
UFCG
SUPRA OMNES LUX LUCIS

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

Provendo feedback estrutural sobre projetos de algoritmo no ensino da programação utilizando Testes de Design

Caio Batista Oliveira

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: Educação em computação

João Arthur Brunet Monteiro
(Orientador)

Campina Grande, Paraíba, Brasil

O48p Oliveira, Caio Batista.
Provendo feedback estrutural sobre projetos de algoritmo no ensino da programação utilizando testes de desing / Caio Batista Oliveira. - Campina Grande, 2022.
60 f. : il. Color

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2021.
"Orientação: Prof. Dr. João Arthur Brunet Monteiro".
Referências.

1. Ensino de Programação. 2. Educação em Ciência da Computação. 3. Análise Estática. 4. Design Tests. 5. Feedback Personalizado em Programação. I. Monteiro, João Arthur Brunet. II. Título.

CDU 004.4:37(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA ITAPUANA SOARES DIAS GONÇALVES CRB-15/93

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO CIENCIAS DA COMPUTACAO
Rua Aprigio Veloso, 882, - Bairro Universitario, Campina Grande/PB, CEP 58429-900

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

CAIO BATISTA OLIVEIRA

PROVENDO FEEDBACK ESTRUTURAL SOBRE PROJETOS DE ALGORITMO NO ENSINO DA PROGRAMAÇÃO UTILIZANDO TESTES DE DESIGN

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: 10/12/2021

Prof. Dr. JOÃO ARTHUR BRUNET MONTEIRO, Orientador, UFCG

Prof. Dr. WILKERSON DE LUCENA ANDRADE, Examinador Interno, UFCG

Prof. Dr. TIAGO LIMA MASSONI, Examinador Interno, UFCG

Profa. Dra. YUSKA PAOLA COSTA AGUIAR, Examinadora Externa, UFPB

Documento assinado eletronicamente por **TIAGO LIMA MASSONI**, **COORDENADOR(A) ADMINISTRATIVO(A)**, em 10/12/2021, às 15:47, 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 **JOAO ARTHUR BRUNET MONTEIRO**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 10/12/2021, às 16:00, 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 **WILKERSON DE LUCENA ANDRADE**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 13/12/2021, às 09:43, 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 **Yuska Paola Costa Aguiar**, **Usuário Externo**, em 14/12/2021, às 18:23, 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 **1995558** e o código CRC **93F42EE7**.

Referência: Processo nº 23096.077582/2021-39 SEI nº 1995558

Resumo

As disciplinas introdutórias de programação normalmente possuem um grande número de estudantes matriculados por semestre, o que dificulta os professores a terem um contato individualizado para sanar dúvidas. Assim, a correção das atividades ocorre tipicamente por meio de ferramentas automáticas. No ensino da programação tanto a correção funcional como a correção da forma estrutural de uma solução algorítmica (projeto de algoritmo) são importantes. Existem diversas técnicas para a parte funcional, dentre as mais populares estão implementações de Juízes Online, entretanto não existem ferramentas amplamente difundidas que otimizem esta segunda correção. A área de estudo de técnicas para a verificação da parte de estrutura de dados e projeto é reduzida e sem grandes referências do estado da arte, deixando professores com poucos recursos e ferramentas que possam aplicar na correção da forma das soluções desenvolvidas pelos alunos. Assim, alunos podem desenvolver soluções estruturalmente incorretas, usando de funções ou estruturas de dados não permitidas, prejudicando o aprendizado da lógica de programação e limitando o contato de professores e alunos para resolução de dúvidas. Propomos neste trabalho uma abordagem de análise estática de código para detecção de problemas estruturais em projetos de algoritmo. O objetivo dessa abordagem é prover ao tutor uma forma de detectar um projeto de algoritmo, sem que seja necessário executar o algoritmo e o ajude a assegurar a estrutura do que está analisando, como o uso de funções proibidas e padrões específicos de codificação. Para isso, desenvolvemos uma ferramenta, Python Design Wizard, que fornece uma API, abstraída da AST de Python, possibilitando a criação de testes de design do código. A pesquisa conduzida neste mestrado envolve a validação da implementação, usabilidade e dos conceitos utilizados para construção da ferramenta. Esta validação é composta por duas partes, na primeira conduzimos uma avaliação quantitativa executando testes de design em 1714 programas de alunos de Programação I da Universidade Federal de Campina Grande, utilizando a ferramenta Python Design Wizard, para detecção de algoritmos de ordenação. Na segunda parte, utilizamos a mesma ferramenta e a técnica de Think Aloud Protocol para conduzir entrevistas com profissionais de educação e extrair informações se a solução proposta pode influenciar positivamente na aprendizagem da programação. Nossos resultados incluem uma ferramenta capaz de detectar algoritmos de ordenação, dentre algoritmos de alunos, no nosso estudo

quantitativo e uma coletânea de frases com *feedback* positivo sobre o conceito de testes de design, comprovando a facilidade da compreensão dos testes implementados na ferramenta e sua utilidade na área de educação.

Keywords. Ensino de programação, Qualidade de código, Design Tests, Feedback personalizado em programação, Padrões de codificação, Educação em ciência da computação.

Abstract

Introductory programming courses usually have a large number of students enrolled per semester, which makes it difficult for teachers to have individual contact to answer questions. Thus, exercises correction typically occurs through automatic tools. In teaching programming both the functional correction and the correction of the structural form of an algorithmic solution (algorithm design) are important. There are several techniques for the functional part, among the most popular are Online Judges implementations, however there are no widely used tools that optimize this second correction. The area of study of techniques for verifying data structure and design is reduced and without major standart references, leaving teachers with few resources and tools that they can apply to correct the form of the solutions developed by the students. Thus, students can develop structurally incorrect solutions, using functions or data structures that are not allowed, impairing the learning of programming logic and limiting the contact of teachers and students to solve doubts. In this work, we propose a static code analysis approach to detect structural problems in algorithm designs. The goal of this research is to provide the tutor a way to detect an algorithm design without having to execute the algorithm and help him to ensure the integrity of the code structure, such as the use of prohibited functions and specific coding patterns. In order to do that, we developed a tool, Python Design Wizard, which provides us an API, abstracted from Python's AST, enabling the creation of code design tests. The research conducted in this Masters involves the validation of the implementation, usability and concepts used to build the tool. This validation is composed of two parts, the first one we conducted a quantitative evaluation running design tests in 1714 Programing I student programs at the Federal University of Campina Grande, using the Python Design Wizard, in order to detect sorting algorithms. In the second part, we use the same tool and the Think Aloud Protocol technique to conduct interviews with education professionals and extract information on whether the proposed solution can positively influence programming learning. Our results include a tool capable of detecting sorting algorithms, among student algorithms, in our quantitative study and a collection of sentences with positive *feedback* on the concept of design tests, proving the easy understanding of the tests implemented in the tool and its usefulness in the education field.

Agradecimentos

À Deus primeiramente, sem Ele nada seria.

À Raquel, meu amor e meu apoio durante todo esse tempo.

À minha avó, que hoje está feliz onde quer que esteja.

À minha família, que nunca deixou de acreditar em mim.

Aos meus amigos, que me ajudaram tanto.

À meu orientador, João Arthur Brunet Monteiro, sem ele não seria possível este trabalho.

Aos participantes do experimento executado para avaliação deste trabalho.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

1.1 Contextualização

O ensino da programação tem ganhado cada vez mais significância com o passar dos anos, motivado principalmente pelo avanço tecnológico. O interesse crescente na área de Tecnologia da Informação e o Programa de Apoio à Reestruturação e Expansão da Universidades Federais (Reuni) fizeram com que as turmas de cursos introdutórios de programação crescessem. Consequentemente, a quantidade de exercícios que devem ser corrigidos pelos professores também cresceu.

Diversos estudos na área de educação tentam elencar os principais problemas do ensino da programação, como o estudo de Altadmri [AB15], mostrando que erros mais difíceis de se corrigir e analisar crescem de acordo com o tempo de estudo, migrando de erros sintáticos para semânticos. Outro estudo relacionado a estes problemas é de Carter [CDP15] que demonstra uma diferença considerável de qualidade (segundo uma métrica própria) entre o código dos alunos produzido em provas e nos exercícios propostos em sala de aula.

Para minimizar estas divergências, os estudos de Leite [Lei15] e Papastergiou [Pap09] apresentam propostas de melhorias no ensino da programação, fazendo os alunos focarem na resolução de problemas de programação em forma de exercícios com mais frequência. Consequentemente causando um acúmulo de correções para os professores durante o semestre.

Devido a essa grande quantidade de exercícios, o tempo que os professores teriam que empregar para prover um *feedback* sobre todos os aspectos das respostas (manualmente)

seria inviável; Assim, utilizam-se muitas vezes de ferramentas de correções automáticas, semelhantes a *coding contests* como citada em Janzen [JCH13] e Singh [SGS13].

Em alguns cursos introdutórios de programação, nos quais os professores implementam a esta ideia de Juízes Online [AJ14], a correção da solução submetida é feita com testes criados pelos professores e de forma automática. Esses testes têm como foco, porém, a corretude da solução e não sua estrutura [KT04].

Além da correção funcional, é também importante a forma como é construída essa solução, sendo denominada Projeto de Solução, Projeto de Algoritmo ou Design de Algoritmo. A segunda denominação provém do fato que, o algoritmo sendo uma sequência de passos que transforma um dado de entrada em um dado de saída, é uma solução em forma de código. Esta última tem o sentido de projeto ou desenho estrutural de um algoritmo.

Devido a pouca abordagem na literatura sobre como verificar automaticamente este projeto de algoritmo, comumente é utilizada a correção funcional. Os professores focam em testar apenas "o que" e não "o como" das soluções dos alunos, deixando de lado a verificação da consistência entre solução e a especificação. Existem estudos sobre análise estática aplicada no contexto de ensino a programação [ASF16] [TRB04], entretanto são abordagens para a qualidade de código e não o projeto de algoritmo.

A principal consequência da abstenção desta correção é que os alunos podem criar um projeto de algoritmo utilizando-se de bibliotecas e funções pré-definidas, as quais os professores não desejariam introduzir naquele momento. Assim, o aluno deixa de exercitar o raciocínio lógico para resolver a questão apresentada da forma que o professor pretendia. Este tipo de prática é chamada *Violação de Projeto*.

Podemos exemplificar este problema na seguinte situação. Um professor utiliza desse tipo de correção automática para verificar a funcionalidade correta das soluções dos alunos e propõe que implementem o algoritmo *bubblesort* para ordenar uma série de listas, porém um dos alunos implementa o algoritmo *insertion sort* que resolve o mesmo problema de uma outra forma. Os testes automáticos não acusarão erros de funcionalidade, mas o aluno não terá correspondido às expectativas do professor em relação ao projeto de sua solução. A área que aborda os aspectos do design de algoritmo num exercício ainda é pouco explorada na academia para a parte de educação. Entretanto este campo da ciência está diretamente relacionado com análise estática de código [JAC09], que contém estudos comprovando que

muitos problemas, não detectáveis com o programa em execução, podem ser detectados com uma análise estrutural dos comandos [Hua+04].

Uma abordagem popular para análise estática de código é proposta por Fagan [Fag76] que nomeou de "inspeção manual". Essa técnica consiste na inspeção de um código por um time de desenvolvedores, avaliando passo a passo a estrutura do código. Esse tipo de abordagem depende de muita atenção e tempo dedicado para garantir *feedback* correto.

No contexto de educação esse tipo de abordagem enfrenta dois grandes problemas. O primeiro é que tipicamente professores lecionam disciplinas de programação sozinhos ou em pequenos grupos, dificultando a inspeção manual de todos os exercícios dos alunos. O segundo problema é que a quantidade de soluções, anteriormente mencionada, torna o processo repetitivo e inviável a correção manual para os professores.

Há ferramentas e soluções para análise estática em diferentes contextos. Como por exemplo ArchJava, o trabalho de Aldrich [ACN02] ou mesmo Design Wizard proposto por Brunet [BGF09], ambas ferramentas de análise estática implementadas em Java com o intuito de garantir a conformidade de requisitos com código do software. Porém essas soluções não utilizam abordagens voltadas para o ensino de programação, uma API validada com tutores ou mesmo manipulações e checagens em baixo nível (atribuições de variáveis e *loops*).

Somando a falta de métodos focados no projeto de solução dos alunos, com a necessidade de *feedback* [CDP15] e a falta de tempo de um contato individualizado com os alunos [RS05], o design de algoritmo é deixado tipicamente de lado e a importância da correção se volta somente para a funcionalidade deles. Acreditamos que podemos avançar na solução deste problema através da abordagem proposta neste trabalho de pesquisa e que teria contribuições relevantes para o ensino de programação.

1.2 Objetivo da pesquisa

O objetivo desta pesquisa é prover professores uma abordagem para detectar estaticamente problemas estruturais em códigos de alunos, fornecendo assim, um *feedback* mais preciso em relação a forma como alunos respondem exercícios.

Com esse tipo de abordagem os professores poderão somar à correção funcional, o resultado dos seus testes estruturais, complementando o processo de correção de exercícios em

cursos introdutórios de programação.

1.3 Solução proposta

Para avançar na solução dos problemas abordados e ajudar professores a otimizarem seu tempo nas disciplinas introdutórias de programação, propomos uma abordagem de inspeção automatizada de código baseada em análise estática, voltada especificamente para a educação. Na nossa abordagem utilizamos regras de especificação semelhantes ao estudo de Brunet [BGF09] para podermos criar uma ferramenta que auxiliou a analisar nossa proposta de extensão do ensino da programação, na qual observamos a viabilidade da correção estrutural dos programas de alunos.

A principal diferença entre os testes de design e nossa extensão destes testes é que o nível de abstração tem uma variação mais abrangente. Esta variação vai desde parâmetros e variáveis até detecção de *loops* encadeados, verificando assim, a conformidade sintática de acordo com uma semântica traduzida em testes pela nossa ferramenta.

A extensão do ensino da programação utilizando testes de design proposta neste estudo tem o intuito de verificar desde chamadas de métodos até operações mais complexas como operações em listas e recursividade. Podendo garantir requisitos estruturais em forma de simples checagens em um teste de unidade.

Nossa ideia na abordagem é que os professores possam criar suas suítes de teste tanto para as funcionalidades como para o design de algoritmo, dando ao professor a possibilidade de prover um *feedback* mais completo sobre os exercícios dos alunos, bem como a garantia da originalidade das suas soluções. Com essa abordagem podemos detectar indicativos de plágio e uso de algoritmos prontos, sem a necessidade de execução.

A nossa implementação deste conceito intitula-se **Python Design Wizard**, que fornece uma API para construção de testes de design na linguagem Python. Essa API é construída a partir de uma abstração da árvore sintática de Python (AST) [Klu12] [Fou09] em forma de entidades e relações. Tornando-a, assim, mais simples de se ler, utilizar e manipular.

1.4 Questões de pesquisa

As questões de pesquisa que orientam nossa pesquisa envolvem respectivamente a ferramenta a qual desenvolvemos e a extensão do conceito de testes de design.

As duas primeiras questões tem relação direta com a implementação e funcionalidade da ferramenta. São elas:

Q1. É possível detectar algoritmos de ordenação de forma estática?

Q2. É possível avaliar a estrutura de soluções de alunos utilizando testes em Python?

Algoritmos de ordenação não conseguem ser detectados facilmente de forma automática por operações simples, porém não são tão complexos ao ponto de precisarmos de um longo período de tempo para implementação do ferramental. Esta é a motivação da escolha para este tipo de algoritmo.

As duas questões seguintes se referem à extensão do teste de design proposto nesta pesquisa e da usabilidade da ferramenta no contexto do ensino da programação. São elas:

Q3. O conceito de testes de design é facilmente entendido?

Q4. O uso de uma ferramenta com esse conceito tem alguma utilidade?

1.5 Avaliação

Nesta pesquisa a avaliação consistiu em dois estudos distintos. Os estudos avaliaram primeiramente a precisão da ferramenta, especificamente na detecção de algoritmos e seu uso em um ambiente de ensino de programação. No segundo momento, a facilidade de compreensão do conceito abordado, focando-se principalmente na criação e compreensão de testes e na usabilidade da ferramenta.

No primeiro estudo da validação do Python Design Wizard como ferramenta, utilizamos um estudo de caso com 1714 programas correspondentes a soluções de 21 questões diferentes de alunos do semestre 2017.2 da turma de Programação 1, do curso de Ciência da

Computação da Universidade Federal de Campina Grande. Nosso estudo envolveu a detecção de algoritmos de ordenação nestas soluções e dentro de uma amostra que continha especificamente operações não-triviais com listas, que poderiam confundir a avaliação da ferramenta nos testes.

Como forma de validação para o segundo estudo utilizamos o *Think Aloud Protocol* [Lew82] [Cla04], que é um formato de experimento no qual o participante é orientado a responder uma série de perguntas ou resolver um problema, verbalizando tudo que pode sobre o que está pensando, para depois ser transcrito, passar por um processo de categorização e ser utilizado como fonte de *feedback*. No nosso caso, este experimento foi feito com 16 pesquisadores de três diferentes universidades, os quais foram instruídos a ler e explicar testes de design e depois disso construir um outro teste utilizando a API do Python Design Wizard.

Nossos resultados para o primeiro estudo apontam a capacidade de detecção da ferramenta para algoritmos. Além disso, testes de design foram feitos em uma segunda parte, para diminuir a chance de falsos negativos. O que corrobora para segurança de nossos resultados utilizado no contexto de ensino. Para o segundo experimento os resultados apontam que os pesquisadores não tiveram dificuldades em entender o conceito da ferramenta e verificamos a utilidade da ferramenta para diferentes funções na área de ensino da programação.

1.6 Contribuições

As contribuições deste trabalho podem ser elencadas como:

- Desenvolvimento de uma ferramenta chamada Python Design Wizard, que implementa o uma extensão em baixo nível do conceito de Testes de Design em Python, fazendo uso do Unittest (*framework* de testes), amplamente utilizado em cursos de programação.
- Documentação com exemplos de como utilizar a ferramenta Python Design Wizard, para a correção de programas e manipulação da árvore sintática.
- Uma coleção de testes de exemplo, para detecção de problemas estruturais comuns de alunos que estão no início da aprendizagem de programação.
- Conjunto de entrevistas com potenciais usuários da ferramenta validando seu conceito e a facilidade de compreensão dos métodos fornecidos pelo Python Design Wizard.

- Estudo para detecção de algoritmos de ordenação, com dados reais de alunos de um curso introdutório de programação.

1.7 Estrutura do Documento

O documento dessa dissertação está dividido em capítulos que estão organizados da seguinte forma: **Capítulo 2** tem a fundamentação teórica utilizada para essa pesquisa. Depois no **Capítulo 3** discutimos sobre utilização de juízes online como forma de avaliação no ensino da programação, sobre testes de design e avaliação estática. Em seguida no **Capítulo 4** temos nosso capítulo de avaliação, com a metodologia utilizada e resultados **Capítulo 5** nossa seção de conclusão.

Capítulo 2

Fundamentação Teórica

A fundamentação teórica necessária para compreensão desta dissertação pode ser agrupada em dois grandes aspectos: contexto do ensino da programação atualmente e as formas de garantir solução correta de softwares. No contexto do ensino explicaremos como os conceitos de projeto de algoritmo e design de software são relevantes para os professores e como as correções de exercícios são incompletas atualmente. Já para o segundo aspecto, abordaremos as principais diferenças das análises dinâmicas e estáticas, bem como os testes e ferramentas que utilizam essas técnicas para correção automática, abordagem proposta nesta pesquisa.

2.1 Projeto de algoritmo

Pesquisadores da área do ensino da programação se deparam constantemente com diversos problemas, dentre eles a otimização dos conteúdos abordados no ensino e possibilitar os professores a terem mais tempo dedicado com seus alunos [Tob+01] são dois assuntos que direcionam nossa pesquisa.

Com conteúdos extensos e de grande importância, alguns cursos introdutórios de programação tem seu conteúdo dividido em unidades. Essa divisão permite aos professores abordar temas de forma granular e com iterações mais curtas de avaliação.

Este é o caso do curso de Ciência da Computação da Universidade Federal de Campina Grande, no qual estão contempladas 10 unidades com carga de conhecimento crescente na disciplina de Programação 1. Nesta disciplina as primeiras unidades são conhecimentos gerais sobre uso de variáveis, as unidades do meio do curso são sobre uso de listas e laços

até, por fim, unidades que abordam uso de dicionário e operações com matrizes.

Todas as unidades e seus respectivos conteúdos são listados na Tabela 2.1:

Tabela 2.1: Unidades de estudo da disciplina introdutória de programação da UFCG.

Unidade	Descrição	Unidade	Descrição
1	Conceitos elementares de programação.	6	Funções.
2	Escrevendo programas simples.	7	Estruturas de dados: listas.
3	Condições, alternativas e funções.	8	Ordenação
4	Laços definidos.	9	Estruturas de dados: sequências de sequências e matrizes.
5	Laços indefinidos.	10	Estruturas de dados: mapas.

A principal forma de fixação dos conteúdos, em forma de unidade, são exercícios de problemas computacionais. Esses exercícios abordam um ou mais aspectos das unidades em que se encontram e progridem em dificuldade conforme os alunos avançam nos conteúdos.

Para que o aluno possa resolver esses problemas, deve propor e implementar um Projeto de Algoritmo, ou ainda, um Design de Algoritmo. No livro *"Projeto de Algoritmos Com Implementações em Pascal e C"* [Ziv+04], Projeto de Algoritmo é definido como a idealização ou formalização de um passo a passo para resolver um problema. A implementação desse projeto é a solução dos alunos em forma de software para resolver o problema proposto pelos professores.

O ensino introdutório feito de forma incorreta ou incompleta pode levar a maus hábitos e problemas na vida profissional dos alunos [RRR03]. Sendo essencial assim uma forma de corrigir o projeto de algoritmo dos alunos e permitir um *feedback*, possibilitando ele a refletir sobre as diferenças entre o resultado alcançado e o que era esperado.

2.2 Análise Estática x Análise Dinâmica

Na Engenharia Software a preocupação com a qualidade do software está diretamente relacionada com o grau de semelhança entre a implementação e os requisitos previamente

estabelecidos [Lu+16]. A garantia de uma evolução seguindo as especificações do projeto requerem formas de validar e verificar soluções.

O princípio de V&V (validação e verificação) é utilizado amplamente para garantir a conformidade dos softwares. A validação está ligada ao princípio de funcionalidade correta da solução, detêm-se ao "que foi feito" e não "como foi feito". Por outro lado, a verificação aborda o princípio de estrutura correta solução, detêm-se ao "como foi feito" e não "o que foi feito" [WF89].

Validação e verificação não são princípios opostos, mas sim complementares. Seu uso correto fornece uma detecção precisa de erros de alto risco, que possam comprometer o software, avalia constantemente os requisitos na implementação, fornece informações sobre a qualidade e o progresso do desenvolvimento e dá a oportunidade de ajustes a curto prazo em implementações [WF89].

A Análise Dinâmica é a técnica que utiliza o princípio da validação para garantir a correta funcionalidade do software. Avalia se dado um *input* o *output* é o esperado. Um tipo de implementação dessa análise é conhecida por *Black-box testing*, ou também, Teste Caixa Preta. Sendo precisa quanto a funcionalidade, porém pouco precisa em relação ao projeto da solução e requer alto custo de processamento [Bal99].

Analogamente a Análise Estática utiliza o princípio da verificação. Mais rápida e menos custosa que a análise anterior, não depende da execução de um software [AB05]. É altamente precisa em relação a construção da solução, porém menos precisa em sua funcionalidade. A Análise Estática avalia se dado um *input* ele passa por todos os passos que deveria e se não comete nenhum efeito colateral, como por exemplo, alterar objetos e listas de escopo global. Uma de suas implementações se chama *White-box testing*, ou também, Teste Caixa Branca.

Existem diversas ferramentas que implementam uma, ou ambas as análises [HCO11] [Lu+16] [VWH12], [VD00]. Estas ferramentas, porém, no contexto de educação, são quase exclusivamente dedicadas à Análise Dinâmica.

2.3 Correção de exercícios

O elevado número de alunos traz o desafio da análise e correção se suas soluções [Tob+01]. Os professores não têm tempo hábil para poder corrigir todos os exercícios, recorrendo assim

a soluções automatizadas para este fim.

A principal tática adotada para solucionar o problema do volume de exercícios são implementações de testes automatizados [JCH13] [SGS13] [GPL16]. Testes estes que são aplicados aos programas criados pelos alunos, fornecendo dados sobre a funcionalidade correta da solução.

Em alguns casos, professores utilizam ferramentas que aplicam os testes automaticamente nas soluções dos alunos em um sistema de integração contínua, chamados Juízes Online [AJ14]. O curso de Ciência da Computação da UFCG utiliza uma implementação própria de um Juiz Online, chamado TST.

Como as ferramentas utilizadas pelos professores se baseiam em Análise Dinâmica, os dados extraídos dos resultados são relacionados à funcionalidade correta da solução. A falta de ferramentas de Análise Estática nesses Juízes Online faz com que a estrutura dessas soluções seja ignorada. Sendo permitido assim que alunos utilizem estruturas de dados e algoritmos que o impedem de exercitar a lógica na construção do projeto de acordo com o que o professor ensinou. Esse tipo de violação de regras de projeto impede uma orientação mais precisa no ensino da programação.

2.4 Conceito de testes de design

Para o completo entendimento desse estudo, essa seção de discussão sobre o conceito detalhado dos componentes e as aplicações de Testes de Design se faz necessária.

O estudo conduzido por Brunet em *"Design tests: An approach to programmatically check your code against design rules"* [BGF09], traz o conceito de Testes de Design como uma abordagem para construção de testes que tem como objetivo verificar se a estrutura do código está de acordo com as especificações, ao invés de se focar na funcionalidade do programa.

Um Teste de Design é executável de forma automática e corresponde diretamente à implementação de uma regra de design. Brunet descreve ainda que os testes de design tem três características fundamentais:

- São testes especificados na mesma linguagem que o código avaliado
- São executáveis automaticamente

- Checam como o software está sendo construído

O Teste de Design é um conceito válido apesar destes surgimentos de padrões de projeto, pois utiliza abstrações que se aplicam independente da forma de organização arquitetural, sejam a pacotes, componentes, módulos, iteradores, etc. Essas abstrações são denominadas: Entidades e Relações e são descritas da seguinte forma:

- **Entidades**: parte de um software, que contém N relações N pertencente aos naturais positivos
- **Relações**: conexão ou relacionamento entre duas entidades ou mais entidades, sendo sempre feita de um *caller* para um *callee* ou de um *callee* para um *caller*. Na forma de 1..N ou N..1 (relação inversa)
- **Caller**: Entidade que invoca uma ação. Ex: Função de *print*
- **Callee**: Entidade objeto dessa ação. Ex: *String* passada como parâmetro na função *print*

O exemplo a seguir (Código Fonte 2.1) ilustra como o teste de design pode ser aplicado em um tipo de implementação de um algoritmo com diferentes variações.

Código Fonte 2.1: Fibonacci utilizando recursão

```
def fibonacci(n): # Entity
    if(n <= 1):
        return n
    else:
        return(fibonacci(n-1) + fibonacci(n-2))
```

No exemplo do Código Fonte 2.1 temos uma implementação recursiva do algoritmo que gera o n-ésimo elemento da sequência Fibonacci. Supondo que esta é uma solução de uma questão da qual o professor pediu para implementar o algoritmo de forma recursiva e não iterativa, o teste de design faria uma checagem da relação da entidade, representada no exemplo pela função Fibonacci, para com ela mesma, através de uma chamada de função.

Tendo esta relação representada da seguinte forma em pseudocódigo:

Código Fonte 2.2: Relações diretas das entidades

```
ASSERT Entity.calls(Entity) == TRUE
```

A relação inversa seria bastante parecida e pode ser representada desta forma:

Código Fonte 2.3: Relações inversas das entidades

```
ASSERT Entity.is_called(Entity) == TRUE
```

Neste exemplo tanto o Caller como o Callee são representados pela função Fibonacci. Ela ao mesmo tempo realiza e é objeto da ação.

Baseando-se nessa abordagem, Brunet apresenta uma implementação de uma ferramenta para a construção de testes de design em Java, utilizando o *framework* JUnit. Essa ferramenta é chamada Design Wizard e serviu como guia para a implementação da nossa ferramenta utilizada nesta pesquisa.

2.5 Design Wizard e Testes de Design

A implementação desses testes pode ser realizada em diferentes etapas no processo de construção de um software. Comumente os softwares têm em seu princípio um projeto, que contém as especificações a serem seguidas, como uma espécie de planta de um prédio para a Engenharia Civil e os testes de design podem garantir, se firmados nessa etapa de planejamento, que o código continue fiel às especificações.

Com a evolução da Engenharia de Software novos padrões de projeto surgiram, alguns exemplos são: MVC, Repository Pattern e Atomic Design. Todos estes padrões, independente da linguagem a qual se foquem, tem como objetivo ter um controle maior ao longo da evolução da implementação do projeto e diminuir as deformidades na estruturação, ajudando a legibilidade e manutenção do software.

Na mesma proposta do conceito, Brunet apresenta uma ferramenta que implementa a abordagem de Teste de Design chamada **Design Wizard** e que funciona independentemente do padrão de projeto arquitetural. Esta ferramenta faz uma leitura da árvore sintática de Java e abstrai as entidades em classes (objetos), criando regras de implementação tomando como base as relações entre os objetos, fazendo uma implementação muito parecida com um teste unitário comum do JUnit.

A ferramenta proposta por Brunet tem o foco em projetos de software e requisitos de empresas de TI. Sendo uma de suas evoluções de uso criar documentações de código au-

tomatizadas apenas extraindo dos testes de design implementados. Testado em empresas, provou-se escalável e utilizável no dia-a-dia dos desenvolvedores.

Apesar de ser uma ferramenta comprovadamente útil e que reproduz fielmente o conceito de testes de design, o Design Wizard é implementado em Java, dificultando a sua utilização em contextos de cursos introdutórios de programação. Comumente Python é utilizado nestes cursos, devido a sua facilidade de compreensão e popularidade [Sri17]. Além disso, o foco do Design Wizard não foi idealizado para utilização de professores e pesquisadores, não oferece uma linguagem API amigável para não-desenvolvedores rotineiros.

2.6 Implementação da ferramenta e Testes de design

O Design Wizard é a primeira implementação do conceito de Testes de Design. É uma ferramenta que provê uma API para a criação de testes para garantir o cumprimento das regras de design de forma abstrata, utilizando os conceitos de entidades e relações. Utiliza uma biblioteca para a manipulação de bytecode em Java chamada ASM [BLC02], fazendo assim uma leitura estática do código, sem a necessidade da execução do código.

A leitura estática proporciona eficiência em relação ao tempo para a avaliação dos testes criados com o Design Wizard em projetos Java. Essa eficiência é comprovada em um experimento realizado por Brunet no mesmo estudo citado anteriormente. Uma análise de proporção de tempo versus tamanho para leitura completa de arquivos Jar, resultou em dados que comprovam que a ferramenta não tem problemas em fazer a leitura do bytecode mesmo para arquivos maiores que 10 MB.

Como a abordagem de Testes de Design pode ser mapeada para testes de unidade, a ferramenta resolve o problema da criação de um novo *framework* de testes apenas utilizando a própria biblioteca padrão de testes de Java, o JUnit. Os casos de teste são interpretados como testes de regras de design do código, fazendo com que sua escrita seja semelhante ao que os desenvolvedores estão acostumados no seu dia-a-dia.

Pseudocódigo do problema de dependência cíclica de pacotes (Código Fonte 2.5):

Código Fonte 2.4: Pseudocódigo teste pacote com dependência cíclica

```
packages = getAllPackages()
FOREACH packages as package:
```

```
    FOREACH package.child as child:
        ASSERT child.calls(package.name) == FALSE
```

Transformado para a linguagem do Design Wizard, estilo JUnit (Código Fonte 2.6):

Código Fonte 2.5: Teste Design Wizard dependência cíclica

```
public class DesignTest extends TestCase {
    public void testCyclicDependences() throws IOException {
        DesignWizard dw = new DesignWizard("project.jar");
        Collection<PackageNode> allPackages = dw.getAllPackages();
        for (PackageNode packageNode : allPackages) {
            Set<PackageNode> callersPackages = packageNode.
             getCallersPackages();
            for (PackageNode caller : callersPackages) {
                assertFalse(caller.getCallersPackages().
                 contains(packageNode));
            }
        }
    }
}
```

Neste exemplo (Código Fonte 2.5 e 2.6) de dependência cíclica mostrado por Brunet ilustra como é a implementação dos testes de design com a ferramenta.

Entrando em detalhes no funcionamento do Código Fonte 2.6 temos mas duas primeiras linhas as declarações da classe e do método que compõem o teste. Na terceira linha o Design Wizard faz o processamento do arquivo a ser analisado. Em seguida armazena todas as entidades do tipo "pacote" em uma variável que é utilizada como objeto de iteração do *loop* na linha seguinte. Dentro do corpo deste primeiro loop, todos os pacotes que chamam o pacote da iteração corrente são armazenados em uma outra variável, que também vai servir de parâmetro para o próximo loop. Por fim, dentro do corpo do *loop* interno existe uma condição que checa a não existência de uma chamada de um pacote do *loop* interno para o externo. Criando assim uma restrição de chamada cíclica de pacotes.

Apesar da grande praticidade do Design Wizard, ele não atende a necessidade principal deste estudo que é a utilização do conceito de testes de design para auxiliar professores no ensino da programação. A ferramenta foi criada com objetivo de criar amarras arquiteturais

em softwares robustos, possuindo um nível de abstração muito mais alto do que o utilizado no dia-a-dia dos professores de ensino da programação.

Capítulo 3

Testes de Design no contexto de ensino de programação

Nesta seção apresentamos o conceito de testes de design. Depois, discutimos esse conceito dentro do contexto de ensino de programação, listando os principais problemas da área e como a pesquisa deste mestrado apresenta uma proposta de solução para alguns deles utilizando uma extensão desse conceito, utilizando o Python Design Wizard como ferramenta.

3.1 Discussão sobre problemas no ensino de programação

Estudamos o estado da arte sobre problemas enfrentados atualmente pelos professores no ensino da programação, analisando mais profundamente o contexto do curso de Ciência da Computação das universidades federais, em especial a UFCG.

A grande quantidade de alunos gera semestralmente uma quantidade de respostas de exercícios que exigiria um tempo hábil para uma correção manual o qual os professores não possuem, fazendo com que professores e monitores recorram a sistemas de submissão [AJ14], integrados à correção automatizada dos exercícios dos alunos.

A correção automatizada utiliza *frameworks* de testes nativos das linguagens ensinadas, Python e Unittest no caso da UFCG. Teste unitários ajudam na detecção de problemas, especificamente da parte funcional, garantindo que os alunos tenham um *feedback* sobre a corretude da sua solução de forma rápida e precisa.

Este tipo de processo ajuda os professores a terem mais tempo para focarem nos conteú-

dos a serem lecionados, porém, também abre uma margem para soluções que não estejam estruturalmente condizentes com o ensino e regras dos professores. Como podemos perceber no exemplo do Código Fonte 3.1 abaixo, a questão pedia a implementação de um algoritmo de ordenação e o aluno apenas fez uma chamada para a API nativa de Python, para o método *sort*.

Código Fonte 3.1: Desvio de padrão de projeto usando função nativa

```
def my_custom_sort(array):
    array.sort()
    return array
```

No exemplo do Código Fonte 3.2 temos uma situação diferente que se encaixa no mesmo problema. A questão propunha ao aluno para implementar o algoritmo *Insertion Sort*, porém, foi implementado o *Bubble Sort*.

Código Fonte 3.2: Desvio de padrão de projeto usando algoritmo diferente

```
def my_insertion_sort(array):
    n = len(array)
    for i in range(n-1):
        for j in range(0, n-i-1):
            if array[j] > array[j + 1] :
                array[j], array[j + 1] = array[j + 1], array[j]
```

Ambos os exemplos trazem o problema que não pode ser solucionado com o processo e o ferramental atual dos professores, porém estão corretos do ponto de vista funcional. Dúvidas de alunos surgem sobre diferentes implementações para um mesmo problema e é impossível o contato individualizado para todos.

A solução proposta neste trabalho é abordar este ponto e sanar os problemas da correção estrutural, integrando uma ferramenta de correção baseada em análise estática, de fácil utilização, diminuindo o tempo dedicado a dúvidas que poderiam ser resolvidas imediatamente pelo próprio sistema de correção e aumentando a produtividade no ensino da programação.

3.2 Ferramental: Python Design Wizard

O Python Design Wizard foi desenvolvido com o mesmo propósito inicial do Design Wizard em prover uma API para a criação de testes de design. Utiliza como fonte de sua análise estática a própria AST (árvore sintática) de Python, combinando o conceito de testes de unidade com o *framework* nativo de testes da linguagem, análogamente ao JUnit no Design Wizard, o Unittest.

A ferramenta para seu funcionamento, recebe como *inputs* um arquivo em Python que vai ser analisado estruturalmente e um conjunto de testes que serão executados nesse arquivo. O conjunto de testes deverá estar listado no arquivo de **scripts** de configuração.

Nos exemplos abaixo temos um arquivo para ser analisado, o teste em questão e o arquivo de configuração que lê os teste que vão ser executados.

Código Fonte 3.3: programa_exemplo.py

```
def double(n):
    new_number = n * 2
    return double
```

Código Fonte 3.4: test_assign.py

```
def test_assign(self):
    self.dw.design_populate_all_entities()
    self.assertTrue(
        self.dw.design_get_entity('assign_field') == []
    )
```

Código Fonte 3.5: scripts.json

```
{
    "scripts":[
        "test_assign.py"
    ]
}
```

O Código fonte 3.3 mostra um programa simples em Python que cria uma função chamada **Double**. Esta função recebe como parâmetro um número inteiro, cria uma variável

atribuindo duas vezes o valor deste parâmetro e retorna o valor desta nova variável na terceira linha.

No código do teste, Código Fonte 3.4, após a definição do teste, na segunda linha o Python Design Wizard cria todas as entidades do *programa_exemplo.py* e suas relações. Em seguida o teste checa a não existência de um campo de atribuição dentro do *programa_exemplo.py*.

Por fim no Código Fonte 3.5, temos o aquivo que no formato de *JSON* recebe todos os testes em uma lista, com a chave *scripts*.

Para realizar um teste com o Python Design Wizard precisamos realizar a seguinte operação (Código Fonte 3.6):

Código Fonte 3.6: Comando para executar Python Design Wizard

```
$ python -m demo.demo_interact {{path/to/folder}} {{path/to/json/folder
    }}/scripts.json t
```

Utilizando a linha de comando do Python é passado um primeiro parâmetro, nomeado **m**, para executar a API como um programa Python. Logo depois, o nome **demo_interact** é como é nomeada a API internamente, passado como principal parâmetro do comando. Após o nome da API mais um parâmetro é passado contendo o caminho da pasta onde estão todos os arquivos para análise. Depois é passado o caminho até o arquivo de *scripts*, contendo o nome de todos os testes a serem executados. Todos os testes devem estar em uma pasta chamada **tests/scripts** dentro da ferramenta. O último parâmetro (**t**) é um parâmetro de recursividade, habilitando a execução de testes em sub diretórios dentro do diretório passado como parâmetro.

Adaptando o comando com os exemplos (Códigos fonte 3.3, 3.4 e 3.5), assumindo que o *programa_exemplo.py* está no diretório **alunos** e o *scripts.json* está no diretório corrente. Temos o resultado o comando a seguir (Código Fonte 3.7):

Código Fonte 3.7: Comando para executar Python Design Wizard com exemplos

```
$ python -m demo.demo_interact alunos/ ./scripts.json t
```

Os resultados do Python Design Wizard tem a forma de ponto para cada teste passante e o nome do teste em caso de falha, então caso executássemos o comando do exemplo, teríamos o seguinte resultado (Código Fonte 3.8):

Código Fonte 3.8: Resultado Python Design Wizard com exemplos

```
$ python -m demo.demo_interact alunos/ ./scripts.json t

Directory: alunos

test_assign programa_exemplo.py
```

O teste que foi mostrado como exemplo (Código Fonte 3.4), falharia neste caso, pois existe um campo de atribuição dentro do programa analisado (Código Fonte 3.3). Caso mudássemos a condição de checagem para **!= []** ao invés de **== []**, teríamos este outro resultado (Código Fonte 3.9), pois agora estaríamos esperando pelo menos um campo de atribuição:

Código Fonte 3.9: Resultado Python Design Wizard com exemplos

```
$ python -m demo.demo_interact alunos/ ./scripts.json t

Directory: alunos

. programa_exemplo.py
```

A API provida pelo Python Design Wizard contém uma série de métodos e abstrações de relações que tornam mais simples que a própria implementação original em Java. Devido a facilidade de leitura do código Python a redução de tamanho dos testes é considerável, como pode ser observado no Código Fonte 3.7.

Código Fonte 3.10: Teste Python Design Wizard dependência cíclica

```
def test_cycle_dependency(self):
    self.dw.design_populate_all_entities()
    all_imports = self.dw.design_get_all_by_type('import')
    for single_import in all_imports:
        self.assertFalse(
            single_import.design_get_callees_from_entity_relation('
                IMPORTS')
             .find(single_import.name)
        )
```

Este teste garante a mesma regra de design testada no exemplo anterior (Código Fonte 2.5), porém de forma mais sucinta.

Explicando detalhadamente o que faz este teste faz, temos primeiramente a definição do teste, semelhante ao que foi apresentado no exemplo anterior (Código Fonte 3.4), sendo a segunda linha a criação das entidades e suas relações pelo Python Design Wizard. Em seguida, todos os comandos de importação são capturados e atribuídos a uma nova variável chamada **all_imports**. Depois da criação da variável é feito um *loop* para iterar sobre cada uma das importações. Dentro do *loop* existe uma checagem, esperando resultado negativo, para a uma busca dentro da importação que está sendo iterada, por uma chamada de importação para ela mesma, ou outra que chame ela e que também seja chamada por essa mesma importação.

3.3 Especificação: Python Design Wizard

Nós desenvolvemos o Python Design Wizard, que provê uma API de fácil uso que poderá ser utilizada nesta abordagem de sistemas de correção. Com pouco tempo de estudo, até mesmo para professores que não tenham um contato direto com programação diariamente, podem utilizar a ferramenta, como mostraremos no nosso estudo na seção a seguir.

O Python Design Wizard está estruturado da seguinte maneira em seu repositório, por módulos:

- **api**: Contém um módulo definindo a classe PythonDW, que contêm tanto os métodos de processamento quando manipulação das entidades.
- **data**: Contém módulos com exemplos de programas a serem testados.
- **demo**: Contém um *script* de interação simples que roda testes utilizando a API do PythonDW para testar quaisquer programas que forem passados como parâmetros.
- **design**: Contém todas as classes utilizadas na definição de entidades e relações.
- **tests**: Contém os próprios testes de unidade da API do PythonDW e mais exemplos de testes criados utilizando a API.
- **utils**: Contém todos os *enums* de mapeamento que são utilizados na API.

Para utilizar a API basta instanciar a classe principal e vão ser criadas 3 estruturas, uma contendo a árvore sintática original da linguagem, um dicionário inicialmente vazio para armazenar as entidades com relações e por fim, o *enum* de mapeamento das entidades originais

da AST para as criadas pelo Python Design Wizard. O método inicial que é chamado ao se instanciar pode ser observado no código abaixo (Código Fonte 3.7).

Código Fonte 3.11: Classe PythonDW

```
class PythonDW:
    def __init__(self):
        self.ast_tree = []
        self.entities = {}
        self.ast_elements_dict = AstEntityTypeEnum.ast_entity_dict
```

O método parse (Código Fonte 3.8) é essencial para a leitura de qualquer programa para o qual se deseja escrever testes. A leitura dele é feita como a leitura de qualquer arquivo em Python, porém caminhando por todos os nós encontrados na árvore sintática e criando uma relação de pai e filho entre os nós.

Código Fonte 3.12: Função Parse

```
def parse(self, file_path):
    read_file = open(file_path, 'r')
    self.ast_tree = ast.parse(read_file.read())

    for node in ast.walk(self.ast_tree):
        for child in ast.iter_child_nodes(node):
            child.parent = node
```

Mesmo após o parse ser concluído as entidades não estão populadas com as relações, para isso são usados métodos auxiliares da API. A principal motivação desta escolha é que nem todos os casos de teste necessitam de um processamento da árvore completa. Otimizando espaço e tempo no cálculo das relações.

Esse tipo de pré-processamento é uma etapa que não existe nas abordagens de análise dinâmica e apesar de onerar os testes com uma etapa a mais, todas as outras operações de verificação (acesso) e manipulação (deleção e adição) são em tempo constante O(1), pois são feitas através de acessos a chaves de um dicionário de entidades.

Como pode ser observado no código a seguir (Código Fonte 3.9), apenas as entidades de *loops* e as operações dentro deles são populadas no campo *entities*.

Código Fonte 3.13: Função para popular entidades de loop

```
def design_populate_loop_entities(self, loop_name='for'):
    loop_entities = self.entities.get(loop_name)
    if (loop_entities is None):
        return

    for loop in self.entities.get(loop_name):
        self.create_body_loop(loop)
```

Todos os outros métodos são utilizados para manipulação das entidades e suas relações, como nos dois últimos exemplos (Código Fonte 3.10). O primeiro verifica se um nó da árvore sintática é ligado diretamente a outro por uma relação de parentesco, já o segundo retorna todas as entidades que têm uma determinada relação com a entidade principal, que é passada como parâmetro.

Código Fonte 3.14: Funções auxiliares PythonDW

```
def is_leaf_from_branch(self, branch_node, leaf):
    response = False
    while not isinstance(leaf.ast_node, self.ast_elements_dict['module'])
        :
        if leaf.ast_node.parent != branch_node.ast_node:
            leaf.ast_node = leaf.ast_node.parent
        else:
            response = True
            break
    return response

def design_get_relations_from_entity(self, entity_name, type_relation):
    entity = self.get_entity_by_name(entity_name)
    return entity.get_relations_by_type(type_relation) if entity != ''
        else []
```

Por ser uma abstração da árvore sintática, grande parte das operações e casos teste da linguagem são aplicáveis e implementáveis pela ferramenta. Podendo ser utilizado até mesmo para a geração de relatórios sobre as violações estruturais mais frequentes em problemas computacionais.

A utilização do Python Design Wizard não se resume somente a correção de exercícios.

O conceito de testes de design pode ser ensinado em disciplinas voltadas a testes e análise estática de código, servindo assim de exemplo para a implementação e execução deste tipo de teste.

O repositório do Python Design Wizard pode ser encontrado na seguinte url: `https://github.com/Caio-Batista/python-dw`.

3.4 Avaliação da API

O Python Design Wizard nasce como uma implementação do Design Wizard proposto por João Arthur Brunet [BGF09], porém com o foco no ensino de programação. A ferramenta fornece uma API que abstrai a árvore sintática de Python, construindo relações e entidades, utilizando a abordagem de Análise Estática. Sendo possível assim, criação de testes para avaliar a estrutura dos projetos de algoritmo dos alunos, de forma automática.

Um estudo com dados históricos de alunos da UFCG foi realizado para garantir a precisão da ferramenta em detectar de forma estática algoritmos de ordenação. Esse estudo utilizou um teste de design que avaliou o projeto de algoritmo de 1714 submissões, todas de um mesmo período. O teste utilizado tinha o intuito de avaliar, como um caso base, a simplicidade da API para testar regras de estrutura complexas de forma precisa. Mostrando assim a possibilidade de criação para outras regras, por indução.

Para avaliar a usabilidade da nossa ferramenta e a facilidade de compreensão dos conceitos abordados, utilizamos o **Think Aloud Protocol** [Lew82]. O processo deste estudo consiste na observação de pessoas durante um experimento, pedindo a todo tempo que expressem suas ideias sobre o que estão fazendo. Essas frases são transcritas e depois passam por um processo de catalogação em categorias chamadas *Tags*. Se encerra o processo extraindo aprendizados dessas *Tags*. Essa técnica aplicada em diversos trabalhos [SC07] [ESM07] [SM08] mostra a eficiência em extrair e traduzir *feedback* valioso para avaliar interfaces e sistemas.

No nosso caso usamos uma abordagem parecida com Clarke [Cla04] que utiliza o Think Aloud Protocol para avaliar uma API. Fizemos entrevistas com pesquisadores e professores de 3 universidades diferentes em duas partes. A primeira de perguntas e respostas e depois para implementação. Avaliando assim aspectos de compreensão e usabilidade.

3.5 Considerações finais

A nossa ferramenta fornece a possibilidade de se avaliar regras de construção de forma automática, através de testes de design. A nossa motivação para a pesquisa se deve ao fato da ausência de propostas para este tipo de avaliação voltados para a educação e ensino da programação. Já que a funcionalidade dos projetos de algoritmo já é corrigida hoje por diversas ferramentas, atacamos a área pouco abordada, equipando o professor com mais informação.

O professor sendo munido dessa informação, permite assim um *feedback* mais preciso para o aluno, detecção de plágios e um direcionamento mais completo, além de permitir mais tempo para o professor se focar nas atividades da disciplina e nas ementas das unidades.

Capítulo 4

Avaliação

Nossa pesquisa levou à formulação de quatro questões de pesquisa relativas à corretude da ferramenta e quanto à sua facilidade de uso no contexto de ensino da programação. São elas: **Q1)** *É possível detectar algoritmos de ordenação de forma estática?* **Q2)** *É possível avaliar a estrutura de soluções de alunos utilizando testes em Python?* **Q3)** *O conceito de testes de design é facilmente entendido?* **Q4)** *O uso de uma ferramenta com esse conceito tem alguma utilidade?*

Nesta seção descrevemos os dois estudos realizados para responder às questões. O primeiro estudo está ligado às duas primeiras perguntas e o segundo às duas últimas. Sendo eles: **1)** uma avaliação quantitativa de validação da API, verificando o grau de confiança nas respostas dos testes implementados pelo Python Design Wizard e; **2)** um estudo qualitativo em que profissionais validam a abordagem da ferramenta no contexto de educação.

4.1 Validação API

Para verificarmos se a nossa implementação dos testes de design está funcionando corretamente de acordo com o conceito proposto e respondermos as duas primeiras questões de pesquisa, foi feito um experimento com dados históricos de alunos da UFCG, aplicados a uma série de testes implementados com a API do Python Design Wizard. Coletando resultados para fortalecer nosso grau de confiança na nossa solução.

Este teste teve como base a detecção de algoritmos de ordenação nas submissões dos alunos, sem a necessidade de execução dos seus códigos. A amostra originária de certas

questões aplicadas pelos professores em problemas envolvendo operações com listas e iterações.

4.1.1 Metodologia

Visão geral

A divisão de unidades no ensino da programação no curso de Ciência da Computação facilita a aplicação de exercícios focados em pequenas porções de conteúdo. No contexto da UFCG o sistema que faz esta aplicação e correção automática é o TST.

Como pode ser observado no esquema a seguir (Figura 4.1), o TST funciona como um intermediário do professor e do aluno. O professor disponibiliza as questões no Juíz Online e os alunos podem acessar as questões de acordo com a unidade a qual estão estudando. Enviam suas submissões que são corrigidas e devolvidas como um *feedback* quase imediato. O professor por sua vez obtém um relatório completo de todas as soluções e problemas de execução dos programas dos alunos.

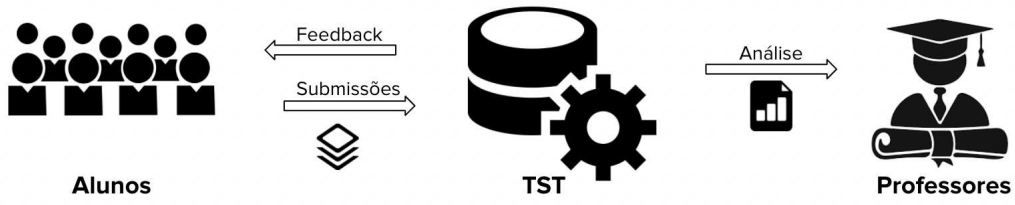

Figura 4.1: Fluxo TST (UFCG Programação 1)

Para nosso experimento simulamos uma integração com o TST, criando e aplicando testes de design em submissões de alunos ao sistema. Os testes avaliaram a existência de algoritmos de ordenação utilizando *For* nos programas dos alunos. A escolha do *For* nesse estudo se deu pelo fato do *For* e *While* terem ambos o mesmo custo de checagem e manipulação com a nossa API, sendo os testes para detecção de algoritmos de ordenação com *While* muito semelhantes aos testes com *For*. Então os resultados de um tipo de *loop* para o outro, por indução, seriam equivalentes, requerendo apenas algumas pequenas mudanças em parâmetros de métodos. Simplificamos o escopo do problema apenas para fim de análise e não por

complexidade.

Todos os dados coletados foram do semestre 2017.2 e se focaram em soluções para questões das unidades de 7 e 8 do curso. Estas unidades abordam os conceitos de listas, operações com repetições (iterações) e problemas de ordenação.

Após coletar os resultados dos testes, foi feita uma revisão manual da amostra para assegurar a avaliação correta dos testes que apontaram uso de estruturas de ordenação. Como mostra o esquema a seguir (Figura 4.2), o Python Design Wizard faz o papel da ferramenta que avalia automaticamente as submissões dos alunos e entrega o relatório para o professor.

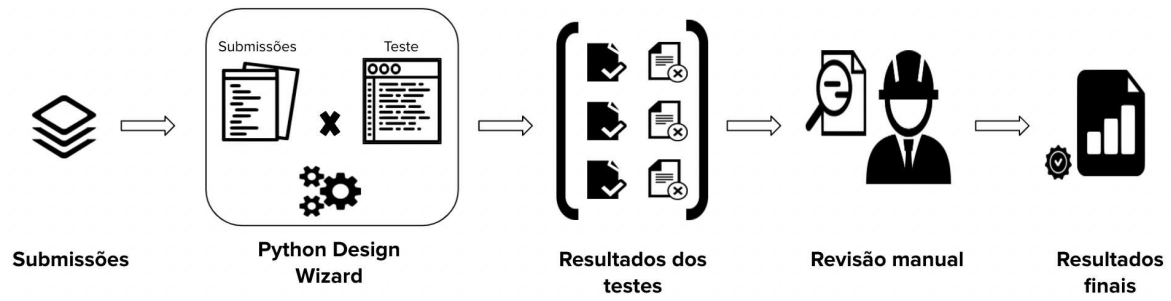

Figura 4.2: Fluxo primeiro experimento de validação da ferramenta

Todos os resultados foram verificados duas vezes manualmente, e aplicados as métricas que serão descritas na subseção a seguir. Abaixo temos dois exemplos de um programa que passaria no teste (Código Fonte 4.1) e outro resultaria em uma falha (Código Fonte 4.2).

Código Fonte 4.1: Exemplo de programa sem ordenação

```
def my_mean_3(array):
    sum_all = 0
    for e in array:
        sum_all += e

    return sum_all/3.0
```

Código Fonte 4.2: Exemplo de programa com ordenação

```
def my_sort(array):
    for i in range(1, len(array)):
        key = array[i]
```

```
        j = i-1

        while j>= 0 and key < array[j]:
            array[j+1] = array [j]
            j -= 1
        array[j+1] = key
```

A escolha das unidades não foi feita de forma aleatória. As unidades 7 e 8 são as que mais têm elementos em comum com algoritmos de ordenação, então se a API tem êxito em cenários que são potencialmente confusos para o teste, temos a confiança na precisão da solução no contexto de problemas utilizados por professores. Respondendo assim, às duas primeiras questões de pesquisa: **Q1)** *"É possível detectar algoritmos de ordenação de forma estática?"* e **Q2)** *"É possível avaliar a estrutura de soluções de alunos utilizando testes em Python?"*.

Teste utilizado

Para a criação do teste deste experimento utilizamos as funções fornecidas pela API do Python Design Wizard. O teste consiste nas seguintes etapas: **1)** um pré-processamento, populando as entidades abstraídas da AST do programa; **2)** uma checagem para identificar um laço do tipo *For*; **3)** um passo de processamento, recuperando todos os nós filhos deste primeiro *For*; e **4)** uma última checagem verificando se existe troca de posições de elementos de um *array*, através de atribuições.

A forma do teste pode ser observada abaixo e cada uma das funções utilizadas serão explicadas a seguir:

Código Fonte 4.3: Teste utilizado para detectar ordenação

```
def test_sorting_usage(self):
    self.dw.design_populate_all_entities()
    self.assertTrue(
        self.dw.design_get_relations_from_entity('for1', 'HASLOOP') != []
    )
    callees = self.dw.design_get_callees_from_entity_relation(
        'for1', 'HASLOOP'
    )
```

```
self.assertTrue(
    self.dw.design_list_entity_has_some_child_as_type(
        callees, 'assign_field'
    )
)
```

- **self**: Todo teste de design recebe o parâmetro self, que contém uma instância do Python Design Wizard, fornecendo sua API.
- **dw**: Objeto Python Design Wizard com todos os métodos da API.
- **assertTrue**: Verificação de condição verdadeira, nativa de Python.
- **design_populate_all_entities**: Método da API que popula o dicionário interno de entidades e relações. Inicia todos os testes de design.
- **design_get_relations_from_entity**: Método da API que recebe uma entidade pelo nome, primeiro parâmetro, e procura um tipo de relação, fornecido no segundo parâmetro. Neste caso estamos testando aqui se existe um *For* e ele tem alguma relação de *loop* dentro dele (relação *HASLOOP*).
- **design_get_callees_from_entity_relation**: Método da API que recupera todas as entidades que são *callees* da relação passada como parâmetro. Aqui novamente o *HASLOOP*.
- **design_list_entity_has_some_child_as_type**: Método da API que verifica para uma lista de entidades se existe alguma outra entidade filha de um tipo específico. Aqui os *callees* são a lista passada como primeiro parâmetro e *assign_field* o tipo a ser verificado. O que esse método está fazendo é checando internamente se existe uma manipulação de lista, através de atribuições na lista que está sendo iterada pelo *For*.

Dados

Os dados deste experimento foram retirados todos do curso de Ciência da computação da Universidade Federal de Campina Grande. Especificamente da disciplina de Programação 1 e podem ser sumarizados da seguinte forma:

- **Período relativo a origem da amostra**: 2017.2
- **Quantidade da amostra**: 1714 programas
- **Origem da amostra**: Programas enviados ao TST por alunos, anonimizados
- **Unidades as quais foram retiradas a amostra**: 7 e 8
- **Motivo da escolha das unidades**: Unidades com operações não triviais, operações com listas e uso de *For* e operações com ordenação.
- **Quantidade de tipos de questões**: 21 questões diferentes

Métricas

Nosso objetivo neste estudo foi avaliar o grau de confiança da ferramenta em detectar algoritmos de ordenação e de não apontar ordenação quando não existir. Para isso utilizamos três métricas, a precisão, o *recall*, a acurácia e a acurácia balanceada, que podem ser compreendidas pelas fórmulas a seguir:

$$Precision = \frac{ICO}{(ICO+INCO)}$$

$$Recall = \frac{ICO}{(ICO+NICO)}$$

$$Accuracy = \frac{ICO+NINCO}{(ICO+INCO+NINCO+NICO)}$$

$$BalancedAcc = \frac{Recall+TNR}{2}$$

- **ICO**: Identificados Continham Ordenação
- **INCO**: Identificados Não Continham Ordenação
- **NINCO**: Não Identificados Não Continham Ordenação
- **NICO**: Não Identificados Continham Ordenação
- **TNR**: True Negative Rate (Verdadeiro negativo sobre a soma de verdadeiros negativos e falsos positivos)

Estas métricas nos ajudam a entender aspectos diferentes da detecção da ferramenta. A precisão como métrica nos dá a informação sobre a maior quantidade de programas identificados com ordenação detectados. O *Recall* é usado para entendermos como a ferramenta se avaliou bem nos programas que não foram identificados com ordenação. As duas últimas métricas são sobre o aspecto geral de classificação da ferramenta, se em geral, o Python Design Wizard acertou quando pedimos para identificar ordenações.

Especificamente a Acurácia balanceada (**BalancedAcc**) foi usada como uma métrica de segurança. Em caso de amostra desbalanceada esta métrica normaliza os dados, dando mais segurança dos resultados independente da quantidade de amostras positivas e negativas.

4.1.2 Resultados

Para este primeiro experimento os resultados são descritos em duas partes. Primeiramente, são apresentados a quantidade de programas detectados com ordenação com exemplo e a aplicação dessa quantidade nas métricas utilizadas. A segunda parte discute os resultados e responde as questões de pesquisa.

Aplicados às métricas

Após a aplicação do teste na amostra, todos os 1714 programas que foram selecionados para o experimento, foram revisados manualmente duas vezes e os resultados podem ser encontrados na Tabela 4.1.

Tabela 4.1: Resultados dos testes de detecção de ordenação aplicados com programas de alunos

Status	Valor booleano	Quantidade
Identificados Continham Ordenação	Verdadeiro positivo	49
Identificados Não continham Ordenação	Falso positivo	15
Não Identificados Não Continham Ordenação	Verdadeiro negativo	1641
Não Identificados Continham Ordenação	Falso negativo	9
True Negative Rate	-	0,99

Aplicando os resultados às métricas temos que:

$$Precision = \frac{ICO}{(ICO+INCO)} = \frac{49}{(49+15)} \approx 77\%$$

$$Recall = \frac{ICO}{(ICO+NICO)} = \frac{49}{(49+9)} \approx 84\%$$

$$Accuracy = \frac{ICO+NINCO}{(ICO+INCO+NINCO+NICO)} =$$

$$= \frac{(49+1641)}{(49+15+1641+9)} \approx 99\%$$

$$BalancedAcc = \frac{Recall+TNR}{2} =$$

$$= \frac{0{,}84+0{,}99}{2} \approx 92\%$$

Análise

Nosso resultado mostra que na maioria dos casos os programas foram corretamente classificados com ordenação ou não pelo Python Design Wizard. A ferramenta consegue detectar na maior parte dos casos, precisamente, se existe um algoritmo de ordenação de forma estática em programas de alunos. Um exemplo de programa o qual foi detectado ordenação pode ser visto no exemplo abaixo (Código Fonte 4.4).

Código Fonte 4.4: Exemplo de programa de aluno com ordenação

```
def insertion_sort(numeros):
    for c in range(len(numeros)):
        for i in range(1, len(numeros)):
            if numeros[i-1] > numeros[i]:
                numeros[i-1],numeros[i] = numeros[i],numeros[i-1]
```

O aluno utilizou uma implementação do algoritmo de ordenação *Insertion Sort* e foi detectado pela ferramenta. Neste caso independente do problema que a questão pedia para ser resolvido nosso resultado não é invalidado, pois aqui estamos validando a capacidade de detectar a ordenação e não a correção do problema em si.

Aqui, no Código Fonte 4.5, temos um exemplo de um programa analisado que o teste não detectou ordenação.

Código Fonte 4.5: Exemplo de programa de aluno sem ordenação

```
b=float(raw_input())
h=float(raw_input())
area=(b*h)/2.0
s=''
area=str(area)
for i in range(0, len(area)):
    s+=area[i]
    if    i=='.':
        s+=area[i+1]
        s+=area[i+2]
        break
print area
```

Os nossos resultados são fortalecidos com nossa abordagem de revisão manual e com as métricas de acurácia e *recall*. Além disso, o Python Design Wizard possui testes internos de unidade e funções com cobertura acima da 80%, diminuindo nosso risco relacionado a construção da ferramenta.

Os testes de cobertura da ferramenta podem ser sumarizados em:

- 33 testes de unidade da ferramenta
- 100% de cobertura de funções
- 86% (aproximadamente) de cobertura de branches

Respondendo assim às duas primeiras questões de pesquisa (Q1 e Q2) nas quais não só conseguimos construir testes que abstraem o conceito de testes de design como também são eficientes na detecção de algoritmos de ordenação.

Q1. É possível detectar algoritmos de ordenação de forma estática?
R1. Nossos resultados do experimento quantitativo mostram que sim, para a grande parte dos casos.

Q2. É possível avaliar a estrutura de soluções de alunos utilizando testes em Python?
R2. Conseguimos avaliar algoritmos de ordenação e com nossos resultados acreditamos que podemos estender isso para outros tipos de estruturas.

4.2 Estudo qualitativo sobre testes de design com Python Design Wizard

A fim de avaliarmos qualitativamente a nossa solução e respondermos as duas questões restantes: **Q3)** *"O conceito de testes de design é facilmente entendido?"* e **Q4)** *"O uso de uma ferramenta com esse conceito tem alguma utilidade?"*. Apresentamos o Python Design Wizard e colhemos *feedback* com possíveis usuários da ferramenta. Estes usuários são

pesquisadores e professores de diferentes universidades no Brasil, com pesquisas especificamente na área de educação ou testes.

Análogo ao estudo de Brunet [BGF09], utilizando o Think Aloud Protocol obtivemos informações para a avaliação da nossa solução, propondo uma correção estática estrutural de código, no contexto do ensino da programação. Além desta validação também obtivemos ideias para trabalhos futuros.

4.2.1 Metodologia

Visão geral

Verificar apenas as impressões dos profissionais sobre a documentação da ferramenta não nos daria a segurança necessária para responder às questões de pesquisa referentes a qualidade da nossa solução, por isso utilizamos o Think Aloud Protocol em uma entrevista de duas etapas. A primeira parte consistiu apenas em interpretação de um conjunto testes de design apresentados e a segunda, em criação de outros testes propostos.

O Think Aloud Protocol é um técnica de coleta de informações que tem sua eficácia comprovada em diversos estudos [ESM07] [SC07] [WK14]. Como demonstra o esquema (Figura 4.3), o Think Aloud Protocol consiste em 4 partes principais: **1)** etapa de coleta da informação com o a pessoa ou conjunto de pessoas que pode validar o objeto de pesquisa; **2)** etapa de tradução ou transcrição das informações coletadas; **3)** uma etapa para atribuir tags semânticas a trechos transcritos, para serem contabilizadas e analisadas; e concluindo na **4)** etapa de organização dos insights obtidos.

Figura 4.3: Fluxo Think Aloud Protocol

Nosso uso do Think Aloud Protocol consistiu em entrevistas feitas com profissionais de forma remota e tiveram seus áudios gravados e transcritos. Fizemos o processo de categorização e coletamos todo o *feedback* necessário para a validação da nossa solução com Python Design Wizard. No esquema seguinte (Figura 4.4) vemos as fases da técnica aplicada a este experimento, contudo, existe uma fase a mais de revisão, a qual introduzimos para maior segurança e confiabilidade dos resultados.

Figura 4.4: Fluxo da entrevista usando Think Aloud Protocol

Todos os entrevistados tiveram seus nomes suprimidos durante a transcrição dos resultados. Categorizando e ordenando os participantes apenas pela ordem das entrevistas cronologicamente.

Entrevista

Na primeira parte da entrevista foram apresentados 3 testes diferentes criados com a API do Python Design Wizard, em ordem de complexidade crescentes. Esta parte da entrevista tinha como objetivo extrair informações sobre nossa implementação do conceito dos testes de design e da facilidade de compreensão de profissionais que nunca tiveram contato com a ferramenta.

O primeiro teste apresentado, detalhadamente descrito no Código Fonte 4.6, testa se um determinado programa tem chamadas para as funções do tipo *sort*, *map*, *print* e uma função definida pelo usuário chamada *func2*. Estas chamadas têm suas quantidades conferidas de forma específica para cada função.

Questões para interpretar

Código Fonte 4.6: Primeira questão da entrevista

```
def test_xpto(self):
    self.dw.design_populate_all_entities()

    num_field_calls = self.dw.design_get_qtd_calls_function("sort")
    self.assertEqual(num_field_calls, 4)

    num_field_calls = self.dw.design_get_qtd_calls_function("map")
    self.assertEqual(num_field_calls, 1)

    num_field_calls = self.dw.design_get_qtd_calls_function("print")
    self.assertEqual(num_field_calls, 4)

    num_field_calls = self.dw.design_get_qtd_calls_function("func2")
    self.assertEqual(num_field_calls, 1)
```

- **self**: Todo teste de design recebe o parâmetro self, que contém uma instância do Python Design Wizard, fornecendo sua API.
- **dw**: Objeto Python Design Wizard com todos os métodos da API.
- **assertTrue**: Verificação de condição verdadeira, nativa de Python.
- **design_populate_all_entities**: Método da API que popula o dicionário interno de entidades e relações. Inicia todos os testes de design.
- **design_get_qtd_calls_function**: Método da API que retorna a quantidade de chamadas para uma função passada por parâmetro.

O segundo teste apresentado (Código Fonte 4.7), apesar de ser menor em quantidade de linhas é mais complexo, por utilizar estruturas de iteração e relações. Neste teste, a verificação é feita para detectar se existe pelo menos um laço aninhado, começando com um *For* externo. A verificação é feita verificando se existe uma entidade *For* e se ela tem alguma relação de *loop* internamente, ou seja, laços aninhados.

Código Fonte 4.7: Segunda questão da entrevista

```
def test_xpto(self):
    self.dw.design_populate_all_entities()
    self.assertTrue(
        self.dw.design_get_relations_from_entity('for1', 'HASLOOP') != []
    )
```

- **self**: Todo teste de design recebe o parâmetro self, que contém uma instância do Python Design Wizard, fornecendo sua API.
- **dw**: Objeto Python Design Wizard com todos os métodos da API.
- **assertTrue**: Verificação de condição verdadeira, nativa de Python.
- **design_populate_all_entities**: Método da API que popula o dicionário interno de entidades e relações. Inicia todos os testes de design.
- **design_get_relations_from_entity**: Método da API que recebe uma entidade pelo nome, primeiro p arâmetro, e procura um tipo de relação, fornecido no segundo parâmetro. Neste caso estamos testando aqui se existe um *For* e ele tem alguma relação de *loop* dentro dele (relação *HASLOOP*).

Por fim, o último teste (Código Fonte 4.8) é exatamente o mesmo que foi aplicado no experimento de validação da API, para descobrir um algoritmo de ordenação implementado com laço *For*. Como as funções deste teste já foram explicadas no primeiro experimento vamos ocultar nesta seção

Código Fonte 4.8: Terceira questão da entrevista

```
def test_xpto(self):
    self.dw.design_populate_all_entities()
    self.assertTrue(
        self.dw.design_get_relations_from_entity('for1', 'HASLOOP') != []
    )
    callees = self.dw.design_get_callees_from_entity_relation(
        'for1', 'HASLOOP'
    )
    self.assertTrue(
```

```
        self.dw.design_list_entity_has_some_child_as_type(
            callees, 'assign_field'
        )
    )
```

Todos os testes que foram apresentados tiveram seus nomes modificados para test_xpto, evitando qualquer tipo de enviesamento na resposta do entrevistado pelo nome do teste.

A segunda parte da entrevista pode ser observada no Figura 4.5, em qual consistiam em 4 tópicos para serem implementados, sendo os dois primeiros com um nível de dificuldade básica e os dois últimos com maior nível de complexidade porém, solucionáveis com uma pesquisa direta na documentação da ferramenta.

Questões para desenvolver

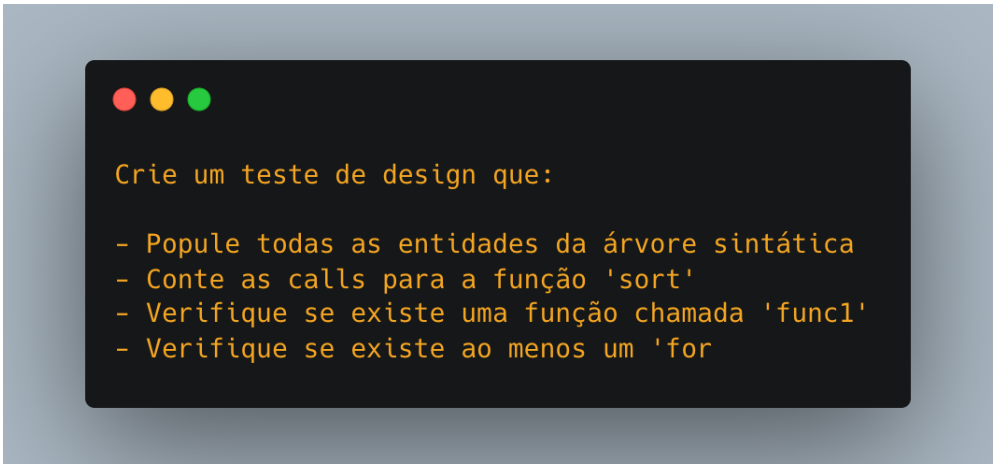

Figura 4.5: Perguntas da entrevista para serem desenvolvidas

Ao final das entrevistas foram reservados alguns minutos para o *feedback* direto sobre a ferramenta e a abordagem, deixando a pessoa entrevistada livre para qualquer comentário.

Dados

Como mencionado anteriormente, os dados deste experimento foram retirados de entrevistas com profissionais (professores e pesquisadores) de universidades diferentes do Brasil, envolvidos diretamente com o ensino da programação e/ou testes.

Importante mencionarmos que apenas os entrevistados 3, 5 e 6 não possuiam experiência ensinando Python ou contato direto com a linguagem de programação no seu dia-a-dia.

Podendo ser sumarizados da seguinte forma:

- **Quantidade de entrevistados**: 16
- **Quantidade de universidades diferentes**: 3
- **Duração das entrevistas**: Aproximadamente 20 minutos (média)
- **Tipo de entrevista**: Remota, por Hangouts
- **Quantidade de problemas para análise**: 3 testes
- **Quantidade de problemas para implementação**: 4 tópicos

As *Tags* utilizadas durante a fase de *tagging* (categorização) foram definidas em relação a abordagem de Testes de Design e sobre a ferramenta Python Design Wizard. As descrições, siglas bem como seus valores semânticos estão na tabela a seguir (Tabela 4.2):

Tabela 4.2: Tags utilizadas nas entrevistas

Nome	Sigla	Valor booleano
Feedback positivo ferramenta	FPF	Positivo
Feedback positivo abordagem	FPA	Positivo
Feedback negativo ferramenta	FNF	Negativo
Feedback negativo abordagem	FNA	Negativo
Dificuldade de uso	DU	Negativo
Dificuldade de compreensão	DC	Negativo
Facilidade de uso	FU	Positivo
Facilidade de compreensão	FC	Positivo
Compreensão do problema	CP	Positivo
Resposta correta	RC	Positivo
Resposta errada	RE	Negativo
Resposta parcialmente certa	RPC	Positivo
Perguntas próximas a resposta	PPR	Positivo
Perguntas distantes da resposta	PDR	Negativo
Direcionamento para a documentação	DPD	Negativo
Direcionamento direto	DID	Negativo

Métricas

Para este estudo a métrica utilizada para medir nossos resultados não poderia ser estritamente quantitativa porque é um estudo qualitativo.

Nosso principal ponto neste estudo foi, pois, analisar o *feedback* e extrair dele os aprendizados sobre a abordagem e a ferramenta. Sendo assim, nossos aprendizados através da interpretação do *feedback* vão responder diretamente nossas questões de pesquisa Q3 e Q4: **Q3)** *"O conceito de testes de design é facilmente entendido?"* e **Q4)** *"O uso de uma ferramenta com esse conceito tem alguma utilidade?"*.

4.2.2 Resultados

Antes de entrarmos na discussão sobre os aprendizados, vamos sumarizar a quantidade de cada *tag* e o total de frases categorizadas negativamente e positivamente. Os resultados individuais podem ser encontrados na Tabela 4.4 e os resultados totais na Tabela 4.3.

Tabela 4.3: Quantidade totais de frases negativas e positivas

Valor booleano	Quantidade
Positivo	461
Negativo	264

Tabela 4.4: Quantidade de frases por Tags

Nome	Sigla	Quantidade
Feedback positivo ferramenta	FPF	52
Feedback positivo abordagem	FPA	39
Feedback negativo ferramenta	FNF	43
Feedback negativo abordagem	FNA	12
Dificuldade de uso	DU	83
Dificuldade de compreensão	DC	74
Facilidade de uso	FU	61
Facilidade de compreensão	FC	103
Compreensão do problema	CP	95
Resposta correta	RC	87
Resposta errada	RE	12
Resposta parcialmente certa	RPC	13
Perguntas próximas a resposta	PPR	11
Perguntas distantes da resposta	PDR	6
Direcionamento para a documentação	DPD	16
Direcionamento direto	DID	18

Diferentemente da seção de resultados do experimento anterior, aqui apresentaremos os aprendizados das entrevistas por tópicos discutindo e analisando cada um com trechos e

dados que fortaleçam nossas afirmações.

Aprendizado 1

O conceito de testes de design foi facilmente entendido

Sumarização

- 452 frases selecionadas, 348 positivas e 104 negativas, sobre o conceito
- 15 entrevistados entenderam o conceito apenas com a introdução da entrevista
- 10 entrevistados já tinham ideias de como implementar o teste pedido antes mesmo de consultar a documentação
- 8 entrevistados fizeram perguntas coerentes sobre o conceito ou a evolução dele

Frases com tag dos entrevistados

> "Pelo que eu entendi o 'for' também é uma entidade, então nessa linha aqui eu estou pegando todas as ocorrências do 'for' e retornando uma lista com essas ocorrências e eu verifico se elas são diferentes de lista vazia."
>
> (Entrevistado 15, UFCG)

> "Tipos de relação e tipos de 'field abstraction', fico na dúvida se isso todas essas relações e entidades são suficientes para tudo que eu precisaria, mas acho que para seu contexto acho que isso seria interessante."
>
> (Entrevistado 10, UFCG)

> "[...] eu gostei sim, achei interessante o jeito de fazer esses testes, achei legal. Meu feedback é totalmente positivo de verdade."
>
> (Entrevistado 13, UFPB)

> "Que o nó de atribuição é o nó pai mas o valor é o nó filho, e ai você ter resolvido isso é uma coisa boa."
>
> (Entrevistado 12, UFCG)

> "Eu não entendi essa última, você quer que eu verifique se existe pelo menos um 'for' dentro da 'func1'?"
>
> (Entrevistado 7, UFCG)

> "Não sei se as entidades salvas aqui elas ficam salvas tipo isso aqui, a chave sendo o nome dos métodos também não sei, deixa eu ver aqui, tipo está me parecendo que se eu chamar um 'design get entity' passando 'func1' talvez funcione."
>
> (Entrevistado 9, UFCG)

Análise do aprendizado

Neste quesito a quantidade de frases positivas é três vezes maior que as negativas. A facilidade de interpretação de quase todos os entrevistados em relação aos testes era nítida e quase unânime. Até mesmo para a implementação dos testes, que deveria ser mais demorada pelo desconhecimento da abordagem e da ferramenta, foi feita de maneira natural por mais da metade dos entrevistados. Nem mesmo necessitando perguntar, fizeram corretamente o que era pedido nas questões. Com este resultado conseguimos validar a abordagem utilizada na criação da API do Python Design Wizard para testes de design.

A fala do entrevistado 12 mostra como em pouco tempo foi compreendido uma das abordagens de implementação visando garantir o cumprimento de regras estruturais do código.

Observamos nas duas últimas duas falas, entrevistados 7 e 9, uma dificuldade que marca as frases que foram classificadas como negativas para a compreensão do conceito. O alto índice, ainda que menor que o positivo, foi causado pela falta de familiaridade com testes num geral e não especificamente dos testes de design. Em outros casos notamos que o conceito ficou confuso na forma como foi implementado na ferramenta a relação entre as entidades e como manipulá-las.

Aprendizado 2

Testes construídos com a ferramenta são rapidamente compreendidos por programadores Python

Sumarização

- 239 frases selecionadas, 113 positivas e 126 negativas, sobre a ferramenta
- Apenas 2 vezes foi preciso direcionamento para a criação dos testes
- 14 entrevistados explicaram os dois primeiros testes em menos de dois minutos
- 15 conseguiram fazer ao menos um teste
- 10 reutilizaram funções vistas na primeira parte de análise

Frases com tag dos entrevistados

> "[...] e ai esse 'HASLOOP' quer dizer que existe um loop dentro desse cara, entendeu? Tipo um loop aninhado, nesse caso."
>
> (Entrevistado 9, UFCG)

> "Ou seja para sumarizar, ele pega todas as entidades que tem na árvore sintática do python e verifica a quantidade de vezes que os métodos foram chamados."
>
> (Entrevistado 8, UFCG)

> "[...] ah, todas as outras funções é a mesma coisa, ele vai contar a quantidade de chamadas a função que você tá passando como parâmetro."
>
> (Entrevistado 7, UFCG)

> "Sendo que eu eu não sei se tem que colocar alguma coisa antes aqui ou uma variável de configuração pra salvar isso aqui."
>
> (Entrevistado 13, UFPB)

> "Sim, to tentando entender... como faz um tempo que não vejo Python, tenho que relembrar um pouco da sintaxe também."
>
> (Entrevistado 5, UFPB)

Análise do aprendizado

A grande quantidade de frases categorizadas como negativas nesse caso foram em sua maioria por problemas de uso com a documentação da ferramenta. Muitas vezes os entrevistados tentavam entender só lendo o teste e em alguns casos até mesmo codificar um teste novo só com o que foi visto na questão proposta anteriormente na entrevista. O entrevistado 13, que logo depois de ler a documentação conseguiu utilizar a ferramenta e entender os testes, comentou sobre esta dificuldade de compreensão.

Outro problema observado foi que apesar de utilizarmos abstrações de relações e entidades no Python Design Wizard, em alguns casos ainda foi visível a dificuldade de compreensão e de uso de algumas funções com a sintaxe mais baixo nível. Este caso em específico foi comentado pelo entrevistado 5 que disse não ter tanta familiaridade com Python.

Apesar das frases negativas, podemos observar que os três entrevistados (9,8 e 7) fizeram as afirmações antes mesmo de consultar a API, interpretando corretamente os testes apresentados. Estas falas combinadas com a quantidade de entrevistados que conseguiram fazer pelo menos um teste e os que reutilizaram as funções vistas, mostram o aspecto intuitivo da nossa API, tanto para a interpretação como para criação de testes.

Aprendizado 3

A ferramenta tem utilidade na área da educação

Sumarização

- 8 entrevistados apontaram o uso da ferramenta para educação
- 4 daqueles disseram que utilizariam a ferramenta no seu dia a dia
- 4 sugerem o uso da ferramenta em disciplinas de testes e arquitetura de software
- Apenas 2 não conseguiram apontar utilidades da ferramenta

Frases com tag dos entrevistados

> "[...] esse teste a gente pode... tanto testar o design de uma aplicação real, mas pela minha vivência também pela parte de ensino acho que seria muito

interessante esse tipo de teste, tanto para validar o que os alunos fizeram [...].”

(Entrevistado 1, UFCG)

“Então o design wizard poderia ajudar os professores nesse sentido, ajudando a verificar se os alunos estão utilizando essas funções que os professores não esperam que os alunos utilizem”

(Entrevistado 11, UFCG)

“Eu acho, entretanto, que dentro da disciplina de P2, [...] eu acho que teste de design poderia ser também incentivado, junto com os testes de unidade, porque eles abstraem mais a ideia do que você precisa testar.”

(Entrevistado 7, UFCG)

“Pensando rápido acho que para correção, a primeira coisa que vêm na cabeça é correção de prova.”

(Entrevistado 5, UFPB)

Análise do aprendizado

Metade dos entrevistados disse diretamente que percebiam utilidade da API para o contexto de educação e mais de um entrevistado falou sobre correção de provas e problemas de alunos (Entrevistados 1 e 5), que é o tema desta pesquisa. Estas falas fortalecem a aplicação da nossa solução para correção estrutural no contexto de ensino da programação e não somente limitado a disciplinas introdutórias.

Aprendizado 4

Houve dificuldade na implementação dos testes

Sumarização

- 3 entrevistados apontaram melhorias na documentação
- 6 entrevistados confundiram o uso de uma função na criação do teste

- 8 entrevistados precisaram consultar mais de uma vez a api para criar os últimos dois testes

Frases com tag dos entrevistados

> "[...] eu sei que ah, parâmetros eu já sei o que é um parâmetro mas se for uma pessoa que tem menos experiência ainda que eu vai dificultar"
>
> (Entrevistado 6, UFCG)

> "[...] por ser tão linha de comando, eu tive alguma dificuldade"
>
> (Entrevistado 16, UFCG)

> "A minha dúvida é se ele pega só coisas do Python ou do... funções criadas pelo próprio usuário? Porque eu to vendo aqui pelos exemplos que eu tenho um 'for while if' que são três coisas nativas do Python, então não sei se ele realmente pega coisa que não é nativa."
>
> (Entrevistado 10, UFPB)

Análise do aprendizado

Houveram por algumas vezes dificuldades na implementação dos testes pedidos (especialmente os dois últimos), por falta de clareza na documentação ou mesmo no propósito do método utilizado da API. Através destas falas, tanto a documentação como a própria API foi melhorada nos pontos requisitados, facilitando a compreensão e abstraindo mais o nível da API em métodos ainda mais simples e intuitivos.

Desdobramentos

- Testes de design podem ser utilizados para criar regras e restrições de arquiteturas de software
- A ferramenta foi melhorada através do *feedback* das entrevistas
- Entrevistados com menos experiência com programação tiveram mais dificuldade

- A documentação da ferramenta foi melhorada
- Os entrevistados se guiaram pelos exemplos e não pela descrição da API

Frases com tag dos entrevistados

> "Talvez a documentação não esteja bacana, talvez colocar mais exemplos... o teu exemplo mostrado é melhor do que a documentação, sabe?"
>
> (Entrevistado 14, UFCG)

> "[...] esse tipo de abordagem é interessante para fazer esse tipo de amarra, eu consigo dar mais transparência para o que está ruim e o que está bom dentro do nosso software."
>
> (Entrevistado 2, FACISA)

Análise do aprendizado

Durante algumas entrevistas os exemplos da API se mostraram extremamente importantes e cruciais para o bom desenvolvimento e interpretação dos testes de design. Este *feedback* em especial foi motivador de uma reestruturação na nossa documentação do Python Design Wizard, provendo não somente testes de exemplo como programas para serem testados utilizando os métodos da API.

Dois dos entrevistados que relataram ter menos experiência com testes sentiram dificuldades, porém, somente um não conseguiu, ao final da entrevista, desenvolver o que foi pedido.

Por fim, mais de um entrevistado citou a utilização do Python Design Wizard para validação estrutural de softwares, relação entre camadas e regras de comunicação entre módulos. Este conceito, porém, foi abordado e validado na pesquisa de Brunet e não é nosso foco como descoberta.

Análise geral

Compreendemos no aspecto geral que apesar de algumas dificuldades, a aceitação da abordagem e da API no contexto de educação esteve validado e que alguns dos entrevistados

gostariam de utilizar no dia-a-dia do ensino da programação, ou mesmo utilizá-la para ensino de testes em outras disciplinas avançadas de programação.

Nossos resultados fundamentados por estas falas nos trazem confiança na resposta positiva para as duas últimas questões de pesquisa: Q3 e Q4. Não só percebemos a facilidade de compreensão para uma ferramenta que implementa o conceito de testes de design, como também o interesse de profissionais na utilização da mesma para o contexto de ensino da programação.

Q3. O conceito de testes de design é facilmente entendido?
R3. Acreditamos pelas declarações dos entrevistados e de perguntas relacionadas ao entendimento do conceito que a compreensão é fácil.

Q4. O uso de uma ferramenta com esse conceito tem alguma utilidade?
R4. Apoiados nas declarações dos entrevistados sobre as aplicações da ferramenta no cotidiano de professores e tutores do ensino da programação, acreditamos que seriam potenciais usuários da ferramenta, para otimizar correções de programas.

Capítulo 5

Conclusão

5.1 Pesquisa com Python Design Wizard

Esta pesquisa de mestrado apresenta um estudo sobre a extensão do conceito de testes de design que pode ser aplicada no contexto de ensino da programação para sanar problemas de professores no que tange a correção estrutural de programas. Como meio para realizar este estudo foi desenvolvida uma ferramenta chamada Python Design Wizard que é uma implementação análoga ao Design Wizard apresentado no estudo de referência deste trabalho, contudo, feito na linguagem que predomina o ensino da programação atualmente nos cursos de computação e com uma API voltada para professores.

Existem diversas propostas para solucionar problemas relacionados à correção de exercícios de alunos em larga escala. Estas abordagens contudo não focam na parte estrutural, o que pode comprometer toda uma linha de aprendizagem do aluno se for deixada de lado. Os professores se focam então apenas na parte funcional da correção, deixando de lado como as soluções são implementadas e se os alunos quebram regras de design pré-estabelecidas. A correção estrutural utilizando técnicas de inspeção manual demanda uma quantidade muito maior de tempo que os professores detêm e ainda está sujeita a falhas humanas por cansaço ou mesmo desatenção. Por isso, a aplicação de uma técnica baseada em análise estática de código visando verificação de regras de design pode auxiliar professores neste problema.

O Python Design Wizard, ferramenta desenvolvida nesta pesquisa, é usada como uma implementação da técnica de testes de design para esta correção estrutural de soluções de alunos. A ferramenta disponibiliza uma API construída como uma camada de abstração

acima da árvore sintática da linguagem Python, transformando nós em entidades que contém relações que podem ser testadas. Os testes criados com o Python Design Wizard utilizam as mesmas estruturas e operadores que os testes nativos da linguagem, não necessitando de qualquer dependência extra. Além disso, a API da ferramenta é voltada para correção de problemas, com métodos curtos e verbosos, para professores de qualquer tipo de curso de programação utilizarem.

Para a avaliação da nossa abordagem e ferramenta utilizamos dois estudos. Os dois estudos tinham como objetivo validar a ferramenta e avaliar a sua utilização por professores e pesquisadores. Para isso foi feito um estudo de caso com testes criados com a API do Python Design Wizard aplicados a programas de alunos, visando validar a precisão da ferramenta em detectar algoritmos de ordenação de forma estática. Em seguida foi feito um estudo qualitativo utilizando o Think Aloud Protocol, visando desta vez extrair *feedback* de potenciais interessados da ferramenta, sobre a praticidade da API e sua utilidade no dia-a-dia do ensino da programação. No primeiro nossos resultados, combinados a um teste de suporte, confirmaram que a ferramenta tem o comportamento correto e que consegue detectar com precisão algoritmos de ordenação. No segundo, com conjunto de *feedback* aprendemos que apesar da ferramenta e da documentação necessitarem de algumas modificações ela é compreendida facilmente por professores/pesquisadores e ambos enxergam potencial e utilidade no ensino da programação.

5.2 Trabalhos Futuros

Como próximos passos da nossa pesquisa e trabalhos futuros, pretendemos fazer estudos longitudinais com turmas de programação que terão durante o semestre exercícios corrigidos estruturalmente com testes utilizando a API do Python Design Wizard. Idealmente utilizaremos a mesma estrutura de Juízes Online, como o TST, apenas acoplando a parte de correção do projeto de algoritmo, adicionando o *feedback* com o já existente sobre a parte funcional das soluções dos alunos. Mediremos após o semestre o quanto de tempo extra os professores puderam despender em atividades que não de correção, a evolução das notas de provas e a presença de Violações de Código em comparação com semestres anteriores.

Bibliografia

[ACN02] Jonathan Aldrich, Craig Chambers e David Notkin. “ArchJava: Connecting software architecture to implementation”. Em: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE. 2002, pp. 187–197.

[AB15] Amjad Altadmri e Neil C.C. Brown. “37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data”. Em: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE ’15. Kansas City, Missouri, USA: Association for Computing Machinery, 2015, pp. 522–527. ISBN: 9781450329668. DOI: 10.1145/2676723.2677258. URL: https://doi.org/10.1145/2676723.2677258.

[AJ14] Fábio P Alves e Patrıcia Jaques. “Um ambiente virtual com feedback personalizado para apoio a disciplinas de programação”. Em: *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*. Vol. 3. 1. 2014, p. 51.

[ASF16] Eliane Araujo, Dalton Serey e Jorge Figueiredo. “Qualitative aspects of students’ programs: Can we make them measurable?” Em: *2016 IEEE Frontiers in Education Conference (FIE)*. IEEE. 2016, pp. 1–8.

[AB05] Cyrille Artho e Armin Biere. “Combined static and dynamic analysis”. Em: *Electronic Notes in Theoretical Computer Science* 131 (2005), pp. 3–14.

[Bal99] Thomas Ball. “The concept of dynamic analysis”. Em: *Software Engineering—ESEC/FSE’99*. Springer. 1999, pp. 216–234.

[BGF09] Joao Brunet, Dalton Guerrero e Jorge Figueiredo. “Design tests: An approach to programmatically check your code against design rules”. Em: *2009 31st International Conference on Software Engineering-Companion Volume*. IEEE. 2009, pp. 255–258.

[BLC02] Eric Bruneton, Romain Lenglet e Thierry Coupaye. "ASM: a code manipulation tool to implement adaptable systems". Em: *Adaptable and extensible component systems* 30.19 (2002).

[CDP15] Jason Carter, Prasun Dewan e Mauro Pichiliani. "Towards Incremental Separation of Surmountable and Insurmountable Programming Difficulties". Em: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Kansas City, Missouri, USA: Association for Computing Machinery, 2015, pp. 241–246. ISBN: 9781450329668. DOI: 10.1145/2676723.2677294. URL: https://doi.org/10.1145/2676723.2677294.

[Cla04] Steven Clarke. "Measuring API usability". Em: *Dr. Dobb's Journal Windows* (2004), S6–S9.

[ESM07] Brian Ellis, Jeffrey Stylos e Brad Myers. "The factory pattern in API design: A usability evaluation". Em: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 302–312.

[Fag76] ME Fagan. "Design and code inspections to reduce errors in program development". Em: *IBM Systems Journal* 15.3 (1976), pp. 182–211.

[Fou09] Python Software Foundation. *32.2. ast — Abstract Syntax Trees*. 2009. URL: https://docs.python.org/2/library/ast.html (acedido em 06/10/2021).

[GPL16] Jianxiong Gao, Bei Pang e Steven S Lumetta. "Automated feedback framework for introductory programming courses". Em: *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. 2016, pp. 53–58.

[HCO11] William GJ Halfond, Shauvik Roy Choudhary e Alessandro Orso. "Improving penetration testing through static and dynamic analysis". Em: *Software Testing, Verification and Reliability* 21.3 (2011), pp. 195–214.

[Hua+04] Yao-Wen Huang et al. “Securing web application code by static analysis and runtime protection”. Em: *Proceedings of the 13th international conference on World Wide Web*. 2004, pp. 40–52.

[JAC09] William JACKSON. *Static vs dynamic code analysis*. 2009. URL: `https://gcn.com/articles/2009/02/09/static-vs-dynamic-code-analysis.aspx` (acedido em 06/10/2021).

[JCH13] David S Janzen, John Clements e Michael Hilton. “An evaluation of interactive test-driven labs with WebIDE in CS0”. Em: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 1090–1098.

[Klu12] T. Kluyver. *Green Tree Snakes the missing Python AST docs*. 2012. URL: `https://greentreesnakes.readthedocs.io/en/latest/` (acedido em 06/10/2021).

[KT04] Moez Krichen e Stavros Tripakis. “Black-box conformance testing for real-time systems”. Em: *International SPIN Workshop on Model Checking of Software*. Springer. 2004, pp. 109–126.

[Lei15] Raquel Machado Leite. “Uma proposta para o ensino de programação de computadores na Educação Básica”. Em: (2015).

[Lew82] Clayton Lewis. *Using the"thinking-aloud"method in cognitive interface design*. IBM TJ Watson Research Center Yorktown Heights, NY, 1982.

[Lu+16] Yafeng Lu et al. “How does regression test prioritization perform in real-world software evolution?” Em: *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 535–546.

[Pap09] Marina Papastergiou. “Digital game-based learning in high school computer science education: Impact on educational effectiveness and student motivation”. Em: *Computers & education* 52.1 (2009), pp. 1–12.

[RS05] André Luıs Alice Raabe e JMC da Silva. “Um ambiente para atendimento as dificuldades de aprendizagem de algoritmos”. Em: *XIII Workshop de Educação em Computação (WEI’2005). São Leopoldo, RS, Brasil*. Vol. 3. sn. 2005, p. 5.

[RRR03] Anthony Robins, Janet Rountree e Nathan Rountree. “Learning and teaching programming: A review and discussion”. Em: *Computer science education* 13.2 (2003), pp. 137–172.

[SGS13] Rishabh Singh, Sumit Gulwani e Armando Solar-Lezama. “Automated feedback generation for introductory programming assignments”. Em: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 2013, pp. 15–26.

[Sri17] KR Srinath. “Python–the fastest growing programming language”. Em: *International Research Journal of Engineering and Technology (IRJET)* 4.12 (2017), pp. 354–357.

[SC07] Jeffrey Stylos e Steven Clarke. “Usability implications of requiring parameters in objects’ constructors”. Em: *29th International Conference on Software Engineering (ICSE’07)*. IEEE. 2007, pp. 529–539.

[SM08] Jeffrey Stylos e Brad A Myers. “The implications of method placement on API learnability”. Em: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 2008, pp. 105–112.

[Tob+01] Carlos Miguel Tobar et al. “Uma arquitetura de ambiente colaborativo para o aprendizado de programação”. Em: *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*. Vol. 1. 1. 2001, pp. 367–376.

[TRB04] Nghi Truong, Paul Roe e Peter Bancroft. “Static analysis of students’ Java programs”. Em: *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*. Citeseer. 2004, pp. 317–325.

[VWH12] André Van Hoorn, Jan Waller e Wilhelm Hasselbring. “Kieker: A framework for application performance monitoring and dynamic software analysis”. Em: *Proceedings of the 3rd ACM/SPEC international conference on performance engineering*. 2012, pp. 247–248.

[VD00] Jeffrey S Vetter e Bronis R De Supinski. “Dynamic software testing of MPI applications with Umpire”. Em: *SC’00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE. 2000, pp. 51–51.

[WF89] Dolores R Wallace e Roger U Fujii. “Software verification and validation: an overview”. Em: *Ieee Software* 6.3 (1989), pp. 10–17.

[WK14] Jacqueline Whalley e Nadia Kasto. “A qualitative think-aloud study of novice programmers’ code writing strategies”. Em: *Proceedings of the 2014 conference on Innovation & technology in computer science education*. 2014, pp. 279–284.

[Ziv+04] Nivio Ziviani et al. *Projeto de algoritmos: com implementações em Pascal e C*. Vol. 2. Thomson Luton, 2004.