Nathalia da Cruz Alves

CODEMASTER: UM MODELO DE AVALIAÇÃO DO PENSAMENTO COMPUTACIONAL NA EDUCAÇÃO BÁSICA ATRAVÉS DA ANÁLISE DE CÓDIGO DE LINGUAGEM DE PROGRAMAÇÃO VISUAL

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Mestre em Ciência da Computação.

Orientadora: Prof.ª Dr. rer. nat. Christiane Gresse von Wangenheim, PMP

Coorientador: Prof. Dr. Jean Hauck

Florianópolis
2019

Ficha de identificação da obra elaborada pelo autor
através do Programa de Geração Automática da Biblioteca Universitária
da UFSC.

Alves, Nathalia da Cruz
CodeMaster: Um modelo de avaliação do pensamento computacional na Educação Básica : através da análise de código de linguagem de programação visual / Nathalia da Cruz Alves ; orientadora, Christiane Gresse von Wangenheim, coorientador, Jean Carlo Rossa Hauck, 2019.
143 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós Graduação em Ciência da Computação, Florianópolis, 2019.

Inclui referências.

1. Ciência da Computação. 2. Avaliação. 3. Pensamento computacional. 4. Educação Básica. 5. Análise estática de código. I. Wangenheim, Christiane Gresse von. II. Hauck, Jean Carlo Rossa. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

Nathalia da Cruz Alves

CodeMaster: Um modelo de avaliação do pensamento computacional na Educação Básica através da análise de código de linguagem de programação visual

Esta Dissertação foi julgada adequada para obtenção do Título de mestre e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 26 de fevereiro de 2019.

Prof. José Luís Almada Güntzel, Dr.
Coordenador do Curso

Banca Examinadora:

Prof.ª Christiane Gresse von Wangenheim, Dr.ª rer. nat. PMP
Orientadora
Universidade Federal de Santa Catarina

Prof. Jean Carlo Rossa Hauck, Dr.
Coorientador
Universidade Federal de Santa Catarina

Prof. Adriano Ferreti Borgatto, Dr.
Universidade Federal de Santa Catarina

Prof. Dalton Francisco De Andrade, Dr.
Universidade Federal de Santa Catarina

Prof. Ricardo Azambuja Silveira, Dr.
Universidade Federal de Santa Catarina

AGRADECIMENTOS

Agradeço a tudo e a todos que de alguma forma contribuíram para o planejamento, execução e conclusão deste trabalho.

Especialmente, agradeço à professora Christiane, a orientadora certa, mesmo nas horas incertas; pelas suas orientações, prestatividade, diligência e apoio durante todo o período de desenvolvimento deste trabalho.

Agradeço ao professor Jean pela sua assessoria e amparo na realização de atividades intrincadas, pertinentes a este trabalho, as quais não estariam realizadas sem sua significativa contribuição.

Agradeço aos membros atuais e antigos do GQS (Grupo de Qualidade de Software) e da Iniciativa Computação na Escola que contribuíram efetivamente para a conclusão deste trabalho.

Agradeço aos professores que aceitaram participar como membros da banca, ao professor Adriano que prestou explanações com clareza em relação a conceitos de estatística até então por mim desconhecidos.

Agradeço à CAPES pelo suporte financeiro durante todo o período de desenvolvimento deste trabalho, por meio da concessão de bolsa de mestrado. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

There certainly are things that cannot be told in words, but that can only be said by people who have exhausted their use of words.

Tanaka Yoshiki

RESUMO

O ensino do pensamento computacional, já na Educação Básica, é de suma importância para preparar os alunos para os desafios do século XXI. Desta forma, surge a necessidade de avaliação das competências adquiridas em relação ao pensamento computacional. A avaliação pela análise do código criado pelo aluno como resultado de atividades abertas é uma forma que permite verificar quais conceitos foram efetivamente aplicados no processo de ensino-aprendizagem. Mesmo já existindo algumas abordagens de forma pontual, principalmente para a linguagem de programação Scratch, ainda não existe um modelo de avaliação mais abrangente e sistematicamente validado. Desta forma, o objetivo do presente trabalho é desenvolver sistematicamente um modelo de avaliação genérico independente de uma linguagem de programação visual (VPL) com base na literatura e no estado da arte. O modelo é instanciado por uma rubrica voltada à avaliação de programas criados com a VPL App Inventor e implementado evoluindo a ferramenta web CodeMaster. A avaliação da confiabilidade e validade do modelo é realizada por uma avaliação em larga escala com mais de 88 mil aplicativos desenvolvidos com App Inventor. Os resultados da avaliação indicam que o modelo é válido e confiável. Por meio da disponibilidade do modelo, espera-se facilitar e reduzir o esforço necessário para avaliação de atividades de programação no contexto de ensino de computação na Educação Básica, suportando assim a sua ampla aplicação em escolas brasileiras.

Palavras-chave: avaliação, pensamento computacional, análise estática de código, linguagem de programação visual, Educação Básica.

ABSTRACT

Teaching computational thinking already in K-12 is important to prepare students for the challenges of the 21st century. Therefore, there is a need to assess the acquired competences related to computational thinking. An assessment through the analysis of the code created by the student as a result of ill-defined activities is a way that allows to verify what concepts were effectively applied in the teaching-learning process. And, although, there exist already some specific approaches mainly for the programming language Scratch, there still does not exist a broader assessment model that has been systematically validated. Thus, the objective of this research is to systematically develop a generic model independent of a visual programming language (VPL) based on literature and the state of the art. The model is instantiated through a rubric aimed at assessing programs created with App Inventor, and implemented by evolving the web tool CodeMaster. Its reliability and validity are evaluated through a large-scale evaluation of the instantiation of the model, with more than 88,000 App Inventor programs. The results indicate that the model is reliable and valid. The model is expected to facilitate and reduce the effort to assess programs created by students in the context of teaching computing in K-12 and thus support the broad application in Brazilian schools.

Keywords: assessment, computational thinking, static code analysis, visual programming language, K-12 Education.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BNCC – Base Nacional Comum Curricular
CSTA – *Computer Science Standards*
GQM – *Goal Question Metric*
IEC – *International Electrotechnical Commission*
INE – Departamento de Informática e Estatística
ISO – *International Standard Organization*
MEC – Ministério da Educação
PPGCC – Programa de Pós-Graduação em Ciência da Computação
SQuaRE – *Standards named systems and software Quality Requirements and Evaluation*
TCT – Teoria Clássica dos Testes
TRI – Teoria da Resposta ao Item
VPL – *Visual Programming Language*

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta uma contextualização do problema que se almeja solucionar neste trabalho e o método de pesquisa adotado para tal. São apresentados os objetivos gerais e específicos, bem como a delimitação do escopo deste trabalho. Ao final, são apresentadas as contribuições esperadas e a estrutura desta dissertação.

1.1 CONTEXTUALIZAÇÃO

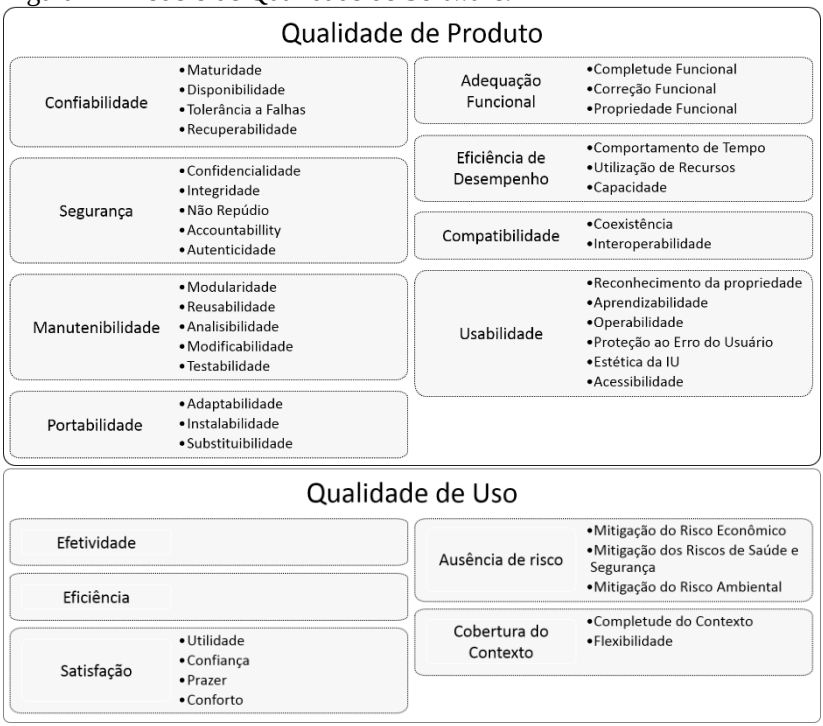
Os produtos de software estão aumentando rapidamente e estão sendo usados em quase todas as atividades da vida humana. Neste contexto, a qualidade do software é um fator essencial relacionado ao sucesso de negócios e à segurança humana. Consequentemente, medir e avaliar a qualidade de um produto de software tornou-se uma tarefa importante. A qualidade do software é o grau em que o produto de software satisfaz necessidades explícitas e implícitas quando usado sob condições especificadas (ISO/IEC 25010 – *Systems and software Quality Requirements and Evaluation – SQuaRE*, 2011). Para desenvolver um produto de software de qualidade satisfatória, é necessário que a qualidade seja identificada, planejada, medida e avaliada em todo o seu ciclo de vida, utilizando métricas de qualidade baseadas em um modelo de qualidade (ISO/IEC 25010, 2011).

A qualidade de software abrange diversos aspectos que são definidos, por exemplo, pela norma ISO/IEC 25010 (2011). Esta norma apresenta um *framework* composto por dois modelos (veja Figura 1): (i) Qualidade de Produto que aborda os fatores externos e internos de qualidade do produto e (ii) Qualidade de Uso que aborda os fatores de uso do software.

Existem diversas técnicas que podem ser utilizadas para medir/avaliar a qualidade. Entre elas, técnicas que medem a qualidade baseada no código do produto de software. Essas podem ser classificadas em duas categorias: análise estática e análise dinâmica. A análise estática tipicamente faz uma revisão do código fonte e da documentação (BOURQUE; FAIRLEY, 2014) e a análise dinâmica é realizada por meio de testes de software. Diferente da análise dinâmica, a análise estática é realizada sem executar o código ou considerar uma entrada específica (AYEWAH et al., 2008). Em vez de tentar provar que o código cumpre suas especificações, a análise estática busca violações de práticas de programação recomendadas e fornece *feedback* sobre a

qualidade do código do software medindo diversas métricas pré-estabelecidas.

Figura 1 - Modelo de Qualidade de Software.



Fonte: adaptado de ISO/IEC 25010 (2011) – versão traduzida.

A análise estática de código geralmente é realizada por uma ferramenta automatizada que analisa todo o código fonte do programa para um conjunto de critérios predefinidos (BELLER et al., 2016). Estas ferramentas podem ser configuradas para detectar problemas funcionais, tais como vazamento de memória e lógica incorreta, ou problemas de manutenção de código, como, por exemplo, desacordo com práticas recomendadas de programação e violações de convenções de estilo (BELLER et al., 2016).

Atualmente existem diversas ferramentas para análise estática de código alinhadas às características do modelo de qualidade de software (ISO/IEC 25010, 2011) para diferentes linguagens de programação como Python, Java, C, C++, etc. (NIST, 2017). Cada ferramenta possui objetivos distintos, podendo ser desde objetivos mais simples como a

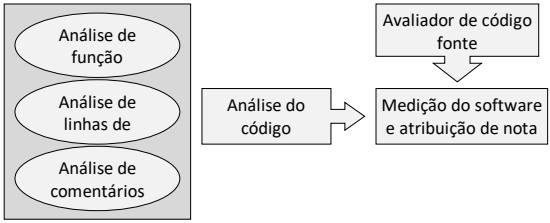
busca por erros de sintaxe, a objetivos mais complexos como a busca por falhas críticas de segurança. Tipicamente, estas ferramentas utilizam métodos derivados da tecnologia de compiladores, como grafo de fluxo de controle e análise léxica, para facilitar a manipulação e análise do código fonte (SOTIROV, 2005).

Estas ferramentas também podem ser utilizadas no contexto educacional para facilitar o processo de avaliação de atividades práticas de programação de alunos (YULIANTO; LIEM, 2014). O ensino da computação, geralmente, engloba o ensino de programação por meio de atividades práticas nas quais os alunos criam programas de software. Como parte de uma unidade instrucional, estes programas devem ser avaliados observando a aplicação de conceitos de algoritmos, variáveis, controle de fluxo, modularidade, etc. (CSTA, 2016; SINTHSIRIMANA; PANJABUREE, 2013). Estes analisadores de código voltados ao contexto educacional, além de realizarem uma análise de código, geralmente possuem (SINTHSIRIMANA; PANJABUREE, 2013):

- Um modelo de análise de código: Tipicamente baseado em uma rubrica, isto é, uma grade de critérios, detalha o que deve ser analisado.
- Um modelo de atribuição de nota: Define a alocação de nota para o programa do aluno a partir dos resultados gerados pela análise de código.
- *Feedback* instrucional: Gera sugestões de melhorias e correções, específicas para o programa avaliado, com base nos dados obtidos na análise de código.

Um exemplo típico é o analisador criado por SINTHSIRIMANA e PANJABUREE (2013). A ferramenta analisa o número de linhas de um programa, o processo de função e a legibilidade (*readability*) dos comentários (Figura 2).

Figura 2 - Arquitetura de um analisador de código voltado ao contexto educacional.



Fonte: adaptado de Sinthsirimana e Panjaburee (2013) – versão traduzida.

Quando realizada de forma manual pelo professor, a análise do código-fonte é um processo trabalhoso e repetitivo, pois exige que o professor realize a mesma análise minuciosa para cada programa individualmente. Além disso, o professor deve possuir conhecimento avançado já que há inúmeras formas de programar soluções para uma mesma atividade. Sendo assim, uma ferramenta deste tipo auxilia o processo de ensino e aprendizagem já que é capaz de avaliar vários programas em um curto período de tempo de forma coerente e fornecer *feedback* detalhado de forma imediata aos alunos (YULIANTO; LIEM, 2014).

Atualmente propõe-se iniciar o ensino de computação já na Educação Básica. Para o ensino de computação na Educação Básica, existem linguagens e ferramentas direcionadas ao contexto do ensino nesse nível educacional para iniciantes, as quais tipicamente usam VPL (*Visual Programming Languages*) baseada em blocos (CSTA, 2016). Esse tipo de linguagem é indicado como sendo mais adequado para o ensino de computação de novatos, pois permite aos alunos concentrarem-se apenas na programação lógica, sem o obstáculo de erros de sintaxe que são encontrados em linguagens tradicionais baseadas em texto (CSTA, 2016). Uma VPL é qualquer linguagem de programação que permite ao usuário criar programas por meio da manipulação de elementos gráficos, em vez de especificá-los textualmente (GOLIN; REISS, 1990). Exemplos de VPL baseada em blocos são: Scratch, App Inventor e Snap!.

Para facilitar a avaliação de atividades práticas de programação na Educação Básica também já foram criadas ferramentas de análise estática de código para VPL baseada em blocos. Exemplos incluem as ferramentas Dr. Scratch, a qual avalia algumas práticas de programação como paralelização e sincronização, empregadas na VPL Scratch (MORENO-LEÓN; ROBLES, 2015), Ninja Code Village (OTA; MORIMOTO; KATO, 2016), a qual avalia conceitos como controle e funções, também sobre a VPL Scratch, e CodeMaster (GRESSE VON WANGENHEIM et al., 2018), a qual avalia práticas do pensamento computacional móvel na VPL App Inventor e conceitos do pensamento computacional tanto para App Inventor como para Snap!.

1.2 PROBLEMA

Apesar de já existirem alguns analisadores para VPL baseada em blocos, observa-se que as ferramentas existentes têm modelos de avaliação bastante variados. Atualmente, não existe um modelo

conceitual genérico, independente de uma linguagem de programação, que identifique quais características do código devem ser analisadas. Os modelos também não definem para qual etapa da Educação Básica são voltados e não apresentam uma avaliação sobre sua validade e confiabilidade.

Além disso, alguns analisadores são direcionados à avaliação de atividades fechadas de programação, com solução predefinida, ao invés de atividades abertas nas quais os alunos precisam realizar todas as atividades do ciclo de engenharia de software, dentro de um contexto de ensino construtivista, visando também o desenvolvimento de habilidades do século XXI como criatividade e pensamento crítico (p21.org).

Neste contexto, há a necessidade de um método válido e confiável que forneça suporte na definição, execução e avaliação de programas criados com VPL baseada em blocos. Assim, a principal pergunta no contexto desta pesquisa consiste em:

Pergunta de pesquisa: Como fornecer uma avaliação automatizada da qualidade de trabalhos práticos de programação com VPL baseada em blocos, dentro do contexto educacional da Educação Básica, de forma confiável e válida?

1.3 OBJETIVOS

1.3.1 Objetivo geral

O objetivo geral deste trabalho é desenvolver e avaliar um modelo conceitual de análise estática de código de VPL baseada em blocos dentro do contexto educacional de avaliação de atividades abertas na Educação Básica.

1.3.2 Objetivos específicos

Os objetivos específicos do presente trabalho são:

Objetivo I: Análise do estado da arte em relação a modelos de análise e avaliação de código referente a VPL baseada em blocos no contexto da Educação Básica.

Objetivo II: Avaliação em termos de confiabilidade e validade de um avaliador de código.

Objetivo III: Desenvolvimento de um modelo conceitual de avaliação genérico, identificando de forma sistemática características do

código a serem analisadas a partir da análise qualitativa da literatura existente e da análise de um avaliador de código.

Objetivo IV: Evolução da implementação de um avaliador de código com base no modelo genérico desenvolvido.

Objetivo V: Avaliação de uma instância do modelo desenvolvido em um estudo de caso avaliando a sua confiabilidade e validade.

Delimitação do escopo do trabalho: O escopo do modelo proposto limita-se a analisadores estáticos de código para VPL baseada em blocos orientadas ao contexto educacional da Educação Básica. O foco neste grupo de linguagens é motivado pela ausência observada de uma abordagem planejada especificamente para o contexto educacional.

1.4 MÉTODO DE PESQUISA

O método de pesquisa utilizado neste trabalho é caracterizado como uma pesquisa exploratória aplicada de abordagem multimétodo (SAUNDERS et al., 2009). Trata-se de uma pesquisa exploratória, pois envolve a análise do estado da arte. Possui natureza aplicada, haja vista compreende o desenvolvimento de um modelo conceitual para análise estática de código. Tem abordagem multimétodo, pois utiliza técnicas qualitativas e quantitativas, como procedimentos de pesquisa bibliográfica e documental (GIL, 2010), mapeamento sistemático da literatura (PETERSEN et al., 2008), Modelo ADDIE (BRANCH, 2009), entre outros. Para a avaliação do modelo é realizado um estudo de caso (YIN, 2001) usando a abordagem GQM – *Goal Question Metric* (BASILI; CALDIERA; ROMBACH, 1994). As etapas de pesquisa são apresentadas na Figura 3.

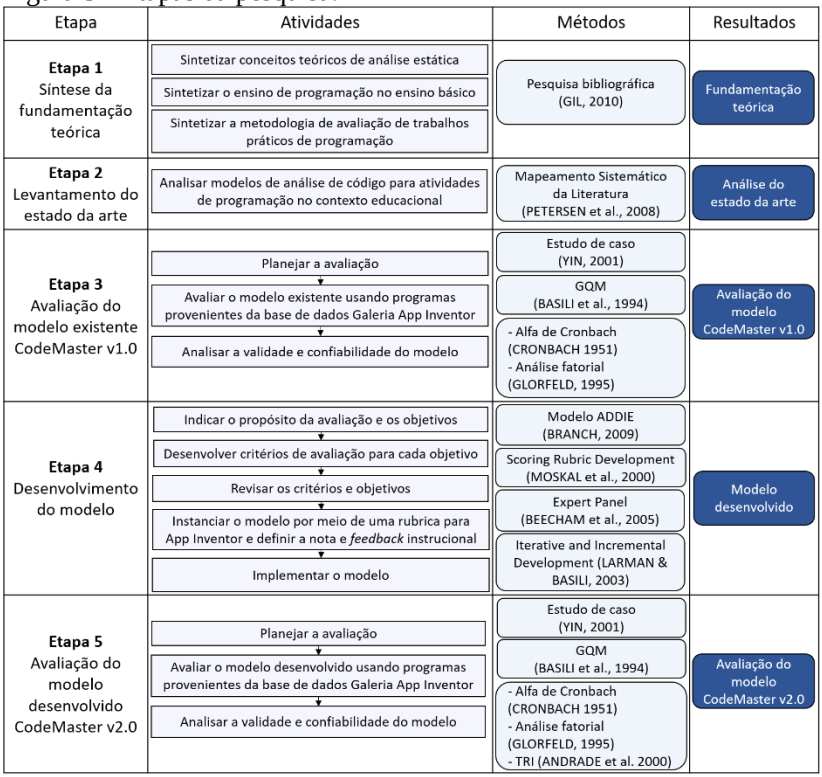
A seguir é apresentada uma descrição detalhada de cada etapa.

Síntese da fundamentação teórica: O objetivo desta etapa é a definição de conceitos de ensino de programação na Educação Básica, metodologia de avaliação de trabalhos práticos de programação no contexto educacional e análise estática. Para isso é realizada uma pesquisa bibliográfica (GIL, 2010) e feita uma síntese teórica sobre conceitos relacionados ao escopo deste trabalho.

Levantamento do estado da arte: Nesta etapa é realizado um mapeamento sistemático de literatura (PETERSEN et al., 2008) com o objetivo de levantar o estado da arte sobre modelos existentes de análise automatizada de atividades de programação no contexto da Educação Básica. Na etapa de definição é estabelecida uma pergunta de pesquisa, seus objetivos e os critérios para inclusão e exclusão de resultados. É definida também a *string* de busca e as bases de dados nas quais a

pesquisa é realizada. Como resultado, é definido o protocolo de busca. Na execução são realizadas buscas nas bases de dados a partir da *string* definida. Usando os critérios de inclusão/exclusão, trabalhos relevantes são selecionados. Na etapa de análise dos resultados, são extraídas informações dos trabalhos selecionados e é realizada uma análise e discussão identificando pontos fortes e fracos de cada trabalho.

Figura 3 - Etapas da pesquisa.



Fonte: elaborada pela autora.

Avaliação em larga escala do modelo CodeMaster v1.0: Identificado na revisão da literatura como um dos principais modelos existentes, é feita uma avaliação de larga escala do modelo existente CodeMaster (GRESSE VON WANGENHEIM et al., 2018). Essa avaliação é feita por meio de um estudo de caso (YIN, 2001) com aplicativos coletados da Galeria AppInventor, os quais são analisados com a ferramenta CodeMaster v1.0. A partir desses dados de avaliação

obtidos pelo CodeMaster v1.0, é realizada uma análise estatística do modelo existente sobre a confiabilidade, por meio do coeficiente Alfa de Cronbach (CRONBACH, 1951) e a validade, por meio de uma análise fatorial (GLORFELD, 1995). Os dados são discutidos e interpretados de forma a identificar sua confiabilidade e validade.

Desenvolvimento do modelo: Seguindo o modelo de design instrucional ADDIE (BRANCH, 2009) e a abordagem de desenvolvimento de rubricas conforme proposto por Moskal e Leydens (2000) o modelo é desenvolvido. Com base no *framework* CSTA (CSTA, 2016) são definidos os objetivos de aprendizagem a serem avaliados. Com base nessa definição são desenvolvidos critérios de avaliação para cada objetivo. O modelo é revisado em relação à validade por meio de um painel de especialistas (BEECHAM et al., 2005). A partir do modelo desenvolvido é instanciada uma rubrica para a VPL App Inventor levando-se em consideração também os resultados do mapeamento sistemático da literatura e da avaliação do modelo CodeMaster v1.0. A partir da rubrica instanciada, é completado o suporte automatizado no CodeMaster para suportar a rubrica definida. A implementação da rubrica segue um processo iterativo e incremental (LARMAN; BASILI, 2003) incluindo a análise de requisitos, modelagem, construção de software e testes.

Avaliação em larga escala do modelo CodeMaster v2.0: O modelo desenvolvido é avaliado por um estudo de caso (YIN, 2001). A partir do método GQM (BASILI; CALDIERA; ROMBACH, 1994) é definido o que deve ser avaliado em relação ao modelo desenvolvido CodeMaster v2.0 e quais instrumentos de coleta de dados são utilizados. Para avaliar o modelo são coletados aplicativos da Galeria AppInventor, os quais são analisados com o avaliador de código CodeMaster v2.0. A partir desses dados é realizada uma análise estatística sobre a confiabilidade do modelo por meio do coeficiente Alfa de Cronbach (CRONBACH, 1951). Além disso, a validade é analisada por meio de uma análise fatorial exploratória (GLORFELD, 1995) e por meio da Teoria da Resposta ao Item (TRI), usando o Modelo de Resposta Gradual (SAMEJIMA, 1969) os parâmetros de inclinação e de dificuldade dos itens são calibrados (ANDRADE; TAVARES; VALLE, 2000). Além disso, os dados são discutidos e interpretados de forma a identificar a confiabilidade e validade do modelo desenvolvido.

1.5 CONTRIBUIÇÕES

O objetivo desde trabalho está inserido na linha de pesquisa de Engenharia de Software do PPGCC/INE/UFSC ligado à área de qualidade de software. O presente trabalho apresenta as seguintes contribuições:

Contribuição no âmbito científico. A principal contribuição científica do presente trabalho na área de engenharia de software/qualidade de software é o desenvolvimento de um modelo conceitual de análise estática de código para a avaliação de qualidade de produtos de software programados com VPL baseada em blocos. A partir do modelo desenvolvido pode-se criar sistematicamente novos analisadores de código para VPL baseada em blocos.

Outra contribuição é o levantamento do estado de arte de forma sistemática fornecendo uma visão geral sobre esta questão de pesquisa. Além disso, são apresentadas evidências empíricas em relação a sua validade e confiabilidade.

Contribuição no âmbito tecnológico. Em relação ao âmbito tecnológico o presente trabalho contribuirá na evolução do CodeMaster (GRESSE VON WANGENHEIM et al., 2018) visando a implementação das principais métricas do modelo proposto como modelo de avaliação. A evolução do CodeMaster (GRESSE VON WANGENHEIM et al., 2018) será disponibilizada publicamente no site da Iniciativa Computação na Escola (CNE, 2019).

Contribuição no âmbito social. Ao dispor de um modelo para analisadores de código em VPL baseada em blocos, pode-se utilizar este tipo de software para o ensino de computação na Educação Básica. De forma concreta, o impacto social deste trabalho é automatizar e reduzir o tempo necessário para a correção de programas criados por alunos, melhorando a avaliação e o ensino de programação por pessoas não necessariamente especialistas na área. Desta forma, a disponibilidade de um analisador dessa natureza pode contribuir a ampliar a integração do ensino de computação de forma interdisciplinar na Educação Básica em escolas Brasileiras. Com um software deste tipo, é possível dar uma avaliação e um *feedback* mais completo e rico de informações de forma individual/personalizada. Assume-se que ao receber uma avaliação mais detalhada e imediata contribui-se na aprendizagem e consequentemente na qualidade do ensino, contribuindo, desta forma, na preparação de indivíduos preparados para os desafios século XXI.

1.6 ESTRUTURA DO TRABALHO

Este trabalho está dividido em 7 capítulos. No capítulo 2, são descritos os fundamentos teóricos para facilitar a compreensão dos principais conceitos utilizados nesta pesquisa. No capítulo 3 é fornecida uma visão geral sobre o estado da arte das abordagens de avaliação de código em VPL. No Capítulo 4 é apresentada a avaliação do modelo CodeMaster v1.0 para a VPL App Inventor por meio de um estudo de caso usando uma base de dados de mais de 88 mil programas.

O Capítulo 5 apresenta o desenvolvimento do modelo e a instanciação do modelo por meio de uma rubrica para a avaliação de programas criados com App Inventor. O modelo é voltado para avaliação do pensamento computacional via código de programas, criados pelo aluno como resultado de atividades abertas. É apresentada também a evolução da ferramenta CodeMaster. No Capítulo 6 é apresentada a avaliação do modelo CodeMaster por meio da instanciação da rubrica VPL App Inventor. A avaliação é feita por meio de um estudo de caso, usando uma base de dados de mais de 88 mil programas. A conclusão e as considerações finais são apresentadas no Capítulo 7.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 ENSINO DE COMPUTAÇÃO NA EDUCAÇÃO BÁSICA

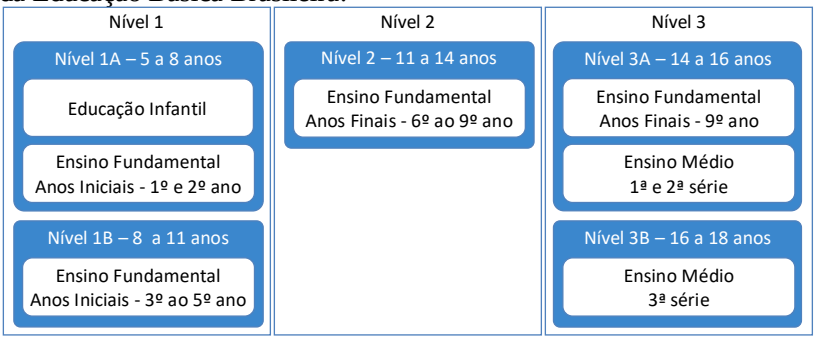
Atualmente a computação encontra-se em evidência na maioria das atividades. Independentemente da área de atuação de um profissional, é importante que este conheça os fundamentos e princípios básicos da computação para que possa exercer sua atividade de forma plena. No entanto, a população em geral não é tão bem informada nesta área como deveria ser, existindo uma lacuna de conhecimento acerca da ciência da computação em todos níveis (CSTA, 2016).

Com o objetivo de mitigar os problemas gerados por esta falta de conhecimento básico, surgiram diversas iniciativas ao redor do mundo no sentido de popularizar o ensino de computação para todos (BOCCONI, 2016). A inserção de computação já na Educação Básica é vista como uma das abordagens mais efetivas para esta questão. Em todos os lugares do mundo vários esforços estão sendo feitos nesse sentido. Recentemente, no Brasil, em dezembro de 2018, o MEC homologou a BNCC - Base Nacional Comum Curricular - que apresenta o desenvolvimento do pensamento computacional dentro da área de Matemática (MEC, 2018).

> A área de Matemática, no Ensino Fundamental, centra-se na compreensão de conceitos e procedimentos em seus diferentes campos e no **desenvolvimento do pensamento computacional**, visando à resolução e formulação de problemas em contextos diversos. (MEC; BNCC, 2018, p. 471).

É neste contexto mundial que foi desenvolvido o *K-12 Computer Science Framework* pela Associação de Professores de Ciência da Computação – *Computer Science Teachers Association* (CSTA, 2016), composta por especialistas em educação e computação do mundo todo. O *framework* serve como um guia de alto nível que pode ser usado para o desenvolvimento de currículos personalizados especificamente de computação.

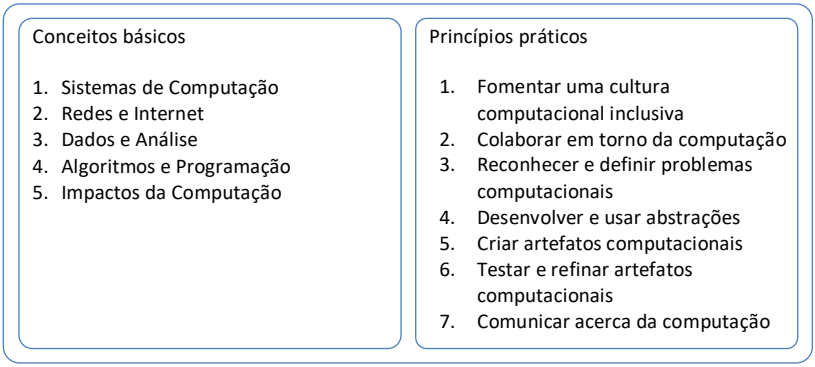
Figura 4 - Níveis de ensino de computação localizados para o contexto da Educação Básica Brasileira.



Fonte: elaborada pela autora com base em CSTA (2016) e BNCC (2018).

O *framework* CSTA K-12 apresenta três níveis gerais com objetivos de aprendizagem para faixas etárias compreendidas desde a Educação Infantil até o Ensino Médio (Figura 4). Para o contexto da Educação Básica Brasileira, pode-se dizer que o nível 1 corresponde a faixa de alunos da Educação Infantil e Ensino Fundamental – Anos Iniciais. O nível 2 é direcionado a alunos do Ensino Fundamental – Anos Finais. Finalmente, o nível 3, apresenta objetivos voltados para alunos do Ensino Fundamental – Anos Finais e Ensino Médio.

Figura 5 - Conceitos e práticas do *K-12 Computer Science Framework*.



Fonte: adaptado de CSTA (2016) – versão traduzida.

Visando a aprendizagem da computação na Educação Básica, o CSTA K-12 apresenta **conceitos básicos** e **princípios práticos** (Figura 5). Os conceitos básicos são divididos em categorias que representam

grandes áreas no campo da computação. Os princípios práticos referem-se a comportamentos que alunos alfabetizados computacionalmente utilizam para se envolver plenamente com os conceitos básicos computação (CSTA, 2016). Cada conceito básico contém uma série de subconceitos que se relacionam com um ou mais princípios práticos. As progressões de aprendizagem via níveis, conforme apresentados na Figura 4, fornecem um vínculo que liga o aprendizado de conceitos básicos e princípios práticos dos alunos da Educação Infantil ao Ensino Médio (CSTA, 2016).

2.1.1 Pensamento computacional

O pensamento computacional refere-se aos processos de pensamento envolvidos na criação de soluções algorítmicas, ou passo-a-passo, que podem ser executadas por um computador (WING, 2006). É uma competência relevante universal, não apenas reservada a cientistas da computação (WING, 2006).

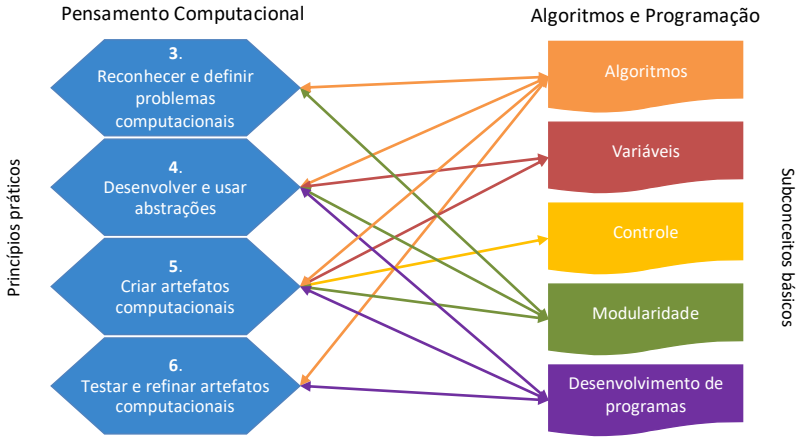
> O pensamento computacional é uma maneira pela qual os **seres humanos resolvem problemas**; não é uma forma de fazê-los pensar como computadores. Computadores são néscios e monótonos; seres humanos são inteligentes e criativos. (WING, 2006, p. 35 – traduzido).

O *framework* CSTA K-12 (2016) delineia o pensamento computacional pelos princípios práticos 3, 4, 5 e 6 (Figuras 5 e 6). Os princípios práticos 1, 2 e 7 são independentes (Figura 5), isto é, consistem em práticas gerais da ciência da computação que complementam o pensamento computacional.

Segundo o *framework* CSTA K-12 (2016), a ciência da computação oferece oportunidades únicas para desenvolver o pensamento computacional e as práticas do *framework* podem ser aplicadas a outras áreas além da ciência da computação. No entanto, no contexto deste trabalho é explorada a relação entre o conceito (e subconceitos) de **algoritmos e programação** e a prática do **pensamento computacional** (Figura 6), conforme proposto pelo *K-12 Computer Science Standards* (CSTA, 2017), para avaliar o produto de software criado pelo aluno. Além disso, a etapa do ensino explorada é entre o Ensino Fundamental – Anos Iniciais (Figura 4 – nível 1B), Ensino

Fundamental – Anos Finais (Figura 4 – nível 2) e Ensino Médio (Figura 4 – nível 3A).

Figura 6 – Relação entre a prática do pensamento computacional e o conceito (e subconceitos) de algoritmos e programação.



Fonte: elaborada pela autora com base em CSTA (2017).

Um **algoritmo** é uma sequência de passos projetados para realizar uma tarefa específica (CSTA, 2016). Algoritmos podem ser implementados por meio de uma linguagem de programação e são usados para controlar todos os sistemas de computação, possibilitando as pessoas resolverem problemas e se comunicarem com o mundo de novas maneiras. Algoritmos são projetados para serem realizados por seres humanos e computadores. Na Educação Infantil e Ensino Fundamental – Anos Iniciais, os alunos aprendem sobre algoritmos simples do mundo real. À medida que progridem, os alunos aprendem sobre o desenvolvimento, combinação e decomposição de algoritmos, bem como a avaliação de algoritmos concorrentes (Tabela 1).

Tabela 1 - Objetivos de aprendizagem do subconceito de algoritmo para o Ensino Fundamental e Ensino Médio (CSTA, 2017).

Nível 1B (8-11 anos)	**Nível 2 (11-14 anos)**	**Nível 3A (14-16 anos)**
1B-AP-08. Comparar e refinar vários algoritmos para a mesma tarefa e determinar qual é o mais adequado. (*P6.3, P3.3*)	**2-AP-10.** Usar fluxogramas e/ou pseudocódigo para resolver problemas complexos como algoritmos. (*P4.4, P4.1*)	**3A-AP-13.** Criar protótipos que utilizam algoritmos para resolver problemas computacionais aproveitando-se de conhecimentos e interesses pessoais anteriores. (*P5.2*)

Variáveis servem para armazenar e manipular dados de programas de computador. Na Educação Infantil e Ensino Fundamental – Anos Iniciais, os alunos aprendem que diferentes tipos de dados, como palavras, números ou imagens, podem ser usados de diferentes maneiras. À medida que progridem, os alunos aprendem sobre variáveis e formas de organizar grandes coleções de dados em estruturas de dados de crescente complexidade (Tabela 2).

Tabela 2 - Objetivos de aprendizagem do subconceito de variáveis para o Ensino Fundamental e Ensino Médio (CSTA, 2017).

Nível 1B (8-11 anos)	**Nível 2 (11-14 anos)**	**Nível 3A (14-16 anos)**
1B-AP-09. Criar programas que usam variáveis para armazenar e modificar dados. (*P5.2*)	**2-AP-11** Criar variáveis com definição clara que representam diferentes tipos de dados e executem operações sobre seus valores. (*P5.1, P5.2*)	**3A-AP-14.** Usar listas para simplificar soluções, generalizando problemas computacionais ao invés de usar repetidamente variáveis simples. (*P4.1*)

As estruturas de **controle** especificam a ordem na qual as instruções são executadas dentro de um algoritmo ou programa. Na Educação Infantil e Ensino Fundamental – Anos Iniciais, os alunos aprendem sobre a execução sequencial e estruturas de controle simples. À medida que progridem, os alunos expandem sua compreensão para combinações de estruturas que suportam a execução complexa (Tabela 3).

Tabela 3 - Objetivos de aprendizagem do subconceito de controle para o Ensino Fundamental e Ensino Médio (CSTA, 2017).

Nível 1B (8-11 anos)	**Nível 2 (11-14 anos)**	**Nível 3A (14-16 anos)**
1B-AP-10. Criar programas que incluam sequências, eventos, laços e condicionais. (P5.2)	**2-AP-12.** Projetar e desenvolver iterativamente programas que combinam estruturas de controle, incluindo laços aninhados e condicionais compostos. (*P5.1, P5.2*)	**3A-AP-15.** Justificar a seleção de estruturas de controle específicas quando os *trade-off*s envolvem implementação, legibilidade e desempenho do programa e explicar os benefícios e desvantagens das escolhas feitas. (*P5.2*)
		3A-AP-16. Projetar e desenvolver iterativamente artefatos computacionais para uso prático, expressão pessoal ou para resolver uma questão social usando eventos para iniciar instruções. (*P5.2*)

A **modularidade** envolve dividir tarefas complexas em tarefas mais simples e combiná-las para criar algo mais complexo. Na Educação Infantil e Ensino Fundamental – Anos Iniciais, os alunos aprendem que algoritmos e programas podem ser projetados dividindo tarefas em partes menores e recombinando soluções existentes. À medida que progridem, os alunos aprendem a reconhecer padrões para usarem soluções gerais e reutilizáveis para cenários comuns e descrevem claramente as tarefas de forma amplamente utilizável (Tabela 4).

Tabela 4 - Objetivos de aprendizagem do subconceito de modularidade para o Ensino Fundamental e Ensino Médio (CSTA, 2017).

Nível 1B (8-11 anos)	**Nível 2 (11-14 anos)**	**Nível 3A (14-16 anos)**
1B-AP-11. Decompor (quebrar) problemas em subproblemas menores e gerenciáveis para facilitar o processo de desenvolvimento. (*P3.2*)	**2-AP-13.** Decompor problemas e subproblemas em partes para facilitar o projeto, implementação e revisão de programas. (*P3.2*)	**3A-AP-17.** Decompor problemas em componentes menores por meio de análises sistemáticas, usando construtos como procedimentos, módulos e/ou objetos. (P3.2)
1B-AP-12. Modificar, *remixar* ou incorporar partes de um programa existente em seu próprio trabalho, para desenvolver algo novo ou adicionar recursos mais avançados. (*P5.3*)	**2-AP-14.** Criar procedimentos com parâmetros para organizar o código e torná-lo mais fácil de reutilizar. (*P4.1, P4.3*)	**3A-AP-18.** Criar artefatos usando procedimentos dentro de um programa, combinações de dados e procedimentos, ou programas independentes, mas inter-relacionados. (*P5.2*)

O **desenvolvimento de programas** é feito por meio de um processo de engenharia de software que muitas vezes é repetido até que o programador e/ou cliente estejam satisfeitos com a solução. Na Educação Infantil e Ensino Fundamental – Anos Iniciais, os alunos aprendem como e por que as pessoas desenvolvem programas. À medida que progridem, os alunos aprendem sobre os *trade-offs* no projeto do programa associado a decisões complexas envolvendo restrições de usuários, eficiência, ética e testes.

Tabela 5 - Objetivos de aprendizagem do subconceito de desenvolvimento de programas para o Ensino Fundamental e Ensino Médio (CSTA, 2017).

Nível 1B (8-11 anos)	**Nível 2 (11-14 anos)**	**Nível 3A (14-16 anos)**
1B-AP-13. Usar um processo iterativo para planejar o desenvolvimento de um programa, incluindo	**2-AP-15.** Procurar e incorporar comentários (*feedback*) de membros da equipe e usuários para	**3A-AP-19.** Conceber e desenvolver programas sistematicamente para grandes públicos,

Nível 1B (8-11 anos)	Nível 2 (11-14 anos)	Nível 3A (14-16 anos)
... as perspectivas dos outros e considerando as preferências de usuário. (*P1.1, P5.1*)	... refinar uma solução que atenda às necessidades dos usuários. (*P2.3, P1.1*)	... incorporando *feedback* dos usuários (*P5.1*).
1B-AP-14. Observar os direitos de propriedade intelectual e atribuir o devido reconhecimento ao criar ou *remixar* programas. (*P7.3*)	**2-AP-16.** Incorporar código, mídia e bibliotecas existentes em programas originais e dar reconhecimento (ao autor original). (*P4.2, P5.2, P7.3*)	**3A-AP-20.** Avaliar as licenças que limitam ou restringem o uso de artefatos computacionais ao usar recursos, como p.ex. bibliotecas. (*P7.3*)
1B-AP-15. Testar e depurar (identificar e corrigir) erros um programa ou algoritmo para garantir que ele seja executado conforme planejado. (*P6.1, P6.2*)	**2-AP-17.** Testar sistematicamente e refinar programas usando vários de casos de teste. (*P6.1*)	**3A-AP-21.** Avaliar e aperfeiçoar artefatos computacionais para torná-los mais utilizáveis e acessíveis. (*P6.3*)
1B-AP-16. Assumir papéis variados, com orientação do professor, ao colaborar com colegas durante o projeto, implementação e etapas de revisão do desenvolvimento do programa. (*P2.2*)	**2-AP-18.** Distribuir tarefas e manter um cronograma para o projeto quando se está desenvolvendo artefatos computacionais de forma colaborativa. (*P2.2*)	**3A-AP-22.** Projetar e desenvolver artefatos computacionais trabalhando em equipe, usando ferramentas colaborativas. (*P2.4*)
1B-AP-17. Descrever as escolhas feitas durante o desenvolvimento do programa usando comentários de código, apresentações e demonstrações. (*P7.2*)	**2-AP-19.** Documentar programas para torná-los mais fáceis de entender, testar e depurar. (*P7.2*)	**3A-AP-23.** Documentar decisões de projeto usando texto, gráficos, apresentações e/ou demonstrações no desenvolvimento de programas complexos. (*P7.2*)

2.1.2 Atividades de ensino do pensamento computacional

Existem diversas maneiras de desenvolver o pensamento computacional. Apesar de ser um conceito relativamente novo na área da educação, tendo sido definido formalmente em 2006 por Wing (2006) e adicionado na BNCC em 2018, no Brasil já foram realizados vários estudos nesse contexto (SANTOS; ARAUJO; BITTENCOURT, 2018). Atualmente já existem unidades instrucionais baseadas nas diretrizes CSTA K-12 para se ensinar computação na Educação Básica (GROVER; PEA, 2013; LYE; KOH, 2014; ALVES et al., 2017; DANIEL et al., 2017). Estas unidades abordam principalmente o pensamento computacional e conceitos de algoritmos e programação por meio da aprendizagem baseada em problemas.

Tipicamente, as atividades práticas de unidades instrucionais para desenvolver o pensamento computacional consistem em atividades

abertas (*ill-structured*) de programação. Atividades abertas envolvem a criação de artefatos computacionais, como por exemplo, criação de jogos, animações e aplicativos (BRENNAN; RESNICK, 2012; SEITER; FOREMAN, 2013; DANIEL et al., 2017; ALVES et al., 2017) de forma livre a partir de um tema predefinido. No início da unidade são apresentados os subconceitos básicos de algoritmos e programação, como por exemplo, variáveis, estruturas de controle, etc., e suas aplicações na prática por meio de um programa exemplo. A partir disso, os alunos concebem uma ideia com base num tema e começam a implementá-la com uma VPL baseada em blocos por meio de um ambiente de aprendizagem construcionista. Estas unidades utilizam a abordagem de atividades abertas a partir de um tema predefinido.

Cabe ressaltar que há também a abordagem de atividades fechadas as quais são integralmente predefinidas como, por exemplo, as atividades da iniciativa *Hour of Code* (CODE, 2015) e *Code Combat* (CODECOMBAT, 2016). As atividades fechadas apresentam uma descrição completa de tudo o que o aluno deve programar, isto é, contam com um roteiro de instruções e o resultado da atividade é o mesmo para todos (veja a comparação na Tabela 6). As atividades fechadas são importantes para que o aluno possa comparar sua solução com a solução esperada sempre que sentir necessidade, permitindo que crie mais confiança sobre sua capacidade de solucionar problemas. Tipicamente, são vistas como sendo mais convenientes para gerenciar o processo de ensino-aprendizagem, pois basta uma comparação com o gabarito para que a solução seja avaliada.

Os problemas do mundo real, no entanto, não seguem essa estrutura de atividade fechada e são mais similares às atividades abertas (DANIELS et al., 2007). As atividades abertas possuem várias possíveis soluções, além disso o julgamento sobre qual caminho tomar para chegar a uma solução cabe ao aluno (LYE; KOH, 2014; GIJSELAERS, 1996). A vantagem de utilizar atividades abertas é que se incentiva e recompensa a criatividade e o pensamento analítico dos alunos (SURIF; IBRAHIM; DALIM, 2014). Além disso, o processo de desenvolvimento para criar programas úteis e eficientes é estimulado, já que envolve a escolha de quais informações utilizar e como processá-las e armazená-las, separando grandes problemas em pequenos, recombinando soluções existentes e analisando diferentes soluções.

Tabela 6 - Comparação entre atividades abertas e fechadas (SURIF et al., 2014).

Tópico	**Atividade fechada (*well-structured*)**	**Atividade aberta (*ill-structured*)**
Solução	É conhecida e sempre existe.	Existem várias soluções. Algumas vezes não há solução.
Informação de apoio	Todas as informações necessárias são dadas.	Não são dadas todas as informações necessárias, o aluno deve buscá-las.
Conceitos necessários	Os conceitos necessários para a solução são delimitados numericamente e são organizados passo-a-passo.	Envolve vários conceitos e o aluno tem a liberdade de escolher qual conceito usar.
Método de solução	Tipicamente não existe um método preferido, diferentes métodos chegarão na mesma solução desejada.	Tem várias alternativas de caminhos que chegam em diferentes soluções.
Avaliação	As soluções ou são corretas ou são incorretas	As soluções podem ser consideradas parcialmente corretas ou incorretas. Além disso julgamentos sobre o contexto podem ser requeridos.

2.1.3 Ambiente de VPL baseada em blocos

Ambientes de programação consistem em plataformas que permitem a execução de programas. Os ambientes de programação baseada em blocos são restritos a Linguagens de Programação Visual (*Visual Programming Language* – VPL). As VPLs baseada em blocos permitem a programação via blocos de comandos gráficos em vez de textuais (Figura 7).

Figura 7 – Exemplo de categorias e código em uma VPL baseada em blocos.

Fonte: elaborada pela autora com base em scratch.mit.edu.

Estes ambientes assemelham os blocos de programação a elementos de um *puzzle*, isto é, a peças de um quebra-cabeça (WEINTROP; WILENSKY, 2015) em que cada bloco possui uma cor

que corresponde à cor da sua categoria (Figura 7). Para programar, deve-se encaixar os blocos de uma forma lógica. Desta forma, a maioria das VPL baseada em blocos não permite a geração de erros sintáticos, haja vista o formato gráfico de cada bloco não permite o encaixe de blocos que não satisfazem a sintaxe da linguagem (WEINTROP; WILENSKY, 2015).

Atualmente existem diversos ambientes de VPL baseada em blocos. Exemplos são Alice, Agent Sheets, App Inventor, Blockly, Scratch e Snap! (Tabela 7). Esses ambientes permitem a criação de programas complexos, como por exemplo, jogos 2D e 3D, aplicativos móveis e animações.

Tabela 7 - Exemplos de ambientes de programação de VPL baseada em blocos.

Ambiente	Escopo
Alice (www.alice.org)	Permite criar programas 3D, como jogos, animações, etc.
Agent Sheets (www.agentsheets.com)	Permite criar jogos usando design de games.
App Inventor (appinventor.mit.edu)	Permite criar aplicativos móveis para Android.
Blockly (developers.google.com/blockly)	Permite praticar programação via desafios de programação.
Scratch (scratch.mit.edu)	Permite criar programas, como jogos 2D, animações, etc.
Snap! (byob.berkeley.edu)	Permite criar programas, como jogos, animações, etc. e praticar conceitos avançados de programação.

Estas linguagens contêm diversas funções e comandos básicos que refletem conceitos de algoritmos e programação, como por exemplo, estruturas de controle, variáveis, etc. Algumas linguagens também possuem suporte a conceitos mais avançados ou de outras dimensões do pensamento computacional. No caso de Snap! há comandos que refletem conceitos mais complexos, como por exemplo, cálculo lambda (HARVEY et al., 2012). App Inventor, voltado à criação de aplicativos móveis, contém comandos relacionados à subdimensão do pensamento computacional móvel, envolvendo comandos que usam recursos do *smartphone*.

Entre as VPL baseada em blocos mais proeminentes da atualidade estão Scratch, AppInventor e Snap!. Observando que para Scratch já existem diversas soluções de análise automatizada de programas resultantes de atividades abertas, neste trabalho será focada a análise de programas na linguagem App inventor resultantes de

atividades abertas, seguindo o escopo do CodeMaster v1.0 (GRESSE VON WANGENHEIM et al., 2018).

2.1.3.1 App Inventor

App Inventor é um ambiente de programação com uma VPL baseada em blocos para a criação de aplicativos móveis para dispositivos Android (Figura 8). É um projeto de código aberto que foi originalmente criado pela Google e atualmente é mantido pelo *Massachusetts Institute of Technology* (MIT). O App Inventor fornece um ambiente de programação *on-line* baseado em navegador com um editor *drag-and-drop* (arraste e solte). A versão atual é o App Inventor 2.0, sendo que o App Inventor Classic foi retirado de produção em 2015. Para visualizar a execução de um aplicativo, é necessário um *smartphone* rodando o sistema operacional Android. App Inventor está disponível em várias línguas, inclusive em português do Brasil.

Figura 8 - Interface do ambiente App Inventor.

Fonte: elaborada pela autora com base em appinventor.mit.edu.

Com o App Inventor, um aplicativo móvel pode ser criado em dois estágios. Primeiro, os componentes da interface do usuário são definidos no Designer (Figura 8). O Designer também permite especificar componentes não visuais, como sensores, componentes sociais e de mídia, que acessam recursos do telefone ou de outros aplicativos. Em um segundo estágio, na área de Blocos, o comportamento do aplicativo é especificado via programação, conectando blocos visuais (Figura 8). Alguns blocos representam eventos, condições ou ações para um componente de aplicativo específico (por exemplo, botão pressionado, tirar uma foto com a câmera), enquanto outros representam conceitos de algoritmos e programação, como variáveis, laços, etc. (TURBAK et al., 2017).

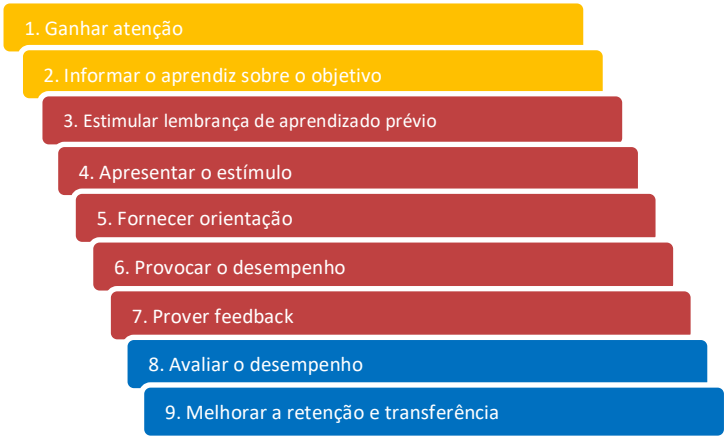
Tabela 8 - Componentes da área de Designer e comandos da área de Blocos do App Inventor.

	Componente	Descrição
Designer	Interface de Usuário	Todos os componentes visíveis do aplicativo para criar a interface visual, como botões, imagens, caixas de texto, etc.
	Organização	Componentes que auxiliam na organização dos componentes visíveis da interface do usuário.
	Mídia	Todos os componentes de mídia, como câmera, reprodutores de mídia, tradutores, etc.
	Desenho e Animação	Componentes que permitem ao usuário desenhar e visualizar animações.
	Mapas	Componentes que permitem inserção de mapas com marcadores.
	Sensores	Componentes que interagem com os sensores do dispositivo móvel, como acelerômetro, scanner de código de barras, giroscópio, localização, pedômetro, etc.
	Social	Componentes que permitem que o aplicativo se comunique com outros aplicativos sociais, como e-mail, chamadas telefônicas, mensagens de texto, Twitter, etc.
	Armazenamento	Componentes que permitem a criação de diferentes tipos de bancos de dados para armazenar dados.
	Conectividade	Componentes que permitem a conexão com outros dispositivos ou aplicativos via Bluetooth, WEB ou API (*Application Programming Interface*).
	LEGO® MINDSTORMS®	Componentes que proveem uma interface de alto nível para o robô LEGO MINDSTORMS
	Experimental	Componentes em fase de testes.
	Extensões	Opção que permite adicionar bibliotecas com blocos adicionais de programação.
Blocos	**Comando**	**Descrição**
	Controle	Comandos responsáveis pelo controle de laços, condicionais e iterações em listas
	Lógica	Comandos responsáveis por operações lógicas em variáveis e operandos booleanos.
	Matemática	Comandos responsáveis por operações matemáticas em variáveis, incluindo operações básicas como adição, subtração, multiplicação e divisão, bem como operações mais complexas, como seno, cosseno etc.
	Texto	Comandos responsáveis pela manipulação de texto, incluindo comandos para concatenação de *strings*, divisão de *strings*, etc.
	Listas	Comandos responsáveis pela criação e manipulação de dados de lista.
	Cores	Comandos responsáveis pela definição das cores dos componentes da aplicação.
	Variáveis	Comandos responsáveis pela criação e manipulação de variáveis.
	Procedimentos	Comandos responsáveis pela definição de procedimentos e funções.
	Eventos	Comandos que tratam eventos disparados em relação aos componentes do Designer.
	Get and Set	Comandos que alteram atributos dos componentes do Designer.

2.2 AVALIAÇÃO DE ATIVIDADES ABERTAS

A avaliação é uma etapa importante no processo de ensino-aprendizagem. Comumente, a avaliação tem como objetivo avaliar o desempenho de alguém ou de algo, isto é, busca verificar se o que está sendo avaliado demonstra as competências desejáveis. A partir do resultado da avaliação, uma nota pode ser atribuída ao objeto avaliado e um *feedback* personalizado pode ser fornecido (BRANCH, 2009). A avaliação é importante tanto para o aluno, de forma a conhecer seu desempenho, como para o professor, para melhorar a retenção e transferência de conhecimento (Figura 9).

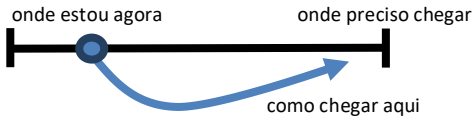
Figura 9 - Nove eventos de aprendizagem de Gagne.



Fonte: adaptado de Gagne (1965) – versão traduzida.

Para aprender, um aluno precisa saber três coisas: o que é considerado um bom desempenho em uma determinada atividade; como seu próprio desempenho se relaciona com o bom desempenho; e o que fazer para fechar esta lacuna (STEGEMAN; BARENDSEN; SMETSERS, 2016).

Figura 10 - Esquema de *feedback* instrucional.



Fonte: adaptado de Sadler (1989) – versão traduzida.

O CSTA recomenda que cada objetivo de aprendizagem seja claro, observável e mensurável, ou seja, espera-se que o aluno entenda o que deve ser feito e que o professor tenha uma forma de verificar e medir o artefato/ação produzido pelo aluno (CSTA, 2016).

De acordo com o design instrucional, uma avaliação pode ser classificada em: **diagnóstica**, **formativa** e **somativa**. A avaliação diagnóstica é realizada no início do processo e consiste em avaliar ou identificar os conhecimentos dos alunos acerca do tema antes da intervenção instrucional (BRANCH, 2009). A avaliação formativa é o processo de coletar dados sobre a aprendizagem do aluno durante o ensino, de forma a revisar o processo de instrução para que este seja mais efetivo. A avaliação formativa é muito importante para o aluno pois permite que o mesmo compreenda o seu processo de aprendizagem e identifique deficiências e/ou dificuldades na aprendizagem (GIPPS, 1994). A avaliação somativa é o processo de coleta de dados após o processo de ensino e tem caráter classificatório, tipicamente alocando-se uma nota a fim de determinar o grau de atingimento dos objetivos de aprendizagem (BRANCH, 2009). São utilizados critérios gerais baseados nos objetivos de aprendizagem de forma a sintetizar a aprendizagem do aluno tendo em evidência a perspectiva de conclusão, pois é realizada no final do processo educacional (BRANCH, 2009).

A avaliação de atividades abertas de programação tipicamente é realizada de forma somativa, ou seja, é realizada ao final do processo instrucional, atribuindo-se uma nota sobre o programa/artefato do aluno já totalmente desenvolvido.

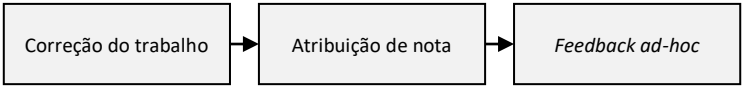
A atribuição de nota pode se referir a um valor quantitativo ou qualitativo, isto é, podem ser atribuídos conceitos, como por exemplo, de ‘A’ a ‘E’, ou valores dentro de um intervalo definido, como, por exemplo, de 0 a 10, e porcentagens de 0 a 100. No Brasil não há uma padronização para atribuição de notas na avaliação. No entanto, no Ensino Fundamental, usualmente, adota-se a alocação de notas de forma quantitativa no intervalo [0, 10], na qual alunos que desempenharam a tarefa perfeitamente dentro do esperado recebem a nota máxima, ou seja, a nota 10 (CME, 2011). Para atribuir uma nota, pode-se utilizar um **sistema de pontuação dicotômico**, **politômico** ou **composto** (Tabela 9).

Tabela 9 - Tipos de pontuação (ALBANO, 2017).

Dicotômica	Politômica	Composta
Atribui um de apenas dois valores possíveis.	Atribui um de três ou mais valores possíveis.	Combina notas de várias áreas ou componentes.
São mutuamente exclusivos e de fácil atribuição.	É menos direta e menos objetiva do que a pontuação dicotômica, sendo difícil manter o significado consistente de categorias atribuídas como parcialmente corretas.	Cada componente pode ter uma pontuação dicotômica, politômica ou um modelo de medição.
O exemplo mais comum é a pontuação que representa uma resposta de sim ou não e certo ou errado.	Pode ser encontrado como conceitos na forma: insuficiente, regular, bom, excelente, ou na forma de nota num intervalo [0,10].	Pode consistir na soma de componentes dicotômicos, politômicos ou resultado de um modelo de medição via uma fórmula complexa.

Na avaliação manual de atividades abertas, tipicamente o professor realiza a avaliação dos tópicos avaliados. A partir disso, atribui uma nota individualmente e fornece *feedback* instrucional dependendo das questões observadas nos trabalhos dos alunos (Figura 11).

Figura 11 - Processo de avaliação manual de atividades abertas de programação.



Fonte: elaborada pela autora.

Não há consenso na literatura sobre quais critérios de medição devem ser comuns para avaliação de atividades abertas de programação (ALA-MUTKA; JARVINEN, 2004; BRENNAN; RESNICK, 2012; GROVER; COOPER; PEA, 2014). Normalmente, a funcionalidade correta, o bom projeto e estilo de programação são características desejáveis. No entanto, as definições e pesos relativos destas características na avaliação variam muito. Diferentes professores enfatizam características diferentes, baseando seus critérios em sua experiência pessoal e no sentido dos objetivos do curso (ALA-MUTKA; JARVINEN, 2004; FITZGERALD et al., 2013). Outros examinam somente erros de programação mais comuns como: seleção de item fora do limite, que consiste na tentativa de obter o item de índice maior do que o comprimento de uma lista, por exemplo; tentativa de realizar divisão por zero; excesso e/ou falta de argumentos quando se tenta invocar um método com mais/menos argumentos do que a quantidade

de parâmetros que foi definido para ele; e ausência de retorno, quando há tentativa de invocar um método sem retorno em uma instrução que requer um valor de retorno.

2.2.1 Rubrica

Para se realizar uma correção uniforme e consistente de atividades abertas de programação costuma-se utilizar rubricas. Rubricas consistem em ferramentas para definição de métricas de avaliação mais transparentes e eficazes (LOBATO et al., 2007).

> As rubricas permitem um **detalhamento mais claro** do processo de avaliação e ao mesmo tempo **facilitam o diagnóstico** de problemas específicos dentro do processo de ensino-aprendizagem. (LOBATO et al., 2007, p. 5).

Uma rubrica é uma ferramenta que auxilia na avaliação do trabalho do aluno. É uma grade de critérios que consiste em:

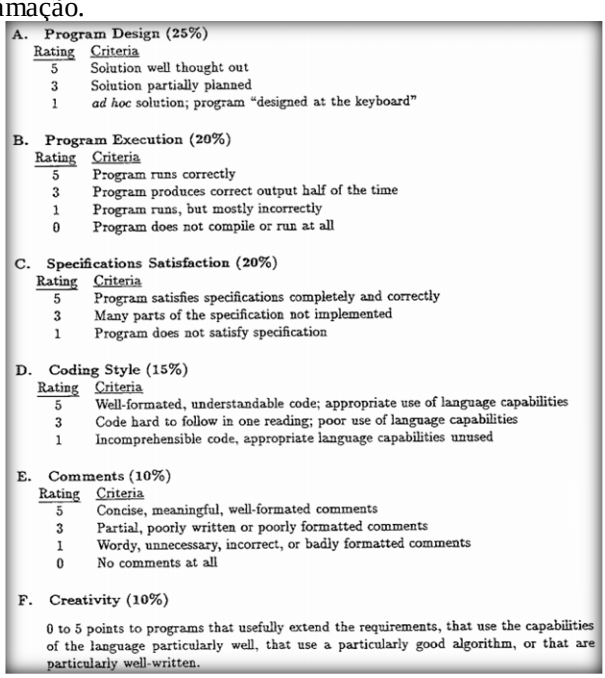
(1) um conjunto de **critérios**, para identificar a característica que deve ser medida,

(2) uma determinada **escala de classificação** com vários níveis de desempenho que podem ser atingidos junto com um sistema de pontuação (qualitativo ou quantitativo) relacionado a cada nível de desempenho, e

(3) **descritores verbais** que detalham textualmente para cada critério os níveis de desempenho (veja um exemplo na Figura 12).

A rubrica pode ser usada para calcular notas e para fornecer *feedback*. Uma nota consiste em uma indicação do nível de desempenho refletido pela(s) pontuação(ões) específica(s) de uma avaliação. As rubricas foram desenvolvidas para realizar a avaliação somativa e classificação/atribuição de nota, isto é, sem fornecer *feedback* relacionado ao conteúdo. No entanto, elas podem ser utilizadas para funcionar como uma ferramenta de avaliação formativa, de forma a fornecer *feedback* aos alunos que podem usá-las para melhorar seu desempenho (STEGEMAN; BARENDSEN; SMETSERS, 2016).

Figura 12 - Exemplo de rubrica utilizada em disciplinas de Programação.



A. Program Design (25%)

Rating	Criteria
5	Solution well thought out
3	Solution partially planned
1	*ad hoc* solution; program "designed at the keyboard"

B. Program Execution (20%)

Rating	Criteria
5	Program runs correctly
3	Program produces correct output half of the time
1	Program runs, but mostly incorrectly
0	Program does not compile or run at all

C. Specifications Satisfaction (20%)

Rating	Criteria
5	Program satisfies specifications completely and correctly
3	Many parts of the specification not implemented
1	Program does not satisfy specification

D. Coding Style (15%)

Rating	Criteria
5	Well-formated, understandable code; appropriate use of language capabilities
3	Code hard to follow in one reading; poor use of language capabilities
1	Incomprehensible code, appropriate language capabilities unused

E. Comments (10%)

Rating	Criteria
5	Concise, meaningful, well-formated comments
3	Partial, poorly written or poorly formatted comments
1	Wordy, unnecessary, incorrect, or badly formatted comments
0	No comments at all

F. Creativity (10%)

0 to 5 points to programs that usefully extend the requirements, that use the capabilities of the language particularly well, that use a particularly good algorithm, or that are particularly well-written.

Fonte: Howatt (1994).

A avaliação do desempenho do aluno foca na capacidade de aplicação de conceitos aprendidos em contextos de atividades abertas em que não há somente uma resposta correta para determinada atividade (WIGGINS, 1993). Neste tipo de avaliação, o foco é no **artefato/resultado final** e não em como os resultados foram atingidos. Busca-se verificar se os objetivos de aprendizagem, que definem o que aluno deve ser capaz de fazer ao final de uma unidade instrucional (DRISCOLL; WOOD, 2007), foram atingidos.

2.2.2 Avaliação automatizada de programas

A avaliação automatizada é feita por meio de um software que que avalia diversos fatores e características de qualidade. Tipicamente, este software implementa um modelo de avaliação baseado em uma rubrica contendo as características a serem analisadas. Essa avaliação pode ser feita por meio da **análise dinâmica** (caixa-preta) ou **análise**

estática (caixa-branca) (KOYYA; LEE; YANG, 2013; ALA-MUTKA, 2005).

As técnicas de análise estática examinam o código-fonte sem executar o programa. Podem analisar características como estilo de codificação, métricas de software, erros de programação, design e características específicas relacionadas à estrutura do programa. Além disso, podem realizar a detecção de plágio (ALA-MUTKA, 2005) e analisar o fluxo de controle e documentação do software, incluindo requisitos, especificações de interface e modelos (BOURQUE; FAIRLEY, 2014). Tipicamente, a análise estática de um programa em uma VPL baseada em blocos provê informações sobre a presença de certos blocos que podem ser mapeados para conceitos abstratos de algoritmos e programação especificados pelo CSTA K-12 (2017), como por exemplo, variáveis e controle, ou ainda podem ser comparados com uma solução correta. Desta forma é possível verificar aspectos da qualidade do programa. No entanto, a análise estática não possibilita a verificação de erros de execução.

As técnicas dinâmicas têm como característica a execução do código (BOURQUE; FAIRLEY, 2014). Geralmente, são técnicas de teste, mas técnicas como a simulação e análise de modelos também podem ser consideradas dinâmicas. Com estas técnicas é possível testar um programa com uma quantidade de casos significativa, comparando a saída gerada com a saída esperada para cada teste (GUPTA; DUBEY, 2012). Assim é possível verificar a corretude do programa e a capacidade do aluno de realizar testes sobre o programa, além de permitir a quantificação do tempo para a execução de um programa, na prática possibilitando a medição da performance.

A avaliação automatizada é aplicada frequentemente a atividades fechadas, em que há uma solução/saída modelo, mas também pode ser aplicada a avaliação de atividades abertas que possuem diversas soluções. Para a avaliação de atividades fechadas tipicamente se faz uma comparação entre a solução/saída do programa do aluno e a solução/saída modelo. Para atividades abertas, comumente se emprega a análise estática, haja vista a análise dinâmica é mais utilizada pela sua capacidade de reconhecer o comportamento correto/incorreto do programa.

2.3 ANÁLISE ESTÁTICA

O processo de análise estática foi concebido com o objetivo de avaliar a qualidade dos programas dos alunos. Na prática, é utilizado

também para ajudar os alunos principiantes a aprender a programar e fornecer aos professores ferramentas de avaliação semiautomáticas (TRUONG; ROE; BANCROFT, 2004). Alguns analisadores estáticos utilizam duas abordagens distintas para análise estática: análise de similaridade estrutural e análise de métricas de engenharia de software (TRUONG; ROE; BANCROFT, 2004).

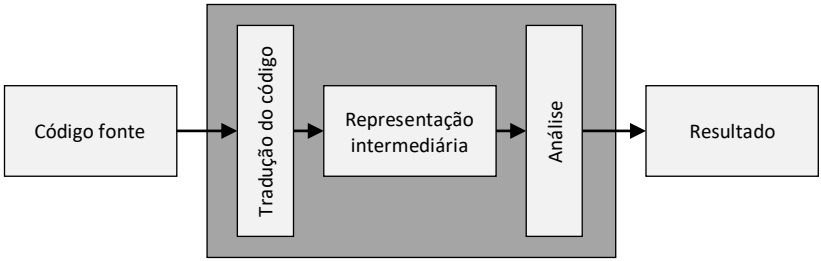
A **análise de similaridade estrutural** tem como objetivo verificar como a estrutura da solução do aluno se compara com uma solução modelo. O *feedback* para os alunos e instrutores indica a semelhança das soluções do aluno e do modelo. Esta técnica é tipicamente usada para programas resultantes de atividades fechadas (*well-structured*), pois é difícil antecipar todas as soluções possíveis para um problema. Uma das formas de se contornar isso é se caso o sistema não encontrar uma correspondência entre a solução do estudante e todas as soluções modelo disponíveis, então a solução do estudante é enviada ao docente para revisão.

A **análise de métricas de engenharia de software** é feita de forma quantitativa. É tipicamente usada para medir a qualidade do software. Esta análise é baseada em métricas de complexidade de software e diretrizes de boas práticas de programação para avaliar a qualidade das soluções dos alunos. São usadas métricas como por exemplo, complexidade ciclomática, métricas de Halstead e métricas de acoplamento e coesão. Algumas ferramentas também realizam a análise estática do código examinando a documentação (BOURQUE; FAIRLEY, 2014). A análise de métricas também pode ser usada para avaliar a aplicação de conceitos de algoritmos e programação relacionados às práticas pensamento computacional em um programa resultante de uma atividade aberta (*ill-structured*). Nesse caso, as definições dos critérios de uma rubrica, ligados aos objetivos de aprendizagem, são utilizadas como métricas de análise do código para análise do atendimento aos fatores.

A realização do processo de análise estática tipicamente consiste na análise léxica e sintática (*parsing*) do código que se deseja analisar. A partir das análises é realizada a avaliação de elementos relacionados as métricas definidas (Figura 13). Este processo é o mesmo realizado por um compilador quando se está a realizar a análise léxica e sintática com objetivo de fazer a tradução do código para uma representação intermediária. A partir do *parsing*, o código é traduzido para uma representação intermediária de forma a facilitar o processo de análise. A análise é realizada sobre esta representação intermediária do código com bases nas regras e métricas definidas. Os resultados obtidos na análise

são apresentados para o usuário, podendo ser em forma textual ou gráfica (JONES, 2013).

Figura 13 - Processo genérico de análise estática.



Fonte: adaptado de Jones (2013) – versão traduzida.

Cabe ressaltar que para a fase de análise, dependendo do objetivo do analisador de código, as regras utilizadas podem ser fixas ou configuráveis pelo usuário (SINTHSIRIMANA; PANJABUREE, 2013). Em analisadores com regras fixas, as métricas e procedimentos de análise já são predefinidas e o usuário pode ou não ter a liberdade de escolher quais e quantas regras serão utilizadas. Em analisadores com regras configuráveis, é possível que o usuário crie suas próprias regras que serão utilizadas para analisar o código. Tipicamente, estas regras devem ser definidas em sintaxe especificada pelo próprio software.

3 ESTADO DA ARTE

Neste capítulo é apresentado o estado da arte de abordagens para analisar e/ou avaliar o desempenho do aluno em relação à programação e pensamento computacional por meio do código-fonte de VPL baseada em blocos no contexto da Educação Básica. Para isto, é realizado um mapeamento sistemático da literatura seguindo os procedimentos definidos por Petersen et al. (2008). Os resultados do mapeamento são também publicados em Alves et al. (2018).

3.1 DEFINIÇÃO

O mapeamento sistemático da literatura tem como objetivo identificar, classificar e interpretar pesquisas disponíveis por meio de critérios de qualificação claros e reproduzíveis em relação ao tema deste trabalho (PETERSEN et al., 2008). A pergunta de pesquisa deste mapeamento é:

Pergunta de pesquisa: Quais abordagens existem para analisar o desempenho do aluno a partir da análise do código de programas criados com VPL baseada em blocos no contexto da Educação Básica e quais as suas características?

Para responder essa pergunta, são definidas as seguintes perguntas de análise:

Análise do programa

PA1. Quais abordagens existem e quais as suas características?

PA2. Quais conceitos de algoritmos e programação relacionados à prática do pensamento computacional são analisados?

PA3. Como os conceitos de algoritmos e programação relacionados à prática do pensamento computacional são analisados?

Avaliação e *feedback* instrucional

PA4. Se e como é gerada uma nota?

PA5. Se e como o *feedback* instrucional é apresentado?

Automatização da avaliação

PA6. Se e como a abordagem foi automatizada?

Bases de dados: São consideradas como fonte de pesquisa os artigos indexados pela ferramenta de busca SCOPUS (https://www.scopus.com) com acesso livre por meio do Portal CAPES[1].

[1] Um portal para o acesso de trabalhos científicos, gerenciado pelo Ministério da Educação do Brasil, disponível para instituições autorizadas, como universidades e agências de pesquisa (www.periodicos.capes.gov.br).

Critérios de inclusão e exclusão: De acordo com o objetivo de pesquisa, são considerados somente artigos na língua inglesa cujo foco seja apresentar uma abordagem para análise e/ou avaliação de conceitos de algoritmos e programação relacionados à prática do pensamento computacional a partir do código-fonte. A análise deve ser feita examinando programas criados com VPL baseada em blocos no contexto da Educação Básica. Além disso, são considerados também artigos que abordam conceitos comumente abordados na Educação Básica, mas que podem ter sido usados em outros estágios educacionais (como o Ensino Superior). Os artigos devem ter sido publicados nos últimos 21 anos, entre janeiro de 1997, ano seguinte à criação da primeira VPL (TEMPEL, 2013), e julho de 2018.

São excluídos artigos que apresentam uma abordagem para analisar programas em linguagens de programação textuais, artigos que apresentam uma abordagem para avaliar conceitos de computação com base em outras fontes que não sejam o código-fonte desenvolvido pelo aluno, como por exemplo, provas, entrevistas, etc., ou executam análise de código fora de um contexto educacional, como por exemplo, avaliação de qualidade de software.

Critérios de qualidade: São considerados apenas artigos que apresentam informações substanciais para se extrair informações referente às perguntas de análise. São excluídos artigos que apresentam, por exemplo, somente um resumo de uma proposta e para os quais não são encontradas mais informações detalhadas.

Termos de busca: Com base na pergunta de pesquisa os termos de busca e seus sinônimos são derivados (Tabela 10). O termo *code analysis* é escolhido pela razão de expressar o conceito principal a ser pesquisado. Os termos *static analysis*, *grading* e *assessment* são escolhidos, pois em pesquisas prévias verificou-se que a maioria dos trabalhos utilizam este termo para análise de código com foco em avaliação. O termo *visual programming* é escolhido para restringir a pesquisa a trabalhos que focam em linguagens de programação visual. Também são definidos como sinônimos para este termo algumas VPL baseada em blocos proeminentes, voltadas ao contexto da Educação Básica, como Scratch, App Inventor e Snap!.

Tabela 10 - Termos de busca.

Conceito	**Sinônimo**
code analysis	static analysis, grading, assessment
visual programming	visual language, scratch, app inventor, snap, blockly

Usando esses termos de busca a *string* de busca é calibrada e adaptada de acordo com a base de dados.

String de busca: (“code analysis" OR "static analysis" OR grading OR assessment) AND (“visual programming" OR "visual language" OR scratch OR "app inventor" OR snap OR blockly)

Metodologia de seleção: Inicialmente os artigos são excluídos com base na área de conhecimento, ou seja, artigos publicados em outras áreas que não estejam relacionadas a este tema são excluídos, como, por exemplo, Medicina, Biologia, etc. Após isso, os resultados são analisados com base no título, resumo e palavra-chave. Os trabalhos resultantes dessa análise constituem em resultados potencialmente relevantes. Esses trabalhos são analisados na íntegra, além de se verificar suas referências de forma a identificar trabalhos não encontrados nos resultados desse mapeamento de literatura. Os trabalhos potencialmente relevantes que não possuem informações substanciais referente às perguntas de análise são excluídos. Após essas etapas, tem-se como resultado o conjunto de artigos relevantes.

3.2 EXECUÇÃO DA BUSCA E SELEÇÃO DE ARTIGOS

A execução da busca foi realizada em agosto de 2018, pela autora do trabalho e revisada por um pesquisador sênior, conforme definido. A autora do presente trabalho realizou a pesquisa inicial. Foi aplicada a metodologia de seleção de artigos na qual todos os resultados encontrados via SCOPUS foram analisados por meio do título, resumo e palavra-chave. Foram selecionados 36 artigos potencialmente relevantes os quais foram analisados na íntegra, além do título, resumo e palavra-chave, para identificar trabalhos compatíveis com todos os critérios de inclusão/exclusão.

A seleção dos artigos relevantes foi realizada pela autora do trabalho e revisada por um pesquisador sênior na qual se discutiu sobre a inclusão ou exclusão de trabalhos que geraram dúvidas até se chegar a um consenso. Dentre os artigos potencialmente relevantes analisados na íntegra, foram selecionados 23 artigos relevantes. A Tabela 11 apresenta a quantidade de artigos encontrados e selecionados por etapa do processo de seleção.

Tabela 11 - Quantidade de artigos no SCOPUS em cada etapa de seleção.

Resultados da busca	Resultados analisados	Resultados potencialmente relevantes	Resultados relevantes
2550	2550	36	23

Nos resultados iniciais da busca, foram encontrados artigos cujo foco é a avaliação da aprendizagem do aluno por meio de outras formas de avaliação que não são por meio de análise do código-fonte, como provas (WEINTROP; WILENSKY, 2015). Também foram identificados artigos analisando estatisticamente outras questões como, por exemplo, a forma que alunos novatos programam usando VPL (AIVALOGLOU; HERMANS, 2016) ou erros comuns em programas criados em Scratch (TECHAPALOKUL, 2017) e, portanto, excluídos do mapeamento. Foram encontrados artigos descrevendo abordagens para a compreensão de programas (ZHANG; SURISETTY; SCAFFIDI, 2013; KECHAO et al., 2012) ou voltadas a linguagens textuais (KECHAO et al., 2012), ou seja, não voltadas à VPL baseada em blocos, e por esta razão também foram excluídos deste mapeamento. Artigos que apresentam uma abordagem para avaliar outros conceitos que não incluem algoritmos e programação, como por exemplo, inovação, também foram excluídos (HWANG; LIANG; WANG, 2016). Artigos completos ou em andamento que satisfazem todos os critérios de inclusão, mas não apresentam informações suficientes com relação às questões de análise foram excluídos devido ao critério de qualidade (GROVER et al., 2016; CHEN et al., 2017).

Tabela 12 - Artigos relevantes.

ID	Título do artigo	Referência
1	A Method for Measuring of Block-based Programming Code Quality	(KWON; SOHN, 2016a)
2	A Framework for Measurement of Block-based Programming Language	(KWON; SOHN, 2016b)
3	Assessment of Computer Science Learning in a Scratch-Based Outreach Program	(FRANKLIN et al., 2013)
4	Hairball Lint-inspired Static Analysis of Scratch Projects	(BOE et al., 2013)
5	Automatic detection of bad programming habits in scratch A preliminary study	(MORENO; ROBLES, 2014)
6	Dr. Scratch - A web tool to automatically evaluate scratch projects	(MORENO-LEÓN; ROBLES, 2015)
7	Comparing Computational Thinking Development Assessment Scores with Software Complexity Metrics	(MORENO-LEÓN; ROBLES; ROMÁN-GONZÁLEZ, 2016)
8	On the Automatic Assessment of Computational Thinking	(MORENO-LEÓN

	Skills - A Comparison with Human Experts	et al., 2017)
9	Real time assessment of computational thinking	(KOH et al., 2014a)
10	Early validation of computational thinking pattern analysis	(KOH et al., 2014b)
11	Towards the Automatic Recognition of Computational Thinking for Adaptive Visual Language Learning	(KOH et al., 2010)
12	Recognizing Computational Thinking Patterns	(BASAWAPATNA et al., 2011)
13	ITCH Individual testing of computer homework for scratch assignments	(JOHNSON, 2016)
14	Modeling the Learning Progressions of Computational Thinking of Primary Grade Students	(SEITER; FOREMAN, 2013)
15	Ninja code village for scratch - Function samples function analyzer and automatic assessment of computational thinking concepts	(OTA; MORIMOTO; KATO, 2016)
16	The fairy performance assessment: measuring computational thinking in middle school	(WERNER et al., 2012)
17	Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts?	(DENNER; WERNER; ORTIZ, 2012)
18	Scrape: A tool for visualizing the code of scratch programs	(WOLZ; HALLBERG; TAYLOR, 2011)
19	Quizly - A live coding assessment platform for App Inventor	(MAIORANA; GIORDANO; MORELLI, 2015)
20	Autograding and Feedback for Snap!: A Visual Programming Language.	(BALL; GARCIA, 2016)
21	Autograding for Snap!	(BALL, 2017)
22	CodeMaster – Automatic Assessment and Grading of App Inventor and Snap! Programs	(GRESSE VON WANGENHEIM et al., 2018)
23	Analysis of Scratch Projects of an Introductory Programming Course for Primary School Students	(FUNKE; GELDREICH; HUBWIESER, 2017)

Observa-se que os artigos encontrados, relacionados ao foco da pesquisa, foram publicados dentro do período dos últimos 9 anos (Figura 14), demonstrando a recente popularização de VPL baseada em blocos dentro do contexto educacional.

Figura 14 - Quantidade de publicações relevantes por ano sobre abordagens de análise de código de VPL no contexto educacional.

Fonte: elaborada pela autora.

3.3 EXTRAÇÃO DE DADOS

Os dados foram extraídos dos artigos de forma a responder às questões de pesquisas conforme especificado na Tabela 13.

Tabela 13 - Especificação dos dados extraídos.

Pergunta de análise	Item	Descrição
PA1. Quais abordagens existem e quais as suas características?	Nome da abordagem	Indicando o nome ou autor da abordagem.
	VPL baseada em blocos	Indicando qual é a linguagem alvo da abordagem.
	Tipo de atividade	Indicando se a abordagem é destinada à avaliação de atividades fechadas ou abertas.
PA2. Quais conceitos de algoritmos e programação relacionados à prática do pensamento computacional são analisados?	Conceitos do CSTA	Indicando quais conceitos são analisados em relação ao *framework* do CSTA.
	Conceitos adicionais	Indicando conceitos analisados além dos propostos pelo *framework* do CSTA.
PA3. Como os conceitos de algoritmos e programação relacionados à prática do pensamento computacional são analisados?	Tipo de análise de código	Indicando se a abordagem utiliza análise automatizada (estática ou dinâmica) ou manual.
PA4. Se e como é gerada uma nota?	Tipo de nota	Indicando que método(s) a abordagem utiliza para atribuir uma nota (pontuação dicotômica, pontuação politômica e/ou pontuação composta).

PA5. Se e como o *feedback* instrucional é apresentado?	Tipo de avaliação	Indicando se é feita uma avaliação somativa ou formativa.
	Feedback instrucional	Indicando se a abordagem apresenta *feedback* instrucional.
PA6. Se e como a abordagem foi automatizada?	Categorias de usuário	Indicando para quais categorias de usuários a ferramenta oferece módulos específicos (professor, aluno, administrador, instituição).
	Plataforma de acesso	Indicando as plataformas cuja ferramenta pode ser acessada.
	Licença	Indicando o tipo de licença (grátis, proprietária).
	Língua	Indicando em que língua(s) a abordagem está disponível.

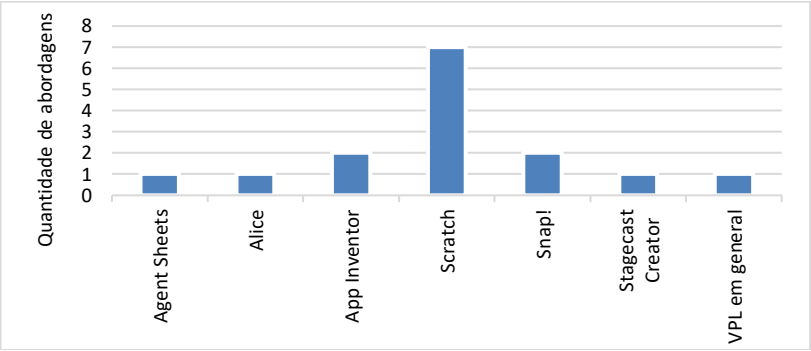
3.4 ANÁLISE DE DADOS

Esta seção apresenta a análise dos dados extraídos dos artigos de acordo com as questões de pesquisa definidas na Seção 3.1. São apresentadas as abordagens encontradas e suas características gerais. São transcritos os elementos que cada abordagem analisa, qual o tipo de análise feita e como ela é feita. São apresentadas informações indicando se a abordagem atribui uma nota para o projeto do aluno, qual o tipo de avaliação que é realizada, se apresenta *feedback* instrucional e, por fim, alguns aspectos de implementação.

Quais abordagens existem e quais as suas características?

Foram encontrados 23 artigos relevantes que descrevem 14 abordagens diferentes, haja vista alguns dos artigos apresentam a mesma abordagem, porém sob perspectiva diferente. A maioria das abordagens é voltada para a linguagem de programação Scratch (Figura 15). As demais são direcionadas às linguagens App Inventor, Alice, Agent Sheets e Stagecast Creator.

Figura 15 - Visão geral das abordagens para linguagens de programação visual baseadas em blocos.



Fonte: elaborada pela autora.

As abordagens diferem em relação ao tipo de atividade de programação para a qual foram projetadas, incluindo abordagens para avaliar atividades fechadas (*well-structured*) para as quais uma solução é definida antecipadamente, bem como atividades abertas (*ill-structured*) com várias possíveis soluções (Tabela 14).

Tabela 14 - Visão geral das características das abordagens encontradas.

ID	**Nome da abordagem**	**VPL baseada em blocos**	**Tipo de atividade de programação**	
			Atividade fechada	**Atividade aberta**
1 - 2	Abordagem de Kwon e Sohn	VPL em geral		x
3 - 4	Hairball	Scratch	x	
5 - 8	Dr. Scratch	Scratch		x
9 - 12	CTP/PBS/REACT	Agent Sheets	x	
13	ITCH	Scratch	x	
14	PECT	Scratch		x
15	Ninja Code Village	Scratch		x
16	Fairy Assessment	Alice	x	
17	Abordagem de Denner et al.	Stagecast Creator	x	
18	Scrape	Scratch		x
19	Quizly	App Inventor	x	
20 - 21	Lambda	Snap!	x	
22	CodeMaster	App Inventor e Snap!		x
23	Abordagem de Funke et al.	Scratch		x

Quais conceitos de algoritmos e programação relacionados à prática do pensamento computacional são analisados?

Todas as abordagens realizam uma análise referente ao desempenho, medindo a competência de conceitos de algoritmos e programação relacionados à prática do pensamento computacional. Como trata-se de uma análise orientada ao produto, isto é, uma análise sobre o código-fonte do programa, são procurados **indicadores de aprendizagem de conceitos previamente definidos** verificando se há a aplicação desses conceitos no código. Desta forma, é examinado se comandos (ou blocos), conceitualmente relacionados a esses conceitos, estão presentes ou ausentes no código (BRENNAN; RESNICK, 2012; MORENO-LEÓN; ROBLES; ROMÁN-GONZÁLEZ, 2016; GRESSE VON WANGENHEIM et al., 2018). De acordo com a definição do *framework* do CSTA (2016), a maioria das abordagens analisa quatro dos cinco subconceitos relacionados a algoritmos e programação, são eles: controle, algoritmos, variáveis e modularidade (Tabela 15). Nenhuma das abordagens analisa o subconceito de desenvolvimento de programas. Outras abordagens, como CTP (KOH et al., 2014b), analisam padrões de pensamento computacional, incluindo geração, colisão, transporte e difusão.

Algumas abordagens manuais analisam também elementos relacionados ao conteúdo do programa, como a criatividade e a estética (KWON; SOHN, 2016b; WERNER et al., 2012). A funcionalidade do programa é analisada apenas via análise dinâmica ou manual. Duas abordagens (KWON; SOHN, 2016b; DENNER et al., 2012) analisam o nível de completude, analisando se o programa tem várias funções, ou no caso de jogos, se tem vários níveis de dificuldade. A abordagem Ninja Code Village (OTA; MORIMOTO; KATO, 2016) analisa se existem algumas funções de propósito geral no código, como por exemplo, se em um jogo uma função de pular com um personagem foi implementada. Três abordagens analisam elementos relacionados à organização do código e apenas duas abordagens (manuais) analisam a usabilidade (DENNER; WERNER; ORTIZ, 2012; FUNKE; GELDREICH; HUBWIESER, 2017).

Tabela 15 - Visão geral dos conceitos analisados.

Abordagem	Conceitos analisados														
	Conceitos do CSTA					Conceitos adicionais									
	Algoritmos	Variáveis	Controle	Modularidade	Desenvolvimento de programas	Técnica de projeto de algoritmos	Criatividade	Estética	Funcionalidade	Completude	Inicialização	*Computational Thinking Patterns*	Detecção de funções típicas	Organização e documentação	Usabilidade
Abordagem de Kwon e Sohn	x	x	x	x		x	x	x	x	x					
Hairball	x		x								x				
Dr.Scratch	x	x	x	x											
CTP/PBS/ REACT	x	x	x									x			
ITCH	x	x	x	x					x						
PECT	x	x	x	x							x				
Ninja Code Village	x	x	x	x									x		
Fairy Assessment	x	x	x	x			x								
Abordagem de Denner et al.	x	x	x	x				x	x	x				x	x
Scrape	x	x	x	x							x				
Quizly	x	x	x	x					x						
Lambda	x	x	x	x											
CodeMaster	x	x	x	x										x	
Abordagem de Funke et al.	x	x	x	x					x					x	x

Para avaliar as competências de pensamento computacional apresentadas na Tabela 15, as abordagens analisam o código para detectar a presença de comandos de programação específicos. Assim, é inferida, a partir da presença de comandos específicos, a aprendizagem de conceitos e práticas do pensamento computacional. Por exemplo,

para avaliar a competência em relação ao conceito estruturas de controle, o qual especifica a ordem em que instruções são executadas dentro de um algoritmo ou programa, é verificado se o programa do aluno contém um comando de laço (*loop*).

Algumas abordagens também analisam competências específicas relacionada às características da linguagem de programação e/ou tipo de programa, como por exemplo, padrões de pensamento computacional em jogos (KOH et al., 2014b). No entanto, apenas com base no código criado não é possível analisar algumas das práticas computacionais, como por exemplo, reconhecimento e definição de problemas computacionais.

A avaliação de outros aspectos complexos, como a criatividade, também é difícil de automatizar, refletida pelo fato de que nenhuma abordagem automatizada com relação a esse critério foi encontrada. Para avaliar esse tipo de aspecto mais subjetivo, outras formas complementares de avaliação devem ser usadas, tais como entrevistas baseadas em artefatos, relatórios e trabalhos (BRENNAN; RESNICK, 2012).

Como os conceitos de algoritmos e programação relacionados à prática do pensamento computacional são analisados?

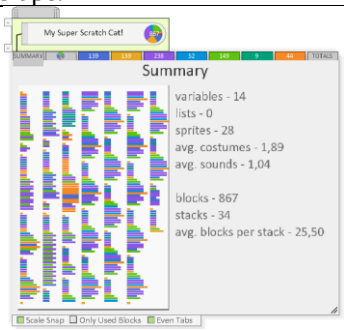
As abordagens analisam o código de diferentes maneiras, as quais podem ser classificadas em abordagens automatizadas, via análise estática ou dinâmica, e não automatizadas, via análise manual. A maioria das abordagens encontradas realiza a análise estática. Isso está relacionado ao fato de que o tipo de análise empreendida depende da natureza da atividade de programação. Somente no caso de atividades fechadas, com uma solução conhecida antecipadamente, é possível comparar o código do programa dos alunos com o código de uma implementação modelo.

Cabe ressaltar que todas as abordagens focadas na análise de atividades abertas são feitas por meio da análise estática, com base na detecção da presença de comandos relacionados aos conceitos de algoritmos e programação. A análise estática geralmente inclui uma análise quantitativa de comandos presentes no código. Isso permite identificar quais e com que frequência cada comando é usado. Esta abordagem baseia-se na premissa de que a presença de um comando relacionado a um conceito indica sua compreensão conceitual. Embora, isso não implique necessariamente que o aluno obteve uma

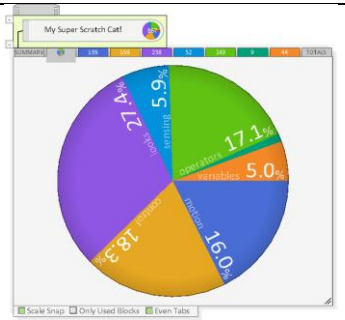
compreensão profunda acerca do respectivo conceito de programação (BRENNAN; RESNICK, 2012).

Com base nas quantidades de comandos utilizados, são realizadas análises adicionais, como por exemplo, cálculos de somas, médias e porcentagens. Os resultados da análise são apresentados de várias formas, incluindo gráficos em diferentes graus de detalhamento. Um exemplo é a ferramenta Scrape (WOLZ; HALLBERG; TAYLOR, 2011) que apresenta várias telas com informações minuciosas dos resultados da análise do código (Figura 16).

Figura 16 - Apresentação dos resultados da análise de um programa por Scrape.

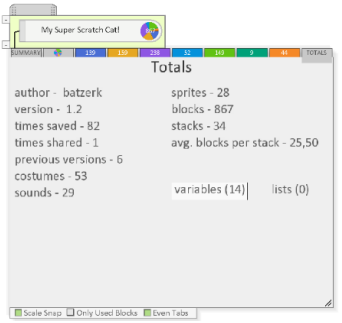


a) Resumo dos resultados da análise.



b) Porcentagens de comandos usados por categoria do Scratch.

c) Exemplo de detalhamento dos comandos utilizados para uma categoria do Scratch.

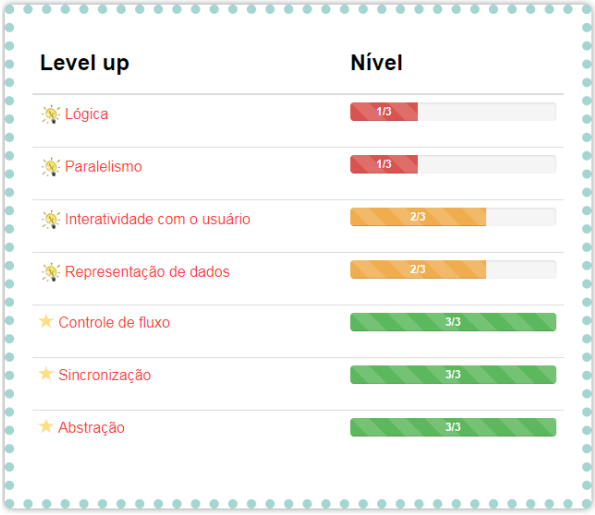


d) Informações gerais do projeto.

Fonte: elaborada pela autora com base na ferramenta Scrape.

Algumas abordagens apresentam os resultados da análise estática em um nível mais abstrato, além da quantificação de comandos (MORENO-LEÓN; ROBLES, 2015; OTA; MORIMOTO; KATO, 2016; GRESSE VON WANGENHEIM et al., 2018). Um exemplo é a ferramenta Dr. Scratch (MORENO-LEÓN; ROBLES, 2015), a qual analisa conceitos como abstração, lógica e paralelismo, fornecendo uma pontuação para cada critério baseado em uma rubrica (Figura 17).

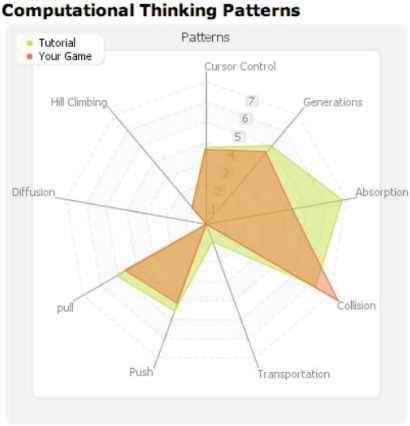
Figura 17 - Análise de um programa em Scratch feita pelo Dr. Scratch.



Fonte: elaborada pela autora com base na ferramenta Dr. Scrach.

As atividades fechadas de programação também podem ser avaliadas usando análise estática, tipicamente comparando o código da solução dos alunos com o código da solução correta implementada pelo professor. Neste caso, a análise é realizada por meio da verificação, isto é, constatando se um determinado conjunto de comandos que está presente na solução do professor, também está presente no programa do aluno. Uma das abordagens que faz esse tipo de análise é CTP (KOH et al., 2014a). Esta abordagem requer que para cada exercício de programação uma solução modelo seja previamente programada por um professor ou designer instrucional. Desta forma, a solução dos alunos, indicada em marrom na Figura 18, é comparada à solução modelo, indicada em verde na Figura 18. Os resultados são apresentados por meio de um gráfico (Figura 18).

Figura 18 - Gráfico comparando a solução do aluno com a do professor.



Fonte: Koh et al. (2014a).

Algumas abordagens utilizam a análise dinâmica. Neste caso, a análise é realizada utilizando testes para determinar se a solução do aluno é correta com base na saída produzida pelo programa (MAIORANA; GIORDANO; MORELLI, 2015). No entanto, a adoção dessa abordagem exige pelo menos a predefinição dos requisitos e/ou casos de teste para a atividade de programação. Outra desvantagem deste tipo de teste de caixa preta é que ele examina a funcionalidade de um programa sem analisar sua estrutura interna.

A análise híbrida, combinando a análise dinâmica, por meio de testes personalizados, com a análise estática, por meio da quantificação de comandos específicos, é encontrada na abordagem ITCH (JOHNSON, 2016). ITCH é direcionada a atividades fechadas, de forma que a avaliação realiza tanto a análise dinâmica quanto a análise estática para a solução do aluno de uma atividade fechada, ao final gerando um relatório personalizado.

Várias abordagens encontradas são aplicadas via análise manual do código independentemente do tipo de atividade, seja ela aberta ou fechada. Um exemplo é a abordagem Fairy Assessment (WERNER et al., 2012) que usa uma rubrica para avaliar o código do aluno resultante de uma atividade fechada. A abordagem PECT apresenta uma rubrica para realizar análises manuais de atividades abertas (SEITER; FOREMAN, 2013). A Tabela 16 resume os tipos de análises adotadas por todas as abordagens.

Tabela 16 - Visão geral dos tipos de análises.

Abordagem	Análise automatizada		Análise manual
	Análise estática	Análise dinâmica	
Abordagem de Kwon e Sohn	x		
Hairball	x		
Dr.Scratch	x		
CTP/PBS/REACT	x		
ITCH	x	x	
PECT			x
Ninja code village	x		
Fairy Assessment			x
Abordagem de Denner et al.			x
Scrape	x		
Quizly		x	
Lambda	x	x	
CodeMaster	x		
Abordagem de Funke et al.			x

Se e como é gerada uma nota?

A avaliação de competências do pensamento computacional com base na análise do código é feita usando diferentes formas de atribuições de nota. Algumas abordagens utilizam uma pontuação dicotômica, avaliando a corretude de um programa como um todo, indicando se está certo ou errado. Um exemplo é Quizly (MAIORANA; GIORDANO; MORELLI, 2015) que testa o programa do aluno e, se a saída do programa estiver correta, apresenta uma mensagem dizendo que o programa está correto, caso contrário incorreto.

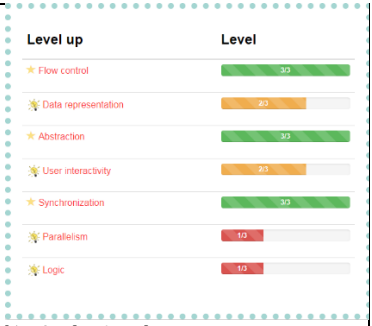
Várias abordagens dividem o programa em **áreas** a serem analisadas e atribuem uma pontuação politômica para cada área. Desta forma, um único programa pode conter diferentes pontuações para cada área. Um exemplo é a abordagem Hairball (BOE et al., 2013) em que cada área pode ser rotulada como (i) correta, quando o conceito foi implementado corretamente, (ii) semanticamente incorreta, quando o conceito foi implementado de uma maneira que nem sempre funciona como pretendido, (iii) incorreta, quando foi implementado incorretamente, ou (iv) incompleto, quando a solução não está finalizada.

Algumas abordagens fornecem uma pontuação composta baseada na soma do conjunto de pontuações politômicas de cada área. Um exemplo é Dr. Scratch (MORENO-LEÓN; ROBLES, 2015), a qual analisa sete áreas (Figura 19a) e atribui uma pontuação politômica para cada área (Figura 19b). Neste caso, presume-se que o uso de blocos de maior complexidade, como "se então, senão" implica em um nível de desempenho mais alto do que quando são usados blocos de menor complexidade, como "se então". Uma pontuação final composta é atribuída ao programa com base na soma das pontuações politômicas. Esta pontuação final composta indica o nível de desempenho por meio da soma de todos os pontos de cada área, os quais são dispostos numa escala ordinal de 0 a 3 pontos. Assim, a pontuação final composta indica que a soma de 0 a 7 pontos se refere ao nível básico, 7 a 14 pontos ao nível desenvolvimento e 15 a 21 pontos ao nível mestre (Figura 19c). Este resultado também é apresentado por meio da imagem de uma mascote, cuja seção de *feedback* apresenta elementos de gamificação no contexto da avaliação (Figura 19c) (MORENO-LEÓN; ROBLES, 2015). A ferramenta também cria um certificado personalizado que pode ser salvo e impresso pelo usuário.

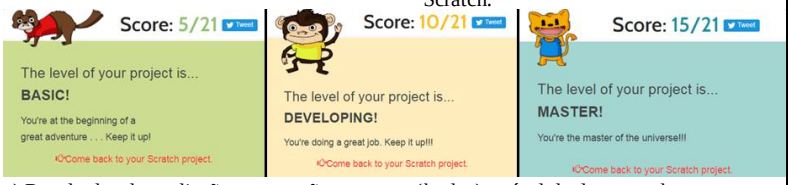
Figura 19 - Exemplos de pontuação pelo Dr. Scratch.

CT concept	Basic	Developing	Proficiency
Parallelization	2 scripts on green_flag	2 scripts on key_pressed, 2 scripts on sprite_clicked on the same sprite	2 scripts on when_I_recieve_message, create_clone, 2 scripts when_%s_is_>_%s, 2 scripts on when_backdrop_change_to
Synchronization	wait	Broadcast, when_I_receive_message, stop_all, stop_program,stop_programs_sprite	wait_until, when_backdrop_change_to, when_I_start_as_clon, broadcast_and_wait
Data representation	Modifiers of properties of sprites	Operations on vars	Operations on lists
Conditional logic	if	if_else	logic operations
Interactivity (UI)	Green_flag	key_pressed, sprite_clicked, ask_and_wait, mouse blocks	when_%s_is_>_%s, video, audio
Algorithmic notions of flow control	Sequence of blocks	Repeat, Forever	repeat_until
Abstraction and problem decomposition	> 1 scripts	> 1 scripts and > 1 sprites	def_block

a) Rubrica usada para avaliar um projeto.



b) Avaliação de um projeto no Dr. Scratch.



c) Resultados da avaliação: pontuação, mascote (*badge*) e nível de desempenho.

Fonte: elaborada pela autora com base em Moreno-León e Robles (2015).

Outra maneira de atribuir uma pontuação composta é baseada em uma soma ponderada das pontuações individuais para cada área (KWON; SOHN, 2016a) ou considerando diferentes variáveis (KOH et al., 2014a). A abordagem apresentada por Kwon e Sohn (2016a), por exemplo, avalia vários critérios para diferentes áreas, atribuindo a cada uma delas pesos diferentes. A Tabela 17 resume o tipo de pontuação adotado pelas abordagens.

Tabela 17 - Visão geral da atribuição de nota por abordagem.

Abordagem	Tipo de pontuação			Não atribui pontuação	Não identificado
	Dicotômica	Politômica	Composta		
Abordagem de Kwon e Sohn			x (fórmula geral)		
Hairball		x (c/ 4 áreas)			
Dr. Scratch		x (c/ 7 áreas)	x (soma das pontuações politômicas)		
CTP/PBS/REACT		x (c/ 7 áreas)	x (fórmula geral)		
ITCH				x	
PECT		x (c/ 24 áreas)			
Ninja code village		x (c/ 8 áreas)			
Fairy Assessment			x (fórmula para cada atividade)		
Abordagem de Denner et al.					x
Scrape				x	
Quizly	x				
Lambda	x				
CodeMaster		x (c/ 15 áreas)	x (soma das pontuações politômicas)		
Abordagem de Funke et al.		x			

Se e como o *feedback* instrucional é apresentado?

A avaliação das abordagens é feita de forma somativa e/ou formativa. Algumas abordagens podem ser usadas para ambas (MORENO-LEÓN; ROBLES, 2015; BOE et al., 2013; GRESSE VON WANGENHEIM et al., 2018; MAIORANA; GIORDANO; MORELLI, 2015). Poucas abordagens apresentam *feedback* instrucional além de

uma pontuação (MORENO-LEÓN; ROBLES, 2015; BOE et al., 2013; GRESSE VON WANGENHEIM et al., 2018; MAIORANA; GIORDANO; MORELLI, 2015). A Tabela 18 resume os tipos de avaliação e *feedback* instrucional.

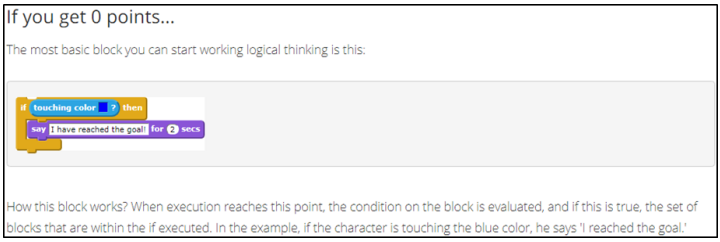
Tabela 18 - Visão geral sobre os tipos de avaliação e *feedback* instrucional.

Abordagem	**Avaliação**		***Feedback* instrucional**
	Somativa	**Formativa**	
Abordagem de Kwon e Sohn	x		
Hairball	x	x	x
Dr.Scratch	x	x	x
CTP/PBS/REACT		x (em tempo real)	
ITCH	x		
PECT	x		
Ninja code village	x	x	
Fairy Assessment	x		
Abordagem de Denner et al.	x		
Scrape			
Quizly		x	x
Lambda		x	
CodeMaster	x	x	x
Abordagem de Funke et al.	x		

Tipicamente, o *feedback* envolve sugestões e dicas de modificações que podem ser feitas para alcançar uma pontuação maior e consequentemente um nível de desempenho melhor. Os estudos não informam em detalhes como esse *feedback* é feito. No entanto, o *feedback* gerado por abordagens que usam análise dinâmica (MAIORANA; GIORDANO; MORELLI, 2015) é baseado na saída obtida pela execução do programa.

Algumas abordagens fornecem *feedback* por meio de um conteúdo com dicas que podem ser consultadas a qualquer momento. Dr. Scratch (MORENO-LEÓN; ROBLES, 2015), por exemplo, fornece uma explicação geral sobre como alcançar uma pontuação mais alta em determinada área avaliada (Figura 20). Similarmente, o CodeMaster (GRESSE VON WANGENHEIM et al., 2018) apresenta a rubrica usada para avaliar o programa. Quizly (MAIORANA et al., 2015), juntamente com a descrição da atividade, fornece um tutorial sobre como resolver as atividades.

Figura 20 - Dicas sobre como obter um desempenho melhor pelo Dr.Scratch.



Fonte: elaborada pela autora com base na ferramenta Dr. Scratch.

Se e como a abordagem foi automatizada?

Apenas algumas abordagens foram automatizadas via ferramentas de software (Tabela 19). Entre elas, a maioria realiza análise estática. Poucas abordagens utilizam análise dinâmica para avaliar a solução do aluno (JOHNSON, 2016; MAIORANA; GIORDANO; MORELLI, 2015; BALL, 2017) Essas ferramentas de software tipicamente se apresentam como orientadas para professores e/ou alunos, com poucas exceções fornecendo também recursos para administradores ou representantes institucionais. Por exemplo, Quizly (MAIORANA; GIORDANO; MORELLI, 2015) apresenta três módulos diferentes dependendo do usuário: administrador, aluno ou professor. Cada módulo tem funcionalidades relacionadas a cada papel, como por exemplo, apenas os professores podem criar atividades.

Em geral, os detalhes de implementação das abordagens não são apresentados na literatura encontrada, com exceção da ferramenta Hairball (BOE et al., 2013) e CodeMaster (GRESSE VON WANGENHEIM et al., 2018). Hairball foi desenvolvida como um conjunto de *scripts* em Python usando o paradigma de orientação a objetos, podendo ser estendida e adaptada para avaliar tarefas específicas. Dr. Scratch (MORENO-LEÓN; ROBLES, 2015) cita que foi implementado com base em Hairball. A análise de programas no Dr. Scratch é desenvolvida na linguagem Python, na qual a implementação de Hairball foi estendida. A ferramenta CodeMaster separa a análise, avaliação e apresentação em diferentes módulos. O sistema *back-end* foi implementado usando Java 8, rodando em um servidor de aplicativos Apache Tomcat 8, usando um banco de dados MySQL 5.7. O componente *front-end* foi implementado com JavaScript usando a biblioteca *Bootstrap* com um layout customizado (GRESSE VON WANGENHEIM et al., 2018)

O acesso às ferramentas é na forma de *scripts*, aplicação web ou desktop. Numa aplicação web, como Dr. Scratch, é permitido que o usuário informe o endereço web do projeto ou envie o arquivo do programa para executar a avaliação diretamente no navegador.

Não foi possível obter informações detalhadas sobre a licença de todas as ferramentas. No entanto, a autora do presente trabalho obteve acesso a uma implementação gratuita de várias ferramentas (Tabela 19). A maioria das ferramentas está disponível apenas em inglês, com exceção de algumas poucas ferramentas que oferecem internacionalização e localização para vários idiomas, como espanhol, inglês, português, etc., facilitando seu uso em diferentes países.

Tabela 19 - Visão geral sobre aspectos das ferramentas.

Abordagem	**Categorias de usuário**	**Plataforma de acesso**	**Licença**	**Língua**
Abordagem de Kwon e Sohn	Professor	Não identificado	Não informado	Inglês
Hairball	Professor	Conjunto de scripts em Python	Gratuita	Inglês
Dr.Scratch	Aluno, Professor e Instituição	Aplicação web	Gratuita	Espanhol, Inglês, Português, etc.
CTP/PBS/REACT	Professor	Aplicação web	Não informado	Inglês
ITCH	Aluno	Conjunto de scripts em Python	Não informado	Inglês
PECT	Professor	Rubrica (não automatizada)	Não informado	Inglês
Ninja code village	Aluno e Professor	Aplicação web	Gratuita	Inglês, Japonês
Fairy Assessment	Professor	Rubrica (não automatizada)	Não informado	Inglês
Abordagem de Denner et al.	Professor	Rubrica (não automatizada)	Não informado	Inglês
Scrape	Professor	Aplicação desktop	Gratuita	Inglês
Quizly	Aluno, Professor e Administrador	Aplicação web	Gratuita	Inglês
Lambda	Aluno e Professor	Aplicação web	Gratuita	Inglês
CodeMaster	Aluno, Professor e Administrador	Aplicação web	Gratuita	Inglês, Português
Abordagem de Funke et al.	Professor	Rubrica (não automatizada)	Não informado	Inglês

3.5 DISCUSSÃO

Levando em consideração a importância do suporte automatizado para a avaliação das atividades de programação, de forma a ampliar o ensino de computação na Educação Básica, observa-se que foram encontradas poucas abordagens. A maioria das abordagens é voltada à análise programas codificados em Scratch, sendo atualmente uma das VPLs baseadas em blocos mais utilizadas, popular em vários países, e disponível em vários idiomas.

As abordagens apresentadas nos artigos podem ser utilizadas para avaliação formativa e/ou somativa. Foram encontradas abordagens para diferentes tipos de atividades de programação, incluindo atividades fechadas com uma solução correta ou bem definida, como também para atividades abertas dentro do contexto de aprendizagem baseada em problemas.

O objetivo da maioria das abordagens é de apoiar o processo de avaliação do aluno pelo professor. No entanto, algumas também são voltadas ao uso pelos alunos para monitorar e orientar seu progresso de aprendizagem. Alguns exemplos são Dr. Scratch (MORENO-LEÓN; ROBLES, 2015), CodeMaster (GRESSE VON WANGENHEIM et al., 2018) e CTP (KOH et al., 2014a) que também fornece *feedback* em tempo real durante a atividade de programação. As abordagens tipicamente fornecem pontuação como *feedback*, baseada na análise do código, incluindo pontuações dicotômicas ou politômicas e pontuações compostas, fornecendo uma nota geral. No entanto, poucas abordagens fornecem dicas ou sugestões (além da pontuação), a fim de orientar o processo de aprendizagem de forma personalizada. Apenas duas abordagens (MORENO-LEÓN; ROBLES, 2015; GRESSE VON WANGENHEIM et al., 2018) usam elementos de gamificação, apresentando o nível de competência de forma lúdica com mascotes.

Para analisar o código criado pelo aluno, as abordagens utilizam análise estática, dinâmica ou manual. A vantagem das abordagens de análise estática que medem certas qualidades do código é que não há necessidade de uma solução predefinida implementada antecipadamente. Assim, a análise estática é uma solução para a avaliação de problemas abertos em contextos de aprendizagem baseados em problemas. No entanto, a inexistência de uma solução predefinida para tais atividades abertas limita sua análise em relação a certas características, não permitindo, por exemplo, validar a corretude do código. Por outro lado, cabe ressaltar que atividades abertas são importantes para estimular o desenvolvimento de conceitos mais

complexos, os quais requerem um esforço cognitivo maior. As abordagens de análise dinâmica podem ser aplicadas para a avaliação de atividades fechadas. No entanto, uma desvantagem, é que a estrutura de código interno não é analisada, isto é, um programa pode ser considerado correto (ao gerar o resultado esperado) mesmo quando os comandos de programação esperados não são usados e/ou o programa não possui um bom projeto de implementação.

Ao se concentrar em avaliações do desempenho a partir da análise do código criado pelos alunos, as abordagens inferem a avaliação de práticas de pensamento computacional relacionadas à algoritmos e programação. Isso ilustra a forte ênfase da análise de competências relacionadas à programação pela maioria das abordagens. Elementos adicionais, como usabilidade, organização do código, documentação, estética ou criatividade, raramente são avaliados, na maioria somente por abordagens manuais. Esta limitação deste tipo de avaliação subjetiva que é menos encontrada nas abordagens automatizadas, baseadas exclusivamente na análise do código, também indica a necessidade de métodos de avaliação alternativos. Esses métodos alternativos podem fornecer uma avaliação mais abrangente especialmente quando se trata de práticas e perspectivas de pensamento computacional (BRENNAN; RESNICK, 2012). Exemplos de avaliações alternativas que complementam a avaliação são observações ou entrevistas. A este respeito, as abordagens baseadas na análise do código podem ser consideradas **uma das formas de avaliar o pensamento computacional**, em vez de serem consideradas o único meio de avaliação e podem (e devem) ser usadas em conjunto com outros tipos de avaliação sem detrimentos. Cabe enfatizar, quando a avaliação de critérios mais objetivos é automatizada, pode-se liberar o tempo do professor para que este possa concentrar-se em outros métodos de avaliação complementares para avaliar critérios mais subjetivos. Assim, é possível prover ao aluno uma avaliação mais completa e mais rápida, na qual a avaliação automatizada concentra-se em aspectos objetivos e repetitivos, e a avaliação manual concentra-se em aspectos subjetivos, únicos e originais.

Observou-se também que não existe um consenso sobre os critérios avaliados (tanto subjetivos, como objetivos), níveis de desempenho e formas de pontuação entre as abordagens encontradas. Alguns artigos indicam o uso explícito de rubricas como base para a avaliação (SEITER; FOREMAN, 2013; WERNER et al., 2012; MORENO-LEÓN; ROBLES, 2015; OTA; MORIMOTO; KATO, 2016; GRESSE VON WANGENHEIM et al., 2018). Isso confirma as

descobertas de Grover e Pea (2013) e Grover, Pea e Cooper (2015) que, apesar dos muitos esforços sobre como avaliar o pensamento computacional, ainda não há consenso sobre as estratégias de avaliação.

Apenas algumas abordagens são automatizadas. Entre estas, algumas não fornecem uma interface de usuário, apresentando os resultados apenas via o terminal que executa os *scripts*. Isso pode dificultar a sua adoção devido à falta de conhecimento técnico de pessoas que não possuem conhecimentos de computação. Outro fator que pode dificultar na prática a adoção generalizada das ferramentas encontradas é a sua disponibilidade apenas em inglês, com apenas algumas exceções disponíveis em várias línguas. Observou-se ainda que essas ferramentas são fornecidas como ferramentas autônomas, ou seja, não são integradas a ambientes de programação e/ou sistemas de gerenciamento de cursos, a fim de facilitar sua adoção nos contextos educacionais existentes.

Esses resultados mostram que, embora existam algumas soluções pontuais, ainda há uma lacuna não só para ferramentas de avaliação automatizada, como também para a definição conceitual de estratégias de avaliação do pensamento computacional, haja vista os estudos demonstram que cada autor cria a sua, independentemente dos demais.

Também foi observada uma falta de contextualização dessas abordagens dentro do ambiente educacional, indicando, por exemplo, a etapa de ensino para a qual elas foram projetadas e também como essas abordagens podem ser completadas por métodos alternativos de avaliação para fornecer um *feedback* mais abrangente, incluindo conceitos e práticas subjetivas, as quais são difíceis de avaliar de forma automatizada. Estes resultados indicam a necessidade de mais estudos, a fim de apoiar em larga escala a inserção do ensino de computação nas escolas de Ensino Fundamental e Médio.

Como resultado desse estudo de mapeamento, várias oportunidades de pesquisa nessa área podem ser identificadas, incluindo a definição sistemática de critérios de avaliação bem definidos para uma grande variedade de VPL baseadas em blocos, especialmente para a avaliação de atividades abertas. no desenvolvimento de diversos tipos de aplicativos (jogos, animações, aplicativos, etc.). Considerando também restrições práticas e uma tendência de cursos online abertos e massivos, o desenvolvimento e melhoria de soluções automatizadas, que permitem uma avaliação e *feedback* rápido e em tempo real para os alunos, pode melhorar ainda mais o processo de aprendizagem dos alunos, bem como reduzir a carga de trabalho dos professores e, assim, ajudar a ampliar a o ensino de computação nas escolas.

3.6 AMEAÇAS À VALIDADE

Como em qualquer mapeamento da literatura, existem algumas ameaças à validade dos resultados. Portanto, foram identificadas ameaças potenciais, e aplicadas estratégias de mitigação para minimizar seu impacto.

Viés de publicação. Os mapeamentos sistemáticos podem sofrer o viés de que os resultados positivos são mais prováveis de serem publicados do que negativos. No entanto, considera-se que os resultados dos artigos, positivos ou negativos, têm pouca influência neste mapeamento sistemático, pois buscou-se caracterizar as abordagens ao invés de analisar seus impactos positivos/negativos na aprendizagem.

Identificação de estudos. Outro risco é a omissão de estudos relevantes. Para mitigar esse risco, a escolha das palavras-chave foi feita em conjunto com um pesquisador sênior de forma a ser tão inclusiva quanto possível. Foram considerados não apenas conceitos básicos, mas também sinônimos. O risco de excluir estudos primários relevantes foi mitigado pelo uso de múltiplas bases de dados (indexadas pelo SCOPUS) que cobrem a maioria das publicações científicas nesta área.

Seleção e extração de dados. As ameaças de seleção e extração de dados dos artigos foram mitigadas por meio de uma definição detalhada dos critérios de inclusão/exclusão. Foi definido e documentado um protocolo rígido para a seleção de artigos, e a autora do presente trabalho conduziu a seleção em conjunto com um pesquisador sênior, discutindo a seleção até ser atingido um consenso. A extração de dados foi dificultada em alguns casos, pois a informação relevante para esse estudo nem sempre estava relatada explicitamente e, portanto, em alguns casos, teve que ser inferida. No entanto, essa inferência foi feita pela autora e meticulosamente revisada por um pesquisador sênior.

4 AVALIAÇÃO EM LARGA ESCALA DO CODEMASTER V1.0

Especificamente voltado às VLPs App Inventor e Snap! já existe o modelo CodeMaster (GRESSE VON WANGENHEIM et al., 2018). CodeMaster possui uma rubrica para a avaliação de conceitos de algoritmos e programação relacionados à prática do pensamento computacional. Dispõe de uma implementação de sistema web gratuita para analisar e avaliar automaticamente projetos em App Inventor ou Snap! resultantes de atividades abertas. CodeMaster pode ser usado tanto por alunos, para obter *feedback* imediato em todo o processo de aprendizagem sobre um projeto específico, quanto por professores, a fim de avaliar e classificar todos os projetos de programação de uma turma.

O modelo adota uma estratégia de avaliação autêntica que mede o desempenho dos alunos com base numa análise estática sobre os programas criados pelos alunos, resultantes de atividades abertas de programação. Para avaliar se o resultado produzido pelos alunos demonstra que eles aprenderam o pensamento computacional, é usada uma rubrica que avalia as competências com base em indicadores predefinidos (disponível no Anexo A). Inspirado no Dr. Scratch (MORENO-LEÓN; ROBLES, 2015), a complexidade dos programas dos alunos é medida em relação a várias subdimensões do pensamento computacional. A rubrica CodeMaster v1.0 para AppInventor contém 15 critérios, doravante chamados de itens. O modelo de pontuação é politômico. Todos os itens possuem 4 níveis e são pontuados numa escala [0, 3], a descrição de cada pontuação está disponível no Anexo A.

Tabela 20 - Itens da rubrica CodeMaster v1.0 para App Inventor.

Conceitos e práticas de pensamento computacional	**Pensamento computacional móvel**
I01. Nomeação	I08. Persistência de dados
I02. Abstração de procedimentos	I09. Telas
I03. Eventos	I10. Interface de usuário
I04. Laços	I11. Sensores
I05. Condicionais	I12. Mídia
I06. Operadores	I13. Social
I07. Listas	I14. Conectividade
I08. Persistência de dados	I15. Desenho e animação

A definição da rubrica em relação ao pensamento computacional é baseada na definição de pensamento computacional proposta por Brennan e Resnick (2012) e no pensamento computacional móvel proposto por Sherman e Martin (2015; 2014). Diferente da definição do CSTA (conforme apresentada na seção 2.1.1), a definição de Brennan e

Resnick propõe a divisão do pensamento computacional em práticas e conceitos, e o pensamento computacional móvel proposto por Sherman e Martin (2015; 2014) adota itens referentes à conceitos representados especificamente em produtos de software criados com App Inventor (Tabela 20).

Inicialmente, a ferramenta CodeMaster foi avaliada somente em termos da sua corretude e usabilidade (GRESSE VON WANGENHEIM et al., 2018). Assim, uma questão em aberto é a confiabilidade e validade da rubrica para servir como ponto de partida para o desenvolvimento do modelo proposto no presente trabalho.

4.1 DEFINIÇÃO E EXECUÇÃO DO ESTUDO

O objetivo deste estudo é analisar a rubrica CodeMaster v1.0. A qualidade da rubrica é avaliada em termos de confiabilidade e validade de construto do ponto de vista de pesquisadores no contexto do ensino de computação na Educação Básica.

Seguindo a abordagem GQM (BASILI; CALDIERA; ROMBACH, 1994) o objetivo do estudo é definido e decomposto em aspectos de qualidade e questões de análise a serem investigadas com base nos dados coletados.

Confiabilidade

AQ1: Existe evidência de consistência interna da rubrica CodeMaster v1.0 para App Inventor?

Validade

AQ2: Existe evidência de validade convergente da rubrica CodeMaster v1.0 para App Inventor?

AQ3: Como os fatores subjacentes influenciam as respostas nos critérios da rubrica CodeMaster v1.0 para App Inventor?

Coleta de dados. Para maximizar o tamanho da amostra, foi realizado o *download* de todos os aplicativos públicos disponíveis e acessíveis da Galeria App Inventor[2] em maio de 2018, armazenando seus arquivos *.aia*. Um arquivo *.aia* é uma coleção compactada de arquivos que inclui um arquivo de propriedades do projeto, arquivos de mídia usados pelo aplicativo e, para cada tela do aplicativo, dois arquivos principais: um arquivo *.bky* e um arquivo *.scm*. O arquivo *.bky* encapsula uma estrutura XML com todos os blocos de programação usados na programação do aplicativo, e o arquivo *.scm* encapsula uma estrutura JSON que contém todos os componentes visuais usados no

[2] http://ai2.appinventor.mit.edu/

aplicativo (TURBAK et al., 2017). Como resultado, foi obtido o código-fonte de 88.864 aplicativos App Inventor.

Avaliação e classificação dos aplicativos App Inventor. Os aplicativos obtidos foram analisados usando a ferramenta CodeMaster v1.0. Dos 88.864 projetos baixados, 88.606 foram analisados com sucesso com o CodeMaster v1.0. Devido a dificuldades técnicas, 256 aplicativos App Inventor não puderam ser analisados. Os resultados da avaliação foram exportados em um arquivo SQL e, em seguida, convertidos em um arquivo CSV. Os dados coletados foram agrupados em uma única tabela para validar a rubrica CodeMaster v1.0 usando a linguagem de programação R.

4.2 ANÁLISE DOS DADOS

Considerou-se a obrigatoriedade de todos os itens para todos os projetos. A Tabela 21 apresenta a frequência da pontuação por itens cuja descrição de cada pontuação está disponível no Anexo A.

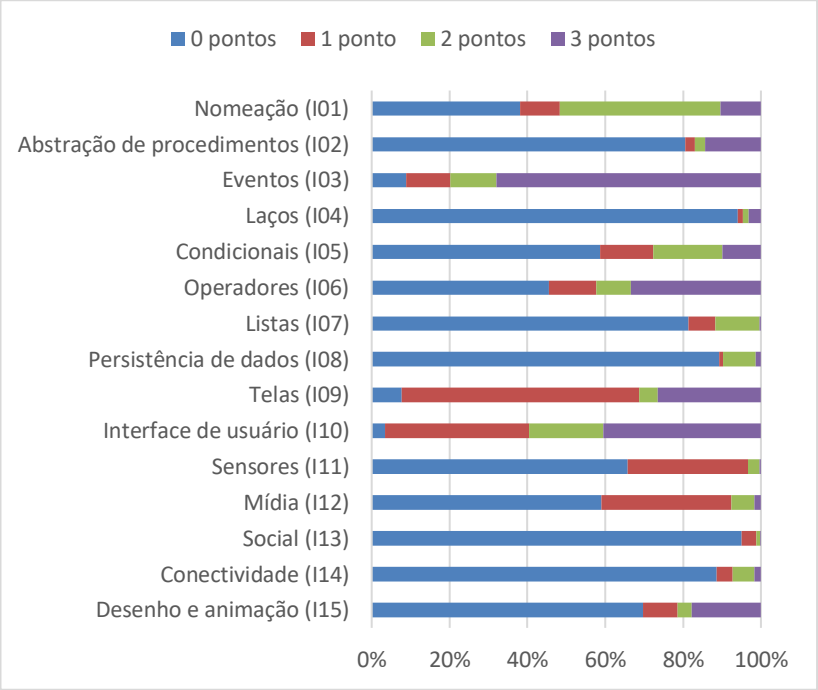
Tabela 21 - Frequência de pontuação por item na rubrica CodeMaster v1.0.

Item	**0**	**1**	**2**	**3**
I01. Nomeação	33811	9055	36537	9205
I02. Abstração de procedimentos	71357	2162	2338	12751
I03. Eventos	7774	10130	10444	60260
I04. Laços	83375	1061	1439	2733
I05. Condicionais	51957	12203	15714	8734
I06. Operadores	40280	10859	7921	29548
I07. Listas	72070	6139	10141	**258**
I08. Persistência de dados	79127	853	7472	1156
I09. Telas	6843	54096	4188	23481
I10. Interface de usuário	2971	32909	16796	35932
I11.Sensores	58315	27315	2702	**276**
I12. Mídia	52351	29506	5244	1507
I13. Social	84225	3326	813	**244**
I14. Conectividade	78485	3619	4949	1555
I15. Desenho e animação	61759	7921	3131	15797

Cabe ressaltar que alguns itens têm poucos dados em algumas pontuações, como os itens 7 (Listas), 11 (Sensores) e 13 (Social) para a pontuação 3 (marcados em negrito na Tabela 21). Apenas o item 3 (Eventos) apresentou pontuação 3 em mais de 60% dos projetos. Isso é explicado devido à VPL App Inventor ser orientada a eventos e, portanto, estimula a utilização desse conceito (TURBAK et al., 2014). A

Figura 21 apresenta as porcentagens de pontuação por itens por meio de um gráfico de barras empilhadas.

Figura 21 - Porcentagem de pontuação por item na rubrica CodeMaster v1.0.



Fonte: elaborada pela autora.

AQ1. Existe evidência de consistência interna da rubrica CodeMaster v1.0 para App Inventor?

Foi analisada a confiabilidade medindo a consistência interna da rubrica CodeMaster v1.0 para App Inventor por meio do coeficiente alfa de Cronbach (1951). O coeficiente alfa de Cronbach indica indiretamente o grau que um conjunto de itens mede um único fator, neste caso, se a rubrica CodeMaster v1.0 para App Inventor mede o pensamento computacional. Tipicamente, valores de alfa de Cronbach entre $0,7 < \alpha \leq 0,8$ são aceitáveis, entre $0,8 < \alpha \leq 0,9$ são bons, e $\alpha \geq 0,9$ são excelentes (DEVELLIS, 2003), indicando assim uma consistência interna do instrumento. Analisando os 15 itens da rubrica CodeMaster

v1.0 para App Inventor, obteve-se um valor aceitável do alfa de Cronbach ($\alpha = 0,79$).

AQ2. Existe evidência de validade convergente da rubrica CodeMaster v1.0 para App Inventor?

Para obter evidências da validade convergente dos itens da rubrica CodeMaster v1.0 para App Inventor, calculam-se as intercorrelações dos itens e correlação total do item (DEVELLIS, 2003). Para a validade convergente, espera-se que os critérios da mesma subdimensão tenham uma correlação mais alta (CARMINES; ZELLER, 1982; TROCHIM; DONNELLY, 2008).

Para analisar as intercorrelações entre os itens de uma mesma subdimensão, utilizou-se a correlação policórica, por ser a mais apropriada para variáveis observadas ordinais (OLSSON, 1979), conforme definidas na rubrica CodeMaster v1.0 (disponível no Anexo A). A matriz mostra o coeficiente de correlação, indicando o grau de correlação entre dois itens ordinais (pares de itens). De acordo com Cohen (1998), uma correlação entre os itens é considerada satisfatória, se o coeficiente de correlação for maior que 0,29, indicando que há uma correlação média ou alta entre os itens. Os resultados da análise são apresentados na Tabela 22 com correlações satisfatórias marcadas em negrito. Valores menores que 0 tiveram sua parte inteira suprimida para uma apresentação de resultados mais compacta sem prejuízos, ou seja, o valor 0,609, por exemplo, é apresentado como apenas ,609.

Tabela 22 - Coeficiente de correlação policórica da rubrica CodeMaster v1.0.

Item	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1,0													
2	**,609**	1,0												
3	**,485**	**,595**	1,0											
4	**,375**	**,582**	,260	1,0										
5	**,475**	**,616**	**,580**	**,476**	1,0									
6	**,593**	**,806**	**,680**	**,486**	**,796**	1,0								
7	**,346**	**,431**	**,320**	**,696**	**,471**	**,392**	1,0							
8	**,300**	**,337**	**,570**	**,425**	**,465**	**,417**	**,443**	1,0						
9	,160	,198	**,720**	,229	**,357**	**,349**	**,295**	**,446**	1,0					
10	**,313**	**,359**	**,640**	,280	**,514**	**,503**	**,358**	**,507**	**,470**	1,0				
11	,261	**,486**	**,480**	,212	**,384**	**,547**	,198	,284	,200	,240	1,0			
12	,035	,139	,260	-,054	,029	-,002	,043	,038	,160	,120	,260	1,0		
13	,045	-,014	**,30**	,150	,123	,011	,099	**,415**	,280	,250	,110	,198	,0	
14	,022	,016	**,350**	,140	,284	,062	,138	**,268**	,180	**,400**	,190	-,169	,220	1,0
15	**,376**	**,582**	**,470**	,071	,215	**,626**	-,050	-,057	,120	,120	**,490**	,139	-,188	-,388

Itens que pertencem à subdimensão de conceitos e práticas do pensamento computacional mostram um grau aceitável de correlação em

quase todos os pares de itens pois não foram detectados valores negativos em nenhum grupo de itens. Apenas o par 3 e 4 (Eventos e Laços) apresenta um valor abaixo de 0,29. No entanto, isso pode ser devido ao fato de que o uso de laços (*loops*) às vezes pode ser substituído pelo uso de eventos em aplicativos do App Inventor (TURBAK et al., 2014). Em geral, pode-se observar uma tendência para os itens dessa subdimensão serem correlacionados, indicando evidência de validade convergente. Os itens dessa subdimensão também mostram algumas correlações elevadas com os itens da subdimensão pensamento computacional móvel. Isso não representa um problema, pois está se avaliando, em geral, subdimensões de um mesmo conceito: o pensamento computacional.

Por outro lado, apenas três pares de itens da subdimensão do pensamento computacional móvel apresentam uma correlação acima de 0,29 (par 10-9, par 10-14 e par 11-15). Os outros itens apresentam correlação baixa ou até mesmo negativa, especialmente, a forte correlação negativa entre o item 14 (Conectividade) e o item 15 (Desenho e animação). Isso está indicando que quanto mais elementos relacionados a Desenho e animação estão presentes em um aplicativo, menos elementos relacionados à Conectividade estão presentes. Isso, de fato, faz sentido, pois para habilitar o desenho, ou para apresentar uma animação por meio de um aplicativo, não se exige nenhum tipo de componente de Conectividade, como por exemplo, solicitações HTTP da web ou conexões Bluetooth. Outros itens dessa subdimensão, itens 12, 13 e 14 (Mídia, Social e Conectividade respectivamente), também apresentam correlações muito baixas com os itens da subdimensão que estão inseridos (pensamento computacional móvel). Isso novamente pode ser explicado pelo fato de que a maioria dos aplicativos não apresenta características de todos esses itens ao mesmo tempo. Por exemplo, um aplicativo pode acessar a câmera do dispositivo (componente de Mídia) sem uso de um componente Social (por exemplo, fazer uma ligação telefônica) ou componentes de Conectividade, sendo a presença de um item já suficiente para o propósito de um aplicativo. Consequentemente, no que diz respeito à avaliação do pensamento computacional móvel, isso pode indicar que nem todos esses itens devem ser considerados na avaliação. Por exemplo, exigir o uso de apenas um desses itens de acordo com a finalidade específica do aplicativo pode ser suficiente, em vez de exigir o uso de todos eles em um aplicativo.

Vários pares de itens de diferentes subdimensões também apresentaram um alto grau de correlação. Novamente, apenas os itens

12, 13 e 14 (Mídia, Social e Conectividade respectivamente), do pensamento computacional móvel, apresentam baixo grau de correlação com a subdimensão de conceitos e práticas do pensamento computacional. No entanto, como já discutido, esses itens também mostraram correlações baixas com itens da mesma subdimensão. Consequentemente, para avaliar as competências de pensamento computacional, essas baixas correlações entre os itens de mesma subdimensão indicam que eles podem não estar mensurando esse fator. Portanto, esses itens serão analisados detalhadamente nas análises posteriores, a fim de verificar se isso é confirmado e se é necessário eliminar esses itens da rubrica.

Correlação item-total.

Complementando a análise anterior, a correlação entre todos os outros itens é analisada. Cada item da rubrica deve ter uma correlação média ou alta com todos os outros itens (DEVELLIS, 2003), pois isso indica que os itens apresentam consistência em comparação com os outros itens, já uma baixa correlação item-total de um item enfraquece a validade da rubrica. Para esta análise, utilizou-se o método da correlação item-total corrigida. O valor de referência para a análise é o mesmo apresentado na seção anterior, considera-se uma correlação satisfatória se o valor for acima de 0,29 (COHEN, 1998). Além disso, é analisado como fica o alfa de Cronbach se o item é retirado, espera-se que nenhum item cause um aumento no alfa de Cronbach se retirado (DEVELLIS, 2003). Os resultados desta análise são apresentados na Tabela 23.

Tabela 23 - Resultados da análise da correlação item-total para a rubrica CodeMaster v1.0 para App Inventor.

Item	Correlação Item-total	Alfa de Cronbach se o item for removido
I01. Nomeação	0,515	0,77
I02. Abstração de procedimentos	0,537	0,77
I03. Eventos	0,596	0,77
I04. Laços	0,280	0,79
I05. Condicionais	0,612	0,76
I06. Operadores	0,709	0,75
I07. Listas	0,350	0,79
I08. Persistência de dados	0,321	0,79
I09. Telas	0,331	0,79
I10. Interface de usuário	0,502	0,77
I11.Sensores	0,429	0,78
I12. Mídia	**0,110**	**0,80**
I13. Social	**0,089**	**0,80**
I14. Conectividade	**0,091**	**0,80**
I15. Desenho e animação	0,329	0,79

A maioria dos valores das correlações item-total estão acima de 0,29. Isso indica que existe uma consistência interna aceitável. O grau de correlação entre os itens mostra que os itens medem a mesma dimensão, indicando evidência de validade convergente. No entanto, novamente, esta análise confirma a questão em relação aos itens 12, 13 e 14 (Mídia, Social e Conectividade respectivamente), pois apresentam uma baixa correlação item-total (marcadas em negrito na Tabela 23). Consequentemente, esses resultados confirmam que esses itens não demonstram uma correlação satisfatória com os demais itens e, portanto, indicam que podem não estar necessariamente medindo o pensamento computacional móvel. Além disso, o valor do alfa de Cronbach, se esses itens forem excluídos, apresenta um aumento. Portanto, esses itens precisam ser analisados, pois podem não contribuir efetivamente para a avaliação. O item 4 (Laços) mostra uma correlação de 0,28, no entanto, ao contrário dos itens mencionados anteriormente, a correlação ainda é muito próxima do valor mínimo esperado (0,29). Novamente, isso pode ser devido ao fato de que o uso de comandos de Laços não é amplamente aplicado nos programas App Inventor devido à natureza do software que está sendo desenvolvido (XIE; ABELSON, 2016; TURBAK et al., 2014). Todos os outros itens mostram uma diminuição no valor do alfa de Cronbach se removidos e demonstram correlação item-total suficiente, indicando, assim, a validade das competências mensuradas.

AQ3. Como os fatores subjacentes influenciam as respostas nos critérios da rubrica CodeMaster v1.0 para App Inventor?

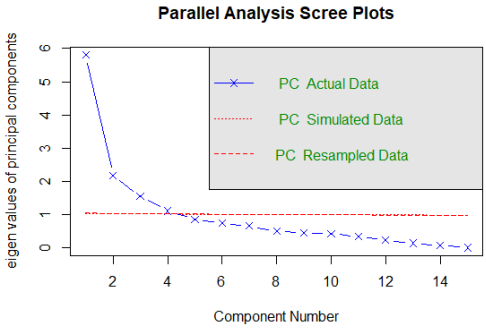
Para identificar o número de fatores que representa a aplicação dos conceitos por meio da programação dos 15 itens da rubrica CodeMaster v1.0 para App Inventor, foi realizada uma análise fatorial. Com base na definição da rubrica CodeMaster v1.0 para App Inventor, assumiu-se que ela é influenciada por duas subdimensões: **conceitos e práticas de pensamento computacional (I)** e **pensamento computacional móvel (II)**.

Para analisar se os critérios da rubrica CodeMaster v1.0 para App Inventor podem ser submetidos a um processo de análise fatorial, foi utilizado o índice Kaiser-Meyer-Olkin (KMO) sendo o mais utilizado na prática (BROWN, 2006). O índice KMO mede a adequação da amostragem com valores entre 0 e 1. Um valor de índice próximo a 1,0 suporta uma análise fatorial e qualquer valor menor que 0,5 indica que provavelmente não é passível de análise fatorial (BROWN, 2006). Analisando o conjunto de critérios da rubrica CodeMaster v1.0 para App

Inventor, foi obtido um índice KMO de 0,80 indicando que a análise fatorial é apropriada neste caso.

O número de fatores retidos na análise é decidido aplicando-se a análise fatorial (GLORFELD, 1995). Foi utilizada a análise paralela que é um método para determinar o número de componentes ou fatores a serem retidos da análise fatorial. Essencialmente, o método funciona criando um conjunto de dados aleatório com o mesmo número de observações e variáveis que os dados originais. Os resultados da análise paralela mostram que 3 fatores devem ser retidos, como mostra o scree plot (Figura 22), indicando três pontos bem acima da linha vermelha. Assim, a análise fatorial indica uma decomposição em 3 subdimensões, diferente da originalmente proposta composta de apenas 2 subdimensões.

Figura 22 - Scree Plot referente à rubrica CodeMaster v1.0.



Fonte: elaborada pela autora.

Uma vez identificado o número de fatores subjacentes, outra questão é determinar quais itens são carregados em qual fator. Para identificar as cargas fatoriais dos itens, utiliza-se um método de rotação (BROWN, 2006). Foi utilizado o método de rotação Oblimin, no qual os fatores podem ser correlacionados (JACKSON, 2005).

O limiar para análise fatorial não é padronizado na literatura: Stevens (1992) considera aceitável valores acima de 0,4 e Comrey e Lee (1992) sugerem o uso de limites mais restritos: 0,32 (ruim), 0,45 (aceitável), 0,55 (bom), 0,63 (muito bom) ou 0,71 (excelente). No presente trabalho, como o tamanho da amostra é bastante grande (88 mil), considera-se que valores de carregamento acima de 0,45 são significativos. A Tabela 24 mostra as cargas fatoriais dos itens

associados aos 3 fatores retidos. O maior fator de carga de cada critério indica a qual fator o critério está **mais** relacionado (marcado em negrito na Tabela 24). Cabe salientar que alguns itens, levando em consideração o limiar de 0,45, podem estar relacionados a mais de 1 fator.

Tabela 24 - Cargas fatoriais para 3 fatores.

Item	Fator 1	Fator 2	Fator 3
I01. Nomeação	0,310	0,345	**0,444**
I02. Abstração de procedimentos	0,356	0,410	**0,688**
I03. Eventos	0,376	**0,920**	0,081
I04. Laços	**0,606**	0,119	0,441
I05. Condicionais	**0,648**	0,333	0,489
I06. Operadores	0,417	0,479	**0,723**
I07. Listas	**0,579**	0,159	0,210
I08. Persistência de dados	**0,605**	0,463	0,009
I09. Telas	0,367	**0,700**	-0,189
I10. Interface de usuário	**0,526**	0,482	0,066
I11.Sensores	0,073	**0,484**	0,365
I12. Mídia	-0,094	**0,344**	-0,098
I13. Social	0,328	**0,350**	-0,343
I14. Conectividade	**0,571**	0,108	-0,257
I15. Desenho e animação	-0,389	0,598	**0,658**

Analisando as cargas fatoriais dos itens (Tabela 24), pode-se observar que, o primeiro fator (fator 1), está mais relacionado ao conjunto dos itens 4, 5, 7, 8, 10 e 14 (Laços, Condicionais, Listas, Persistência de dados, Interface de usuário e Conectividade respectivamente). Segundo Brennan e Resnick (2012), alguns desses itens estão relacionados à subdimensão conceitos e práticas de pensamento computacional. Entretanto, os itens 10 e 14, originalmente relacionados à subdimensão do pensamento computacional móvel, também estão sendo indicados como itens desse fator. Os dados indicam que o item 10 (Interface do usuário) é quase tão relacionado a esses itens mencionados anteriormente (0,526 fator de carga) quanto aos itens de pensamento computacional móvel (0,482 fator de carga).

Os demais itens, itens 1, 2 e 6 (Nomeação, Abstração de procedimentos e Operadores respectivamente) originalmente relacionados à subdimensão de conceitos e práticas do pensamento computacional estão com cargas fatorais mais altas no fator 3, teoricamente relacionado somente à subdimensão de práticas de pensamento computacional. Desta forma, os dados apresentam uma separação desses itens da subdimensão de conceitos de pensamento computacional, conforme proposto pela rubrica de pensamento computacional de Brennan e Resnick (2012). O agrupamento do item 6

(Operadores) como pertencente à subdimensão de práticas de pensamento computacional parece ser inadequado, sendo um critério semanticamente relacionado à subdimensão de conceitos de pensamento computacional (fator 1). No entanto, cabe ressaltar que esse item também mostra um alto fator de carregamento para os outros dois fatores, de modo que, em geral, o item 6 tem uma carga forte nos três fatores. Com base nos resultados da análise fatorial, o item 15 (Desenho e animação) também é agrupado nesse terceiro fator. Semanticamente, esse item, o qual permite definir telas na interface do aplicativo e objetos animados, está mais relacionado à subdimensão do pensamento computacional móvel. No entanto, o alto valor de carregamento no fator 3 pode ser explicado pelo fato de que a criação de uma animação requer tipicamente muitos elementos relacionados à prática do pensamento computacional em geral e não apenas de componentes de aplicações móveis.

As cargas fatoriais do segundo fator indicam fortemente itens relacionados ao pensamento computacional móvel, incluindo itens como Sensores, Mídia e Social (itens 11, 12 e 13 respectivamente), além do item Telas (item 9). Cabe ressaltar o alto valor de carregamento do item Eventos (item 3), sendo o maior valor de carregamento de todos os itens. Esse alto valor pode ser explicado devido à natureza da VPL App Inventor ser fortemente baseada em eventos (TURBAK et al., 2014). Em geral, pode-se observar que os resultados da análise fatorial confirmam parcialmente a estrutura da rubrica original do CodeMaster. Porém, indicam a separação da subdimensão do pensamento computacional em conceitos e práticas conforme proposto por Brennan e Resnick (2012).

Levando em consideração que o objetivo geral da rubrica CodeMaster v1.0 para App Inventor é a avaliação de um único conceito: o pensamento computacional, é importante determinar o quanto os itens estão sendo carregados em **um único fator**. A Tabela 25 mostra as cargas fatoriais dos itens associadas a um fator. Considera-se que valores de carregamento acima de 0,45 são significativos (marcados em negrito na Tabela 25).

Analisando as cargas fatoriais dos itens da Tabela 25, pode-se observar que, a maioria dos itens apresenta valores acima do limiar de 0,45. Entretanto, os itens 12, 13 e 14 (Mídia, Social e Conectividade respectivamente) chamam a atenção com valores de cargas fatoriais bem abaixo do valor definido (0,45). Assim, novamente, esse resultado sugere que esses itens não estão bem formulados, de forma que sua relação com o pensamento computacional não está clara, exigindo uma

revisão de sua definição e possivelmente sua exclusão da rubrica CodeMaster v1.0 para App Inventor.

Tabela 25 - Carregamento em um fator.

Item	Fator
I01. Nomeação	**0,640**
I02. Abstração de procedimentos	**0,874**
I03. Eventos	**0,867**
I04. Laços	**0,704**
I05. Condicionais	**0,828**
I06. Operadores	**0,923**
I07. Listas	**0,548**
I08. Persistência de dados	**0,631**
I09. Telas	**0,579**
I10. Interface de usuário	**0,614**
I11.Sensores	**0,580**
I12. Mídia	0,144
I13. Social	0,226
I14. Conectividade	0,253
I15. Desenho e animação	**0,544**

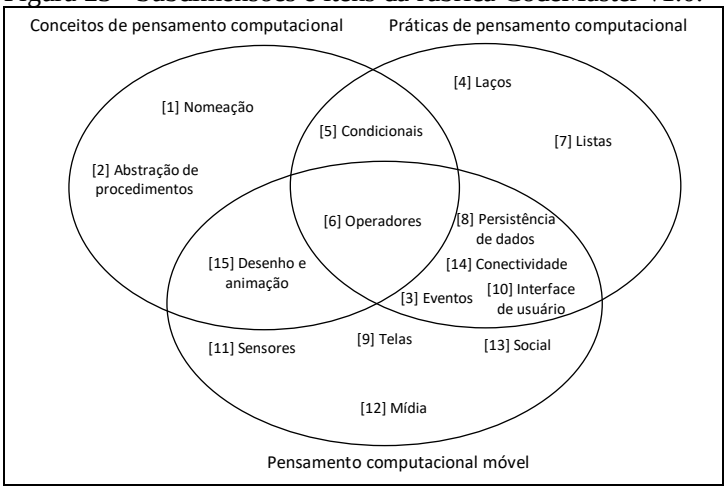
4.3 DISCUSSÃO

Os resultados obtidos, em geral, indicam confiabilidade aceitável (alfa de Cronbach acima de 0,7) e validade da rubrica CodeMaster v1.0 para App Inventor, de forma que avaliam bem algumas competências de pensamento computacional. No entanto, a análise também aponta a necessidade de revisão e reestruturação da rubrica com relação aos itens do pensamento computacional móvel. Especialmente os itens 12, 13 e 14 (Mídia, Social e Conectividade) precisam ser revisados e possivelmente excluídos. Uma razão para a baixa correlação desses itens pode ser devido à maneira em como seus níveis de desempenho estão sendo medidos. Outra causa para o resultado da análise em relação ao item 13 (Social), pode ser o pequeno número de aplicativos (menos de 7% dos programas App Inventor) que usam comandos relacionados a esse item.

A análise fatorial também mostrou que vários itens possuem altas cargas fatoriais em relação a diferentes subdimensões (Figura 23). Por exemplo, o item 6 (Operadores) apresenta altas cargas fatoriais em relação aos três fatores, enquanto outros itens, como Persistência de dados (item 8), Desenho e animação (item 15) e Sensores (item 11), apresentam altas cargas fatoriais em relação a dois fatores. Alguns desses itens podem ser claramente agrupados semanticamente em um fator, como por exemplo, Condicionais (item 5), Operadores (item 6) e

Persistência de dados (item 8) podem ser agrupados como conceitos do pensamento computacional (Figura 23).

Figura 23 - Subdimensões e itens da rubrica CodeMaster v1.0.



Fonte: elaborada pela autora.

Embora essa classificação, baseada na definição formal, possa não estar em total conformidade com os resultados da análise, observa-se que os resultados podem ter sido influenciados pela formulação dos itens, voltada às características da linguagem que é fortemente relacionada ao pensamento computacional móvel. Como por exemplo, o item 8 (Persistência de dados) é formulado com o objetivo de medir especificamente elementos de persistência relacionados a componentes móveis e, portanto, apresentou uma alta carga fatorial no fator relacionado ao pensamento computacional móvel. Assim, mais uma vez, isso indica a necessidade de uma revisão da classificação dos itens em geral.

Um item relevante que apresenta altos fatores de carregamento em dois fatores é o item 3 (Eventos). Isso pode ser explicado pelo fato de que eventos é um conceito básico do pensamento computacional (BRENNAN; RESNICK, 2012), mas também, devido à linguagem App Inventor ser orientada a eventos, esse item também se correlaciona fortemente com o pensamento computacional móvel (TURBAK et al., 2014). Esse resultado ainda confirma as diferenças entre VPL baseada em blocos (como por exemplo, Scratch e App Inventor) e os tipos de artefatos de software criados, (por exemplo, jogos e aplicativos) que

requerem diferentes padrões de uso de comandos e componentes. E, embora os programas Scratch não contenham componentes móveis, já que não se trata de uma VPL voltada à criação de aplicativos móveis, essas diferenças também se estendem ao uso de comandos básicos. Por exemplo, como os programas do App Inventor são essencialmente orientados a eventos, eles apresentam um uso muito menor de comandos de laços (XIE; ABELSON, 2016), bem como a inaplicabilidade do conceito de paralelismo (TURBAK et al., 2014), enquanto os programas Scratch normalmente apresentam um alto uso desses tipos de comandos, já que a linguagem incentiva o uso de laços de forma ilimitada (ARMONI; MEERBAUM-SALANT; BEN-ARI, 2015). Consequentemente, essas diferenças indicam a necessidade de especialização de rubricas, para cada tipo de linguagem de programação e/ou artefatos de software, com itens diferenciados, mas que meçam o mesmo objetivo de aprendizagem geral.

Além disso, os itens que foram originalmente definidos com base em diferentes definições de pensamento computacional (BRENNAN; RESNICK, 2012; SHERMAN; MARTIN, 2015) devem ser revisados de forma a alinhá-los aos objetivos de aprendizagem de pensamento computacional relacionado aos conceitos de algoritmos e programação do CSTA (2016). Ainda, de forma a medir, a partir do código, todas as práticas relacionadas ao pensamento computacional (CSTA, 2016), observou-se também a falta de critérios com relação a outros conceitos importantes de algoritmos e programação, como itens específicos para o conceito de variáveis, manipulação de *strings*, etc.

Ameaças à validade.

Esta avaliação está sujeita a várias ameaças à validade. Portanto, foram identificadas ameaças potenciais e aplicadas estratégias de mitigação para minimizar seu impacto nos resultados. Algumas ameaças estão relacionadas ao projeto do estudo de caso. A fim de mitigar esta ameaça, foi definida e documentada uma metodologia sistemática para o estudo usando a abordagem GQM (BASILI; CALDIERA; ROMBACH, 1994). Outro risco refere-se à qualidade dos dados reunidos em uma única amostra, em termos de padronização de dados. Como o estudo é exclusivamente limitado a avaliações usando o CodeMaster v1.0 para App Inventor, esse risco é minimizado, pois todas as análises foram realizadas de forma automatizada usando a mesma rubrica. Outra questão refere-se ao agrupamento de dados de diferentes contextos. Os aplicativos do conjunto de dados vêm de diversos contextos da comunidade App Inventor, de todos os lugares no mundo, e nenhuma informação adicional sobre o histórico dos criadores dos programas na

Galeria App Inventor está disponível. No entanto, como o objetivo é a análise da validade da rubrica de uma forma independente do contexto, isso não apresenta um problema. Em termos de validade externa, uma ameaça à possibilidade de generalizar os resultados está relacionada ao tamanho da amostra e à diversidade dos dados utilizados para a avaliação. A análise é baseada em dados coletados da Galeria AppInventor, envolvendo uma amostra de 88.606 aplicativos da comunidade do App Inventor. Este é um tamanho de amostra satisfatório, permitindo a geração de resultados significativos.

Em termos de confiabilidade, uma ameaça refere-se à dependência, dos dados e da análise, de pesquisadores específicos. A fim de mitigar essa ameaça, foi documentada uma metodologia sistemática, definindo claramente o objetivo do estudo, o processo de coleta de dados e os métodos estatísticos utilizados para a análise de dados. Outra questão refere-se à escolha correta de métodos estatísticos para análise de dados. Para minimizar essa ameaça, todo o processo foi apoiado e revisado por um especialista na área de estatística.

Cabe ressaltar que nessa análise considerou-se a obrigatoriedade de todos os itens para todos os programas. Contudo, nem sempre todos os itens especificados são realmente necessários em um programa que se deseja avaliar. No entanto, devido à grande quantidade de dados (mais de 88 mil) é inviável, no contexto de uma avaliação em larga escala, observar todos os programas para decidir quais são os itens obrigatórios para cada programa separadamente. Como o objetivo dessa avaliação é avaliar de forma geral todos os itens (ao invés de avaliar e atribuir nota ao projeto de um aluno especificamente) considera-se que essa decisão teve um impacto mínimo.

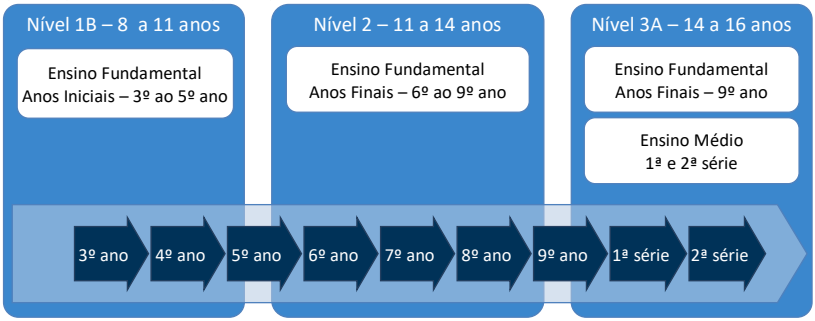
5 DESENVOLVIMENTO DO MODELO

Esse capítulo apresenta o contexto, desenvolvimento e implementação do modelo conceitual de avaliação do pensamento computacional via análise estática de código de VPL baseada em blocos para o contexto da Educação Básica.

5.1 CONTEXTO

Seguindo o modelo de design instrucional ADDIE (BRANCH, 2009), inicialmente o contexto é analisado. Levando em consideração a importância de se inserir o ensino de computação já na Educação Básica e os avanços já feitos nesse sentido no Brasil a partir da BNCC (MEC, 2018) e no mundo (CSTA, 2016), consequentemente, surgem demandas por formas de avaliação para esse contexto. Desta forma, o modelo desenvolvido está inserido dentro do contexto da Educação Básica, especificamente a partir do 3º ano do Ensino Fundamental até a 2ª série do Ensino Médio, para alunos a partir dos 8 anos, até os 16 anos (Figura 24).

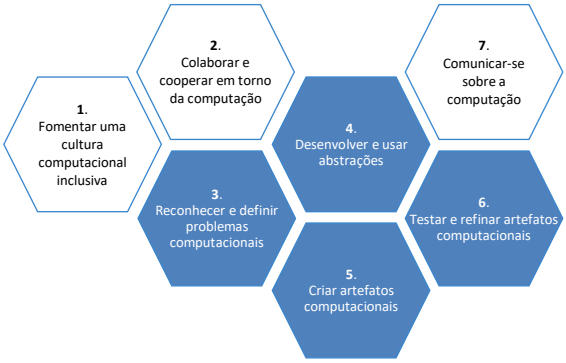
Figura 24 - Contexto educacional do modelo em relação ao CSTA (2016) e à BNCC (2018).



Fonte: elaborada pela autora com base em CSTA (2016) e BNCC (2018).

No contexto da Educação Básica, seguindo a definição do CSTA (2016), o pensamento computacional é delineado pelas práticas 3–6 do *K-12 Computer Science Framework* (Figura 25). Envolve a compreensão acerca da capacidade dos computadores, formulação de problemas a serem resolvidos pelo computador e projeto de algoritmos que possam ser executados por um computador (CSTA, 2016).

Figura 25 - Práticas relacionadas ao pensamento computacional.



Fonte: elaborada pela autora com base em CSTA (2016).

A partir das práticas 3-6, o CSTA apresenta as competências que os alunos devem ter ao final do Ensino Médio (coluna 2 - Tabela 26). Cabe ressaltar que a BNCC também apresenta, em um nível abstrato, dentro da área de Matemática, algumas habilidades fortemente relacionadas às competências descritas pelo CSTA, as quais um aluno do Ensino Médio deve desenvolver (coluna 1 - Tabela 26). Assim, a Tabela 26 relaciona as **competências da BNCC** (apresentadas na coluna 1), com as **competências do CSTA** (apresentadas na coluna 2). Essas competências podem ser desenvolvidas por meio de unidades instrucionais que abordam **conceitos de algoritmos e programação** (coluna 3 - Tabela 26).

Tabela 26 - Competências que os alunos devem ter ao final do Ensino Médio segundo a BNCC (2018) e o CSTA (2016; 2017).

Ao final do Ensino Médio os alunos devem ser capazes de:		**Conceitos de algoritmos e programação CSTA (2016; 2017)**
BNCC (2018)	**Práticas do CSTA (2016; 2017)**	
EM13MAT315. Reconhecer um problema algorítmico, enunciá-lo, procurar uma solução e expressá-la por meio de um algoritmo, com o respectivo fluxograma (BNCC, 2018).	**P3. Reconhecer e definir problemas computacionais**	
	3.1. Identificar problemas complexos, interdisciplinares e reais que podem ser resolvidos computacionalmente.	-
	3.2. Decompor problemas complexos do mundo real em subproblemas gerenciáveis que podem integrar soluções ou procedimentos existentes.	Modularidade
	3.3. Avaliar se é apropriado e viável resolver um problema computacionalmente.	Algoritmos

EM13MAT406. Utilizar os conceitos básicos de uma linguagem de programação na implementação de algoritmos escritos em linguagem corrente e/ou matemática (BNCC, 2018).	**P4. Desenvolver e usar abstrações**	
	4.1. Extrair características comuns de um conjunto de processos inter-relacionados ou fenômenos complexos.	Algoritmos, Variáveis e Modularidade
	4.2. Avaliar as funcionalidades tecnológicas existentes e incorporá-las em novos projetos.	Desenvolvimento de programas
	4.3. Criar módulos e desenvolver pontos de interação que possam se aplicar a várias situações e reduzir a complexidade.	-
	4.4. Modelar fenômenos e processos e simular sistemas para entender e avaliar os resultados potenciais.	Algoritmos e Variáveis
EM13MAT203. Planejar e executar ações envolvendo a criação e a utilização de aplicativos, jogos (digitais ou não), planilhas para o controle de orçamento familiar, simuladores de cálculos de juros compostos, dentre outros, para aplicar conceitos matemáticos e tomar decisões (BNCC, 2018).	**P5. Criar Artefatos Computacionais**	
	5.1. Planejar o desenvolvimento de um artefato computacional usando um processo iterativo que inclua a reflexão e a modificação do plano, levando em consideração os principais recursos, as restrições de tempo e recursos e as expectativas do usuário.	Variáveis, Controle e Desenvolvimento de programas
	5.2. Criar um artefato computacional para uso prática, expressão pessoal ou para tratar de uma questão social.	Algoritmos, Variáveis, Modularidade, Controle e Desenvolvimento de programas
	5.3. Modificar um artefato existente para aprimorá-lo ou personalizá-lo.	Modularidade
--	**P6. Testar e Refinar Artefatos Computacionais**	
	6.1. Testar sistematicamente os artefatos computacionais considerando todos os cenários e usando casos de teste.	Desenvolvimento de programas
	6.2. Identificar e corrigir erros usando um processo sistemático.	Desenvolvimento de programas
	6.3. Avaliar e refinar um artefato computacional várias vezes para aprimorar seu desempenho, confiabilidade, usabilidade e acessibilidade.	Algoritmos e Desenvolvimento de programas

Tipicamente, esses conceitos são ensinados por meio de unidades instrucionais que contêm atividades abertas de programação. Nessas atividades os alunos criam código e, a partir disso, é feita uma avaliação baseada no desempenho. No entanto, alguns conceitos podem ser difíceis de avaliar somente via código. Por essa razão, costumam ser avaliados por meio de outros artefatos. Assim, a avaliação da

aprendizagem dessas unidades instrucionais pode ocorrer de diversas formas, com base em diferentes artefatos, como:

- Prova: um documento com uma série de questões sobre um tema.
- Trabalho: um documento que deve ser elaborado pelo aluno sobre um tema.
- Entrevista: conversa entre professor e aluno, na qual o professor faz as perguntas e o aluno responde.
- Sistema de controle de versão: software que gerencia diferentes versões no desenvolvimento de um documento/software qualquer.
- Arquivos de log do ambiente de programação: documento que contém registro de eventos relevantes dentro de um ambiente de programação.

5.2 DEFINIÇÃO DO MODELO DE AVALIAÇÃO

O modelo é sistematicamente desenvolvido seguindo a abordagem GQM - *Goal, Question Metric* (BASILI; CALDIERA; ROMBACH, 1994). O modelo é definido de forma *top-down* (de cima para baixo) de modo que cada nível é detalhado do elemento mais abstrato até o mais concreto, até se chegar de forma explícita às medidas (BASILI; CALDIERA; ROMBACH, 1994). Desta forma, o modelo é descrito em três níveis:

(1) nível conceitual, em que é definido o objetivo,

(2) nível operacional, em que é definido um conjunto de questões para caracterizar a avaliação, e

(3) nível quantitativo, em que um conjunto de métricas é associado para cada questão de forma a prover uma resposta quantitativa.

O objetivo do modelo é avaliar o pensamento computacional a partir do código-fonte de um programa, em VPL baseada em blocos, criado como solução de uma atividade aberta (*ill-defined*) no contexto da Educação Básica.

A decomposição do nível operacional é baseada na definição de conceitos e práticas do *K–12 Computer Science Framework* (CSTA, 2016), *K-12 Computer Science Standards* (CSTA, 2017) e nos resultados do levantamento do estado da arte (ALVES et al., 2018). Os objetivos de aprendizagem do conceito de algoritmos e programação relacionados ao pensamento computacional são decompostos em

questões. No nível quantitativo, as métricas são descritas indicando a partir de **qual** artefato podem ser medidas e **como** podem ser medidas. Essa indicação é feita porque alguns dos objetivos de aprendizagem não são possíveis de medir via código. Assim, são sugeridas formas alternativas de avaliação ao código, de forma a prover uma avaliação mais completa.

Assim, a Tabela 27 apresenta as perguntas de análise (coluna 1) e os objetivos de aprendizagem relacionados a cada pergunta (coluna 2). A coluna 3 apresenta as métricas que podem ser medidas a partir de vários tipos de artefatos além do código, incluindo prova, trabalho, entrevista, sistema de controle de versão e arquivos de log do ambiente de programação.

Tabela 27 – Decomposição genérica do nível operacional e quantitativo.

Pergunta de análise	**Objetivos de aprendizagem**	**Métricas**
PA1. Qual o nível de desempenho em **algoritmos** com relação às práticas do pensamento computacional? (CSTA, 2016)	**AP1**: Comparar vários algoritmos (semelhanças/diferenças, tamanho, etc.) para uma mesma tarefa determinando qual é o mais adequado. (CSTA, 2017: 1B-AP-08[3])	**Prova/Trabalho**: comparação colocando problemas e solicitando a seleção de alternativas de solução (usando computador ou não)
	AP2: Resolver problemas de forma algorítmica usando uma abordagem apropriada (fluxograma, pseudocódigo, máquina de estado, etc.). (CSTA, 2017: 2-AP-10)	**Prova/Trabalho**: fluxogramas, pseudocódigo, máquina de estado
	AP3: Criar protótipos para resolver problemas computacionais. (CSTA, 2017: 3A-AP-13)	**Trabalho**: Protótipos baixa fidelidade.
	AP4: Incorporar operadores (aritméticos, relacionais, booleanos) em programas. (BRENNAN; RESNICK, 2012; GRESSE VON WANGENHEIM et al., 2018; GROVER; BASU; SCHANK, 2018)	**Código**: verificar se utilizou operadores aritméticos, booleanos e relacionais.
PA2. Qual o nível de desempenho em **representação de dados** com relação às práticas do pensamento computacional? (CSTA, 2016)	**AP5**: Criar programas com variáveis para armazenar, modificar dados e executar operações sobre seus valores. (CSTA, 2017: 1B-AP-09)	**Código**: verificar se criou ou modificou valores de variáveis. **Código**: verificar se criou ou modificou valores de *strings*.
	AP6: Criar variáveis com definição clara sem nomes genéricos ou *default* (CSTA, 2017: 2-AP-11)	**Código**: verificar se os nomes de variáveis são alterados do padrão.

[3]Referência aos objetivos de aprendizagem do *K-12 Computer Science Standards* (CSTA, 2017).

	AP7: Usar listas para simplificar soluções, evitando o uso de variáveis simples repetidamente. (CSTA, 2017: 3A-AP-14)	**Código**: verificar se são usadas listas.
	AP8: Incorporar persistência de dados em programas (SHERMAN et al., 2014; SHERMAN; MARTIN, 2015)	**Código**: verificar se são usados componentes de persistência de dados
PA3. Qual o nível de desempenho em **controle** com relação às práticas do pensamento computacional? (CSTA, 2016)	**AP9**: Criar e desenvolver programas que incluam eventos. (CSTA, 2017: 1B-AP-10)	**Código**: verificar se são usados eventos.
	AP10: Criar e desenvolver programas que incluam laços. (CSTA, 2017: 1B-AP-10; 2-AP-12)	**Código**: verificar se são usados laços.
	AP11: Criar e desenvolver programas que incluam condicionais. (CSTA, 2017: 1B-AP-10; 2-AP-12)	**Código**: verificar se são usados condicionais.
	AP12: Justificar a seleção de estruturas de controle específicas explicando os benefícios e desvantagens. (CSTA, 2017: 3A-AP-15)	**Prova/Entrevista:** questões abertas de escolha com alternativas de estruturas de controle.
	AP13: Incorporar paralelismo em programas (BRENNAN; RESNICK, 2012; GRESSE VON WANGENHEIM et al., 2018)	**Código**: verificar se são usados comandos de paralelismo.
	AP14: Incorporar sincronização em programas (BRENNAN; RESNICK, 2012; GRESSE VON WANGENHEIM et al., 2018)	**Código**: verificar se são usados comandos de sincronização.
PA4. Qual o nível de desempenho em **modularidade** com relação às práticas do pensamento computacional? (CSTA, 2016)	**AP15**: Decompor problemas em subproblemas para facilitar o desenvolvimento e revisão de programas (CSTA, 2017: 1B-AP-11; 2-AP-13)	**Código**: verificar se são criados procedimentos e funções.
	AP16: Modificar ou incorporar partes de um programa existente para desenvolver algo novo, personalizar ou adicionar recursos mais avançados. (CSTA, 2017: 1B-AP-12)	**Sistema de controle de versão**: evidências de versões com adição de código. **Sistema de controle de versão**: programa resultante de um remix com alterações
	AP17: Criar procedimentos para organizar o código e torná-lo mais fácil de reutilizar. (CSTA, 2017: 2-AP-14).	**Código**: verificar se são criados procedimentos e funções.
PA5. Qual o nível de desempenho em **desenvolvimento de programas** com relação às práticas do pensamento computacional? (CSTA,	**AP18**: Usar um processo iterativo para desenvolver um programa (CSTA, 2017: 1B-AP-13; 2-AP-12)	**Trabalho**: Evidências de iterações/projeto e desenvolvimento iterativo **Trabalho**: Análise de contexto (engenharia de usabilidade) /mapa de

2016)		empatia. **Trabalho**: *User stories* (requisitos). **Trabalho**: Protótipos. **Arquivos de log do ambiente de programação**: quantidade de vezes salvo, executado. **Sistema de controle de versão**: evolução de versões.
	AP19: Incorporar *feedback* dos usuários (CSTA, 2017: 2-AP-15; 3A-AP-19)	**Trabalho**: evidência de resultados de entrevistas com usuários ou qualquer outra fonte de solicitação dos usuários. **Trabalho**: relatório de testes de usabilidade.
	AP20: Incorporar código, mídia, biblioteca existente em programas originais e dar reconhecimento (CSTA, 2017: 2-AP-16)	**Código**: verificar se foram incluídas bibliotecas. **Trabalho**: documentação do código-fonte explicitando o código incorporado com o reconhecimento.
	AP21: Testar um programa (CSTA, 2017: 1B-AP-15; 2-AP-17)	**Trabalho**: Plano/casos de teste, estratégia de teste, roteiros e relatórios de teste. **Sistema de controle de versão**: evidências de problemas resolvidos em versões distintas.
	AP22: Depurar um programa (CSTA, 2017: CSTA, 2017: 1B-AP-15; 2-AP-17)	**Arquivos de log do ambiente de programação**: execução seguida por alterações no código.
	AP23: Avaliar artefatos computacionais para melhorar sua qualidade. CSTA, 2017: 3A-AP-21)	**Trabalho**: Relatório de avaliação de usabilidade (avaliação heurística, walkthrough, baseado em modelos etc.), inspeções de código, etc. **Trabalho**: Relatório de avaliação do código (linhas de código, comentários, duplicações, complexidade, etc.).
	AP24: Realizar tarefas de forma	**Sistema de controle de**

	colaborativa, seguindo um cronograma (CSTA, 2017: AB-AP-16; 2-AP-18; 3A-AP-22)	**versão**: proporção de código por usuário.
	AP25: Documentar a análise de requisitos (CSTA, 2017: 1B-AP-17; 2-AP-19)	**Trabalho**: Especificação de requisitos/*user stories*.
	AP26: Documentar o projeto e a implementação (CSTA, 2017: 1B-AP-17; 2-AP-19)	**Trabalho**: Especificação de requisitos/*user stories*, modelagem e planos e relatórios de teste.
PA6. Qual o nível de desempenho em **integração entre elementos da VPL** com relação às práticas do pensamento computacional?	**AP27**: Incorporar elementos da VPL em programas (GRESSE VON WANGENHEIM et al., 2018; SHERMAN et al., 2014; SHERMAN; MARTIN, 2015)	**Código**: verificar se foram inclusos elementos característicos da linguagem.

O modelo desenvolvido foi revisado via um painel de especialistas (BEECHAM et al., 2005) em relação a sua corretude, completude e consistência. O painel de especialistas foi composto por um grupo multidisciplinar de 8 pessoas com experiência em computação e/ou educação. De forma geral, os especialistas consideraram o modelo correto e completo. Não foram indicadas questões de inconsistência. Somente foram levantadas questões para alterações do texto. As sugestões dos especialistas, incluindo alterações e reformulações no texto, foram consideradas a fim de aprimorar o modelo.

5.2.1 Rubrica CodeMaster v2.0 App Inventor

A partir da decomposição genérica do modelo para os níveis operacional e quantitativo, o modelo é instanciado para a VPL App Inventor por meio da definição de uma rubrica. O foco da rubrica é somente a avaliação de conceitos que possuem métricas que podem ser coletadas com base no **código**. A decomposição do nível operacional e quantitativo são especificadas considerando-se as particularidades dessa linguagem de programação voltada a criação de aplicativos móveis. A rubrica é criada a partir da derivação do modelo de medição (seção 5.2) e com base na rubrica CodeMaster v1.0 para App Inventor.

Levando em consideração as questões em relação aos itens Mídia, Social e Conectividade da rubrica CodeMaster v1.0 para App Inventor (seção 4.2), esses itens são analisados sob a perspectiva da sua qualidade. Analisando os indicadores, os referidos itens apresentaram resultados fracos, seja no aumento do alfa de Cronbach caso sejam

retirados da rubrica ou na correlação com outros itens (conforme apresentado na Tabela 23), além do baixo fator de carregamento para um fator (conforme apresentado na Tabela 25). Após uma análise, decidiu-se pela sua exclusão da rubrica evoluída CodeMaster v2.0. O item Interface de Usuário também é excluído devido à sua formulação não estar incluída explicitamente na definição de pensamento computacional do CSTA (2016). Porém, representando um aspecto importante no desenvolvimento de aplicativos, visa-se como trabalho futuro a criação de uma rubrica voltada especificamente ao design de interface de usuário. Além disso, o item Operadores é reformulado. Todos os demais itens presentes na rubrica CodeMaster v1.0 para App Inventor são adaptados de acordo com os objetivos de aprendizagem do CSTA (2017).

A partir da decomposição do modelo genérico, também são criados 5 novos itens: Variáveis, *Strings*, Sincronização, Extensões e Mapas. Esses itens são criados para atender os objetivos de aprendizagem relacionados às práticas do pensamento computacional conforme definidos pelo CSTA (2017). Somente um objetivo de aprendizagem (AP13 – coluna 2, Tabela 27), o qual possui uma métrica definida possível de ser coletada por meio do código no modelo genérico, não foi possível atender. Isso é devido à característica da VPL App Inventor não permitir o paralelismo na linguagem (TURBAK et al., 2014) e, portanto, impossibilitando a avaliação desse item. No total, a rubrica CodeMaster v2.0 para o App Inventor contém 16 itens.

Para cada item são definidos os níveis de desempenho usando escalas ordinais variando de 0 pontos a 3 pontos. Os níveis de desempenho são definidos com base na rubrica CodeMaster v1.0 (GRESSE VON WANGENHEIM et al., 2018). Cabe ressaltar que não necessariamente todos os itens possuem descritores para pontuações 2 e 3, esses itens são definidos usando pontuação dicotômica ou escala ordinal variando de 0 a 2 pontos (Tabela 28).

Tabela 28 - Rubrica CodeMaster v2.0 para App Inventor.

PA1. Qual o nível de desempenho em algoritmos com relação às práticas do pensamento computacional? (CSTA, 2016; 2017)				
Item	**0 pontos**	**1 ponto**	**2 pontos**	**3 pontos**
Operadores: verificar se utilizou operadores aritméticos, booleanos e relacionais.	Não usa operadores	Usa operadores aritméticos.	Usa operadores relacionais.	Usa operadores booleanos (lógicos).

PA2. Qual o nível de desempenho em representação de dados com relação às práticas do pensamento computacional? (CSTA, 2016; 2017)				
Item	**0 pontos**	**1 ponto**	**2 pontos**	**3 pontos**
Variáveis: verificar se criou ou modificou valores de variáveis.	Sem uso de variáveis.	Modificação ou uso de variáveis predefinidas.	Criação e operação com variáveis	
Strings: verificar se criou ou modificou valores de strings.	Sem uso de strings.	Uso do comando de criação de string para alterar textos de elementos.	Criação e operação com strings.	
Nomeação: verificar se os nomes de variáveis são alterados do padrão.	Nenhum ou poucos nomes são alterado do padrão. (menos do que 10%)	De 10 a 25% dos nomes são alterados do padrão.	De 26 a 75% dos nomes são alterados do padrão.	Mais de 76% dos nomes são alterados do padrão.
Listas: verificar se são usadas listas.	Não usa listas.	Usa uma lista unidimensional.	Usa mais de uma lista unidimensional.	Usa uma lista de tuplas (map).
Persistência de dados: verificar se são usados componentes de persistência de dados	Dados são armazenados em variáveis ou propriedades de componentes e não tem persistência quando app é fechado.	Usa persistência em arquivo (File ou Fusion Tables).	Usa algum dos bancos de dados locais do App Inventor (TinyDB).	Usa uma base de dados web tinywebdb ou Firebase do App Inventor (firebase ou TinyWebDB).
PA3. Qual o nível de desempenho em controle com relação às práticas do pensamento computacional? (CSTA, 2016; 2017)				
Item	**0 pontos**	**1 ponto**	**2 pontos**	**3 pontos**
Eventos: verificar se são usados eventos.	Nenhum manipulador de evento é usado (ex. On click).	1 tipo de manipuladores de eventos é usado.	2 tipos de manipuladores de eventos são usados.	Mais de 2 tipos de manipuladores de eventos são usados.
Laços: verificar se são usados laços.	Não usa laços	Usa “While” (laço simples)	Usa “For each” (variável simples)	Usa “For each” (item de lista)
Condicionais: verificar se são usados condicionais.	Não usa condicionais.	Usa apenas “if’s” simples.	Usa apenas “if then else”.	Usa um ou mais “if - else if”.

Sincronização: verificar se são usados comandos de sincronização.	Sem uso de temporizador para sincronização.	Uso de comando de temporizador para sincronização.		
PA4. Qual o nível de desempenho em modularidade com relação às práticas do pensamento computacional? (CSTA, 2016; 2017)				
Item	**0 pontos**	**1 ponto**	**2 pontos**	**3 pontos**
Abstração de procedimentos: verificar se são criados procedimentos e funções.	Não existem procedimentos.	Existe exatamente um procedimento e sua chamada.	Existe mais de um procedimento.	Existem procedimentos tanto para organização quanto para reuso. (Mais chamadas de procedimentos do que procedimentos).
PA5. Qual o nível de desempenho em desenvolvimento de programas com relação às práticas do pensamento computacional? (CSTA, 2016; 2017)				
Item	**0 pontos**	**1 ponto**	**2 pontos**	**3 pontos**
Extensões: verificar se foram incluídas bibliotecas.	Sem uso de comandos de extensões.	Uso de comandos de extensões.		
PA6. Qual o nível de desempenho em integração entre elementos da linguagem com relação às práticas do pensamento computacional?				
Item	**0 pontos**	**1 ponto**	**2 pontos**	**3 pontos**
Sensores	Sem uso de sensores.	Usa um tipo de sensor.	Usa 2 tipos de sensores.	Usa mais de 2 tipos de sensores.
Desenho e Animação	Sem uso de desenho e animação.	Uso de área sensível ao toque.	Uso de animação com bolinha predefinida.	Uso de animação com imagem.
Mapas	Sem uso de comandos de mapas.	Uso do comando de mapa.	Uso de comandos de marcadores de mapas.	
Telas	Apenas uma tela com componentes visuais e que seu estado não se altera com a execução do app (tela informativa).	Apenas uma tela com componentes visuais que se alteram com a execução do app.	Pelo menos duas telas e uma delas altera seu estado com a execução do app.	Duas ou mais telas e pelo menos 2 delas alteram seus estados com a execução do app.

5.2.2 Cálculo da nota e *feedback* instrucional

Com base nas métricas da Tabela 28, o cálculo da nota final da avaliação é feito com base na Teoria Clássica dos Testes, isto é, o resultado observado no teste é resultante da soma do resultado verdadeiro (DEVELLIS, 2006). A nota é definida numa escala de 0 a 10, alinhado à escala de notas tipicamente utilizada nas escolas Brasileiras de Educação Básica. A fórmula 1 apresenta o cálculo realizado para se obter a nota final.

Fórmula 1. Cálculo da nota.

$$Nota = \frac{\sum Pontuação\ recebida\ por\ item}{\sum Pontuação\ máxima\ por\ item} * 10$$

A atribuição de nota é realizada usando o valor da soma das pontuações de cada item, os quais possuem nomeadamente uma pontuação máxima de acordo com os descritores verbais definidos na Tabela 28. A partir da soma de todas as pontuações recebidas por item, o valor é ajustado para a escala de pontuação [0, 10]. A Tabela 29 apresenta um exemplo de atribuição de nota a um projeto.

Tabela 29 - Exemplificação de atribuição de nota pela rubrica CodeMaster v2.0 para AppInventor.

Item	**Pontuação recebida**	**Pontuação máxima**
I01. Operadores	**3**	3
I02. Variáveis	**2**	2
I03. Strings	**2**	2
I04. Nomeação	1	3
I05. Listas	0	3
I06. Persistência	1	3
I07. Eventos	2	3
I08. Laços	**3**	3
I09. Condicionais	1	3
I10. Sincronização	**1**	1
I11. Abstração	2	3
I12. Sensores	1	3
I13. Extensões	**2**	2
I14. Desenho e Animação	**3**	3
I15. Mapas	**2**	2
I16. Telas	**3**	3
Total	30	41
Nota	**7,31**	10

O *feedback* instrucional é apresentado ao aluno para cada item avaliado, indicando qual a pontuação recebida por item (Figura 26). A partir da pontuação obtida por item e a pontuação máxima possível de se obter por item, o aluno pode consultar a rubrica que fica disponibilizada no mesmo ambiente de avaliação. A partir da informação presente na rubrica o aluno pode melhorar seu nível de desempenho e reavaliar seu projeto se assim desejar.

Figura 26 - Apresentação do *feedback* ao aluno.

Fonte: elaborada pela autora com base na implementação do CodeMaster v2.0.

Da mesma forma, o professor, ao utilizar o modelo para avaliar aplicativos de seus alunos, pode consultar a rubrica e apresentar explicações e sugestões de forma transparente sobre como o aluno pode melhorar seu desempenho e, consequentemente, sua nota.

5.3 DESENVOLVIMENTO DA AUTOMATIZAÇÃO DA AVALIAÇÃO

Com base na definição da rubrica CodeMaster v2.0 para App Inventor, o modelo é implementado evoluindo a ferramenta CodeMaster v1.0 (DEMETRIO, 2017), seguindo um processo iterativo e incremental (LARMAN; BASILI, 2003) incluindo a análise de requisitos, modelagem, construção de software e testes.

5.3.1 Análise de requisitos

Evoluindo a ferramenta CodeMaster v1.0 para v2.0, são mantidos todos os requisitos originalmente definidos com algumas alterações (DEMETRIO, 2017). São modificados somente os requisitos: Avaliar

Projeto App Inventor (RF1), Apresentar Avaliação do projeto App Inventor de forma detalhada (RF2), Apresentar Avaliação dos projetos App Inventor de forma resumida (RF3).

Tabela 30 - Requisitos elicitados para o CodeMaster v2.0.

ID	Requisito	Descrição	Artefato Entrada	Artefato Saída
RF1	Avaliar Projeto App Inventor	A ferramenta deve avaliar cada item apresentado na Tabela 28 de acordo com os elementos especificados encontrados no projeto App Inventor e gerar uma nota para cada conceito	Listas de cada tipo de elemento do código.	Uma resposta de avaliação formada pela nota para cada conceito, um nível de competência e um feedback.
RF2	Apresentar Avaliação do projeto App Inventor de forma detalhada	A ferramenta deve apresentar na interface de usuário a avaliação detalhada do seu projeto de acordo com a rubrica apresentada na Tabela 28 Deve apresentar a nota de cada conceito separadamente, bem como a nota final e o nível de competência do aluno e o feedback	Uma avaliação formada pela nota para cada conceito	Interface de usuário exibe a avaliação. Notas de cada conceito, nota final e nível de competência.
RF3	Apresentar Avaliação dos projetos App Inventor de forma resumida	A ferramenta deve apresentar na interface de usuário a avaliação de um conjunto de projetos de forma resumida ao professor de acordo com a rubrica apresentada na Tabela 28.	Um conjunto de avaliações cada uma formada pela nota para cada conceito.	Interface de usuário exibe em tabela a nota de cada conceito, nota final e nível para cada avaliação do conjunto, Além de uma média de todas as notas finais de cada aluno e o total dos projetos submetidos a análise.

Todos os requisitos não-funcionais e casos de uso originalmente definidos no CodeMaster v1.0 (DEMETRIO, 2017) são mantidos.

5.3.2 Modelagem e implementação

A evolução do CodeMaster v1.0 é feita usando a linguagem de programação Java com JSP (linguagem Java para desenvolvimento Web) e o módulo de apresentação utiliza as tecnologias JSP, Java Script, HTML5 e CSS3. Essa escolha é feita por serem linguagens e tecnologias já utilizadas no desenvolvimento da versão 1.0.

Para a **implementação dos novos itens**, no módulo de apresentação são adicionados mais 5 *checkboxs* (Variáveis, Strings, Sincronização, Extensões e Mapas) na tela de professor que contém os critérios que se deseja analisar. Nas telas de resultado do aluno são adicionadas 5 novas linhas para os novos itens e do professor são adicionadas 5 novas colunas. A classe ProjectSettings recebe 5 novas variáveis booleanas para os novos itens. A classe Upload é evoluída de forma a obter o valor das variáveis vindo da requisição do usuário. A classe AppInventorGrade recebe 5 novas variáveis que guardam a pontuação dos itens.

No módulo de análise e avaliação, o ProjectSettings e AppInventorGrade são alteradas da mesma maneira que as respectivas classes do módulo de apresentação. São criadas 5 novas classes que estendem a classe ConceitoCT e implementam o método avaliaCódigo. Por herança, esses novos itens, recebem a instância de código, com todas as estatísticas e estruturas de dados de todos os códigos de telas do projeto App Inventor. A classe AppInventorGrader é evoluída de forma a obter a pontuação dos novos itens, inserir a pontuação na instância de AppInventorGrade e somá-la na variável de pontuação total.

Para a **exclusão dos itens existentes no CodeMaster v1.0**, no módulo de apresentação são excluídos os 4 *checkboxs* referentes aos itens Mídia, Social, Conectividade e Interface de Usuário. São excluídas das telas de resultado do aluno as linhas e do professor as colunas que apresentam esses itens. A classe ProjectSettings deixa de ter 4 variáveis booleanas referentes a esses itens. A classe Upload deixa de obter o valor das respectivas variáveis vindo da requisição do usuário. A classe AppInventorGrade deixa de ter 4 variáveis que guardam a pontuação dos itens.

No módulo de análise e avaliação, o ProjectSettings e AppInventorGrade recebem as mesmas alterações que as respectivas classes do módulo de apresentação. São excluídas as 4 classes que estendem a classe ConceitoCT referentes aos itens. A classe AppInventorGrader deixa de obter a pontuação desses itens, inserir essa

pontuação na instância de AppInventorGrade e somá-la na variável de pontuação total.

Para a **alteração do item existente**, somente é alterada a classe Operadores.

5.3.3 Interface de Usuário

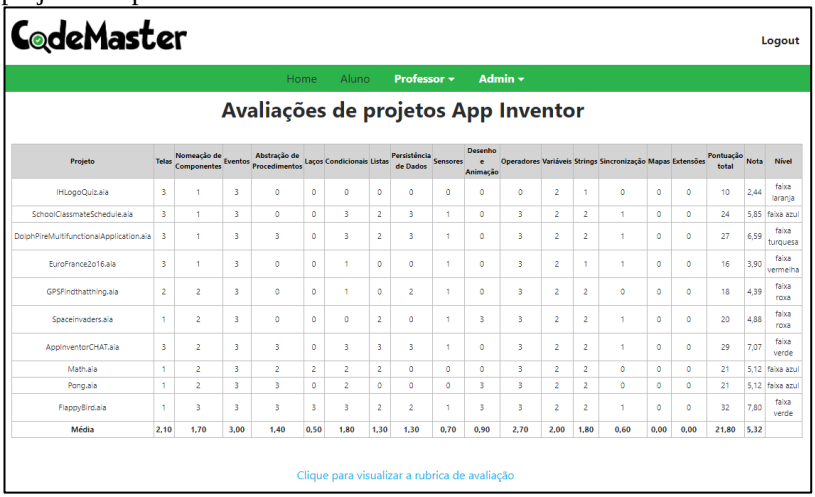
A interface do CodeMaster v2.0 segue o design projetado na versão v1.0. São alteradas as telas de apresentação de resultados tanto do aluno como do professor. A tela que apresenta o resultado da avaliação para o aluno, referente ao RF2 apresentado na Tabela 30, é apresentada na Figura 27.

Figura 27 - Interface apresentando a avaliação de um projeto ao aluno.

Fonte: elaborada pela autora com base na implementação do CodeMaster v2.0.

A tela que apresenta o resultado da avaliação de um conjunto de projetos para o professor, referente ao RF3 apresentado na Tabela 30, é apresentada na Figura 28.

Figura 28 - Interface apresentando a avaliação de um conjunto de projetos ao professor.



Projeto	Telas	Nomeação de Componentes	Eventos	Abstração de Procedimentos	Laços	Condicionais	Listas	Persistência de Dados	Sensores	Desenho e Animação	Operadores	Variáveis	Strings	Sincronização	Mapas	Extensões	Pontuação total	Nota	Nível
IHLogoQuiz.aia	3	1	3	0	0	0	0	0	0	0	0	2	1	0	0	0	10	2,44	faixa laranja
SchoolClassmateSchedule.aia	3	1	3	0	0	3	2	3	1	0	3	2	2	1	0	0	24	5,85	faixa azul
DolphPireMultifunctionalApplication.aia	3	1	3	3	0	3	2	3	1	0	3	2	2	1	0	0	27	6,59	faixa turquesa
EuroFrance2016.aia	3	1	3	0	0	1	0	0	1	0	3	2	1	1	0	0	16	3,90	faixa vermelha
GPSFindthatthing.aia	2	2	3	0	0	1	0	2	1	0	3	2	2	0	0	0	18	4,39	faixa roxa
SpaceInvaders.aia	1	2	3	0	0	0	2	0	1	3	3	2	2	1	0	0	20	4,88	faixa roxa
AppInventorCHAT.aia	3	2	3	3	0	3	3	3	1	0	3	2	2	1	0	0	29	7,07	faixa verde
Math.aia	1	2	3	2	2	2	2	0	0	0	3	2	2	0	0	0	21	5,12	faixa azul
Pong.aia	1	2	3	3	0	2	0	0	0	3	3	2	2	0	0	0	21	5,12	faixa azul
FlappyBird.aia	1	3	3	3	3	3	2	2	1	3	3	2	2	1	0	0	32	7,80	faixa verde
Média	**2,10**	**1,70**	**3,00**	**1,40**	**0,50**	**1,80**	**1,30**	**1,30**	**0,70**	**0,90**	**2,70**	**2,00**	**1,80**	**0,60**	**0,00**	**0,00**	**21,80**	**5,32**	

Fonte: elaborada pela autora com base na implementação do CodeMaster v2.0.

5.3.4 Testes

Foram realizados vários testes voltados à verificação da corretude da ferramenta. Esta verificação foi feita a partir da comparação entre o resultado da avaliação automatizada do projeto, gerada pelo CodeMaster v2.0, e a avaliação manual do projeto feita com base na rubrica apresentada na Tabela 28.

Para executar a avaliação da corretude, foram selecionados aleatoriamente 10 projetos da seção de aplicativos populares da Galeria App Inventor. A comparação entre a avaliação manual e automatizada dos projetos selecionados são apresentados na Tabela 31. Além dessa avaliação ao longo do desenvolvimento do CodeMaster v2.0, vários testes foram feitos com diversos projetos App Inventor. Os erros foram corrigidos, e os resultados mostraram-se consistentes conforme a definição da avaliação apresentada na rubrica na Tabela 28.

Tabela 31 - Comparação entre testes manuais e testes no CodeMaster v2.0

Aplicativo	**Avaliação M (Manual) A (Automatizada)**	**Operadores (I01)**	**Variáveis (I02)**	**Strings (I03)**	**Nomeação (I04)**	**Listas (I05)**	**Persistência (I06)**	**Eventos (I07)**	**Laços (I08)**	**Condicionais (I09)**	**Sincronização (I10)**	**Abstração (I11)**	**Sensores (I12)**	**Extensões (I13)**	**Desenho e Animação (I14)**	**Mapas (I15)**	**Telas (I16)**
AppInventorCHAT	M	3	2	2	2	3	3	3	0	3	1	3	1	0	0	0	3
	A	3	2	2	2	3	3	3	0	3	1	3	1	0	0	0	3
DolphPireMultifunctionalApplication	M	3	2	2	1	2	3	3	0	3	1	3	1	0	0	0	3
	A	3	2	2	1	2	3	3	0	3	1	3	1	0	0	0	3
EuroFrance2o16	M	3	2	**1**	1	0	0	3	0	1	1	0	1	0	0	0	3
	A	3	2	**2**	1	0	0	3	0	1	1	0	1	0	0	0	3
FlappyBird	M	3	2	2	3	2	2	3	3	3	1	3	1	0	3	0	1
	A	3	2	2	3	2	2	3	3	3	1	3	1	0	3	0	1
GPSFindthatthing	M	3	2	2	2	0	2	3	0	1	0	0	1	0	0	0	2
	A	3	2	2	2	0	2	3	0	1	0	0	1	0	0	0	2
IHLogoQuiz	M	0	2	**1**	1	0	0	3	0	0	0	0	0	0	0	0	3
	A	0	2	**2**	1	0	0	3	0	0	0	0	0	0	0	0	3
Math	M	3	2	2	2	2	0	3	2	2	0	2	0	0	0	0	1
	A	3	2	2	2	2	0	3	2	2	0	2	0	0	0	0	1
Pong	M	3	2	2	1	0	0	3	0	2	0	3	0	0	3	0	1
	A	3	2	2	1	0	0	3	0	2	0	3	0	0	3	0	1
SchoolClassmateSchedule	M	3	2	2	1	2	3	3	0	3	1	0	1	0	0	0	3
	A	3	2	2	1	2	3	3	0	3	1	0	1	0	0	0	3
Spaceinvaders	M	3	2	2	2	2	0	3	0	0	1	0	1	0	3	0	1
	A	3	2	2	2	2	0	3	0	0	1	0	1	0	3	0	1

A partir desses resultados, foi corrigido um erro encontrado na avaliação do item Strings. Os projetos foram submetidos novamente ao CodeMaster v2.0 tendo recebidos a pontuação correta. Assim, é possível verificar a corretude do CodeMaster v2.0 para todos os projetos selecionados. Após a correção, todos projetos obtiveram as pontuações esperadas para cada item analisado.

6 AVALIAÇÃO

Este capítulo apresenta a avaliação do modelo por meio de um estudo de caso avaliando a instância do modelo para a rubrica CodeMaster v2.0 para App Inventor. A avaliação é feita em termos de confiabilidade e validade de construto. A confiabilidade refere-se ao grau de consistência dos itens do modelo, pode ser medida por meio do coeficiente alfa de Cronbach (CRONBACH, 1951). A validade de um construto é a capacidade de medir de fato o que se propõe a medir, incluindo a validade convergente, que é medida pelo grau de correlação entre os itens (CARMINES; ZELLER, 1982; TROCHIM; DONNELLY, 2008).

6.1 DEFINIÇÃO E COLETA DE DADOS

O objetivo deste estudo é analisar o modelo por meio da instanciação da rubrica CodeMaster v2.0 para App Inventor. A qualidade do modelo é avaliada em termos de confiabilidade e validade da rubrica do ponto de vista de pesquisadores no contexto do ensino de computação na Educação Básica. Seguindo a abordagem GQM (BASILI; CALDIERA; ROMBACH, 1994) o objetivo do estudo é definido e decomposto em aspectos de qualidade e questões de análise a serem analisadas com base nos dados coletados.

Confiabilidade

AQ1: Existe evidência de consistência interna da rubrica CodeMaster v2.0 para App Inventor?

Validade

AQ2: Existe evidência de validade convergente da rubrica CodeMaster v2.0 para App Inventor?

AQ3: Como os fatores subjacentes influenciam as respostas nos critérios da rubrica CodeMaster v2.0 para App Inventor?

AQ4: Existe evidência de validade das categorias dos itens da rubrica CodeMaster v2.0?

Coleta de dados. Para maximizar o tamanho da amostra, foi realizado o *download* de todos os aplicativos públicos disponíveis e acessíveis da Galeria do App Inventor até maio de 2018, armazenando seus arquivos *.aia*. (exatamente da mesma forma apresentada no capítulo 4). Um arquivo *.aia* é uma coleção compactada de arquivos que inclui um arquivo de propriedades do projeto, arquivos de mídia usados pelo aplicativo e, para cada tela do aplicativo, dois arquivos principais: um arquivo *.bky* e um arquivo *.scm*. O arquivo *.bky* encapsula uma

estrutura XML com todos os blocos de programação usados na programação do aplicativo, e o arquivo *.scm* encapsula uma estrutura JSON que contém todos os componentes visuais usados no aplicativo (TURBAK et al., 2017). Como resultado, foi obtido o código-fonte de 88.864 aplicativos App Inventor.

Avaliação e classificação dos aplicativos App Inventor. Os aplicativos obtidos foram analisados usando a ferramenta CodeMaster v2.0. Dos 88.864 projetos baixados, 88.812 foram analisados com sucesso com o CodeMaster v2.0. Devido a dificuldades técnicas, 52 aplicativos App Inventor não puderam ser analisados. Os resultados da avaliação foram exportados em um arquivo SQL e, em seguida, convertidos em um arquivo CSV. Os dados coletados foram agrupados em uma única tabela para validar o modelo.

Considerou-se a obrigatoriedade de todos os itens para todos os projetos. A Tabela 32 apresenta a frequência da pontuação por itens. Cabe ressaltar que nem todos os itens possuem descritores verbais para pontuação 2, como os itens 10 (Sincronização) e 13 (Extensões) e pontuação 3, como os itens 2 (Variáveis), 3 (Strings), 10 (Sincronização), 13 (Extensões) e 15 (Mapas).

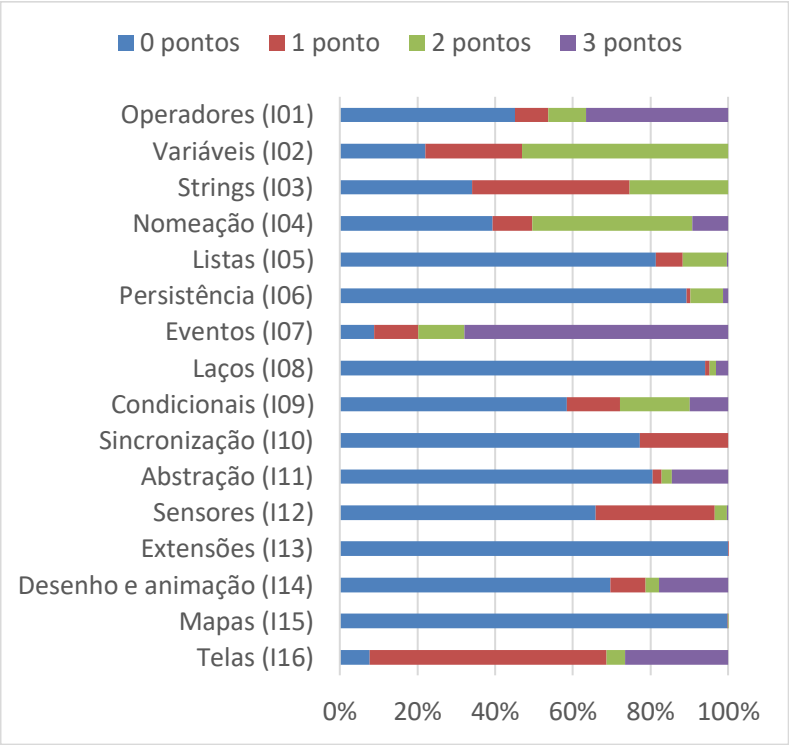
Tabela 32 - Frequência de pontuação por item na rubrica CodeMaster v2.0 para App Inventor.

Item	**0**	**1**	**2**	**3**
I01. Operadores	40003	7659	8678	32472
I02. Variáveis	19616	22109	47087	-
I03. Strings	30241	36118	22453	-
I04. Nomeação	34905	9073	36656	8178
I05. Listas	72228	6154	10166	264
I06. Persistência	79307	857	7490	1158
I07. Eventos	7798	10171	10470	60373
I08. Laços	83546	1066	1456	2744
I09. Condicionais	51938	12189	15960	8725
I10. Sincronização	68574	20238	-	-
I11. Abstração	71501	2166	2344	12801
I12. Sensores	58481	27351	2703	277
I13. Extensões	88812	**0**	-	-
I14. Desenho e Animação	61929	7957	3130	15796
I15. Mapas	88740	**36**	**36**	-
I16. Telas	6855	54211	4216	23530

É importante destacar que alguns itens têm poucos dados em algumas pontuações, como o item 15 (Mapas) para as pontuações 1 e 2 e o item 13 (Extensões) para a pontuação 1 (marcados em negrito na Tabela 32). A baixa porcentagem nas pontuações acima de 0 no item

Mapas (Figura 29) pode ser explicada pela razão de este ser um recurso novo, adicionado há pouco tempo no App Inventor. Assim, para muitos dos aplicativos mais antigos da base de dados, no momento em que foram criados, não havia ainda esse recurso disponível para usar. O item 13 (Extensões) não apresenta nenhum aplicativo com pontuação acima de 0 dentro de um total de 88.812 aplicativos pois a Galeria App Inventor não permite a publicação de aplicativos com extensão. A Figura 29 apresenta as porcentagens de pontuação por itens por meio de um gráfico de barras empilhadas.

Figura 29 - Porcentagem de pontuação por item na rubrica CodeMaster v2.0.



Fonte: elaborada pela autora.

Levando em consideração a frequência nula do item 13 (Extensões), optou-se por excluí-lo da análise. O item 15 (Mapas) foi mantido, ainda que com baixa frequência, porém considerando que se trata de um recurso recentemente adicionado ao App Inventor.

6.2 ANÁLISE DOS DADOS

A partir da Teoria Clássica dos Testes (TCT) (PASQUALI, 1997), são avaliadas as duas principais propriedades psicométricas do modelo: a confiabilidade e a validade. Para avaliar a confiabilidade, relacionada à reprodutibilidade da medida, é utilizado o Alfa de Cronbach (1951). Para avaliar a validade, relacionada à capacidade do modelo medir efetivamente um conceito teórico específico, neste caso, o pensamento computacional, são calculadas as correlações (DEVELLIS, 2003) e utilizada a análise fatorial (GLORFELD, 1995).

Além da análise pela TCT que possui várias limitações, também é feita uma análise pela Teoria da Resposta ao Item (TRI), a teoria da psicometria moderna. Diferente da TCT em que um número de itens contribui positivamente nos resultados, a TRI propõe o uso de escalas mais curtas conservando as propriedades psicométricas do instrumento (SARTES; SOUZA-FORMIGONI, 2013). A TRI parte da suposição de que existe um traço latente a ser medido, neste caso, o pensamento computacional. A análise via TRI é feita utilizando o Modelo de Resposta Gradual (SAMEJIMA, 1969). Os dados coletados foram analisados usando a linguagem de programação R.

AQ1. Existe evidência de consistência interna da rubrica CodeMaster v2.0?

Foi analisada a confiabilidade medindo a consistência interna da rubrica CodeMaster v2.0 para App Inventor por meio do coeficiente alfa de Cronbach (1951). Valores de alfa de Cronbach entre $0,7 < \alpha \leq 0,8$ são aceitáveis, entre $0,8 < \alpha \leq 0,9$ são bons e $\alpha \geq 0,9$ são excelentes (DEVELLIS, 2003), indicando assim uma consistência interna do instrumento. Analisando os 15 itens da rubrica CodeMaster v2.0 (com o item Extensões já excluído desta análise), obteve-se um valor bom do alfa de Cronbach ($\alpha = 0,84$) e acima do valor alfa de Cronbach da rubrica CodeMaster v1.0 ($\alpha = 0,79$).

AQ2. Existe evidência de validade convergente da rubrica CodeMaster v2.0?

Para obter evidências da validade convergente dos itens da rubrica CodeMaster v2.0 para App Inventor, calculam-se as intercorrelações dos itens e correlação total do item (DEVELLIS, 2003). Para a validade convergente, espera-se que os critérios da mesma subdimensão tenham uma correlação mais alta (CARMINES; ZELLER, 1982; TROCHIM; DONNELLY, 2008).

Para analisar as intercorrelações entre os itens de uma mesma subdimensão, utilizou-se a correlação policórica, sendo a mais apropriada para variáveis ordinais observadas (OLSSON, 1979). A matriz mostra o coeficiente de correlação, indicando o grau de correlação entre dois itens ordinais (pares de itens). De acordo com Cohen (1998), uma correlação entre os itens é considerada satisfatória, se o coeficiente de correlação for maior que 0,29, indicando que há uma correlação média ou alta entre os itens. Os resultados da análise são apresentados na Tabela 33 com correlações satisfatórias marcadas em negrito, já sem o item 13 (Extensões).

Apresentados em azul (Tabela 33), estão os itens relacionados ao pensamento computacional conforme definido pelo CSTA (2016; 2017). São incluídos itens referentes à **algoritmos** (PA1), **representação de dados** (PA2), **controle** (PA3) e **modularidade** (PA4), conforme definidos na Tabela 28 – seção 5.2.1. O subconceito de **desenvolvimento de programas** (PA5), não foi possível analisar devido à exclusão do item 13 (Extensões) da análise, sendo este item a métrica de avaliação via código para este subconceito (Tabela 28 – seção 5.2.1). O item foi excluído pela razão de a Galeria App Inventor não permitir aplicativos com extensões. Assim, esse subconceito é avaliado apenas por outros artefatos além do código e, portanto, não é possível fazer a análise de correlação. Itens em vermelho são relacionados à **integração entre elementos da VPL** (PA6 definida na Tabela 28 – seção 5.2.1). Nesse caso, estão inclusos itens especificamente da VPL App Inventor, relacionados ao pensamento computacional móvel.

Tabela 33 - Coeficiente de correlação policórica da rubrica CodeMaster v2.0 para App Inventor.

Item	1	2	3	4	5	6	7	8	9	10	11	12	14	15	16
1	1,0														
2	**,769**	1,0													
3	**,510**	**,541**	1,0												
4	**,584**	**,578**	**,464**	1,0											
5	**,390**	**,553**	**,498**	**,372**	1,0										
6	**,502**	**,528**	**,636**	**,323**	**,446**	1,0									
7	**,717**	**,687**	**,618**	**,634**	**,335**	**,591**	1,0								
8	**,455**	**,550**	**,488**	**,395**	**,696**	**,427**	,257	1,0							
9	**,757**	**,641**	**,570**	**,508**	**,473**	**,468**	**,589**	**,476**	1,0						
10	**,700**	**,609**	**,431**	**,482**	,214	,289	**,671**	**,291**	**,529**	1,0					
11	**,638**	**,747**	**,603**	**,639**	**,428**	**,331**	**,614**	**,583**	**,610**	**,659**	1,0				
12	**,503**	**,517**	**,351**	**,296**	,193	,278	**,483**	,209	**,375**	**,906**	**,486**	1,0			
14	**,500**	**,662**	,093	**,399**	-,033	-,046	**,501**	,081	,227	**,634**	**,559**	**,472**	1,0		
15	,112	,140	,161	,051	,138	,267	,106	,121	,068	,014	,027	,238	-,334	1,0	
16	**,364**	**,337**	**,533**	,257	,289	**,439**	**,754**	,233	**,354**	,274	,195	,203	,099	,260	1,0

O item 1 (Operadores), referente à desempenho em **algoritmos** apresenta alta correlação com praticamente todos os itens da rubrica, conforme esperado. Os itens referentes à desempenho em **representação de dados** (itens 2, 3, 4 e 5), apresentam correlação boa, pois não foram detectados valores abaixo de 0,29 para nenhum par.

Referente à desempenho em **controle**, praticamente todos os itens (6, 7, 8, 9 e 10) também apresentam boa correlação. Contudo, o par de itens 7 e 8 (Eventos e Laços) apresenta um valor um pouco abaixo de 0,29. Isso pode ser explicado pelo motivo que o App Inventor, sendo uma VPL orientada a eventos, favorece o uso desse conceito em detrimento ao uso de laços (TURBAK et al., 2014). Além disso, outros dois pares de itens 5 e 10 (Listas e Sincronização) e 6 e 10 (Telas e Sincronização) também apresentam valores abaixo de 0,29.

O conceito de **integração entre elementos da linguagem** é o que apresenta resultados mais isolados. Nota-se que apenas um par (item 12 e 14) apresenta correlação acima de 0,29. O fato de quase todos os itens não apresentarem correlação acima de 0,29 pode ser entendido pela razão de que aplicativos móveis, tipicamente, têm algum foco especifico, e assim, não necessitam de vários recursos da linguagem para serem úteis. Por exemplo, um bom aplicativo não precisa usar GPS, Bluetooth, Twitter, câmera, etc. Observa-se, uma forte correlação negativa entre os pares de itens Desenho e animação (item 14) e Mapas (item 15). Cabe ressaltar, no entanto, que as correlações do item Mapas podem estar representadas com ruído, haja vista trata-se de um recurso novo adicionado ao App Inventor e fracamente representado no conjunto de dados que contém aplicativos mais antigos.

Correlação item-total.

Complementando a análise anterior, a correlação entre todos os outros itens é analisada. Cada item do instrumento deve ter uma correlação média ou alta com todos os outros itens (DEVELLIS, 2003), pois isso indica que os itens apresentam consistência em comparação com os outros itens. Por outro lado, uma baixa correlação item-total de um item enfraquece a validade da rubrica e, portanto, deve ser eliminado. Para esta análise, utilizou-se o método da correlação item-total corrigida. Os valores de referência para a análise são os mesmos apresentados na seção anterior, considerando uma correlação satisfatória, se o coeficiente de correlação for maior que 0,29 (COHEN, 1998). Além disso, é analisado como fica o alfa de Cronbach se um item for excluído, espera-se que nenhum item cause um aumento substancial no alfa de Cronbach (DEVELLIS, 2003), assim, indicando que todos os itens contribuem para a validade da rubrica. Os resultados desta análise

são apresentados na Tabela 34, mostrando o valor da correlação item-total, bem como o valor do alfa de Cronbach (CRONBACH, 1951), excluindo-se o respectivo item.

Tabela 34 - Resultados da análise da correlação item-total para a rubrica CodeMaster v2.0 para App Inventor.

Item	Correlação Item-total	Alfa de Cronbach se o item for removido
I01. Operadores	0,694	0,82
I02. Variáveis	0,686	0,82
I03. Strings	0,583	0,83
I04. Nomeação	0,585	0,82
I05. Listas	0,364	0,84
I06. Persistência	0,325	0,84
I07. Eventos	0,596	0,82
I08. Laços	0,286	0,84
I09. Condicionais	0,618	0,82
I10. Sincronização	0,562	0,83
I11. Abstração	0,548	0,83
I12. Sensores	0,448	0,84
I14. Desenho e Animação	0,376	0,84
I15. Mapas	**0,015**	**0,85**
I16. Telas	0,324	0,84

A maioria dos valores das correlações item-total estão acima de 0,29. Isso indica que existe uma consistência interna aceitável. O grau de correlação entre os itens mostra que os itens medem o mesmo conceito, indicando evidência de validade convergente. No entanto, esta análise apresenta a baixa correlação do item Mapas (item 15), a qual pode ser explicada pelo fato de ser um recurso novo e ainda pouco usado.

O item 8 (Laços) apresenta uma correlação de 0,28, no entanto, a correlação ainda é muito próxima do valor mínimo esperado (0,29). Novamente, isso pode ser devido ao fato de que o uso de comandos de laços não é amplamente aplicado nos programas App Inventor devido à natureza do software que está sendo desenvolvido (XIE; ABELSON, 2016; TURBAK et al., 2014). Todos os outros itens mostram uma diminuição no valor do alfa de Cronbach se removidos e demonstram correlação item-total suficiente, indicando, assim, a validade das competências mensuradas relacionadas ao pensamento computacional.

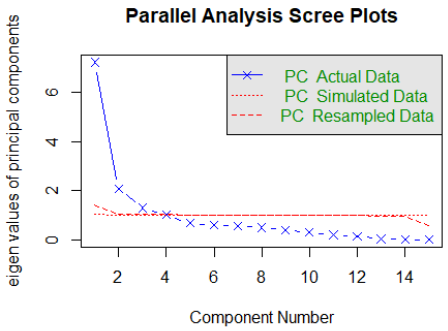
AQ3. Como os fatores subjacentes influenciam as respostas nos itens da rubrica CodeMaster v2.0?

Para identificar o número de fatores que representa a aplicação dos conceitos definidos nos itens da rubrica CodeMaster v2.0 para App Inventor, excluindo-se o item 13 (Extensões) desta análise haja vista o conjunto de dados não possui uma quantidade de aplicativos suficiente para análise, foi realizada uma análise fatorial.

Para analisar se os critérios da rubrica CodeMaster v2.0 para App Inventor podem ser submetidos a um processo de análise fatorial, foi utilizado o índice Kaiser-Meyer-Olkin (KMO) sendo o mais utilizado (BROWN, 2006). O índice KMO mede a adequação da amostragem com valores entre 0 e 1. Um valor de índice próximo a 1,0 suporta uma análise fatorial e qualquer valor menor que 0,5 indica que provavelmente não é passível de análise fatorial (BROWN, 2006). Analisando o conjunto de critérios da rubrica CodeMaster v2.0 para App Inventor, foi obtido um índice KMO de 0,83 indicando que a análise fatorial é apropriada neste caso.

O número de fatores retidos na análise é decidido aplicando-se a análise fatorial (GLORFELD, 1995). Foi utilizada a análise paralela que é um método para determinar o número de fatores a serem retidos da análise fatorial. Após a análise paralela, os resultados mostram que há um fator preponderante, mas ainda indica três pontos acima da linha vermelha (Figura 30).

Figura 30 - Scree Plot referente à rubrica CodeMaster v2.0 para App Inventor.



Fonte: elaborada pela autora.

Pela indicação de 3 pontos acima da linha vermelha, o scree plot sugere que podem haver 3 fatores subjacentes. Assim, para se determinar quais itens são carregados em qual fator, utiliza-se o método de rotação Oblimin, no qual os fatores podem ser correlacionados (JACKSON, 2005). A Tabela 35 mostra as cargas fatoriais dos itens associados aos 3 fatores retidos.

Tabela 35 - Cargas fatoriais para 3 fatores na rubrica CodeMaster v2.0 para App Inventor.

Item	Fator 1	Fator 2	Fator 3
I01. Operadores	0,325	0,074	**0,795**
I02. Variáveis	0,453	0,337	**0,763**
I03. Strings	-0,028	-0,100	**0,801**
I04. Nomeação	0,198	0,174	**0,659**
I05. Listas	-0,070	0,123	**0,690**
I06. Persistência	-0,093	-0,156	**0,786**
I07. Eventos	0,178	-0,157	**0,868**
I08. Laços	-0,004	0,209	**0,768**
I09. Condicionais	0,150	0,048	**0,807**
I10. Sincronização	**0,710**	-0,336	**0,616**
I11. Abstração	0,406	0,241	**0,779**
I12. Sensores	**0,713**	-0,432	0,453
I14. Desenho e Animação	**0,752**	0,298	0,351
I15. Mapas	-0,123	-0,389	0,403
I16. Telas	-0,158	-0,339	**0,702**

O limiar para as cargas fatoriais é baseado no de Comrey e Lee (1992). Eles sugerem o uso de limites mais restritos para classificar as cargas fatoriais: 0,32 (ruim), 0,45 (aceitável), 0,55 (bom), 0,63 (muito bom) ou 0,71 (excelente). Assim, cargas fatoriais boas, acima de 0,55 na Tabela 35, são marcadas em negrito.

Analisando as cargas fatoriais dos itens (Tabela 35), pode-se observar que, o primeiro fator (fator 1), está mais relacionado ao conceito de integração entre elementos da VPL App Inventor, haja vista o item 12 (Sensores) e o item 14 (Desenho e Animação), ambos relacionados a este conceito são agrupados nesse fator. Além desses itens, o item 10 (Sincronização) também apresentou uma alta carga fatorial nesse primeiro fator. Contudo, ele também apresenta uma alta carga fatorial no terceiro fator.

O segundo fator apresenta baixas cargas fatoriais para todos os itens. Nenhum item possui uma carga fatorial acima de 0,5, indicando que não há este segundo fator.

O terceiro fator está mais relacionado ao pensamento computacional conforme definido pelo CSTA (2016). A maioria dos

itens da rubrica CodeMaster v2.0 para App Inventor apresenta uma alta carga fatorial nesse fator.

O item 15 (Mapas), semanticamente relacionado ao primeiro fator, apresenta o maior fator de carregamento no terceiro fator. No entanto, cabe ressaltar que esse item possui uma pequena quantidade de aplicativos com pontuações acima de 0. Isso pode ter gerado ruído no cálculo.

Levando em consideração que o objetivo geral da rubrica CodeMaster v2.0 para App Inventor é a avaliação de um único conceito: o pensamento computacional, é importante determinar o quanto os itens estão sendo carregados em **um único fator**. A Tabela 36 apresenta as cargas fatoriais dos itens associadas a um fator, valores acima de 0,55 são marcados em negrito.

Analisando as cargas fatoriais dos itens da Tabela 36 pode-se observar que a maioria dos itens apresenta um valor de carga acima do limiar de 0,55. Apenas o item 15 (Mapas) possui uma carga fatorial baixa. Contudo, esse item possui poucos aplicativos com pontuações acima de 0, conforme mostrado na Tabela 32. Isso pode ter gerado ruído e dificultado o cálculo, haja vista esse item está relacionado a um recurso recém adicionado ao App Inventor.

Tabela 36 - Carregamento em um fator para a rubrica CodeMaster v2.0 para App Inventor.

Item	Fator
I01. Operadores	**0,875**
I02. Variáveis	**0,868**
I03. Strings	**0,697**
I04. Nomeação	**0,703**
I05. Listas	**0,590**
I06. Persistência	**0,679**
I07. Eventos	**0,860**
I08. Laços	**0,721**
I09. Condicionais	**0,807**
I10. Sincronização	**0,855**
I11. Abstração	**0,882**
I12. Sensores	**0,669**
I14. Desenho e Animação	**0,614**
I15. Mapas	0,262
I16. Telas	**0,574**

AQ4: Existe evidência de validade das categorias dos itens da rubrica CodeMaster v2.0?

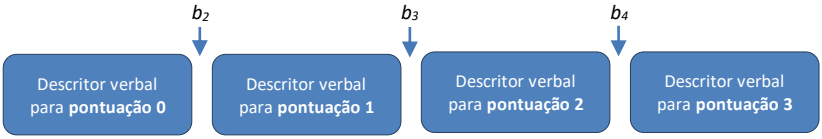
Para analisar a validade das categorias dos itens é realizada uma análise pela TRI (Teoria da Resposta ao Item) (ANDRADE;

TAVARES; VALLE, 2000), utilizando o Modelo de Resposta Gradual – MRG de Samejima (1969), por ser o mais indicado para este modelo. O MRG assume que as categorias de resposta de um item são ordenadas entre si (SAMEJIMA, 1969).

A métrica é estabelecida fixando-se os parâmetros populacionais em $\mu = 0$ e $\sigma = 1$, na qual μ é a média e σ é o desvio-padrão das habilidades da população considerada. A discriminação de uma categoria específica de resposta no MRG depende:

- do parâmetro de inclinação (*a*), comum a todas as categorias do item.
- da distância das categorias de dificuldade adjacentes (b_2, b_3 e b_4 num modelo com 4 categorias – veja Figura 31).

Figura 31 - Parâmetros de dificuldade para itens com 4 categorias de dificuldade adjacentes numa rubrica.



Fonte: elaborada pela autora.

Em relação aos valores desejados no processo de estimação dos parâmetros, também conhecido como calibração, destaca-se que valores acima de 1 para o parâmetro de inclinação (*a*) são considerados bons. Para os parâmetros de dificuldade (b_2, b_3 e b_4) valores tipicamente dentro do intervalo [-5, 5] são esperados, no entanto esta não é uma condição sine qua non. Cabe ressaltar ainda que também é desejado um bom espaçamento entre os valores dos parâmetros de dificuldade (b_2, b_3 e b_4), de forma que todas as categorias tenham uma probabilidade de resposta significativa.

A partir do resultado da calibração de itens, em que $\mu = 0$ e $\sigma = 1$, para o conjunto de 88.812 aplicativos avaliados pelo CodeMaster v2.0, em geral pode-se observar que a maioria dos itens calibrou bem, com valores do parâmetro de inclinação (*a*) acima de 1 (Tabela 37 – coluna 2). Somente o item 14 (Mapas) apresentou um valor baixo para o parâmetro de inclinação. Cabe ressaltar que o erro padrão (EP) do parâmetro de inclinação do item 14 (Mapas) está alto em comparação com o erro padrão dos parâmetros dos demais itens. Isso pode ser explicado pela falta de aplicativos com pontuações acima de 0 neste item, conforme apresentado na Tabela 32, pois trata-se de um recurso

recentemente adicionado ao App Inventor e, portanto, seu uso não está bem representado na base de dados conforme indica o erro padrão bem alto em comparação com os demais itens.

Tabela 37 - Parâmetros da TRI para os itens da rubrica CodeMaster v2.0 para App Inventor.

Item	*a*	EP(a)	*b2*	EP(b2)	*b3*	EP(b3)	*b4*	EP(b4)
I01. Operadores	3,081	0,024	-0,055	0,005	0,210	0,005	0,475	0,005
I02. Variáveis	2,971	0,022	-0,830	0,006	-0,009	0,005		
I03. Strings	1,656	0,012	-0,568	0,007	0,942	0,008		
I04. Nomeação	1,680	0,012	-0,313	0,006	0,067	0,006	1,887	0,012
I05. Listas	1,243	0,014	1,489	0,014	1,996	0,019	5,204	0,070
I06. Persistência	1,572	0,020	1,821	0,016	1,901	0,017	3,356	0,036
I07. Eventos	2,877	0,022	-1,650	0,009	-0,902	0,006	-0,473	0,005
I08. Laços	1,766	0,027	2,138	0,021	2,293	0,023	2,575	0,027
I09. Condicionais	2,323	0,017	0,344	0,005	0,796	0,006	1,571	0,009
I10. Sincronização	2,809	0,029	0,892	0,006				
I11. Abstração	3,184	0,034	0,988	0,006	1,081	0,006	1,189	0,007
I12. Sensores	1,530	0,014	0,636	0,007	2,768	0,022	4,392	0,050
I14. Desenho e Animação	1,324	0,013	0,817	0,008	1,245	0,011	1,454	0,013
I15. Mapas	**0,646**	**0,143**	**11,359**	**2,405**			**12,464**	**2,657**
I16. Telas	1,194	0,010	-2,530	0,019	0,885	0,009	1,099	0,010

Observa-se que, com base nos parâmetros de dificuldade (b_2, b_3 e b_4), nenhum item apresenta parâmetro de dificuldade menor que -2,5 e maior que 5,5. Ainda que alguns itens tenham apresentado pouco espaçamento entre os parâmetros de dificuldade, como os itens 8 e 11 (Laços e Abstração respectivamente), cabe ressaltar que, conceitualmente, os descritores verbais da rubrica CodeMaster v2.0 para esses itens são diferentes. Por esse motivo, não é indicado que sejam agrupados, mesmo que apresentem pouco espaçamento. Os demais itens apresentaram bom espaçamento para os parâmetros de dificuldade (b_2, b_3 e b_4) conforme apresenta a Tabela 37.

Posicionamento na escala.

A partir dos parâmetros calibrados, todos os itens com parâmetro de inclinação (*a*) acima de 1, ou seja, todos itens exceto o item 15 (Mapas), são posicionados numa escala (0,1), isto é, $\mu = 0$ e $\sigma = 1$ (Figura 32). A escala da habilidade é uma escala arbitrária onde o importante são as relações de ordem existentes entre seus pontos e não necessariamente sua magnitude. Os itens foram dispostos nos pontos da escala de acordo com os parâmetros de dificuldade (b_2, b_3 e b_4) calibrados, conforme apresentados na Tabela 37.

Exemplificando, o item 16 (Telas) é posicionado da seguinte forma:

- O valor calibrado para o parâmetro de dificuldade b_2 é de -2,530, portanto, Telas b2 é posicionado no ponto 2,5.
- O valor calibrado para o parâmetro de dificuldade b_3 é de 0,885, portanto, Telas b3 é posicionado no ponto 1,0.
- O valor calibrado para o parâmetro de dificuldade b_4 é de 1,099, portanto, Telas b4 é posicionado no ponto 1,5.

Figura 32 - Posicionamento dos itens na escala.

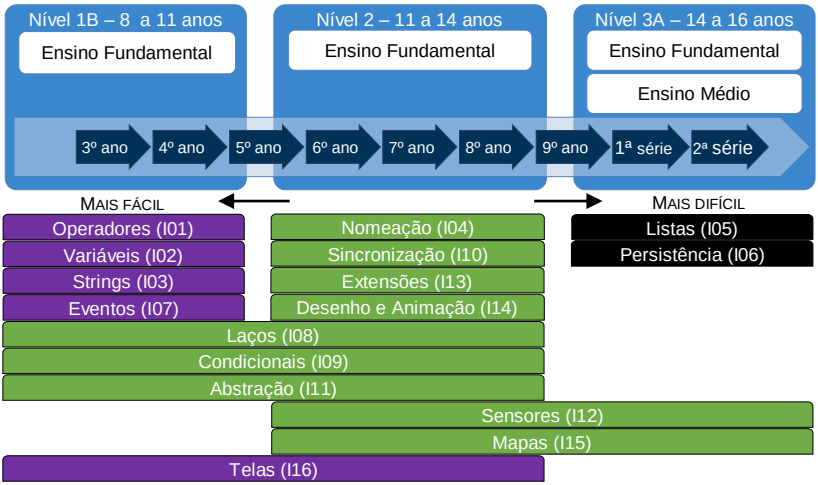
5,5	Listas b4						
5,0							Mais difícil
4,5	Sensor. b4						
4,0							
3,5	Persist. b4						
3,0	Laços b4	Sensor. b3					
2,5	Laços b2	Laços b3					
2,0	Nomea. b4	Listas b3	Persist. b2	Persist. b3	Condic. b4		
1,5	Listas b2	Abst. b3	Abst. b4	Des.Anim. b3	Des. Anim. b4	Telas b4	
1,0	Strings b3	Condic. b3	Sincro. b2	Abst. b2	Sensor. b2	Des. Anim. b2	Telas b3
0,5	Operad.b3	Operad.b4	Nomea. b3	Condic. b2			
0,0	Operad. b2	Variáveis b3	Nomea. b2	Eventos b4			
-0,5	Variáv. b2	Strings b2	Eventos b3				
-1,0							
-1,5	Event. b2						
-2,0							Mais fácil
-2,5	Telas b2						

Cada parâmetro *b* de cada categoria de um item representa a habilidade necessária para uma probabilidade de acerto de 0,50. A partir do posicionamento dos itens na escala (0,1) em que um indivíduo com habilidade 1,50 está 1,50 desvios-padrão acima da habilidade média, esse indivíduo teria probabilidade acima de 0,50 de acertar todos os itens que foram posicionados abaixo do ponto 1,50 da escala.

Nota-se que Listas b4, ou seja, o parâmetro de dificuldade que está entre o descritor verbal para a pontuação 2 e o descritor verbal para a pontuação 3 (veja a Figura 31 e os descritores para o item Listas na Tabela 28), trata-se do item mais difícil da rubrica CodeMaster v2.0 para App Inventor. Esse posicionamento do item está de acordo com o *K-12 Computer Science Standards* (CSTA, 2017). A escala da Figura 32 foi colorida de acordo com as dificuldades dos itens com base no *K-12 Computer Science Standards* (CSTA, 2017). Itens roxos são considerados mais fáceis por serem indicados a serem ensinados já do Ensino Fundamental – Anos Iniciais, já os itens pretos são considerados difíceis pois são indicados a serem ensinados no Ensino Médio ou no último ano do Ensino Fundamental. Os itens verdes são considerados de

dificuldade mediana pois devem ser ensinados no Ensino Fundamental – Anos Finais (Figura 33). Nota-se que os parâmetros de dificuldade dos itens acompanham a dificuldade teórica apontada por CSTA (2017), pois itens roxos (considerados mais fáceis) foram posicionados abaixo do ponto 1,5 da escala, já itens pretos (considerados mais difíceis) ficaram acima de 1,5 e itens verdes ficaram dispersos do meio para cima na escala.

Figura 33 - Dificuldade dos itens de acordo com o *K-12 Computer Science Standards*.



Fonte: elaborada pela autora com base em CSTA (2017).

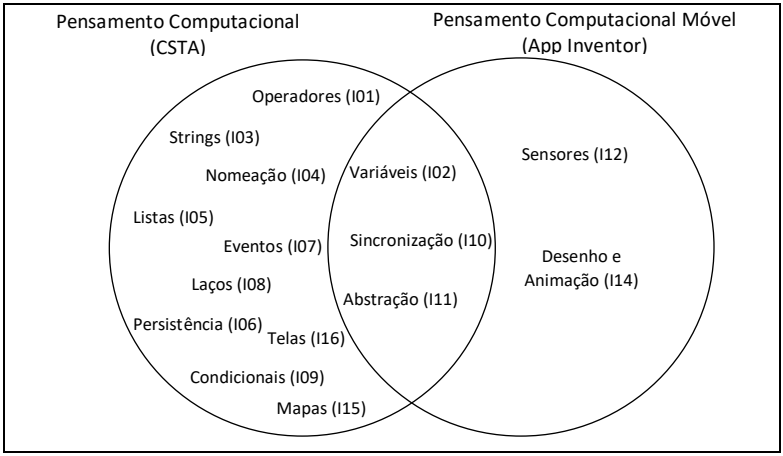
6.3 DISCUSSÃO

Os resultados obtidos indicam boa confiabilidade e validade da rubrica CodeMaster v2.0 para a avaliação das competências de pensamento computacional. No entanto, apesar de o item 15 (Mapas) ter apresentado indicadores inadequados, cabe ressaltar que esse é um recurso adicionado recentemente ao App Inventor. Assim, muitos dos aplicativos utilizados para avaliar a rubrica, por serem mais antigos, não contêm esse recurso. Em relação aos demais itens, todos apresentaram bons indicadores, tanto de confiabilidade, como validade.

Em relação à rubrica CodeMaster v1.0, houve melhora. Os itens da rubrica CodeMaster v2.0 apresentam bons indicadores em relação à confiabilidade, via aumento do Alfa de Cronbach e validade, em que todos os itens, exceto Mapas, apresentaram altos valores de

carregamento em um fator. Apenas o item Mapas apresentou resultados inconclusivos, devido à pequena quantidade de aplicativos que o contém. Além disso, os itens também apresentaram resultados mais consistentes em relação às subdimensões da rubrica, e a análise fatorial mostrou que os itens possuem altas cargas fatoriais em duas subdimensões (Figura 34) conforme originalmente proposto.

Figura 34 - Subdimensões e itens da rubrica CodeMaster v2.0 para App Inventor.



Fonte: elaborada pela autora.

Alguns itens apresentaram altas cargas fatoriais em dois fatores. Por exemplo, o item 10 (Sincronização) apresenta altas cargas fatoriais tanto em relação ao fator de pensamento computacional, conforme definido pelo CSTA (2016), quanto ao fator relacionado ao pensamento computacional móvel, com elementos referentes à VPL App Inventor. Desta forma, pode-se concluir um modelo de avaliação composto por duas subdimensões conforme apresentado na Figura 34.

Por meio da análise via TRI, verificou-se que todos os itens, exceto Mapas, apresentaram bons parâmetros de inclinação. Os itens mais difíceis, com parâmetro b mais elevado, foram os itens 5 e 12 (Listas e Sensores respectivamente). Esse resultado está de acordo com a definição da métrica, a qual contém conceitos considerados avançados na pontuação 3 para o contexto da Educação Básica, refletindo em altos parâmetros de dificuldade. Os itens que apresentaram os menores parâmetros de dificuldade já na pontuação 1 foram os itens 7 e 16 (Eventos e Telas). Esses parâmetros também estão semanticamente

coerentes, haja vista a VPL App Inventor estimula a criação de várias telas para criação de um aplicativo e o uso ilimitado de eventos.

6.3.1Ameaças à validade

Esta avaliação está sujeita a várias ameaças à validade. Portanto, foram identificadas ameaças potenciais e aplicadas estratégias de mitigação para minimizar seu impacto nos resultados. Algumas ameaças estão relacionadas ao projeto do estudo de caso. A fim de mitigar esta ameaça, foi definida e documentada uma metodologia sistemática para o estudo usando a abordagem GQM (BASILI; CALDIERA; ROMBACH, 1994). Outro risco refere-se à qualidade dos dados reunidos em uma única amostra, em termos de padronização de dados. Como o estudo é limitado exclusivamente a avaliações usando o CodeMaster v2.0, esse risco é minimizado, pois todas as análises foram realizadas de forma automatizada usando a mesma rubrica. Outra questão refere-se ao agrupamento de dados de diferentes contextos. Os aplicativos do conjunto de dados vêm de diversos contextos da comunidade App Inventor, de todos os lugares no mundo, e nenhuma informação adicional sobre o histórico dos criadores dos programas na Galeria App Inventor está disponível. No entanto, como o objetivo é a análise da validade da rubrica de uma forma independente do contexto, isso não apresenta um problema. Em termos de validade externa, uma ameaça à possibilidade de generalizar os resultados está relacionada ao tamanho da amostra e à diversidade dos dados utilizados para a avaliação. A análise é baseada em dados coletados da Galeria App Inventor, envolvendo uma amostra de 88.812 aplicativos da comunidade do App Inventor. Este é um tamanho de amostra satisfatório, permitindo a geração de resultados significativos.

Em termos de confiabilidade, uma ameaça refere-se à dependência, dos dados e da análise, de pesquisadores específicos. A fim de mitigar essa ameaça, foi documentada uma metodologia sistemática, definindo claramente o objetivo do estudo, o processo de coleta de dados e os métodos estatísticos utilizados para a análise de dados.

Cabe ressaltar que nessa análise considerou-se a obrigatoriedade de todos os itens para todos os programas. Contudo, nem sempre todos os itens especificados são realmente necessários em um programa que se deseja avaliar. No entanto, devido à grande quantidade de dados (mais de 88 mil) é inviável, no contexto de uma avaliação em larga escala, observar todos os programas para decidir quais são os itens obrigatórios para cada programa separadamente. Como o objetivo dessa avaliação é

avaliar de forma geral todos os itens (ao invés de avaliar e atribuir nota ao projeto de um aluno especificamente) considera-se que essa decisão teve um impacto mínimo.

7 CONCLUSÃO

Como resultado do presente trabalho foi criado um modelo genérico, confiável e válido para a avaliação de programas em VPL criados como resultado de atividades abertas no contexto educacional, respondendo à pergunta de pesquisa.

Como parte do presente trabalho, foi analisado o estado da arte por meio de um mapeamento sistemático publicado em (ALVES et al., 2018) (Objetivo 1). Foram analisados vários trabalhos e verificou-se a ausência de uma definição sistemática de critérios de avaliação bem definidos para uma grande variedade de VPL baseadas em blocos, especialmente para a avaliação de atividades abertas.

Foi conduzida uma avaliação em termos de confiabilidade e validade do modelo de avaliação CodeMaster v1.0 (GRESSE VON WANGENHEIM et al., 2018) em que se verificou a necessidade de revisão e possível exclusão de itens presentes na rubrica (Objetivo 2). A partir dos resultados da avaliação, do mapeamento sistemático e da literatura, foi desenvolvido sistematicamente o modelo, elencando as características do código a serem analisadas (Objetivo 3). O modelo foi instanciado para a VPL App Inventor por meio da definição de uma rubrica e implementado, evoluindo a implementação do CodeMaster v1.0 para v2.0 (Objetivo 4).

O modelo desenvolvido define itens que estão relacionados a etapas específicas da Educação Básica. O modelo, instanciado pela rubrica CodeMaster v2.0 para App Inventor, foi avaliado em termos de confiabilidade e validade (Objetivo 5). Verificou-se que houve melhora nos indicadores como aumento do alfa de Cronbach, bons valores na carga fatorial para um fator e bons parâmetros de inclinação e dificuldade calibrados pela TRI. A dificuldade dos itens reflete a dificuldade descrita pelo *Computer Science Standards* (CSTA, 2017).

Com esse resultado do presente trabalho disponibiliza-se um modelo de avaliação sistemática de programas criados com VPL baseada em blocos e avaliado em larga escala integrando e evoluindo trabalhos relacionados. Esse modelo pode suportar de forma eficiente e efetiva a avaliação no processo de aprendizagem tanto pelo aluno individualmente quanto pelo professor. E assim, contribuir positivamente para uma ampla aplicação do ensino de computação nas escolas Brasileiras.

Como trabalhos futuros sugere-se a integração de outros critérios importantes relacionados tanto a habilidades do século XXI, como a

criatividade, quanto a outros aspectos essenciais para a qualidade de produtos de software, como por exemplo a usabilidade.

Com um foco voltado à avaliação da aprendizagem, podem também ser explorados em trabalhos futuros a estimação de parâmetros via Teoria da Resposta ao Item com dados provenientes de contextos conhecidos, como por exemplo, escolas brasileiras que tenham o currículo alinhado à BNCC. A partir de parâmetros calibrados por meio de dados provenientes de contextos conhecidos pode-se construir uma escala de habilidades, sendo possível fazer uma interpretação pedagógica dos itens. Os parâmetros calibrados podem ser implementados diretamente no CodeMaster v2.0, de forma a automatizar a avaliação, atribuindo a nota via TRI e não via TCT (Teoria Clássica dos Testes).

REFERÊNCIAS

AIVALOGLOU, E.; HERMANS, F. How kids code and how we know: An exploratory study on the Scratch repository. **Proceedings of the 2016 ACM Conference on International Computing Education Research**, Melbourne, VIC, Australia, 2016. p. 53-61.

ALA-MUTKA, K.; JARVINEN, H.-M. Assessment process for programming assignments. **Proceedings of IEEE International Conference on Advanced Learning Technologies**, Joensuu, Finland, 2004.

ALA-MUTKA, K. M. A Survey of Automated Assessment Approaches for Programming Assignments. **Computer Science Education**, v. 15, n. 2, 2005. p. 83-102.

ALBANO, A. D. **Introduction to educational and psychological measurement: Using R**. Disponível em: <https://cehs01.unl.edu/aalbano/intro-measurement-r/> Acessado em 15 abr 2017.

ALVES, N. D. C.; VON WANGENHEIM, C. G.; HAUCK, J. Approaches to Assess Computational Thinking Competences Based on Code Analysis in K-12 Education: A Systematic Mapping Study. **INFORMATICS IN EDUCATION**, 2019. Artigo aceito para publicação.

ALVES, N. D. C. et al. Ensino de Computação de Forma Multidisciplinar em Disciplinas de História no Ensino Fundamental - Um Estudo de Caso. **Revista Brasileira de Informática na Educação**, v. 24, n. 3, 2017. p. 31-46.

ANDRADE, D. F.; TAVARES, H. R.; VALLE, R. C. **Teoria da Resposta ao Item: Conceitos e Aplicações**. São Paulo: ABE — Associação Brasileira de Estatística, 4º SINAPE, 2000.

ARMONI, M.; MEERBAUM-SALANT, O.; BEN-ARI, M. From Scratch to “Real” Programming. **ACM Transactions on Computing Education**, v. 14, n. 4, 2015.

AYEWAH, N. et al. Using Static Analysis to Find Bugs. **IEEE Software**, v. 25, n 5, 2008. p. 22-29.

BALL, M. Implementing "In-Lab" Autograding for Snap! (Abstract Only). **Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education**, Seattle, Washington, USA, 2017. p. 703-703.

BALL, M. A.; GARCIA, D. D. Autograding and Feedback for Snap!: A Visual Programming Language (Abstract Only). **Proceedings**

of the 47th ACM Technical Symposium on Computing Science Education, Memphis, Tennessee, USA, 2016. p. 692-692.

BASAWAPATNA, A. et al. Recognizing Computational Thinking Patterns. **Proceedings of the 42nd ACM Technical Symposium on Computer Science Education**, Dallas, TX, USA, 2011. p. 245-250.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. **Goal Question Metric Paradigm**. 2 ed. MARCINIAK, J. J.:Encyclopedia of Software Engineering, 1994, John Wiley & Sons.

BEECHAM, S. et al. Using an Expert Panel to Validate a Requirements Process Improvement Model. **Journal of Systems and Software**, v. 76, n. 3, 2005. p. 251-275.

BELLER, M.; BHOLANATH, R.; MCINTOSH, S.; ZAIDMAN, A. Analyzing the state of static analysis: A large-scale evaluation in open source software, **IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**, Suita, Japão, 2016. p. 470-481.

BOCCONI, S. et al. Developing computational thinking in compulsory education – Implications for policy and practice. **Technical report, European Union Scientific and Technical Research Reports**. EUR 28295 EN, Publications Office of the European Union.

BOE, B. et al. Hairball: Lint-inspired Static Analysis of Scratch Projects, **Proceeding of the 44th ACM technical symposium on Computer science education**, Denver, Colorado, USA, 2013. p. 215-220.

BOURQUE, P.; FAIRLEY, R. E. **Guide to the software engineering body of knowledge** SWEBOK (R): v. 3 IEEE Computer Society Press, 2014.

BRANCH, R. M. **Instructional Design: The ADDIE Approach**. New York: Springer, 2009.

BRENNAN, K.; RESNICK, M. New frameworks for studying and assessing the development of computational thinking. **Proceedings of the Annual Meeting of the American Educational Research Association**, Vancouver, Canada, 2012.

BROWN, T. A. **Confirmatory factor analysis for applied research**. New York: The Guilford Press, 2006.

CARMINES, E. G.; ZELLER, R. A. **Reliability and validity assessment**. 5ª ed. Beverly Hills: Sage Publications Inc, 1982.

CHEN, G. et al. Assessing elementary students' computational thinking in everyday reasoning and robotics programming. **Computers & Education**, v. 109, 2017. p. 162-175.

CME. **RESOLUÇÃO CME Nº02/2011**. Conselho Municipal de Educação de Florianópolis, 2011.

CNE, 2019. **Iniciativa Computação na Escola**. Disponível em: <http://www.computacaonaescola.ufsc.br/> Acessado em: out 2018.

CODE, 2015. **Hour of Code**. Disponível em: <https://hourofcode.com> Acessado em: out 2017.

CODECOMBAT, 2016. **CodeCombat**. Disponível em: <https://codecombat.com/> Acessado em: mar 2018.

COHEN, J.. **Statistical Power Analysis for the Behavioral Sciences**. New York: Routledge Academic, 1998.

COMREY, A. L.; LEE, H. B. **A first course in factor analysis**. 2ª ed. Hillsdale NJ: Lawrence: Erlbaum Associates, 1992.

CRONBACH, L. J. Coefficient alpha and the internal structure of tests. **Psychometrika**, v. 16, n. 3, 1951. p. 297–334.

CSTA. **K-12 Computer Science Framework**. Computer Science Teachers Association, 2016.

CSTA. **K-12 Computer Science Standards**. Computer Science Teachers Association, 2017.

DANIEL, G. T.; VON WANGENHEIM, C. G.; MEDEIROS, G. A. S.; ALVES, N. C. Ensinando a Computação por meio de Programação com App Inventor. **Anais do Computer on the Beach**, 2017. p. 357-365.

DANIELS, M.; CARBONE, A.; HAUER, A.; MOORE, D. **Panel - ill-structured problem solving in engineering education**. Milwaukee, WI, USA, Frontiers In Education Conference - Global Engineering: Knowledge Without Borders, Opportunities Without Passports, 2007.

DEMETRIO, M. F. **Desenvolvimento de um analisador e avaliador de código de App Inventor para ensino de computação**. Trabalho de Conclusão de Curso. Graduação em Ciência da Computação: Universidade Federal de Santa Catarina, 2017.

DENNER, J.; WERNER, L.; ORTIZ, E. Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts?. **Computers & Education**, v. 58, n. 1, 2012. p. 240-249.

DEVELLIS, R. F. Classical Test Theory. **Medical Care**, v. 44, n. 11, p. 50-59, 2006.

DEVELLIS, R. F. **Scale development: theory and applications**. Thousand Oaks: SAGE Publications, 2003.

DRISCOLL, A.; WOOD, S. **Developing Outcomes-Based Assessment for Learner- Centered Education: a Faculty Introduction**. Sterling, VA, USA: Stylus Publishing, 2007.

FITZGERALD, S. et al. What are we thinking when we grade programs?. **Proceeding of the 44th ACM technical symposium on Computer science education**, Denvier, Colorado USA, 2013. p. 471-476.

FRANKLIN, D. et al. Assessment of Computer Science Learning in a Scratch-Based Outreach Program. **Proceeding of the 44th ACM technical symposium on Computer science education**, Denver, Colorado, USA, 2013. p. 371-376.

FUNKE, A.; GELDREICH, K.; HUBWIESER, P. Analysis of Scratch Projects of an Introductory Programming Course for Primary School Students. **Proceeding of IEEE Global Engineering Education Conference**, Atenas, Grécia, 2017.

GAGNE, R. M. **Conditions of Learning**. Holt, Rinehart and Winston, 1965.

GIJSELAERS, W. H. Connecting problembased practices with educational theory. In: L. W. &. W. H. G. (Eds.), ed. **Bringing problem-based learning to higher education: Theory and practice**. San Francisco: Jossey-Bass, 1996. p. 13-21.

GIL, A. C. **Como elaborar projetos de pesquisa**. 5 ed. São Paulo: Atlas, 2010.

GIPPS, C.. **Beyond testing: Towards a theory of educational assessment**. London: The Falmer, 1994.

GLORFELD, L. W. An improvement on Horn's parallel analysis methodology for selecting the correct number of factors to retain. **Educational and Psychological Measurement**, v. 55, n. 3, 1995. p. 377-393.

GOLIN, E. J.; REISS, S. P. The Specification of Visual Language Syntax. **Journal of Visual Languages and Computing**, v. 1, n. 2, 1990. p. 141-157.

GRESSE VON WANGENHEIM, C. et al. CodeMaster - Automatic Assessment and Grading of App Inventor and Snap! Programs. **Informatics in Education**, v. 17, n. 1, 2018. p. 117-150.

GROVER, S.; BASU, S.; SCHANK, P. What We Can Learn About Student Learning From Open-Ended Programming Projects in Middle School Computer Science. **Proceedings of the 49th ACM Technical Symposium on Computer Science Education**, Baltimore, MD, 2018.

GROVER, S.; BIENKOWSKI, M.; NIEKRASZ, J.; HAUSWIRTH, M. Assessing problem-solving process at scale. **Proc. of the Third ACM Conference on Learning @ Scale**, New York, NY, USA, 2016. p. 245-248.

GROVER, S.; COOPER, S.; PEA, R. Assessing computational learning in K-12. **Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education**, Uppsala, Suécia, 2014. p. 57–62.

GROVER, S.; PEA, R. Computational Thinking in K–12 A review of the state of the field. **Educational Researcher**, v. 42, n. 1, 2013. p. 38-43.

GROVER, S.; PEA, R.; COOPER, S. Designing for deeper learning in a blended computer science course for middle school students. **Journal Computer Science Education**, v. 25, n. 2, 2015. p. 199-237.

GUPTA, S.; DUBEY, S. K. Automatic Assessment of Programming assignment. **Computer Science & Engineering: An International Journal**, v. 2, 2012. p. 315-323.

HARVEY, B.; GARCIA, D.; PALEY, J.; SEGARS, L. Snap!: (build your own blocks). **Proceedings of the 43rd ACM Technical Symposium on Computer Science Education**, Raleigh, NC, USA, 2012.

HOWATT, J. W. On criteria for grading student programs. **ACM SIGCSE Bulletin**, v. 26, n. 3, 1994. p. 3-7.

HWANG, G.; LIANG, Z.; WANG, H. An Online Peer Assessment-Based Programming Approach to Improving Students' Programming Knowledge and Skills. **Proceedings of the Int. Conference on Educational Innovation through Technology**, Tainan, Taiwan, 2016.

ISO/IEC 25010, 2011. **Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE)**, ISO/IEC, 2011.

JACKSON, J. E. Oblimin Rotation. **Encyclopedia of Biostatistics**, John Wiley & Sons, Ltd 2005.

JOHNSON, D. E. ITCH: Individual Testing of Computer Homework for Scratch Assignments. **Proceedings of the 47th ACM Technical Symposium on Computing Science Education**, Memphis, USA, 2016.

JONES, T. Static and dynamic testing in the software development life cycle. **IBM Developer**.

Disponível em: <https://www.ibm.com/developerworks/library/se-static/index.html> Acessado em: mai 2017.

KECHAO, W.; TIANTIAN, W.; XIAOHONG, S.; PEIJUN, M. Overview of Program Comprehension. **Proceedings of the Int. Conference Computer Science and Electronics Engineering**, London, UK, 2012. p. 23-25.

KOH, K. H.; BASAWAPATNA, A.; BENNETT, V.; REPENNING, A. Towards the Automatic Recognition of Computational Thinking for Adaptive Visual Language Learning, **IEEE Symposium on Visual Languages and Human-Centric Computing**, Leganes, Espanha, 2010.

KOH, K. H.; BASAWAPATNA, A.; NICKERSON, H.; REPENNING, A. Real Time Assessment of Computational Thinking, **IEEE Symposium on Visual Languages and Human-Centric Computing**, Melbourne, VIC, Australia, 2014a.

KOH, K. H.; NICKERSON, H.; BASAWAPATNA, A.; REPENNING, A. Early Validation of Computational Thinking Pattern Analysis. **Proceedings of the 2014 conference on Innovation & technology in computer science education**, 2014b. p. 213-218.

KOYYA, P.; LEE, Y.; YANG, J. Feedback for Programming Assignments Using Software-Metrics and Reference Code. **International Scholarly Research Notices**, 2013.

KWON, K. Y.; SOHN, W.-S. A Method for Measuring of Block-based Programming Code Quality, **International Journal of Software Engineering and Its Applications**, v. 10, n. 9, 2016a.

KWON, K. Y.; SOHN, W.-S. A Framework for Measurement of Block-based Programming Language, **Proceedings of the 10th International Workshop Series Convergence Works**, 2016b.

LARMAN, C.; BASILI, V. Iterative and incremental developments: a brief history. **IEEE Computer**, v. 36, 2003. p. 47–56.

LOBATO, A. S. et al. Uma rubrica para avaliação de cursos de programação centrada em avaliação automática. **SBIE - Workshop de Ambientes de apoio à Aprendizagem de Algoritmos e Programação**, 2007.

LYE, S. Y.; KOH, J. H. L. Review on teaching and learning of computational thinking through programming: What is next for K-12?. **Computers in Human Behavior**, v. 41(C), 2014. p. 51-61.

MAIORANA, F.; GIORDANO, D.; MORELLI, R. Quizly: A Live Coding Assessment Platform for App Inventor. **IEEE Blocks and Beyond Workshop**, Atlanta, GA, USA, 2015.

MEC, 2018. Base Nacional Comum Curricular, Brasil.

MORENO, J.; ROBLES, G. Automatic Detection of Bad Programming Habits in Scratch: A Preliminary Study, **Proceedings of the IEEE Frontiers in Education Conference**, Madri, Espanha, 2014.

MORENO-LEÓN, J.; HARTEVELD, C.; ROMÁN-GONZÁLEZ, M.; ROBLES, G. On the Automatic Assessment of Computational Thinking Skills: A Comparison with Human Experts. **Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems**, New York, NY, USA, 2017.

MORENO-LEÓN, J.; ROBLES, G. Dr. Scratch: a Web Tool to Automatically Evaluate Scratch Projects. **Proceedings of the Workshop in Primary and Secondary Computing Education**, Londres, Reino Unido, 2015.

MORENO-LEÓN, J.; ROBLES, G.; ROMÁN-GONZÁLEZ, M. Comparing Computational Thinking Development Assessment Scores with Software Complexity, **Proceedings of the IEEE Global Engineering Education Conference**, Abu Dhabi, United Arab Emirates, 2016.

MOSKAL, B. M.; LEYDENS, J. A. Scoring Rubric Development: Validity and Reliability. **Practical Assessment, Research & Evaluation**, v. 7, n. 10, 2000.

NIST. **Source Code Security Analysis Tool Functional Specification Version 1.1**. NIST Special Publication 500-283 Disponível em: <https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html> Acessado em mar 2017.

OLSSON, U. Maximum likelihood estimation of the polychoric correlation coefficient. **Psychometrika**, v. 44, n. 4, 1979. p. 443–460.

OTA, G.; MORIMOTO, Y.; KATO, H. Ninja code village for scratch: Function samples/function analyser and automatic assessment of computational thinking concepts. **IEEE Symposium on Visual Languages and Human-Centric Computing**, Cambridge, UK, 2016. p. 238-239.

PASQUALI, L. **Psicometria: Teoria e aplicações**. Brasília, DF: Editora da Universidade de Brasília, 1997.

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic mapping studies in software engineering. **Proceedings of the 12th international conference on Evaluation and Assessment in Software Engineering**, Italy: BCS Learning & Development, 2008. p. 68-77.

SADLER, D. R. Formative assessment and the design of instructional systems. **Instructional science**, v. 18, n. 2, 1989. p. 119-144.

SAMEJIMA, F. A. Estimation of latent ability using a response pattern of graded scores. **Psychometric Monograph**, v. 17, 1969.

SANTOS, P. S. C.; ARAUJO, L. G. J.; BITTENCOURT, R. A. A Mapping Study of Computational Thinking and Programming in Brazilian K-12 Education. **FIE 2018 - 48th Annual Frontiers In Education Conference**, San Jose, California, 2018.

SARTES, L. M. A.; SOUZA-FORMIGONI, M. L. O. Avanços na psicometria: da Teoria Clássica dos Testes à Teoria de Resposta ao Item. **Psicologia: Reflexão e Crítica**, Porto Alegre, v. 26, n. 2, 2013. p. 241-250.

SAUNDERS, M. N. K., LEWIS, P.; THORNHILL, A. **Research Methods for Business Students**. 5 ed. Harlow:Prentice Hall, 2009.

SEITER, L.; FOREMAN, B. Modeling the Learning Progressions of Computational Thinking of Primary Grade Students. **Proceedings of the ninth annual international ACM conference on International computing education research**, La Jolla, CA, USA, 2013. p. 59-66.

SHERMAN, M.; MARTIN, F. The assessment of mobile computational thinking. **Journal of Computing Sciences in Colleges**, v. 30, n. 6, 2015. p. 53–59.

SHERMAN, M.; MARTIN, F.; BALDWIN, L.; DEFILIPPO, J., 2014. **App Inventor Project Rubric – Computational Thinking through Mobile Computing**. Disponível em <https://nsfmobilect.files.wordpress.com/2014/09/mobile-ct-rubric-for-app-inventor-2014-09-01.pdf> Acessado em: dez 2018.

SINTHSIRIMANA, S.; PANJABUREE, P. A Development of Computer Programming Analyzer: A Case Study on PHP Programming Language. **Journal of Industrial and Intelligent Information**, v. 1, n. 3, 2013.

SOTIROV, A. **Automatic Vulnerability Detection Using Static Source Code Analysis**. Dissertação de mestrado. Department of Computer Science, Graduate School of University of Alabama, 2005.

STEGEMAN, M.; BARENDSEN, E.; SMETSERS, S. Designing a rubric for feedback on code quality in programming courses. **Proceedings of the 16th Koli Calling International Conference on Computing Education Research**, 2016. p. 160-164.

STEVENS, J. P. **Applied multivariate statistics for the social sciences**. 2ª ed. Hillsdale,NJ: Lawrence Erlbaum Associates, 1992.

SURIF, J., IBRAHIM, H. N.; DALIM, S. F. Problem Solving: Algorithms and Conceptual and Open-ended Problems in Chemistry. **Procedia - Social and Behavioral Sciences**, v.116, 2014. p. 4955-4963.

TECHAPALOKUL, P. Sniffing Through Millions of Blocks for Bad Smells. **Proc. of the ACM SIGCSE Technical Symposium on Computer Science Education**, NY, USA, 2017.

TEMPEL, M. Blocks Programming. **CSTA Voice**, v. 9, n. 1, 2013. p. 3-4.

TROCHIM, W. M.; DONNELLY, J. P. **Research methods knowledge base**. 3ª ed. Mason, OH: Atomic Dog Publishing, 2008.

TRUONG, N.; ROE, P.; BANCROFT, P. Static Analysis of Students' Java Programs. **Australian Computing Education Conference**, v. 30, 2004.

TURBAK, F.; MUSTAFARAJ, F.; SVANBERG, M.; DAWSON, M. Identifying and analyzing original projects in an open-ended blocks programming environment. **Proceedings of the 23rd Int. Conference on Visual Languages and Sentient Systems**, Pittsburgh, PA, USA, 2017.

TURBAK, F. et al. Events-first programming in APP inventor. **Journal of Computing Sciences in Colleges**, v. 29, n. 6, 2014. p. 81-89.

WEINTROP, D.; WILENSKY, U. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. **Proceedings of International Computing Education Research**, Omaha, Nebraska, USA, 2015.

WERNER, L.; DENNER, J.; CAMPE, S.; KAWAMOTO, D. C. The Fairy Performance Assessment: Measuring Computational Thinking in Middle School. **Proceedings of the 43rd ACM technical symposium on Computer Science Education**, Raleigh, North Carolina, USA, 2012. p. 215-220.

WIGGINS, G. P. **The Jossey-Bass education series. Assessing student performance: Exploring the purpose and limits of testing**. San Francisco: Jossey-Bass, 1993.

WING, J. M. Computational thinking. **Communications of the ACM**, v. 49, n. 3, 2006. p. 33–35.

WOLZ, U.; HALLBERG, C.; TAYLOR, B. Scrape: A tool for visualizing the code of scratch programs. **Proceedings of the 42nd ACM Technical Symposium on Computer Science Education**, Dallas, Texas, USA. 2011.

XIE, B.; ABELSON, H. Skill progression in MIT App Inventor. **IEEE Symposium on Visual Languages and Human-Centric Computing**, Cambridge, UK, 2016. p. 213–217.

YIN, R. K. **Case study research: design and methods**. Ed. 2, Thousand Oaks: SAGE Publications, 2001.

YULIANTO, S. V.; LIEM, I. Automatic Grader for Programming Assignment Using Source Code Analyzer. **International Conference on Data and Software Engineering**, Bandung, Indonesia 2014.

ZHANG, Y.; SURISETTY, S.; SCAFFIDI, C. Assisting comprehension of animation programs through interactive code visualization. **Journal of Visual Languages & Computing**, v. 24, n. 5, 2013. p. 313–326.

APÊNDICE A – Elementos analisados pelas abordagens no estado da arte

Detalhamento extraído dos artigos encontrados durante o mapeamento sistemático apresentado no capítulo 3. As informações sobre os elementos analisados foram transcritas sem modificações, com tradução para o português quando possível, mantendo-se a mesma nomenclatura encontrada em cada artigo. Cabe ressaltar que alguns artigos utilizam nomenclatura diferente para elementos equivalentes.

Abordagem	**Detalhamento dos elementos analisados**
Abordagem de Kwon e Sohn	Programação: - Comandos - Parâmetros - Variável - Gatilhos (*trigger*) - Condicional - Iteração - Tipo de dados - Entrada e Saída - Estrutura de dados - Concorrência - Rotinas Conceitos avançados: - Algoritmos - Divisão e Conquista - Design de interação Conteúdo: - Criatividade - Estética - Funcionalidade - Nível de completude
Hairball	- Estado inicial (*Initial State: costume, visibility, orientation, position, size, background*) - Sincronização de fala/som (*Say/Sound Synchronization*) - Troca de messages (*Broadcast and Receive*) - Animações complexas (*Complex Animation*: *costumes, motion, timing, and repetition control structures such as loops*)
Dr.Scratch	– Lógica – Paralelismo – Interatividade com o usuário – Representação de dados – Controle de fluxo – Sincronização – Abstração

CTP/PBS/ REACT	CTP (Computational Thinking Patterns): - *Hill Climbing* - Controle de cursor - *Generations* - Absorção - Colisão - Transporte - *Push* - *Pull* - Difusão
ITCH	Testes personalizados realizados em tempo de execução sobre 5 tarefas predefinidas que abordam: - Tarefa 1: conversão de um algoritmo em pseudocódigo para Scratch contendo laços e condicionais - Tarefa 2: jogo de aventura com variáveis de estado, condicionais e troca de mensagens entre atores - Tarefa 3: implementação de ordenação por seleção destrutiva (*destructive selection sort*) usando *random*, operações com listas e variáveis - Tarefa 4: conversão binário-decimal usando mudança de costumes no Scratch, função piso (*floor function*), concatenação de *strings* e criação de uma subrotina por meio de um bloco customizado - Tarefa 5: digitalização de valores analógicos dada uma taxa de amostragem e tamanho da amostra usando a altura de um áudio
PECT	Pensamento computacional: - Procedimentos e algoritmos - Decomposição de problemas - Paralelização e sincronização - Abstração - Representação de dados Variáveis de evidência: - Aparência - Som - Movimento - Variáveis - Sequência e laços - Expressões booleanas - Operadores condicionais - Coordenadas - Eventos de interação com usuário - Paralelização - Inicialização de posição - Inicialização de aparência. Variáveis de Padrões de Design: - Aparência da animação

	- Movimento da animação - Diálogo - Colisão - Pontuação - Interação com o usuário
Ninja code village	– Condicional – Laços – Procedimento – Procedimento compartilhado – Dados – Eventos – Paralelismo – Interatividade com o usuário – Funções padrões utilizadas
Fairy Assessment	Três tarefas para analisar: - Pensamento algorítmico - Uso efetivo de abstração e modelagem - Criatividade / Inovação
Abordagem de Denner, Werner e Ortiz	Programação: - Paralelismo - Uso de *random* - Teste sobre variável - Uso de variável global - Uso de variável de um ator - Interação de um ator de forma condicional - Eventos ou condições - Funcionalidade de "porta" (outra forma de execução condicional) Organização de código e documentação: - Regras estranhas (*extraneous rule)* - Nomes dos personagens - Nomes de variáveis - Nome das regras (*rule names*) - Regras de comentários utilizadas - Regras de agrupamento (*rule grouping*) - Caixas de regras (*rule boxes*) Design para usabilidade: - Temas incorporados - Mudanças de aparência de personagens - Tipo de aparência do personagem - Vínculo entre as cenas - Múltiplas fases de jogo - Instruções claras - Objetivo (como ganhar-perder) claro - Funcionalidade
Scrape	- Movimento - Aparência

	- Som - Caneta - Variáveis - Eventos - Controle - Sensores - Operadores - Mais blocos
Quizly	Quizzes - Comandos básicos - Expressões - Controle - Procedimentos - Manipular eventos - Funções
CodeMaster	Conceitos e práticas do pensamento computacional - Nomeação - Abstração de procedimentos - Eventos - Laços - Condicionais - Operadores - Listas - Persistência de dados Pensamento computacional móvel - Persistência de dados - Telas - Interface de usuário - Sensores - Mídia - Social - Conectividade - Desenho e animação
Abordagem de Funke et al.	Requisitos - Vários atores - Movimento de atores - Iteração - Condicional Conceitos de programação - Sequência - Variáveis - Listas - Manipulação de eventos - Threads - Coordenação e Sincronização - Interação com o teclado - Números aleatórios

	- Lógica Booleana - Interação Dinâmica <u>Organização de código</u> - Código morto (blocos irrelevantes) - Nomeação de atores - Nomeação de variáveis <u>Operabilidade</u> - Funcionalidade - Personalização de atores - Personalização de palco - Interatividade - Usabilidade - Tipo de projeto <u>Níveis de Compreensão</u> - Pré estrutural - Uni estrutural - Multi estrutural - Relacional - Solução estendida (*extended abstract*)

ANEXO A – RUBRICA CODEMASTER v1.0

Item	Pontuação			
	0	**1**	**2**	**3**
Telas	Apenas uma tela com componentes visuais e que seu estado não se altera com a execução do app (tela informativa).	Apenas uma tela com componentes visuais que se alteram com a execução do app.	Pelo menos duas telas e uma delas altera seu estado com a execução do app.	Duas ou mais telas e pelo menos 2 delas alteram seus estados com a execução do app.
Interface de Usuário	Utiliza apenas 1 elemento visual sem alinhamento.	Utiliza 2 ou mais elementos visuais sem alinhamento.	Utiliza mais de 5 elementos visuais com 1 tipo de alinhamento.	Utiliza mais de 5 elementos visuais com 2 ou mais elementos de alinhamento.
Nomeação: Variáveis e procedimentos	Nenhum ou poucos nomes são alterado do padrão. (menos do que 10%)	De 10 a 25% dos nomes são alterados do padrão.	De 26 a 75% dos nomes são alterados do padrão.	Mais de 76% dos nomes são alterados do padrão.
Eventos	Nenhum manipulador de evento é usado (ex. On click).	1 tipo de manipuladores de eventos é usado.	2 tipos de manipuladores de eventos são usados.	Mais de 2 tipos de manipuladores de eventos são usados.
Abstração de procedimentos	Não existem procedimentos.	Existe exatamente um procedimento e sua chamada.	Existe mais de um procedimento.	Existem procedimentos tanto para organização quanto para reuso. (Mais chamadas de procedimentos do que procedimentos).
Laços	Não usa laços	Usa “While” (laço simples)	Usa “For each” (variável simples)	Usa “For each” (item de lista)
Condicionais	Não usa condicionais.	Usa apenas “if’s” simples.	Usa apenas “if then else”.	Usa um ou mais “if - else if”.
Operações Lógicas e Matemáticas	Não usa operadores	Usa um tipo de operador.	Usa dois tipos de operadores.	Usa mais de 2 tipos de operadores.
Listas	Não usa listas.	Usa uma lista unidimensional.	Usa mais de uma lista unidimensional.	Usa uma lista de tuplas (map).
Persistência de dados	Dados são armazenados	Usa persistência em	Usa algum dos bancos de	Usa uma base de dados web

	em variáveis ou propriedades de componentes e não tem persistência quando app é fechado.	arquivo (File ou Fusion Tables).	dados locais do App Inventor (TinyDB).	tinywebdb ou Firebase do App Inventor (firebase ou TinyWebDB).
Sensores	Sem uso de sensores.	Usa um tipo de sensor.	Usa 2 tipos de sensores.	Usa mais de 2 tipos de sensores.
Mídia	Sem uso de mídias.	Usa um tipo de mídia.	Usa 2 tipos de mídia.	Usa mais de 2 tipos de mídia.
Social	Sem uso de componentes sociais.	Usa um tipo de componente social.	Usa 2 componentes visuais.	Usa mais de 2 componentes visuais.
Conectividade	Sem uso de componentes de conectividade.	Uso de iniciador de atividades (chama início de outro app no celular).	Uso de conexão Bluetooth.	Uso de conexão web, baixo nível.
Desenho e Animação	Sem uso de desenho e animação.	Uso de área sensível ao toque.	Uso de animação com bolinha predefinida.	Uso de animação com imagem.