fim, na última afirmação "Dedico-me a fazer uma questão de programação até que..."(figura 5.10), os alunos precisaram assinalar dentro de algumas opções o quanto exploram os exercícios propostos. Com destaque, 33% dos alunos afirmam que costumam melhorar sua solução mesmo que esteja passando em todos os casos de teste; enquanto que 31% afirmam que trabalham na solução até que seja aceita nos casos de teste exemplo e em outros que imagina.

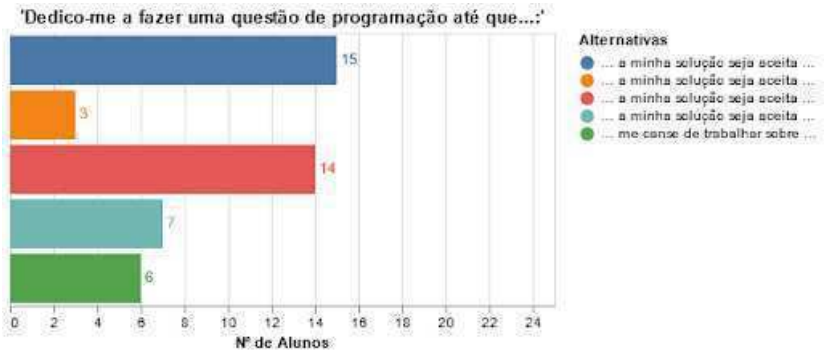


Figura 5.10: Quantificando o nível de empenho dos alunos sobre os exercícios.

De modo geral, os dados coletados pelo questionário permitem considerar que os alunos que participaram do estudo dominam de maneira mediana o conteúdo visto até o momento e

que consideram importante os resultados providos dos testes sobre suas soluções, explorando as possibilidades e os diversos cenários que seus programas precisam abranger para atender ao que é especificado.

5.2.2 Análise das métricas coletadas

Como mencionado, com base em algumas métricas, objetivamos analisar o desempenho dos alunos realizando o contraste entre diferentes estratégias de resolução de exercícios de programação, sobretudo a estratégia que buscamos viabilizar tendo o oráculo como suporte. Iniciamos a análise com a métrica tempo de resolução, seguida pelo número de submissões e, por último, a taxa de corretude funcional das respostas.

Tempo de resolução (T)

Neste ponto analisamos o tempo total de resolução de cada um dos quatro exercícios propostos com o intuito de investigar, principalmente, se alunos que utilizaram o oráculo convergem mais rapidamente para as respectivas soluções corretas.

No gráfico de boxplots e violinos a seguir (figura 5.11) são apresentados os resultados. Os violinos representam a concentração (ou densidade) das informações. Basicamente, as curvas mais largas indicam uma maior frequência de pontos. Por convenção, como os alunos utilizaram o oráculo em dois momentos, nomeamos o método de "Oráculo A"e "Oráculo B".

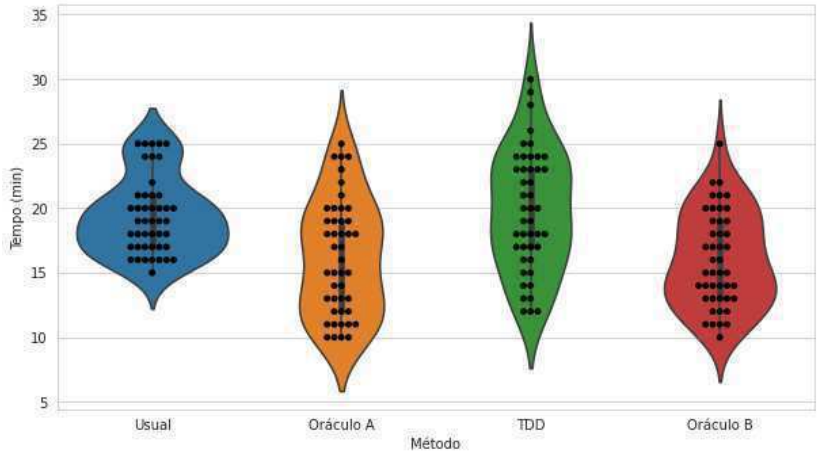


Figura 5.11: Distribuição do tempo para cada método aplicado.

É possível observar que, tomando como *baseline* o maior tempo nas estratégias usual

e TDD, o tempo de resolução utilizando o oráculo não ultrapassa este limite. Isto revela que o oráculo não chega a influenciar negativamente no processo de resolução dos exercícios, mesmo sendo uma ferramenta extra que os alunos precisam utilizar para alcançar suas soluções.

Em números, a tabela 5.2 revela que os tempos para submissão da última versão do código são razoavelmente menores para as estratégias de resolução que utilizam o oráculo como apoio. Além disso, definimos intervalos de confiança de 95% para a mediana.

Tabela 5.2: Sumarização dos resultados relativos ao tempo de resolução dos exercícios.

Método	Tempo (min)		
	Média	Mediana	ICs
Usual	19,44 minutos	19 minutos	[18, 20]
Oráculo A	16,22 minutos	16 minutos	[13, 18]
TDD	18,67 minutos	18 minutos	[16, 21]
Oráculo B	16 minutos	15 minutos	[14, 17]

Para realizar o teste de Wilcoxon para amostras pareadas, precisamos realizar comparações dois a dois. Sendo assim, realizamos o teste de hipótese para o cenário (1) Usual vs. Oráculo A e (2) TDD vs. Oráculo B. Considerando um valor-p = 0,05, obtivemos um p-valor = 0,0009 para o caso (1) e p-valor = 0,003 para o caso (2). Como o p-valor obtido em ambos os casos foi menor que o estabelecido, podemos rejeitar a hipótese nula e considerar que existe uma diferença significativa na variável tempo de resolução, favorecendo o oráculo. Formalmente, ou seja, os alunos que resolvem as questões utilizando o oráculo como apoio apresentam **T** menor que na estratégia usual e na estratégia TDD.

Outro ponto de vista que pode ser explorado é sobre como, individualmente, se comportou o tempo de codificação de cada aluno nas duas estratégias em comparação direta. Como tivemos um n = 45 alunos, plotamos em dois gráficos de colunas empilhadas (Figuras 5.12 e 5.13) o tempo gasto em cada estratégia.

Na comparação Usual vs. Oráculo A percebemos que 69% dos alunos apresentam tempo de resolução menor se utilizando o oráculo, enquanto que 60% também apresentam tempo menor para esta mesma estratégia na comparação TDD vs. Oráculo B. Em alguns casos,

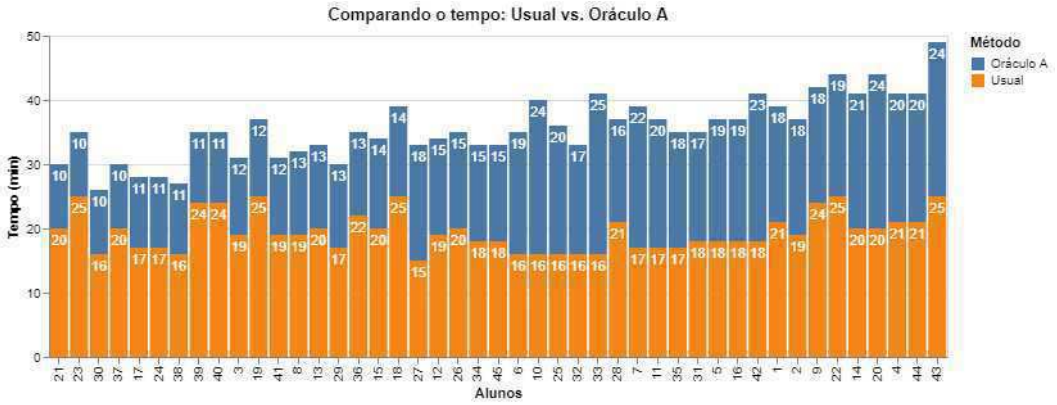


Figura 5.12: Tempo gasto, por aluno, para resolver cada questão proposta na primeira sessão do estudo.

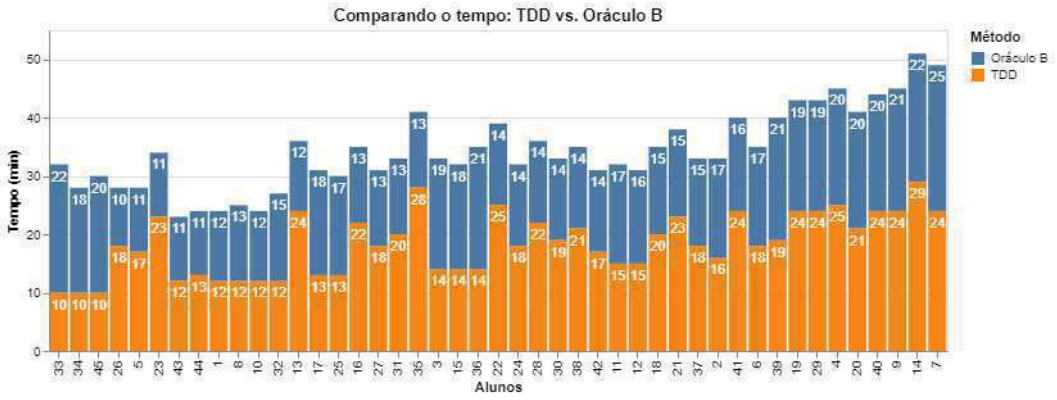


Figura 5.13: Tempo gasto, por aluno, para resolver cada questão proposta na segunda sessão do estudo.

alunos gastaram quase que metade do tempo que levaram para resolver o primeiro exercício; assim como houveram alunos que passaram mais tempo para alcançar a solução quando utilizando o oráculo, o que pode ser justificado por uma série de fatores, como complexidade do exercício ou fatores externos.

Número de submissões (NSub)

Com esta variável, o objetivo também foi de analisar se os alunos que utilizaram o oráculo convergem mais rapidamente para a solução aceita em todos os casos de teste, mas agora observando o número de vezes que submeteram à plataforma até a última submissão. Importante destacar que houve um processo minucioso na análise das submissões, de maneira a desconsiderar para a contagem final de submissões aqueles código que foram submetidos

mais de uma vez sem qualquer alteração.

Iniciando com uma análise quantitativa, na Figura 5.14 podemos fazer uma análise sobre a métrica. Observamos que o espectro de número de submissões nas abordagens que não utilizam o oráculo é razoavelmente maior que nas demais, além da média também apresentar o mesmo comportamento.

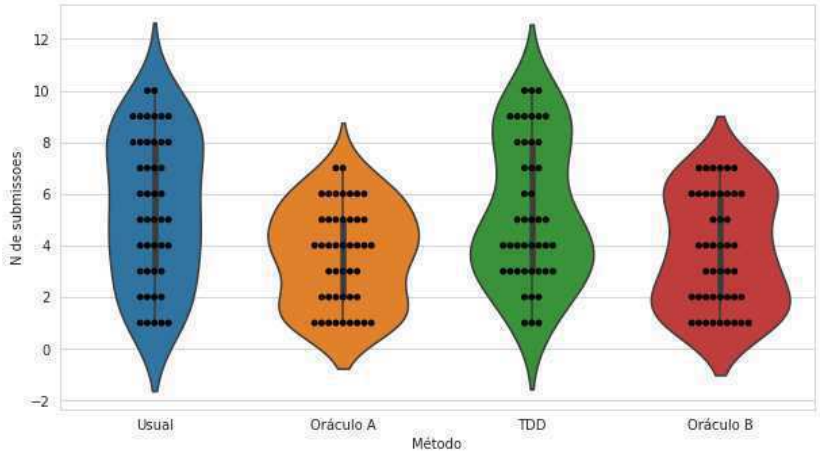


Figura 5.14: Distribuição do número de submissão para cada método aplicado.

Assim como com o tempo de resolução, para realizar o teste de Wilcoxon para amostras pareadas, precisamos realizar comparações dois a dois. Sendo assim, realizamos o teste de hipótese para o cenário (1) Usual vs. Oráculo A e (2) TDD vs. Oráculo B. Considerando um valor-p = 0,05, obtivemos um p-valor = 0,001 para o caso (1) e p-valor = 0,01 para o caso (2). Como o p-valor obtido em ambos os casos foi menor que o estabelecido, podemos rejeitar a hipótese nula e considerar que existe uma diferença significativa na variável número de submissões, favorecendo o oráculo. Formalmente, ou seja, os alunos que resolvem as questões utilizando o oráculo como apoio apresentam **NSub** menor que na estratégia usual e na estratégia TDD.

Em números, a tabela 5.3 revela que o número de submissões até a última versão do código é razoavelmente menor para as estratégias de resolução que utilizam o oráculo como apoio. Além disso, definimos intervalos de confiança de 95% para a mediana.

Outro ponto de vista que pode ser explorado é sobre como, individualmente, se comportou o número de submissões de cada aluno nas duas estratégias em comparação direta. Utilizamos o mesmo tipo de visualização que para a variável tempo de resolução, conforme

Tabela 5.3: Sumarização dos resultados relativos ao número de submissões dos exercícios.

Método	Nº de Submissões		
	Média	Mediana	ICs
Usual	5	5	[4, 7]
Oráculo A	4	4	[3, 5]
TDD	5	5	[4, 6]
Oráculo B	4	4	[2, 5]

ilustrado nas Figuras 5.15 e 5.16.

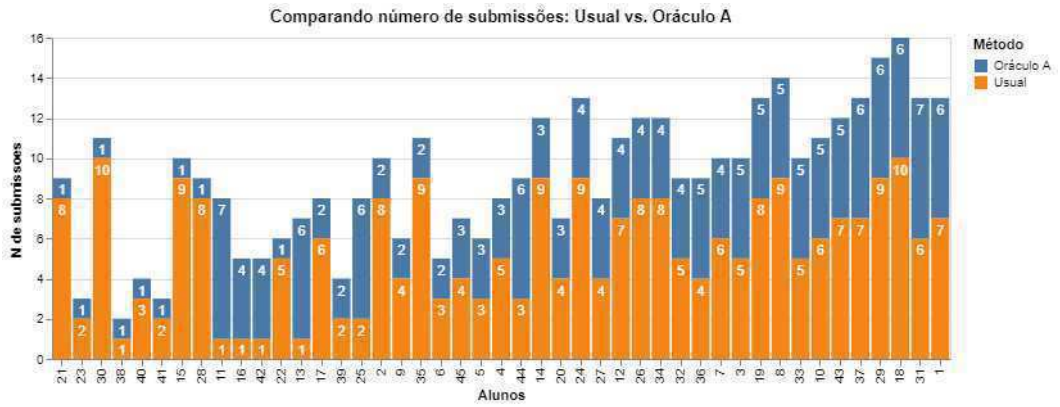


Figura 5.15: Número de submissões, por aluno, para resolver cada questão proposta na primeira sessão do estudo.

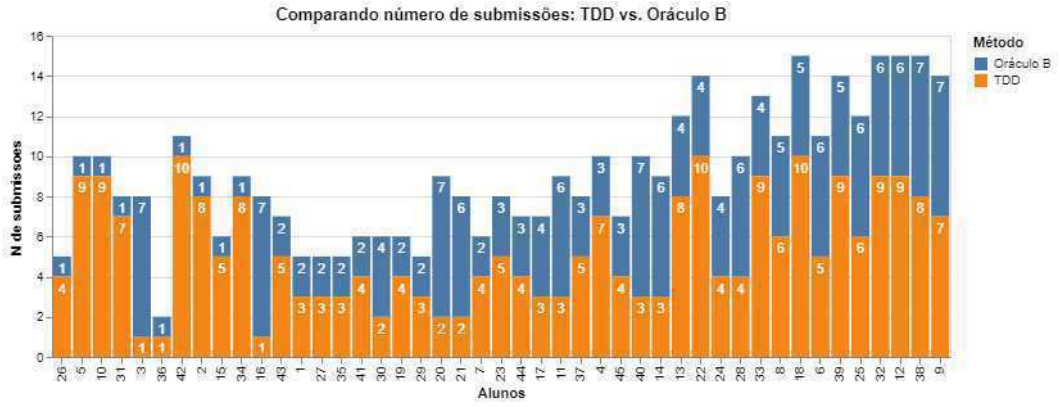


Figura 5.16: Número de submissões, por aluno, para resolver cada questão proposta na segunda sessão do estudo.

Na comparação Usual vs. Oráculo A percebemos que 69% dos alunos apresentam NSub menor se utilizando o oráculo, enquanto que 67% também apresentam NSub menor para esta mesma estratégia na comparação TDD vs. Oráculo B. É perceptível que, apesar da análise conduzida para desconsiderar as submissões consecutivas sem alterações visíveis no código, em alguns casos os alunos realizaram muito mais submissões quando utilizando o oráculo e é isto que nos leva a conduzir uma análise qualitativa com o intuito de entender as particularidades desses casos e sobre a representatividade dos mesmos.

Para a figura 5.15 temos que os alunos 11, 16, 42 e 25 apresentaram NSub muito maior na estratégia que utiliza o oráculo se comparada com a forma usual. Analisando de maneira geral, os alunos tiveram dificuldade no emprego dos operadores aritméticos e sobre como aplicar a função *print()* de acordo com a formatação pedida. O trecho de código da Figura 5.17 representa a forma como os alunos procederam para resolver o exercício e a dificuldade de organização dos operadores aritméticos para alcançar a formatação.

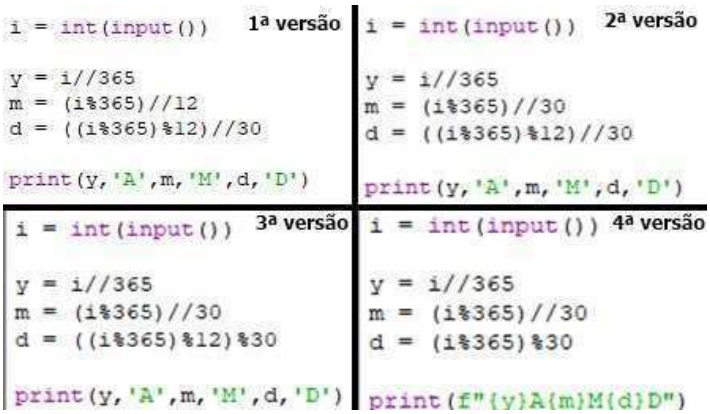


Figura 5.17: Dificuldade na aplicação dos operadores e saída conforme especificada.

Para a figura 5.16, temos que os alunos 3, 16, 20 e 40 apresentaram NSub muito maior na estratégia que utiliza o oráculo se comparada com a forma TDD. Apesar do oráculo, os alunos tiveram dificuldade no entendimento de que o exercício pedia para realizar a soma dos números ímpares consecutivos de acordo com um intervalo, mas que os valores a serem somados não podiam ser os que definiam o intervalo e sim os valores que ficam entre esse intervalo. Além disso, os alunos sentiram a dificuldade de compreender que os números dados como entrada poderiam estar na ordem crescente ou decrescente, e que essa variação necessitaria de verificação para organizar a execução do laço *for*.

O trecho de código da Figura 5.18 representa a forma como os alunos procederam para resolver o exercício e a dificuldade de organização dos possíveis cenários para o exercício.

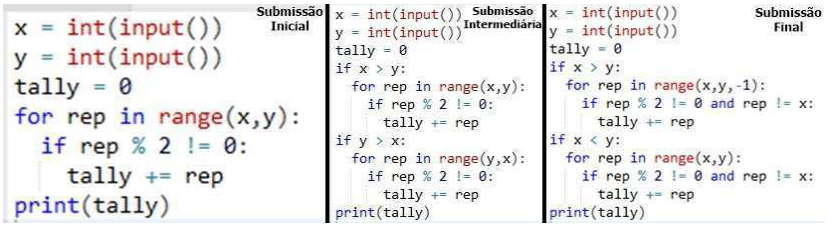


Figura 5.18: Dificuldade na representação de mais de um cenário para o exercício.

Conduzimos também uma análise sobre os exercícios que não tiveram o suporte do oráculo, de maneira a identificar quais as dificuldades que os alunos enfrentaram.

Sobre o exercício aplicado empregando o método usual, o Calculando Desconto, a principal dificuldade dos alunos foi calcular valor final do produto, uma vez que seria necessário subtrair o valor do desconto do valor inicial do mesmo.

No exercício aplicando o método TDD, o Calculando IMC, entre as principais dificuldades estão (1) os operadores de comparação a serem utilizados para considerar o intervalo correto do IMC, (2) confusão nos tipos das variáveis peso e altura durante a leitura e (3) atenção ao resultado do cálculo do IMC e a exibição da mensagem de situação situação - os *prints* deveriam ser iguais ao apresentado na especificação do problema. Os trechos de código a seguir ilustram estes e algumas outras insistências presentes nas soluções.

```
altura=int(input())
peso=float(input())
imc= peso/ (altura**2)
print(imc)
if imc < 18.5:
  print('A baixo do peso')
elif 18.5 >= imc < 25:
  print('Peso normal')
elif 25>= imc < 30:
  print("Sobrepeso")
elif 30>= imc < 35:
  print('Obesidade grau 1')
elif 35>= imc < 40:
 print("Obesidade grau 2")
elif  imc > 40:
 print("Obesidade grau 3")
```

Figura 5.19: Tipo incorreto para a variável altura e inconsistência na mensagem exibida.

```
peso = float(input())
altura = float(input())
imc = peso / (altura ** 2)
print(int(imc))
if imc < 18.5:
    print("Abaixo do peso")
elif imc > 18.5 and imc < 25:
    print("Peso normal")
elif imc >= 25 and imc < 30:
    print("Sobrepeso")
elif imc > 30 and imc < 35:
    print("Obesidade grau 1")
elif imc > 35 and imc < 40:
    print("Obesidade grau 2")
elif imc > 40:
    print("Obesidade grau 3")
```

Figura 5.20: *Casting* para tipo inteiro da variável que armazena o IMC.

```
peso = float(input())
altura = float(input())
imc = peso / altura ** 2
if imc <= 18.5:
  print(imc)
  print("Abaixo do peso")
if imc <= 25 and imc > 18.5:
  print(imc)
  print("Peso normal")
if imc <= 30 and imc > 25 :
  print(imc)
  print("Sobrepeso")
if imc <= 35 and imc > 30:
  print(imc)
  print("Obesidade grau 1")
if imc <= 40 and imc > 35:
  print(imc)
  print("Obesidade grau 2")
if imc < 40:
  print(imc)
  print("Obesidade grau 3")
```

Figura 5.21: Utilização de múltiplos IFs.

Número de testes aceitos (AT)

Sobre esta variável, isoladamente, pouco se pode dissertar sobre os resultados. Todos os alunos que participaram do estudo atingiram uma taxa de correção funcional de 100% em todos os exercícios. Sendo assim, não foi conduzido um teste de hipótese para verificar a conjectura referente a esta variável.

Respostas às questões de pesquisa

Levando em consideração os resultados alcançados, as respostas às questões de pesquisa são apresentadas na Tabela 5.4.

Tabela 5.4: Questões de pesquisa e suas respectivas respostas.

Questão de Pequisa	Resposta
QP1: Os alunos que tentam compreender as especificações dos problemas utilizando oráculo, criando e executando testes, produzem mais programas funcionalmente corretos?	Inconclusivo. Todos os alunos que participaram do estudo submeteram soluções que foram aceitas em todos os testes ocultos. Dessa forma, não foi possível verificar que os alunos que utilizaram o oráculo obtiveram uma taxa de corretude funcional maior.
QP2: Os alunos que tentam compreender as especificações dos problemas utilizando oráculo, criando e executando testes, convergem para a solução correta mais rapidamente?	Sim. Como observado através dos gráficos e corroborado pelo teste de hipóteses, podemos afirmar que os alunos que utilizaram o oráculo alcançam a solução correta mais rapidamente, despendendo menos tempo e submetendo a solução menos vezes.

5.2.3 *Feedback* dos alunos

Em contrapartida aos resultados nativos das análises quantitativa e qualitativa, ao final do estudo os alunos reportaram suas impressões sobre as estratégias através de um formulário. Com este, objetivamos não apenas coletar opiniões dentro de um espectro definido de alternativas, mas também dedicar um espaço livre para que os alunos pudessem discorrer um pouco sobre como o oráculo pode ser melhorado.

Iniciando com a opinião em termos de utilidade e eficácia do oráculo (Figura 5.22), 90% dos alunos julgaram como útil ou importante para planejar melhor suas soluções; corroborando com a escala Likert associada no segundo item do questionário (Figura 5.23). Por fim, quando perguntados com relação a preferência entre as três estratégias, 67% dos alunos apontaram o oráculo (Figura 5.24).

Quanto as comentários abertos, os alunos sugeriram principalmente a possibilidade de interação utilizando apenas as entradas dos problemas de maneira a tomar conhecimento da saída correta, não apenas do *feedback* sobre a saída fornecida corresponder com a entrada.

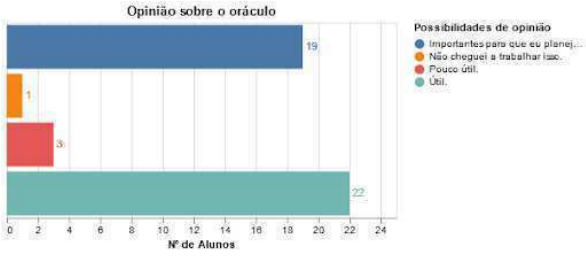


Figura 5.22: Quantificando a opinião dos alunos sobre a utilidade do oráculo.

Figura 5.23: Quantificando a opinião dos alunos através da escala Likert.

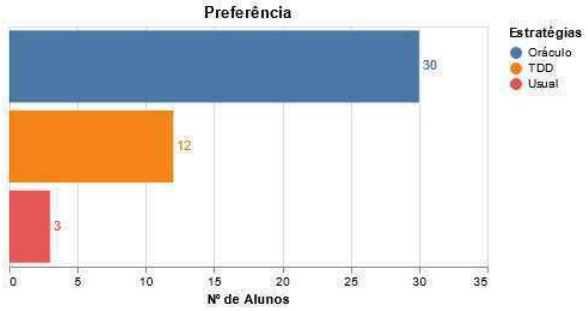


Figura 5.24: Quantificando a preferência sobre as estratégias aplicadas.

Além disso recomendaram também o enriquecimento do *feedback*, pode vir a ajudar os alunos que estão iniciando em programação a entender onde estão errando, tendo uma utilidade muito mais evidente do que simplesmente criar os próprios testes para executar na solução. Os comentários podem ser encontrados no apêndice F.

Capítulo 6

Conclusão

Este capítulo apresentada as conclusões que podemos traçar de acordo com o resultado deste trabalho. A seção 6.1 apresenta a discussão dos resultados, a seção 6.2 menciona as ameaças a validade e, por fim, a seção 6.3 aponta as perspectivas para trabalhos futuros.

6.1 Discussão

A presente dissertação destaca a análise da efetividade de uma estratégia baseada em testes como meio de melhorar o entendimento de enunciados de exercícios de programação. No primeiro estudo, o mapeamento de erros sobre os exercícios, buscamos compreender a natureza dos erros mais comuns de maneira a verificar se estes estavam atrelados ao pouco conhecimento em programação ou à complexidade dos enunciados, de tal maneira que não compreendê-los em sua totalidade impactaria na resolução correta. Observamos que, dentro da taxonomia aplicada, pudemos mapear o tipo de erro, mais precisamente os erros que têm relação com dificuldade de compreensão, para os testes que tratam justamente de determinados detalhes da especificação que os alunos podem não ter cedido a devida atenção e acabou resultando nos testes não aceitos.

Posto isto, pudemos levantar o pressuposto de que se determinados testes escritos pelos instrutores têm o objetivo de "medir"a atenção dos alunos quanto aos detalhes da especificação do problema, fornecer um mecanismo para que o aluno "teste seu entendimento"antes de começar a codificar pode favorecer que os alunos reflitam melhor sobre como sua solução deve se comportar mediante os diversos cenários possíveis. Nomeamos este mecanismo de

"oráculo"e analisamos seu efeito sobre os alunos.

O oráculo foi desenvolvido de tal forma que o "teste de entendimento"pudesse ser feito através de testes de entrada e saída, ou seja, o aluno fornece a entrada para o problema e, como base no que compreendeu da especificação, envia também a saída que imagina que o programa deve exibir para aquela entrada. Como esta ferramenta objetivamos oferecer suporte no primeiro passo do processo de resolução de problemas e obtivemos resultados que retratam esta estratégia como sendo promissora, no que se refere a alcançar soluções corretas mais rapidamente. Além disso, os *feedbacks* fornecidos pelos alunos foram positivos e construtivos de tal forma que guiam alguns dos trabalhos futuros.

6.2 Ameaças à Validade

Cabe ressaltar algumas ameaças a validade com relação a aplicação do oráculo. No que tange a validade interna podemos apontar a execução do estudo em dias diferentes, o que permitiu que os alunos dispusessem de uma janela de tempo para fixar melhor os conteúdos do curso. Atrelado a isto está a heterogeneidade dos participantes, uma vez que o nível de conhecimento das turmas se mostrou misto. Para minimizar estas questões, nos certificamos em executar o estudo em um momento do curso em que todos os alunos já dispusessem do conhecimento prévio para aplicar em seus exercícios de programação.

Com relação a validade externa, podemos mencionar o viés da amostra, uma vez que consideramos duas turmas de programação introdutória que não necessariamente representam a população. A situação pandêmica causada pelo Coronavírus (COVID-19) dificultou o acesso presencial aos alunos e a própria execução do estudo, sendo necessário traçá-lo considerando amostras pareadas. Isto permitiu obter resultados significativos com um tamanho menor de amostra, além de reduzir o tempo necessário para realizar o estudo.

Ainda podemos mencionar também o tempo limite de resolução de cada exercício pode ter impactado no processo reflexivo dos alunos; e também a necessidade de utilizar mais de uma ferramenta durante a resolução dos exercícios, produzindo uma carga de trabalho maior.

6.3 Trabalhos Futuros

Tomando como referência os comentários dos alunos e vislumbrando algumas melhorias no *design* do estudo que consiste na aplicação do oráculo, podemos traçar algumas trabalhos futuros com o intuito de tornar ainda mais evidente os benefícios da estratégia. São eles:

- Reexecução do estudo: composto por apenas um momento, de maneira a minimizar os efeitos do processo de maturação (ameaça interna). Além disso, levando em consideração o agrupamento dos alunos com nível de conhecimento em programação mais aproximado, para um melhor comparativo dos resultados; e também como forma de explorar melhor a métrica que teve resultado inconclusivo;

- Aperfeiçoamento do *feedback* oferecido pelo oráculo: com o objetivo de possivelmente fornecer *hints* mais específicos sobre o processo de resolução dos exercícios sem tornar a solução tão evidente;

- Inclusão de outras formas de interação no oráculo: como comentado pelos alunos, o oráculo também poderia permitir a interação com apenas um elemento do par entrada-saída, por exemplo, o aluno poderia fornecer apenas a entrada ou apenas a saída do problema, de maneira a ter um *feedback* mais voltado para o *design* da solução, não apenas a confirmação da correspondência entre os elementos.

Bibliografia

[1] Kirsti Ala-Mutka. Problems in learning and teaching programming-a literature study for developing visualizations in the codewitz-minerva project. *Codewitz needs analysis*, 20, 2004.

[2] Kirsti Ala-Mutka and H-M Jarvinen. Assessment process for programming assignments. In *IEEE International Conference on Advanced Learning Technologies, 2004. Proceedings.*, pages 181–185. IEEE, 2004.

[3] Kirsti M Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer science education*, 15(2):83–102, 2005.

[4] Eliane Araújo, Matheus Gaudencio, Andrey Menezes, Iury Ferreira, Iara Ribeiro, Alberto Fagner, Lesandro Ponciano, Fabio Morais, Dalton Guerrero, and Jorge Figueiredo. O papel do hábito de estudo no desempenho do aluno de programação. 01 2013.

[5] David P Ausubel. A aprendizagem significativa. *São Paulo: Moraes*, 1982.

[6] Jennifer Ngan Bacquet. Implications of summative and formative assessment in japan–a review of the current literature. *International Journal of Education and Literacy Studies*, 8(2):28–35, 2020.

[7] Ellen F Barbosa, Marco AG Silva, Camila KD Corte, and José C Maldonado. Integrated teaching of programming foundations and software testing. In *2008 38th Annual Frontiers in Education Conference*, pages S1H–5. IEEE, 2008.

[8] Elena García Barriocanal, Miguel-Ángel Sicilia Urbán, Ignacio Aedo Cuevas, and Paloma Díaz Pérez. An experience in integrating automated unit testing practices in an introductory programming course. *ACM SIGCSE Bulletin*, 34(4):125–128, 2002.

[9] Soumya Basu, Albert Wu, Brian Hou, and John DeNero. Problems before solutions: Automated problem clarification at scale. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*, pages 205–213, 2015.

[10] Piraye Bayman and Richard E Mayer. A diagnosis of beginning programmers' misconceptions of basic programming statements. *Communications of the ACM*, 26(9):677–679, 1983.

[11] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[12] Brett A Becker, Cormac Murray, Tianyi Tao, Changheng Song, Robert McCartney, and Kate Sanders. Fix the first, ignore the rest: Dealing with multiple compiler error messages. In *Proceedings of the 49th ACM technical symposium on computer science education*, pages 634–639, 2018.

[13] Jens Bennedsen and Michael Caspersen. Revealing the programming process. volume 37, pages 186–190, 02 2005.

[14] Gunnar Rye Bergersen and Jan-Eric Gustafsson. Programming skill, knowledge, and working memory among professional software developers from an investment theory perspective. *Journal of individual Differences*, 32(4):201, 2011.

[15] Jean Luca Bez, Neilor A Tonin, and Paulo R Rodegheri. Uri online judge academic: A tool for algorithms and programming classes. In *2014 9th International Conference on Computer Science & Education*, pages 149–152. IEEE, 2014.

[16] Ken Blaha, Alvaro Monge, Dean Sanders, Beth Simon, and Tammy VanDeGrift. Do students recognize ambiguity in software design? a multi-national, multi-institutional report. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 615–616. IEEE, 2005.

[17] Jeffrey Bonar and Elliot Soloway. Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human–Computer Interaction*, 1(2):133–161, 1985.

[18] David Boud and Elizabeth Molloy. Feedback in higher and professional education. *Routledge. Taylor & Francis Group. London and New York. Cosh, J.(1998). Peer observation in higher education–a reflective approach. Innovations in Education and Teaching International*, 35(2):171–176, 2013.

[19] Candido Cabo. Fostering problem understanding as a precursor to problem-solving in computer programming. In *2019 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE, 2019.

[20] Simon Caton, Seán Russell, and Brett A Becker. What fails once, fails again: Common repeated errors in introductory programming automated assessments. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, pages 955–961, 2022.

[21] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. On automated grading of programming assignments in an academic institution. *Comput. Educ.*, 41(2):121–131, September 2003.

[22] Peter M Chen. An automated feedback system for computer organization projects. *IEEE Transactions on Education*, 47(2):232–240, 2004.

[23] Michelle Craig, Andrew Petersen, and Jennifer Campbell. Answering the correct question. In *Proceedings of the ACM Conference on Global Computing Education*, pages 72–77, 2019.

[24] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. Intelligent tutoring systems for programming education: a systematic review. In *Proceedings of the 20th Australasian Computing Education Conference*, pages 53–62, 2018.

[25] Draylson Micael De Souza, Jose Carlos Maldonado, and Ellen Francine Barbosa. Progtest: An environment for the submission and evaluation of programming assignments based on testing activities. In *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*, pages 1–10. IEEE, 2011.

[26] Paul Denny, James Prather, Brett A Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. A closer look at metacognitive scaffolding: Solving test cases before

programming. In *Proceedings of the 19th Koli Calling international conference on computing education research*, pages 1–10, 2019.

[27] Paul Denny, James Prather, Brett A Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B Powell. On designing programming error messages for novices: Readability and its constituent factors. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2021.

[28] Chetan Desai, David S Janzen, and John Clements. Implications of integrating test-driven development into cs1/cs2 curricula. *ACM SIGCSE Bulletin*, 41(1):148–152, 2009.

[29] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3):4–es, 2005.

[30] Stephen H Edwards. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing (JERIC)*, 3(3):1–es, 2003.

[31] Stephen H Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 26–30, 2004.

[32] Stephen H Edwards and Manuel A Perez-Quinones. Web-cat: automatically grading programming assignments. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 328–328, 2008.

[33] Emma Enström, Gunnar Kreitz, Fredrik Niemelä, Pehr Söderman, and Viggo Kann. Five years with kattis—using an automated assessment system in teaching. In *2011 Frontiers in education conference (FIE)*, pages T3J–1. IEEE, 2011.

[34] Sandy Garner, Patricia Haden, and Anthony Robins. My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian conference on Computing education-Volume 42*, pages 173–180, 2005.

[35] Anabela Jesus Gomes, Alvaro Nuno Santos, and António José Mendes. A study on students' behaviours and attitudes towards learning to program. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, pages 132–137, 2012.

[36] Jeffrey O Grady. *System engineering planning and enterprise identity*, volume 7. CRC Press, 1995.

[37] Qiang Hao, David H Smith IV, Lu Ding, Amy Ko, Camille Ottaway, Jack Wilson, Kai H Arakawa, Alistair Turcan, Timothy Poehlman, and Tyler Greer. Towards understanding the effective design of automated formative feedback for programming assignments. *Computer Science Education*, 32(1):105–127, 2022.

[38] Tahreem Fatima Hasni and Fakhar Lodhi. Teaching problem solving effectively. *ACM Inroads*, 2(3):58–62, 2011.

[39] Timothy J Hickey. Scheme-based web programming as a basis for a cs0 curriculum. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 353–357, 2004.

[40] Jack Hollingsworth. Automatic graders for programming classes. *Commun. ACM*, 3(10):528–529, October 1960.

[41] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153–156, 2003.

[42] Sheung-Lun Hung, Iam-For Kwok, and Raymond Chan. Automatic programming assessment. *Computers & Education*, 20(2):183–190, 1993.

[43] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research*, pages 86–93, 2010.

[44] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and Renske Weeda. Fostering program comprehension in novice programmers - learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '19, page 27–52, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. Software testing techniques: A literature review. In *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)*, pages 177–182. IEEE, 2016.

[46] Lisa C Kaczmarczyk, Elizabeth R Petrick, J Philip East, and Geoffrey L Herman. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 107–111, 2010.

[47] Rozita Kadar, Naemah Abdul Wahab, Jamal Othman, Maisurah Shamsuddin, and Siti Balqis Mahlan. A study of difficulties in teaching and learning programming: a systematic literature review. *Int. J. Acad. Res. Progress. Educ. Dev*, 10:591–605, 2021.

[48] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *Acm sigcse bulletin*, 37(3):14–18, 2005.

[49] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '01, page 125–180, New York, NY, USA, 2001. Association for Computing Machinery.

[50] Ute Mertens, Bridgid Finn, and Marlit Annalena Lindner. Effects of computer-based feedback on lower-and higher-order learning outcomes: A network meta-analysis. *Journal of Educational Psychology*, 114(8):1743, 2022.

[51] Susanne Narciss and Katja Huth. How to design informative tutoring feedback for multimedia learning. *Instructional design for multimedia learning*, 181195, 2004.

[52] Vicente Lustosa Neto, Roberta Coelho, Larissa Leite, Dalton S Guerrero, and Andrea P Mendonça. Popt: a problem-oriented programming and testing approach for novice students. In *2013 35th international conference on software engineering (ICSE)*, pages 1099–1108. IEEE, 2013.

[53] David J Nicol and Debra Macfarlane-Dick. Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. *Studies in higher education*, 31(2):199–218, 2006.

[54] J Oroma, Herbert Wanga, and Fredrick Ngumbuke. Challenges of teaching and learning computer programming in developing countries: lessons from tumaini university. *DOI: https://doi. org/10.13140/2.1*, 3836, 2012.

[55] Büşra Özmen and Arif Altun. Undergraduate students' experiences in programming: difficulties and obstacles. *Turkish Online Journal of Qualitative Inquiry*, 5(3):1–27, 2014.

[56] Andrew Patterson, Michael Kölling, and John Rosenberg. Introducing unit testing with bluej. *ACM SIGCSE Bulletin*, 35(3):11–15, 2003.

[57] Janice L Pearce, Mario Nakazawa, and Scott Heggen. Improving problem decomposition ability in cs1 through explicit guided inquiry-based instruction. *J. Comput. Sci. Coll*, 31(2):135–144, 2015.

[58] Raymond Pettit and James Prather. Automated assessment tools: Too many cooks, not enough collaboration. *Journal of Computing Sciences in Colleges*, 32(4):113–121, 2017.

[59] G. Polya and J.H. Conway. *How to Solve It: A New Aspect of Mathematical Method*. Penguin mathematics. Princeton University Press, 2004.

[60] James Prather, Raymond Pettit, Brett A Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. First things first: Providing metacognitive

scaffolding for interpreting problem prompts. In *Proceedings of the 50th ACM technical symposium on computer science education*, pages 531–537, 2019.

[61] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 41–50, 2018.

[62] Sandeep Purao and Vijay Vaishnavi. Product metrics for object-oriented systems. *ACM Computing Surveys (CSUR)*, 35(2):191–221, 2003.

[63] Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)*, 18(1):1–24, 2017.

[64] Masura Rahmat, Shahrina Shahrani, Rodziah Latih, Noor Faezah Mohd Yatim, Noor Faridatul Ainun Zainal, and Rohizah Ab Rahman. Major problems in basic programming that influence student performance. *Procedia-Social and Behavioral Sciences*, 59:287–296, 2012.

[65] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer science education*, 13(2):137–172, 2003.

[66] Terry Shepard, Margaret Lamb, and Diane Kelly. More testing should be taught. *Communications of the ACM*, 44(6):103–108, 2001.

[67] Juha Sorva. Notional machines and introductory programming education. *ACM Trans. Comput. Educ.*, 13(2), July 2013.

[68] Juha Sorva and Teemu Sirkiä. Embedded questions in ebooks on programming: useful for a) summative assessment, b) formative assessment, or c) something else? In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pages 152–156, 2015.

[69] James C Spohrer and Elliot Soloway. Alternatives to construct-based programming misconceptions. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 183–191, 1986.

[70] Alistair Sutcliffe and Gordon Rugg. A taxonomy of error types for failure analysis and risk assessment. *International Journal of Human-Computer Interaction*, 10(4):381–405, 1998.

[71] Fabienne M Van der Kleij, Remco CW Feskens, and Theo JHM Eggen. Effects of feedback in a computer-based learning environment on students' learning outcomes: A meta-analysis. *Review of educational research*, 85(4):475–511, 2015.

[72] Jeroen JG Van Merrienboer and John Sweller. Cognitive load theory and complex learning: Recent developments and future directions. *Educational psychology review*, 17(2):147–177, 2005.

[73] Ekaterina Vasilyeva, Mykola Pechenizkiy, and Paul De Bra. Tailoring of feedback in web-based learning: the role of response certitude in the assessment. In *International Conference on Intelligent Tutoring Systems*, pages 771–773. Springer, 2008.

[74] Ashok Kumar Veerasamy, Daryl D'Souza, and Mikko-Jussi Laakso. Identifying novice student programming misconceptions and errors from summative assessments. *Journal of Educational Technology Systems*, 45(1):50–73, 2016.

[75] Jack Wrenn and Shriram Krishnamurthi. Reading between the lines: Student help-seeking for (un) specified behaviors. In *21st Koli Calling International Conference on Computing Education Research*, pages 1–6, 2021.

[76] John Wrenn and Shriram Krishnamurthi. Executable examples for programming problem comprehension. In *Proceedings of the 2019 ACM conference on international computing education research*, pages 131–139, 2019.

[77] Yichen Xie and Dawson Engler. Using redundancies to find errors. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 51–60, 2002.

[78] Stelios Xinogalos. Designing and deploying programming courses: Strategies, tools, difficulties and pedagogy. *Education and Information Technologies*, 21(3):559–588, 2016.

Apêndice A

Exercícios utilizados no mapeamento de erros

Merge Invertido

Escreva a função merge_invertido(l1, l2) que recebe duas listas de inteiros ordenados de forma crescente. A função produz e retorna uma nova lista contendo os elementos das listas ordenados de forma decrescente. Por exemplo, se as listas l1 e l2 forem, respectivamente, [1, 5, 8] e [4, 7, 10, 12], a função deverá retornar a lista [12, 10, 8, 7, 5, 4, 1]. As listas podem não ser do mesmo tamanho e uma delas ou ambas podem ser vazias. Todos os elementos devem aparecer na lista final mesmo que sejam repetidos. Não é permitido usar nenhum método ou função de Python que faça ordenação. Também não é permitido usar nenhum algoritmo de ordenação já existente (bubblesort, insertion sort, selection sort, etc.). Tire proveito da ordenação inicial das listas.

Squeeze

Escreva a função squeeze(lista) que recebe uma lista de inteiros e remove da lista todas os valores duplicados que são vizinhos. Observe que se a lista tiver valores iguais que não estão lado a lado, os valores devem ser mantidos. Observe os asserts para entender melhor a especificação.

Square Code

Escreva a função square_code(mensagem) que recebe uma mensagem de texto (pode conter

letras, espaços e sinais de pontuação - você deve remover o que não for letra) e retorna a mensagem codificada. Por simplicidade, as letras recebidas não possuem acentuação e a mensagem tem pelo menos uma letra válida. Dica: use as funções ceil() e floor() da biblioteca math. A função isalpha() utilizada em strings também pode ajudar.

Pares de palavras com mesma inicial e tamanho

Escreva um programa que identifica quantos pares de palavras de uma lista possuem mesma letra inicial e mesmo tamanho. Seu programa deve, obrigatoriamente, usar a função pares_palavras_com_mesma_inicial_e_tamanho(palavras) que recebe uma string com todas as palavras separadas por e retorna a quantidade de pares que possuem mesma letra inicial e mesmo tamanho. Veja os exemplos e siga a formatação indicada. Importante: você pode assumir que ao menos duas palavras serão informadas na entrada e que todas estarão sempre escritas em minúsculas e não têm caracteres acentuados ou cedilhas.

Apêndice B

Exercícios aplicados no estudo comparativo entre as estratégias

Calculando Desconto (Método Usual)

Na última Black Friday, o gerente de uma loja de perfumes colocou todo o seu estoque em promoção, de acordo com a tabela a seguir:

Código	Condição de Pagamento	Desconto (%)
1	À vista (em espécie)	15
2	Cartão de débito	10
3	Cartão de crédito	5

Construa um programa que solicite ao operador do caixa o preço total da venda, bem como a forma de pagamento. Ao fim, o programa deve informar o valor final a ser pago.

Idade em Dias (Método Oráculo)

Leia um valor inteiro correspondente à idade de uma pessoa em dias e informe-a em anos, meses e dias Obs.: apenas para facilitar o cálculo, considere todo ano com 365 dias e todo mês com 30 dias. Nos casos de teste nunca haverá uma situação que permite 12 meses e alguns dias, como 360, 363 ou 364. Este é apenas um exercício com objetivo de testar raciocínio matemático simples.

Entrada

O arquivo de entrada contém um valor inteiro.

Saída

Imprima a saída conforme exemplo fornecido.

Exemplo de Entrada	Exemplo de Saída
400	1A1M5D
800	2A2M10D
30	0A1M0D

Calculando IMC (Método TDD)

O IMC (Índice de Massa Corporal) é utilizado para avaliar o peso de um indivíduo em relação à sua altura e, como resultado, indica se o indivíduo está com o peso ideal, acima ou abaixo do recomendado. A tabela a seguir indica a situação de um indivíduo adulto em relação ao seu IMC:

IMC	Situação
Abaixo de 18,5	Abaixo do peso
Acima de 18,5 e menor que 25	Peso normal
A partir de 25 e menor que 30	Sobrepeso
Acima de 30 e menor que 35	Obesidade grau 1
Acima de 35 e menor que 40	Obesidade grau 2
Acima de 40	Obesidade grau 3

Para calcular o IMC, é usada a fórmula IMC = peso/(altura)². Construa um programa no qual um usuário informe seu peso e sua altura. A aplicação deve indicar o IMC calculado e a situação do indivíduo.

Soma de Ímpares Consecutivos (Método Oráculo)

Leia 2 valores inteiros X e Y. A seguir, calcule e mostre a soma dos números ímpares entre eles.

Entrada

O arquivo de entrada contém dois valores inteiros.

Saída

O programa deve imprimir um valor inteiro. Este valor é a soma dos valores ímpares que estão entre os valores fornecidos na entrada que deverá caber em um inteiro.

Exemplo de Entrada	Exemplo de Saída
Entrada 1: 6 Entrada 2: -5	Saída: 5

Apêndice C

Questionário aplicado antes do estudo

Tabela C.1: Perguntas propostas no questionário inicial.

Questão	**Alternativas**
1. Eu já tinha conhecimento em programação antes do curso	Likert: 1 a 5
2. Eu domino o conteúdo visto até o momento	Likert: 1 a 5
3. Eu estudo programação resolvendo as questões das listas de exercícios	Likert: 1 a 5
4. Considero o resultado dos testes úteis para fazer corretamente a questão	Likert: 1 a 5
5. Dedico-me a fazer uma questão de programação até que...	a)... a minha solução passe em todos os testes da submissão e eu a considere aceitável. Costumo melhorar a minha solução, mesmo que esteja passando em todos os testes. b)... a minha solução passe em todos os testes da submissão. c)... a minha solução passe nos testes exemplos citados no problema e em outros casos que imagino. d)... a minha solução passe nos testes exemplos citados no problema. e)... me canse de trabalhar sobre ela.

Apêndice D

Questionário aplicado após o estudo

Tabela D.1: Perguntas propostas no questionário final.

Questão	**Alternativas**
Qual a sua opinião sobre a possibilidade de criar a executar testes de entrada e saída (no Oráculo) antes de desenvolver o código referente ao exercício que lhe foi apresentado?	a) Não cheguei a trabalhar isso b) Pouco útil c) Útil d) Importantes para que eu planeje melhor minha solução
Em uma escala de 1 a 5 , tal que a opção 1 representa total reprovação da abordagem apresentada e a opção 5, total aceitação, que nota você atribui para a estratégia em questão (ORÁCULO)?	Likert: 1 a 5
Entre as três formas de desenvolver programas, qual você prefere?	a) Apenas executando os testes ocultos (Usual). b) Apenas criando meus testes e executando-os em minha própria solução (TDD). c) Utilizando primeiro o oráculo, criando e executando testes de entrada e saída (Oráculo).

Apêndice E

Mapeamento dos erros para os casos de teste

Tabela E.1: Mapeamento dos erros para a questão Merge Invertido.

Merge Invertido			
ID	**TR/TE**	**Descrição do(s) erro(s)**	**Classificação**
AL9	2/8	Caso uma das listas seja vazia, a função retorna a outra lista sem fazer o merge.	Regra e Habilidade
AL10	4/8	Problema na construção do laço impede que alguns casos passem. Caso uma das listas seja vazia, a função retorna a outra lista sem fazer o merge.	Regra e Habilidade
AL11	7/8	Aninhamento de laços e confusão com nos índices acabou alterando os elementos das listas. Merge não é feito da maneira correta.	Regra e Conhecimento
AL3	3/8	Não considerou cenários contendo listas vazias, o que ocasiona erro em alguns casos.	Regra
AL12	8/8	Remoção de elementos das listas para compor a lista final e não considerar cenários contendo listas vazias fez com que todos os testes não passassem.	Regra

Tabela E.2: Mapeamento dos erros para a questão Pares de Palavras.

Pares de palavras com mesma inicial e tamanho			
ID	**TR/TE**	**Descrição do(s) erro(s)**	**Classificação**
AL5	2/5	Faltou incluir uma comparação que considerasse a identificação apenas para palavras diferentes.	Regra
AL6	2/5	Fez apenas uma única iteração sobre a lista, o que possibilita a comparação de uma palavra apenas com a próxima.	Regra
AL7	2/5	Faltou fazer a verificação sobre as palavras serem diferentes, o que corrigiria o loop.	Regra
AL8	5/5	Não comparou o caractere inicial de cada string. Forma incorreta de detectar e adicionar as palavras a uma lista.	Regra e Conhecimento

Tabela E.3: Mapeamento dos erros para a questão Square Code.

Square Code			
ID	**TR/TE**	**Descrição do(s) erro(s)**	**Classificação**
AL1	5/6	A codificação é feita da maneira correta, porém não adiciona espaços na mensagem.	Regra
AL2	6/6	Realizou operações e manipulações incoerentes.	Regra e Habilidade
AL3	6/6	Apenas os espaços das mensagens originais são removidos.	Regra
AL4	6/6	Considera um cenário específico para num_linhas e num_colunas. Realizou operações e manipulações incoerentes.	Regra

Tabela E.4: Mapeamento dos erros para a questão Squeeze.

Squeeze			
ID	**TR/TE**	**Descrição do(s) erro(s)**	**Classificação**
AL14	1/7	Não considerou um cenário para lista vazia.	Regra
AL9	2/7	A verificação feita não considera os cenários para todos os elementos repetidos e apenas um repetido.	Regra
AL15	2/7	Não considerou um cenário para lista vazia e com apenas um elemento.	Regra
AL10	3/7	A comparação de maneira decrescente faz com que lista com valores iguais, lista com dois iguais e lista unitária fique vazia.	Regra
AL16	3/7	A comparação de maneira decrescente faz com que lista com valores iguais, lista com dois iguais e lista unitária fique vazia.	Regra
AL17	1/7	Não considerou um cenário para lista vazia.	Regra
AL18	5/7	Fazer pop() nas listas faz com que a comparação no condicional seja prejudicada, implicando em saídas incorretas.	Regra e Conhecimento
AL19	2/7	Não considerou um cenário para lista vazia. Os valores iguais que não estão lado a lado são removidos.	Regra
AL20	3/7	A verificação feita não considera os cenários para todos os elementos repetidos, apenas um repetido e com um único elemento.	Regra
AL3	1/7	Não considerou um cenário para lista vazia.	Regra
AL21	1/7	Não considerou um cenário para lista vazia.	Regra

Apêndice F

Comentários e sugestões dos alunos quanto a estratégia do oráculo

- "O Oráculo poderia dar dicas da resolução e não apenas a respostas."
- "Achei bastante interessante, na resposta de cima gostaria de acrescentar a utilização do oráculo após a execução da minha solução para assim eu confirmar se o resultado estaria correto conforme o feedback do oráculo. Gostei bastante de utilizar esse programa!"
- "É uma ferramenta útil no entendimento dos labs propostos."
- "Gostei desse momento, bem legal o oráculo. Gostei da forma explicativa, apesar de não tá muito por dentro dessas coisas, mas gostei da forma que você explicou e os slides bem legais!"
- "A página poderia ser melhor, está faltando um pouco mais de atenção, mas muito boa para fazer as questões."
- "Quando não é muito sensível a visualização da lógica atrelada à resolução do problema, o oráculo pode vir a ser bastante útil, mas a avaliação da utilidade depende da percepção de dificuldade da resolução do problema pelo usuário."
- "Gostei muito do oráculo!! Aparentemente, o mesmo possui uma excelente precisão, quando baseado no problema proposto."

- "O oráculo é tão útil quanto fazer seus próprios testes, usando uma calculadora para manualmente executar o programa como uma máquina, ele só te diz se você não está completamente louco com a calculadora, idealmente nós poderíamos ter a resposta, assim, fica fácil fazer os testes. O oráculo ajuda, mas quase negligenciável a ajuda."

- "O oráculo serve como validação dos teste que criamos, em situação de muita insegurança é útil para aliviar a ansiedade, mas para questões simples, resolvidas apenas com uma calculadora, eu usaria a calculadora."

- "Legal, começar e você saber que vai dar errado, porém falta um feedback aonde você errou para ficar algo mais didático para quem está começando, pois ainda não tem baseamento na programação e sofre em matemática simples mesmo, com tudo o programa está bastante legal e interativo minha sugestão caso queiram da uma melhorada para fim de aprendizagem."

- "O Oráculo poderia sugerir exemplos de entrada e saída, para que o usuário possa basear a correção de seu código. Desde já, o Oráculo é uma excelente ferramenta de estudo e avaliação."

- "Interessante saber o valor de entrada e saída no começo, por tanto, isso denota o problema a ser resolvido apenas para uma solução, assim, o oráculo, seria mais viável apresentar além de um (está certo) algo como uma solução em formato de algoritmo, sem o código em si."