**Construção e uso de ambiente visual para
o ensino de programação introdutória**

Romenig da Silva Ribeiro

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO
DE
MESTRE EM CIÊNCIAS

Programa de Pós-Graduação em Ciência da Computação
Orientador: Prof. Dr. Leônidas de Oliveira Brandão

São Paulo, 10 de Julho de 2015

Construção e uso de ambiente visual para o ensino de programação introdutória

Esta versão da dissertação contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa da versão original do trabalho, realizada em 12/05/2015. Uma cópia da versão original está disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Leônidas de Oliveira Brandão - IME-USP
- Prof. Dr. Flávio Soares Corrêa da Silva - IME-USP
- Prof. Dra. Juliana Cristina Braga - UFABC

Agradecimentos

Agradeço a Deus, meus pais, à Julia de Athayde (minha futura esposa) e nossos familiares pelo apoio e paciência. Agradeço também ao apoio de todos os amigos que me ajudaram a chegar até aqui. Agradeço ao professor Dr. Leônidas de Oliveira Brandão e sua esposa professora Dra. Anarosa Alves Franco Brandão pelo apoio, orientação e confiança.

Sou muito grato ao amigo Danilo Dalmon, que inspirou de diversas maneiras a realização deste trabalho. A Maria José, pelo apoio e amizade. Também agradeço ao Tulio Faria, meu amigo, pelas palavras de força e inspiração e ajuda quando mais precisei. Agradeço ao Willian Raphael Silva, pela paciência e leitura do trabalho. Ao Fabio Martins, pelas concessões e amizade.

Resumo

RIBEIRO, R. S. **Construção e uso de ambiente visual para o ensino de programação introdutória**. 2015. 78 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2015.

Atualmente, observa-se um significativo crescimento na capacidade humana, resultante do emprego generalizado de sistemas computacionais. Consequentemente, têm sido valorizados o raciocínio lógico e o conhecimento de programação de computadores, observando-se, em vários países, iniciativas para sua introdução no ensino formal desde o ciclo fundamental. Nesse contexto, o ensino superior de ciências exatas apresenta um problema. Se de um lado existe uma maior demanda por graduados que saibam programar computadores, de outro a literatura registra uma alta taxa de insucesso em disciplinas de introdução à programação, indicando ainda que isso leva muitos estudantes a se desmotivarem e desistirem de seus cursos. Entretanto, mais recentemente, a literatura também aponta que a adoção de um paradigma de programação visual pode produzir melhores resultados que a programação textual. Este último modelo de programação é baseada na digitação de código, enquanto que no paradigma visual, os aprendizes utilizam recursos visuais como fluxogramas, ícones ou blocos que representam comandos para construírem os algoritmos. Essa abordagem apresenta resultados promissores, inclusive com estudantes do ensino fundamental. Deste modo, uma questão relevante, foco desta dissertação, é saber se, e como, a programação visual difere da textual em termos de esforços empregados pelos estudantes no processo de aprendizagem. Para isso, optou-se por produzir um novo sistema de programação, baseado em duas outras dissertações realizadas no *Laboratório de Informática na Educação (LIInE)* do IME-USP. A primeira gerou o sistema *iVProg* e a segunda uma *Linha de Produto de Software (LPS)* em 2010. Após estudos, percebeu-se ser mais eficiente produzir uma nova versão do sistema *iVProg* baseado na LPS citada do que evoluir o sistema legado. A partir dessa nova versão do *iVProg*, foi elaborado um experimento para se comparar o modelo de programação textual com o visual. O método empregou o protocolo NASA-TLX para a mensuração da carga de trabalho. Os resultados do experimento sugerem que a abordagem de programação visual demanda menos esforço mental que a programação textual, a partir do que conjectura-se que no modelo visual o aprendiz pode concentrar seus esforços mentais na aprendizagem dos conceitos básicos e fundamentais à programação.

Palavras-chave: programação visual, introdução à programação, *iVProg*, cursos via *Web*, aprendizagem de programação, educação a distância, ambientes virtuais de aprendizagem, e-aprendizagem.

Abstract

RIBEIRO, R. S. **Construction and use of visual environment for teaching introductory programming**. 2015. 78 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2015.

Nowadays, there is significant growth in the human capacity resulting from widespread use of computer systems. Consequently, the logical reasoning and the knowledge of computer programming has been highly valued in several countries. It has been observed initiatives all over the world for its introduction since the elementary school. In this context, higher education sciences presents an apparent paradox. On the one hand there is a greater demand for graduates who know how to program computers, on the other hand literature reports a high failure rate in introductory programming courses and indicates that this demotivate many students, leading them to give up their courses. However, more recently, the literature also pointed out that the adoption of a *visual programming* paradigm can produce better results than the textual programming. In the visual paradigm, learners use visual items, like flowcharts, icons and blocks, that represent commands to build the algorithms. This approach shows promising results, and has been used even with student in elementary schools. Thus, a relevant question, focus of this dissertation is whether, and how, visual programming differs from textual programming in terms of efforts made by students in the learning process. To analyse this question, was produced a new system for visual programming, the *iVProg* version 2, based on two dissertation works in the *Laboratory of Informatics in Education (LInE)* of IME-USP. The first was responsible for the first version of *iVProg*, and the second produced a *Software Product Line (SPL)*. After a study, it became clear that the production of a new sistem using the LPS was better than to improve the legacy *iVProg*. With *iVProg* version 2, an experiment was designed to compare the textual programming model with the visual programming model. The method used was the NASA-TLX protocol for the measurement of workload. The results of the experiment suggest that visual programming approach demands less mental effort than the textual programming, from what is conjectured that in the visual model the learners can focus their mental efforts on learning the fundamental concepts of programming.

Keywords: Visual Programming, introduction to programming, *iVProg*, *Web* courses, programming learning, distance learning, virtual learning environments, e-learning.

Sumário

Lista de Abreviaturas	ix
Lista de Figuras	xi
Lista de Tabelas	xiii
1 Introdução	1
1.1 Problemática	2
1.2 Justificativa	4
1.3 Objetivos	6
1.4 Organização do Trabalho	7
2 Fundamentação	9
2.1 Introdução à Programação	9
2.2 Programação Estruturada	10
2.3 Programação Visual	12
2.4 Programação Visual com o iVProg	14
2.4.1 Módulos de Aprendizagem Interativa	14
2.4.2 Aplicativo de Programação Visual Alice	15
2.4.3 Desenvolvimento da primeira versão do iVProg	17
2.4.4 Casos de uso do iVProg	18
2.5 Trabalhos relacionados	20
3 O aplicativo <i>iVProg2</i>	27
3.1 Descrição das funcionalidades	27
3.2 Desenvolvimento do iVProg2	29
3.2.1 Linha de Produto de Software para iMA	29
3.2.2 Projeto e implementação do <i>iVProg2</i>	30
3.3 <i>iVProg2</i> em <i>HTML5</i>	33
3.4 Considerações sobre o processo de desenvolvimento	34
4 O curso experimental	37
4.1 Protocolo <i>NASA-TLX</i>	37
4.2 <i>Virtual Programming Lab</i> - VPL	39
4.3 Sobre o experimento	40
4.4 Resultados do curso experimental	43

4.4.1	Análise dos inscritos	43
4.4.2	Análise dos dados do <i>NASA-TLX</i>	44
4.4.3	Análise do questionário	48
4.5	Considerações finais	49
5	Conclusão e Trabalhos Futuros	51
5.1	Conclusões	51
5.2	Considerações finais	52
5.2.1	Sobre o desenvolvimento do aplicativo	52
5.2.2	Sobre os resultados do curso	53
5.2.3	Sugestões de trabalhos futuros	53
	Referências Bibliográficas	55

Lista de Abreviaturas

PV	Programação Visual (<i>Visual Programming</i>)
iMA	Módulo de Aprendizagem Interativa (<i>Interactive Learning Module</i>)
SGC	Sistema de Gerenciamento de Curso
Moodle	<i>Modular Object-Oriented Dynamic Learning Environment</i>
MOOC	<i>Massive Open Online Course</i>
VPL	<i>Virtual Programming Lab</i>
NASA-TLX	<i>NASA Task Load Index</i>

Lista de Figuras

1.1	Taxas de reprovação/aprovação em Introdução à Programação no IME-USP	6
2.1	Imagem extraída do trabalho de Glinert e Tanimoto (1984) . A figura ilustra um algoritmo que está esperando a entrada de um valor para o cálculo de seu fatorial.	13
2.2	Classificação de ferramentas de Visualização de Sistemas ; imagem adaptada do trabalho de Price et al. (1993) . O tamanho de cada região não é relevante; para manter a imagem simples, nem toda intersecção está presente.	13
2.3	Três estágios para instanciar uma casa de praia no mundo virtual do Alice	16
2.4	Exemplo de construção de algoritmo no Alice para controle do comportamento de um ser no mundo virtual (cão).	16
2.5	Telas do Alice versão 3.	17
2.6	Telas da primeira versão do iVProg	19
2.7	Tela do Scratch no navegador da internet com auxílio da máquina virtual Flash Player	22
2.8	Tela do Snap! Build Your Own Blocks no navegador da internet.	23
2.9	EToys na Área de Trabalho (em tela cheia).	23
2.10	AgentSheets na Área de Trabalho. Na janela a) temos a área destinada à criação do algoritmo que controla o comportamento dos agentes; b) é a janela contém as condições possíveis de serem observadas; por fim, c) é a janela que contém as ações que o agente pode executar.	24
2.11	PyGoGoBlocos na Área de Trabalho. Fonte: https://master.dl.sourceforge.net/project/br-gogo/DOC	24
2.12	Blockly no navegador de internet.	25
2.13	Tela do aplicativo VSE , obtida no artigo de Msiska e van Zijl (2012)	26
3.1	Tela do aplicativo iVProg2 na área de trabalho. O algoritmo ao fundo calcula o maior entre dois números inseridos pelo usuário e o imprime no console cinza.	28
3.2	Tela do aplicativo iVProg2 na área de trabalho. Em evidência está o menu de contexto.	28
3.3	Padrão Composite. A imagem foi adaptada a partir do livro de Gamma et al. (1994)	32
3.4	Ilustração da estrutura de uma expressão aritmética que pode ser construída no interpretador.	32
3.5	Ilustração da estrutura de uma expressão aritmética que pode ser construída no interpretador.	33

3.6	Captura de Tela com o iVProg HTML5 com exercício para imprimir maior inteiro, integrado ao Moodle via iTarefa	34
4.1	Captura da tela do módulo NASA-TLX no Moodle - proximidade entre atividades e protocolo	38
4.2	Captura da primeira tela do protocolo NASA-TLX : atribuição de valores para as escalas individuais.	39
4.3	Captura da segunda tela do protocolo NASA-TLX : uma das telas destinadas à escolha entre pares para estabelecer os pesos de cada escala	40
4.4	Captura de tela com VPL no Moodle	41
4.5	Fluxograma utilizado para explicação de algoritmo de seleção.	42
4.6	Parte de captura de tela do ambiente no qual se deu o curso. Em destaque na imagem está o módulo “Algoritmos”	43
4.7	Comparação entre boxplots gerados a partir dos dados do G1 (22 respostas) e G2 (12 respostas) para o primeiro bloco de atividades	46
4.8	Comparação entre boxplots gerados a partir dos dados do G1 (6 respostas) e G2 (6 respostas) para o segundo bloco de atividades	46
4.9	Comparação entre boxplots gerados a partir dos dados do G1 (5 respostas) e G2 (7 respostas) para o terceiro bloco de atividades	47

Lista de Tabelas

2.1	Comparação do desempenho entre os anos de 2005, 2010 e 2011 em cursos de programação introdutória	19
2.2	Tabela com trabalhos relacionados.	21
3.1	Comparação entre as métricas do arcabouço, <i>iVProg</i> e <i>iVProg2</i>	35
4.1	Distribuição dos voluntários nos grupos G1 e G2	44
4.2	Número de aprendizes do G1 e G2 que não acessaram o sistema	44
4.3	Número de aprendizes que acessaram o sistema na última semana do curso	44
4.4	Teste WMW aplicado em cada uma das escalas ponderadas de G1 e G2 no primeiro bloco de atividades	45
4.5	Teste WMW aplicado em cada uma das escalas ponderadas de G1 e G2 no segundo bloco de atividades	47
4.6	Teste WMW aplicado em cada uma das escalas ponderadas de G1 e G2 no terceiro bloco de atividades	47

Capítulo 1

Introdução

Na história da computação, os computadores já foram artefatos puramente mecânicos. Com o advento das válvulas, se tornaram eletrônicos e, tempos depois, com a criação de circuitos integrados, tornaram-se máquinas digitais (SEP, 2015). Os primeiros computadores foram criados com o propósito da realização de cálculos numéricos em velocidades superiores as que os humanos desenvolviam. Hoje, ao pensarmos nos cálculos efetuados pela máquina, devemos incluir as buscas por palavras em textos, criptografias de mensagens, consultas a arquivos, impressões, desenhos de gráficos, processamento de dados oriundos de sensores etc. Quando falamos na programação de computadores ou na criação de programas para computadores, nos referimos à elaboração de instruções a fim da realização desses cálculos. Criar um programa, ou programar, significa, portanto, elaborar e representar uma sequência de instruções, numa linguagem exequível pela máquina, alterando seu estado e permitindo que um cálculo seja realizado (Pair, 1990). Assim, é a elaboração de um algoritmo que será executado pelo computador, o que torna comum nos referirmos à “construção de um programa” como a “construção de um algoritmo”.

A programação dessas máquinas é uma atividade que existe, pelo menos, desde 1843, quando Ada Lovelace, filha de Lord Byron, o poeta inglês, criou o primeiro programa para um computador mecânico (Toole e Kim, 1999). Ada, criou uma sequência de passos para calcular os números de Bernoulli¹ no Engenho Analítico projetado por Charles Babbage. No entanto, a máquina de Babbage nunca foi construída devido à falta de dinheiro. Ainda assim, em teoria, o Engenho seria capaz de realizar as quatro operações aritméticas básicas (adição, subtração, divisão e multiplicação). A programação da máquina se daria pela perfuração de cartões que, por sua vez, guiariam as engrenagens do aparato durante a realização dos cálculos (Toole e Kim, 1999). Os primeiros computadores eletrônicos, assim como os eletrônicos digitais, eram muito grandes, chegavam a ocupar salas inteiras e pesavam centenas de quilos. Um representante historicamente importante da classe dos eletrônicos digitais é o *Electronic Numerical Integrator and Computer* o famoso ENIAC. Esse foi o primeiro computador eletrônico digital programável de propósito geral. Criado durante a segunda guerra mundial, ocupava um porão de 135m² numa escola da Universidade da Pensilvânia. Dentre seus componentes físicos haviam cerca de 18 000 válvulas eletrônicas, 70 000 resistores, 10 000 capacitores, 6 000 interruptores e 1 500 relés. Na época, sua construção custou mais de quatrocentos mil dólares ao governo americano (Swaine, 2014).

Até o final dos anos 70 os computadores, além de grandes, eram muito caros e relativamente raros. Ainda no final dos anos 70, a programação de computadores era ensinada apenas nas universidades e era vista de maneira enigmática, destinada apenas a especialistas (Fincher e Utting, 2010). Já na década de 80, os computadores não eram mais colossais como o ENIAC; eles começaram a se tornar populares e os modelos desktops tornavam-se populares. Ainda assim, não existiam muitos sistemas operacionais ou programas disponíveis nas máquinas tal como conhecemos hoje, então os usuários precisavam criar os próprios programas. A criação desses programas não era uma tarefa trivial mesmo para estudantes de Ciência da Computação. Com o passar do tempo, os programas

¹Os números de Bernoulli são encontrados na expansão polinomial de algumas funções trigonométricas. Essas funções já foram utilizadas, por exemplo, para construir tabelas de navegação.

passaram a ser representados por linguagens textuais que possuem uma gramática própria e palavras reservadas. Alguns pesquisadores e autores dessa época diziam que a programação era o novo latim do currículo escolar e era vista como uma pedra de amolar para afiar a mente dos estudantes (Sleeman, 1986).

Uma dificuldade intrínseca da Computação foi mencionada por Dijkstra (2015b) numa de suas anotações. Na época, o avanço da tecnologia permitiu um aumento da ordem de 10^9 na capacidade dos cálculos dos computadores. Segundo Dijkstra, ao criar um programa, um indivíduo precisaria refletir em termos de hierarquias conceituais muito mais profundas do que qualquer humano precisou até aquele momento. De seu ponto de vista, escrever programas era uma forma radicalmente nova de pensar e um desafio intelectual que não tinha precedentes na história da humanidade, pois um programa de computador não resolvia um único problema, era, sim, um gerador de soluções para toda uma classe de problemas.

As dificuldades enfrentadas tanto na atividade de programação quanto no processo de sua aprendizagem alimentam, há anos, uma grande quantidade de pesquisas relacionadas a esses temas. Robins *et al.* (2003), por exemplo, fizeram uma rica revisão da literatura sobre os aspectos educacionais e psicológicos envolvidos no processo de aprendizagem de programação de computadores. Para isso, analisaram textos desde a década de 70 buscando por fatores inerentes a evolução de um indivíduo dito “programador iniciante” até chegar no nível conhecido por “programador experiente”. Durante os estudos, encontraram informações sobre dificuldades na resolução de problemas, compreensão de conceitos abstratos (exemplos: variáveis e ponteiros), utilização de recursos específicos de algumas linguagens de programação etc. No geral, aprender a resolver problemas computacionalmente envolve a aprendizagem de novas ferramentas, uma nova forma de pensar (pensamento sistemático) e uma nova forma de escrever a solução de um problema.

Nesse contexto, este trabalho de mestrado visa contribuir no processo de aprendizagem de programação, reduzindo algumas de suas dificuldades. Mais precisamente, trata-se do estudo da aplicação de uma abordagem denominada Programação Visual a fim de mitigar um conjunto específico de dificuldades enfrentadas pelos iniciantes. Neste texto, ao mencionarmos os cursos de Introdução à Programação, ou simplesmente cursos introdutórios, estaremos nos referindo às disciplinas atinentes tanto aos cursos de nível superior de cunho majoritariamente computacional quanto aqueles cursos de nível superior que possuem a computação como uma extensão do currículo (exemplos: Engenharia, Matemática, Física etc). No geral, dentre os objetivos dos cursos introdutórios, estão o ensino da programação de computadores através do estudo de uma linguagem algorítmica e de exercícios práticos. Além disso, quando os cursos são ainda mais profundos, os alunos fazem um estudo, através de exemplos, da correção, da análise de eficiência e do desenvolvimento de algoritmos e de suas estruturas de dados básicas.

Na seção seguinte, apresentaremos a problemática envolvendo as dificuldades no processo de aprendizagem de programação.

1.1 Problemática

Tradicionalmente, os cursos iniciais em programação possuem como componentes curriculares a introdução à lógica de programação, o desenvolvimento de algoritmos utilizando essa lógica e a aprendizagem de ambientes e linguagens para representar, construir e executar os algoritmos desenvolvidos. Pela experiência do grupo de pesquisa no qual este trabalho foi realizado e em conformidade com outros trabalhos (discutidos a seguir), todas essas novidades podem representar um excesso de informação no processo inicial de aprendizagem e podem atrapalhar o objetivo dos cursos introdutórios. Uma consequência grave dessas dificuldades é a elevada taxa de evasão desses cursos. Uma pesquisa usando os dados do Censo da Educação de 2012, indica que a evasão em cursos de ciências exatas chegou a 44% (Souza *et al.*, 2012) (p. 3).

Há muitos fatores determinantes nessa taxa de abandono; Beaubouef e Mason (2005) citam, por exemplo: assessoria ruim antes e durante a graduação (podendo levar o estudante a uma conclusão errada a respeito da área), falta de habilidades em Matemática e em resolução de problemas,

curso mal planejados, falta de treino e de retroação por parte dos tutores etc.

Na literatura sobre aprendizagem de programação, encontramos descrições como sendo um processo árduo o de conhecer e se habituar aos ambientes e às linguagens algorítmicas, com grande dispendiosidade de tempo por parte dos alunos e, por vezes, dos professores ou tutores. No trabalho de Lahtinen (Lahtinen *et al.*, 2005), por exemplo, um questionário foi aplicado a 559 estudantes e 34 professores de cursos de Introdução à Programação de universidades da Europa. As linguagens predominantes nos cursos eram Java e C++. A análise estatística dos resultados mostra que existe uma correlação forte entre: i) a compreensão das estruturas de programação (seleções, laços de repetição etc.), ii) a aprendizagem da sintaxe das linguagens, iii) a compreensão de como construir um algoritmo para resolver um problema e iv) a divisão de uma funcionalidade em procedimentos e funções ou classes. O coeficiente r de correlação obtido foi 0,534 $\leq r \leq$ 0,637 para um intervalo de confiança de apenas 1%. Além disso, os dados indicam que ou o aluno alcança um bom desempenho em todos os itens citados ou apresenta deficiências em todos eles.

No ano de 2006, Anthony Robins, Patricia Haden e Sandy Garner apresentaram a compilação de dados coletados em cursos de Introdução à Programação ministrados entre 2003 e 2004 (Robins *et al.*, 2006). Em 2003, os dados foram obtidos de 250 alunos e em 2004 de 220 alunos. A coleta foi realizada pelos tutores dos cursos e pelos próprios ministrantes durante o esclarecimento de dúvidas nas aulas em laboratório de informática. Quando um aluno chamava por ajuda, o responsável por esclarecer aquela dúvida anotava a natureza dessa dúvida e computava mais uma unidade aos pedidos dessa categoria. A partir da análise dos dados, eles detectaram que a natureza da maior parte das dúvidas era proveniente do que denominaram, no contexto da linguagem Java, *mecânica trivial*. Segundo os autores, a mecânica trivial representa os problemas relacionados ao esquecimento de parênteses, chaves, ponto e vírgula, erros de digitação, ortografia, convenção de nomes de arquivos, esquecimento de importação de classes, formatação de saída, erros de indentação de código e comentários sem utilidade clara. Dentre 28 categorias de problemas, a mecânica trivial representou cerca de 17% do total de 11 240 pedidos de ajuda aos professores e tutores em 2003. Já em 2004 os pedidos de ajuda devidos a mecânica trivial foram de 15,2% do total de 7768. Nos dois anos, as dúvidas de mecânica trivial foram maiores do que as dúvidas relacionadas a outros assuntos. O segundo lugar em 2003 e 2004, por exemplo, foi a categoria de dúvidas chamada “Preso na concepção do programa”. Nessa categoria estavam as dúvidas de como criar o algoritmo usando a linguagem de programação proposta (Java). Nesse caso, os estudantes sabiam como resolver o problema, mas não conseguiam transpor a solução para o computador. Essa categoria de dúvidas representou cerca de 11,12% em 2003 e 8,23% em 2004.

Os trabalhos de Lahtinen e Robins nos mostram alguns problemas relevantes ao processo de aprendizagem e, além disso, evidenciam a sintaxe das linguagens de programação como um desses problemas. Essa observação é reafirmada, ainda, no trabalho de Paul Denny (Denny *et al.*, 2011). Nele, Denny e os outros autores buscam compreender a barreira da sintaxe para os programadores iniciantes. Com o objetivo de conduzir a pesquisa, formularam as seguintes perguntas: “P1) Quão frequente são os erros de sintaxe nas submissões de exercícios? Essa frequência está relacionada ao desempenho dos alunos no resultado final do curso?” e “P2) Até que ponto os erros de sintaxe atrapalham os estudantes no que diz respeito a receber retroação sobre a lógica de seus programas?”. No total, participaram da pesquisa 330 alunos. Os exercícios que resolveram lhes foram apresentados como cabeçalhos de métodos da linguagem Java. Assim, a tarefa era preencher esses métodos adequadamente. Além disso, como um incentivo, cada aluno ganhou 1% na nota final pela participação no projeto extracurricular. Como tarefa, cada estudante deveria criar 1 exercício e responder a pelo menos 10 outros num período de 4 semanas. No final das atividades, os autores classificaram os alunos participantes num *ranking* baseado no desempenho final na disciplina de Introdução à Programação. A partir daí, dividiram os alunos já classificados em quatro quartis (Q1, Q2, Q3 e Q4) e analisaram os dados para responder as perguntas de pesquisa P1 e P2. Os autores verificaram que a frequência de submissões com problemas de sintaxe é de aproximadamente 50% para Q1, cerca de 60% para Q2, 65% para Q3 e 73% para Q4. Mesmo sendo o menor valor, o número de problemas relacionados a sintaxe em Q1 era muito maior do que os pesquisadores

esperavam. Ademais, o fato do sistema prover a retroação automaticamente implica no não fornecimento de retroação se o código do aluno não puder ser compilado e executado, o que caracteriza um problema ainda maior. Ao analisar os dados para encontrar uma resposta a P2, os autores perceberam uma clara tendência a falha ao final do curso para aqueles alunos que não conseguiram compilar e executar seus códigos durante o experimento.

Além das evidências de que a sintaxe das linguagens representa um problema para iniciantes em programação, também existem trabalhos que buscam compreender melhor o comportamento dos aprendizes durante a compilação de seus códigos (Jadud, 2006) e a influência das mensagens de erros fornecidas pelos compiladores no momento da aprendizagem (Nienaltowski *et al.*, 2008). Esses estudos mostraram que o tempo dispendido para corrigir os erros e compilar os códigos dependem do nível intelectual dos alunos e do tipo de erro encontrado. Além disso, as mensagens de erros apresentadas pelos compiladores, ainda que modificadas para serem mais completas na direção de propor uma solução, não influenciam positivamente na aprendizagem do objeto de estudo. Dessa forma, as dificuldades enfrentadas e o tempo dispendido com o compilador não são significativos para o aprendizado do aluno, portanto representam apenas um fator complicador do processo de ensino e de aprendizagem.

Como citado anteriormente, a experiência do grupo de pesquisa ao qual o autor deste texto pertence e as experiências relatadas por pesquisadores encontradas na literatura, indicam que a aprendizagem de muitos conceitos e de novos ambientes, de modo concomitante, pode ser um fator complicador durante a aprendizagem de programação. Evidentemente, a linguagem de programação também pode ser um dos objetos de estudo nos cursos introdutórios. Entretanto, nossa hipótese é de que mitigando a complexidade de alguns desses elementos no processo inicial da aprendizagem, o estudante poderá se concentrar em uma dificuldade de cada vez. A partir disso, em 2009 foi desenvolvido, no IME-USP, um aplicativo de programação visual chamado iVProg. Na dissertação de mestrado de Kamiya (2010) encontramos a descrição e os resultados dos primeiros testes com o aplicativo. Entretanto, assim como descrevemos na seção 2.4, o iVProg apresentava alguns problemas devido ao código legado e método de desenvolvimento. Em 2012, uma outra dissertação de mestrado tinha como objetivo melhorar: o processo de desenvolvimento, a qualidade do código e a estabilidade dos aplicativos de uma família de sistemas educacionais. O iVProg estava dentre esses aplicativos. Assim, este trabalho dá continuidade aos trabalhos desenvolvidos nessas duas dissertações, reconstruindo o aplicativo de programação visual usando as técnicas propostas por Dalmon (2012) e validando o seu uso num curso experimental de introdução à programação.

Na próxima seção, justificaremos a importância do trabalho desenvolvido nessa direção.

1.2 Justificativa

A importância da criação dos computadores é inegável. Foi graças a eles que já exploramos, em diferentes escalas, o planeta Terra e uma pequena fração do Universo. Conhecer seu funcionamento e entendê-lo como uma extensão das capacidades cognitivas humanas permite que a Ciência avance de forma veloz, superando barreiras antes impostas pelo simples fato de sermos humanos. Em seu artigo publicado na internet, o ex-professor da Universidade de São Paulo e atual professor da Universidade de Stanford, Paulo Blinkstein, afirma que estamos num período de transição na Ciência (Blinkstein, 2008). No parágrafo a seguir ele elucida a importância do domínio dos computadores e a compreensão de suas capacidades:

“Estamos em uma época de transição no mundo científico, em que o pensamento computacional está transformando profundamente a academia e a indústria. Hoje em dia, um cientista em um laboratório de pesquisa de ponta em nada lembra o estereótipo do cientista do século XIX, com seu avental branco, trancado em um laboratório com tubos de ensaio. Em vez disso, ele provavelmente passa a maior parte do tempo em frente a um computador, construindo e estudando modelos computacionais. Um engenheiro industrial, ao tentar redesenhar a linha de produção, não usa só papel e lápis, usa modelos computacionais. Um economista tentando fazer uma projeção de inflação não faz

as contas de cabeça, usa, claro, modelos. A primeira etapa do “pensar computacionalmente” é identificar as tarefas cognitivas que podem ser feitas de forma mais rápida e eficiente por um computador.”

As afirmações de Blinkstein ficam ainda mais claras e são reforçadas com as ações, por exemplo, do governo estadunidense. Em 2013, foi publicado pela Casa Branca o documento com o plano nacional de educação para as áreas de Ciência, Tecnologia, Engenharia e Matemática (conhecidas na literatura como STEM, acrônimo de Science, Technology, Engineering, and Mathematics) (NSTC, 2013). O presidente Barack Obama abriu o discurso com a frase “Avanços na Ciência, Tecnologia, Engenharia e Matemática têm sido, há muito tempo, essenciais para nossa habilidade na fabricação de produtos melhores e mais inteligentes, para a melhoria na saúde pública, o desenvolvimento de fontes de energia mais limpas e eficientes, para a preservação do ambiente, segurança nacional e o crescimento da economia” (NSTC, 2013). No documento há planos de investimentos para a capacitação em informática e tecnologia de 100 mil professores do ensino básico. Afinal, hoje em dia, o raciocínio lógico e a programação de computadores são propostas como habilidades fundamentais para os alunos e sua introdução nos estágios iniciais da educação tem sido adotada em uma taxa crescente. O motivo dessa adoção é atrair uma maior quantidade de crianças para essas áreas no futuro.

Essa demanda por profissionais das ciências (Ciências Exatas e Naturais) é uma realidade em várias das regiões do mundo, incluindo o Brasil. O mercado nacional para os recém formados é bastante favorável e crescente. Em especial, nas áreas de Tecnologia da Informação (TI), segundo uma pesquisa realizada pelo IDC², nosso país já está entre os quatro maiores mercados de Tecnologia da Informação e Comunicação (TIC) do mundo e movimentou cerca de US\$ 233 bilhões em 2012. Esse valor seria equivalente a 51% dos gastos totais com TI da América Latina (BRASSCOM, 2013). Além disso, em entrevista, o presidente da Associação Brasileira das Empresas de Tecnologia da Informação e Comunicação (BRASSCOM), afirmou que o setor de TIC emprega 2,5 milhões de pessoas e a previsão é que o setor demande, na próxima década, mais 1 milhão de pessoas capacitadas (Digital, 2012). Em 2014, outra pesquisa do IDC mostra que mesmo com a crise mundial, copa do mundo e outros fatores desfavoráveis ao crescimento do mercado nacional, o setor cresceu 8,7% entre o segundo semestre de 2013 e primeiro semestre de 2014 (IDC, 2014).

O que gostaríamos de evidenciar é a importância das ciências para o desenvolvimento de uma nação e, também, a programação como componente curricular para colaborar no desenvolvimento dessas ciências. Entretanto, vimos na seção anterior que o processo de aprendizagem de programação pode ser árduo e moroso devido à quantidade de objetos de estudos com os quais os estudantes precisam lidar em concomitância. Todavia, a natureza do problema apresentado pode trazer consequências indesejáveis aos cursos introdutórios ou, ainda, à própria área de Ciência da Computação.

Estudos multi-institucionais comprovam que, mesmo depois de frequentar várias disciplinas envolvendo programação, os alunos ainda possuem várias deficiências em tópicos elementares (Kinnunen e Malmi, 2006). Além disso, alguns alunos têm dificuldade até mesmo para compreender uma simples linha de atribuição de valor a uma variável (Simon, 2011). Ao encontrar muitas dificuldades nos cursos introdutórios, esses alunos ficam desmotivados, o que prejudica a aprendizagem. Pelo impedimento do aprendizado, as dificuldades acarretam em reprovação, o que pode gerar altas taxas de evasão nos cursos de programação (Kinnunen e Malmi, 2006).

Para compreender os índices de reprovação e aprovação nas disciplinas introdutórias, coletamos dados do instituto com o qual autor e orientador possuem vínculo. Os dados foram obtidos a partir do sistema administrativo de cursos da Universidade de São Paulo (USP), denominado JúpiterWeb. Os índices estão registrados na figura 1.1. No período do primeiro semestre de 2007 até o primeiro semestre de 2011, o índice médio de reprovação foi de 47% com pico de 62,9% no segundo semestre de 2009.

²A IDC é uma empresa cujo objetivo é a realização de pesquisas no mercado de Tecnologia da Informação e Comunicação (TIC). <http://www.idcbrasil.com.br/>. Último acesso: 05/01/15.

Figura 1.1: Taxas de reprovação/aprovação em Introdução à Programação no IME-USP

De aspecto mais amplo, Bennedsen e Caspersen (2007) coletaram dados sobre a taxa de falha nos cursos introdutórios de 63 instituições de ensino distribuídas ao redor do mundo. A princípio, o pedido de dados sobre reprovação foi enviado a 575 instituições, porém apenas 16,8% responderam ao pedido e 12,7%, incluídos nos 16,8%, enviaram os dados corretamente. A partir dos dados, calcularam uma taxa média de 33% de reprovação nos cursos introdutórios. Porém, afirmaram a existência de instituições que relataram taxas de 72% e 90% de falha nos Estados Unidos. Os autores especulam que muitas instituições não responderam ao questionário por terem taxas muito elevadas, o que pode lhes representar um motivo de constrangimento.

Agora que contextualizamos o problema e mostramos evidências de sua significância, na próxima seção iremos explicar o objetivo deste trabalho de mestrado.

1.3 Objetivos

Por estar inserido na área de Informática na Educação (IE), a essência desta dissertação é estudar e propor uma solução a um problema educacional utilizando-se de recursos computacionais. Dessa forma, a partir da problemática e da justificativa, podemos definir o **objetivo geral** como: “reconstruir e usar o aplicativo de programação visual *iVProg* e comparar sua carga de trabalho em relação a programação textual”.

Na direção de cumprir o objetivo, aplicamos a técnica proposta em (Dalmon, 2012) durante a reconstrução do aplicativo. Além disso, realizamos uma comparação para avaliar o código-fonte das versões do *iVProg* (seção 3.4). Depois de reconstruído, elaboramos um estudo comparativo entre as abordagens de *Programação Textual* e *Programação Visual*. Assim, para coletar dados oriundos da carga de trabalho de uma abordagem em relação à outra, criamos e aplicamos um curso online, aberto e de cunho experimental. Nesse curso, uma análise das dificuldades encontradas e da carga de trabalho nos forneceu algumas evidências que serão discutidas na seção 4.4. Escolhemos um curso online para aproximar o cenário experimental de uma situação real de ensino nessa modalidade (Educação à Distância) e, também, como uma tentativa de obter o maior volume de dados possível. Além disso, a abordagem online para o curso se mostrou mais adequada devido a grande demanda de tempo que seria incorrida aos voluntários.

Durante o trabalho para cumprirmos o objetivo, alguns subprodutos desta dissertação foram surgindo. A criação desses subprodutos foi direcionada por objetivos mais específicos. Dentre eles, destacamos os seguintes:

- *Criar material instrucional para o curso de Introdução à Programação*: um material que abrange vídeos instrucionais, atividades interativas e textos para referência teórica foram

criados. Esse material poderá ser facilmente distribuído, alterado ou reutilizado, pois está disponível sob licença *Creative Commons* 3.0 na internet.

- *Criar tutoriais de instalação da máquina virtual Java nos ambientes Linux e Windows*: com a finalidade de evitar grandes complicações na instalação da máquina virtual *Java* durante a realização do curso experimental, criamos tutoriais em vídeos mostrando todos os passos: verificação se já possui o *plugin* do *Java* instalado, download adequado para a arquitetura do processador, instalação e teste da máquina virtual *Java*.
- *Criar um interpretador para executar o código gerado pelo iVProg*: durante a reconstrução da ferramenta *iVProg*, foi necessário o planejamento e construção de um interpretador da linguagem visual para a execução dos algoritmos construídos no aplicativo. Esse interpretador é suficientemente flexível para ser estendido ou utilizado em qualquer outra aplicação *Java*.
- *Criar um módulo do protocolo NASA-TLX para Moodle*: O protocolo *NASA-TLX* é um conjunto de questões com a finalidade de mensuração indireta da carga de trabalho durante a realização de uma tarefa. Na seção 4.1, descrevemos o protocolo e como o utilizamos no projeto. Como não existia uma forma adequada de aplicar o protocolo dentro do ambiente *Moodle*, criamos um módulo para aplicá-lo.

Na seção seguinte, há uma descrição da organização do texto.

1.4 Organização do Trabalho

No Capítulo 2, apresentamos os conceitos fundamentais a compreensão deste texto. Nele explicamos os conceitos fundamentais das disciplinas de *Introdução à Programação* (2.1), definimos *Programação Estruturada* (2.2) e *Programação Visual* (2.3), apresentamos o *iVProg* (??) e o aplicativo do qual teve origem (2.4.2). No final do capítulo, apresentamos os trabalhos relacionados (2.5).

No capítulo 3, são apresentados as técnicas utilizadas no desenvolvimento do aplicativo *iVProg2* e dissertamos sobre o seu processo de desenvolvimento (3.2). No final do capítulo, fazemos as considerações a respeito de todo o processo (3.4).

No capítulo 4 apresentamos o curso experimental realizado. Além disso, explicamos sobre o método indireto de aferição da carga de trabalho incorrida aos usuários enquanto realizavam as atividades do curso (4.1). Por fim, analisamos os dados coletados e discutimos os resultados (4.4).

Finalmente, no Capítulo 5 discutimos o trabalho realizado, sintetizamos as conclusões (5.1), fazemos nossas últimas considerações (5.2) e deixamos as sugestões de trabalhos futuros (5.2.3).

Capítulo 2

Fundamentação

Neste capítulo de fundamentação trataremos os conceitos necessários para a compreensão da pesquisa. Assim, começaremos explicando sobre os cursos de Introdução à Programação (2.1), citando o que pode influenciar seu currículo; como exemplo, falaremos da forma de se modelar um programa (paradigma de programação). Daremos mais atenção ao que chamamos de Paradigma de Programação Estruturada (2.2), pois é neste paradigma que o *iVProg* se baseia. Além disso, neste capítulo, falaremos da Programação Visual (2.3) e discutiremos o seu uso no *Alice* (2.4.2) e na primeira versão do *iVProg* (2.4), bem como a relação entre esses dois aplicativos. Também há uma seção onde apresentaremos uma lista de trabalhos relacionados que foram encontrados a partir de uma busca na literatura, apresentando suas contribuições para esta dissertação.

2.1 Introdução à Programação

São nos cursos de Introdução à Programação onde normalmente ocorrem os primeiros contatos dos aprendizes com a lógica e os conceitos básicos da programação de computadores. Esses cursos geralmente são ministrados como disciplinas de um ou dois semestres nas graduações de cunho majoritariamente computacional (exemplos: Bacharelados em Ciência da Computação, Sistemas de Informações etc.) ou nas graduações que possuem a computação como uma extensão do currículo (exemplos: Engenharia, Matemática, Física etc.).

Existem muitos fatores que influenciam nas abordagens desses cursos que vão desde a demanda do mercado de trabalho até a cultura e objetivos das instituições de ensino nas quais esses cursos são realizados. O conteúdo das disciplinas introdutórias depende desses fatores que são determinados no momento de seu planejamento (Pears *et al.*, 2007). Por exemplo, a escolha de um paradigma de programação delimita as linguagens que poderão ser utilizadas, as técnicas para a resolução dos problemas e as ferramentas que apóiam os processos de ensino e aprendizagem.

Seguindo a definição de paradigma de Thomas Kuhn (Kuhn, 2001), a ciência da computação também possui os seus. No livro sobre ensino de programação de Hazzan (Hazzan *et al.*, 2011), os autores, a partir de muitos outros trabalhos, definem *paradigmas de programação* como “heurísticas usadas para resolver problemas algorítmicamente. A partir de um paradigma de programação analisa-se o problema com um ponto de vista específico e, baseando-se nessa análise, propõe-se uma solução para o problema, organizando-a em blocos e definindo a relação entre eles”. Dentre os modelos mais conhecidos e utilizados nas fases iniciais da aprendizagem, em 2001 a força tarefa da ACM e IEEE destacou os de “Programação Estruturada”, “Programação Funcional” e “Programação Orientada a Objetos” (Curricula, 2001). Em 2013, novamente reforçam o mérito desses paradigmas e, além disso, também acrescentam que uma mudança para linguagens mais seguras, gerenciáveis e dinâmicas está acontecendo (Curricula, 2013). A seguir, explicamos um pouco sobre cada um deles a partir da ótica de Tucker e Noonan (2009) e Baranauskas (1994):

- *Programação Estruturada*: neste paradigma as variáveis e o programa são armazenados juntos no computador; a sequenciação de comandos é um dos fatores determinantes para a abstração

do problema durante seu processo de resolução. Além da sequência dos comandos, os conceitos de laço e estrutura de decisão são muito importantes por determinarem o fluxo de execução dessa sequência. Usando essas estruturas, o programador define como o algoritmo será executado pelo computador a fim de alterar os estados do problema. Dessa forma, assim como foi citado por Baranauskas (1994), nesse paradigma programar significa dar ordens ao computador que serão executadas sequencialmente. São exemplos de linguagens de programação estruturada: *C*, *Fortran*, *Cobol*, *Pascal* etc.

- *Programação Funcional*: nesse caso, o programa é modelado como uma coleção de funções matemáticas e evita a alteração de estados e valores. Nesse paradigma, as funções possuem uma entrada (domínio) e podem fornecer uma saída (imagem ou faixa). Essa forma de modelar aproxima a programação da linguagem matemática e uma linha de atribuição da programação estruturada como, por exemplo, “variavel = variavel + 1” não faz sentido nem nas linguagens que seguem esse paradigma nem na própria matemática. São exemplos de linguagens de programação funcional: *Haskell*, *Lisp*, *Scheme*, etc.
- *Programação Orientada a Objetos*: aqui, o programa é modelado como uma coleção de objetos. Esses objetos possuem dados que são armazenados em seus campos. Uma das propriedades desse paradigma é que a alteração desses campos só deve ser realizada pelo próprio objeto a partir de uma mensagem emitida para ele. Por exemplo, se na modelagem de um programa existe um objeto do tipo “computador” e ele tiver um campo que armazena a sua marca, então a marca desse computador será definida por meio de uma mensagem que cuida apenas dessa tarefa. Segundo Baranauskas (1994), o objetivo da orientação a objetos é simular o mundo real, que é povoado por objetos que não sabemos, necessariamente, como funcionam internamente, mas mesmo assim podemos interagir com eles a partir de algum tipo de interface (botões, interruptores etc). São exemplos de linguagens de programação que seguem a Orientação a Objetos: *Smalltalk*, *Java*, *C++*, etc.

Na introdução deste trabalho, definimos a programação como uma atividade na qual devemos “elaborar e representar uma sequência de instruções, numa linguagem exequível pela máquina, alterando seu estado [da máquina] e permitindo que um cálculo seja realizado”. Ao adotarmos essa definição, postulamos que o paradigma abordado é o de Programação Estruturada. O principal motivo para escolhermos esse se dá pela sua utilização no contexto de trabalho do autor e orientador desta dissertação de mestrado. Assim, o trabalho realizado irá na direção de mitigar as dificuldades relacionadas à sintaxe das linguagens de programação estruturada. Portanto, quando descrevermos e mencionarmos os processos estaremos sempre nos referindo à Programação Estruturada, a não ser que declaremos explicitamente que não estamos falando dela.

Na seção seguinte falamos sobre a programação estruturada e alguns de seus conceitos a fim de permitir seu mapeamento nos recursos visuais tais como os propusemos no *iVProg2*.

2.2 Programação Estruturada

Quando os programas começaram a ser armazenados na memória dos computadores, sua criação se dava por comandos em linguagem de máquina (ou linguagem de montagem), conhecida como *Assembly*. Por estar muito próxima da máquina, essa linguagem é dita ser de baixo nível. Um dos problemas desse tipo de linguagem é a complexidade de sua notação, o que resulta em dificuldades de leitura e compreensão do que o programa faz. Quanto mais complexo o problema modelado em *Assembly*, mais complexa é sua representação. Dessa forma, havia a necessidade de linguagens que pudessem ser mais facilmente compreendidas por seres humanos. Assim, linguagens “mais legíveis” e “compreensíveis” começaram a surgir. Dizemos que essas linguagens são de alto nível (Wexelblat, 2014).

Ainda assim, apesar de serem mais legíveis, essas linguagens possuem recursos herdados das linguagens de baixo nível. Um deles, conhecido como *goto*, permite a mudança do fluxo de execução

de um algoritmo para algum ponto específico do código-fonte, indicado por um rótulo definido pelo programador. Entretanto, esse recurso pode deixar um programa bastante confuso se for usado de forma não comedida. Veja um exemplo do uso de *goto*, na linguagem de programação *Pascal* (alto nível), retirado do sítio [TutorialsPoint \(2015\)](#):

```

1 program exGoto;
2 label 1;
3 var
4   a : integer;
5 begin
6   a := 10;
7   (* repeat until loop execution *)
8   1: repeat
9     if( a = 15) then
10      begin
11        (* skip the iteration *)
12        a := a + 1;
13        goto 1;
14      end;
15      writeln('value of a: ', a);
16      a:= a +1;
17    until a = 20;
18 end.
```

Esse algoritmo imprime uma sequência de números de 1 a 19. O comando *goto*, na linha 13, precisa de um rótulo para o qual redireciona a execução do algoritmo. Nesse caso, o rótulo é o número 1, da linha 8; neste ponto se define um laço de repetição, do qual se sai e para o qual se retorna. A saída produzida é a seguinte:

```

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Na história da programação, o próprio professor Edsger Dijkstra nunca subestimou a complexidade dos problemas que poderiam ser resolvidos com o uso dos computadores, tampouco subestimou a dificuldade de se escrever um programa. Entre as décadas de 1960 e 1980, devido à desordem que o comando *goto* poderia gerar em programas mais longos, a comunidade científica começou a discutir maneiras de se evitar o uso desse comando. Um dos documentos bastante conhecidos e que critica o uso de *goto* é o *Go-To Statement considered harmful* de [Dijkstra \(1979\)](#), que também pode ser lido sob título *A case against GO TO statement* ([Dijkstra, 2015a](#)). Dijkstra destacou a dificuldade dos seres humanos em compreender e dar manutenção a uma estrutura demasiadamente grande e complexa, tal como um programa com muitos comandos do tipo *goto*. Dessa forma, para simplificar e organizar a estrutura de um programa, novas linguagens e metodologias foram sendo criadas. Essa busca pela simplificação da estrutura e pela organização e clareza dos programas originou o que hoje conhecemos como **Programação Estruturada**.

Outro fator importante que contribuiu para a diminuição do uso do comando *goto* foi a demonstração feita por Böhm e Jacopini, o **Teorema de Programação Estruturada** ([Böhm e Jacopini, 1966](#)). Esse teorema garante que todo programa *próprio* pode ser escrito com apenas três tipos de estruturas de controle: sequenciais, seletivas e repetitivas. Diz-se que um programa é *próprio* quando: possui somente um ponto de entrada e saída (ou fim) para controle do programa; existem caminhos da entrada até a saída e que passam por todas as partes do programa; e, por fim, possui todas as instruções executáveis, não possuindo laços infinitos ([Aguilar, 2008](#)).

Além da redução a essas três estruturas de controle de fluxo, a programação estruturada também preconiza a organização do programa em módulos, ou seja, em subprogramas especializados. Esses subprogramas são, por vezes, chamados de funções ou subrotinas. A seguir, apresentamos um exemplo de função especializada em calcular o n -ésimo termo da sequência de Fibonacci. Para $n \in \mathbb{N}$, os n primeiros termos da sequência de Fibonacci (F_n) são dados por:

$$F_n = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2; \\ F_{n-1} + F_{n-2} & \text{se } n > 2. \end{cases}$$

Note que neste pequeno trecho de código encontramos os conceitos de estrutura de seleção (**if/else**), de repetição (**for**) e de sequências de comandos dentro do laço para obter o n -ésimo termo (`f_n`).

```
1 //Cálculo do n-ésimo termo de Fibonacci
```

```

2 int fibonacci(int n) {
3   int f_n = 0;
4   int f_1 = 1;
5   int f_2 = 1;
6   int i = 0;
7   if(n == 1 || n == 2) return 1;
8   for (i = 3; i <= n; i++) {
9     f_n = f_1 + f_2;
10    f_2 = f_1;
11    f_1 = f_n;
12  }
13  return f_n;
14 }

```

Para este trabalho, os conceitos e ideias do paradigma de programação estruturada são importantes para justificar as funcionalidades do *iVProg2*. Na seção 3.2.2 são apresentadas explicações sobre as estruturas de seleção, repetição e sequências de comandos possíveis no aplicativo. Na próxima seção, explicamos sobre a *Programação Visual*, trazendo sua definição e exemplos.

2.3 Programação Visual

Historicamente, a *Programação Visual* (PV) começou a aparecer no final da década de 1970. Glinert e Tanimoto (1984), por exemplo, relataram que nos primeiros computadores havia uma coleção de programas “enlatados”. Dessa forma, alguns usuários sentiam vontade, por vezes necessidade, de criar os próprios programas. Entretanto, essas pessoas se surpreendiam com a meticulosidade exigida pela programação e se frustravam com as dificuldades enfrentadas nessa atividade. Além disso, argumentamos na introdução deste texto, especificamente na seção 1.2, que os computadores se tornaram instrumentos de trabalho em muitas áreas do conhecimento. Entretanto, um profissional de uma área que não tenha uma ligação estreita com a Ciência da Computação pode encontrar dificuldades ao se deparar com a necessidade de criar um modelo computacional usando as ferramentas tradicionais da programação. Situações desse tipo serviram de motivação para as pesquisas que deram origem ao recurso conhecido como *Programação Visual* (PV), como o próprio trabalho Glinert e Tanimoto, que criaram o *Pict*, baseado em ícones, que foi um dos primeiros de seu tipo (figura 2.1).

A *Programação Visual* tem o mesmo objetivo de uma linguagem de programação tradicional. Entretanto, na PV o programador utiliza recursos como ícones, fluxogramas, blocos de código ou representações digitais de objetos do mundo real para projetar seu algoritmo. Uma definição de PV é dada por Myers (1990) como “qualquer sistema que permite ao usuário representar um programa em duas (ou mais) dimensões”. No mesmo parágrafo Brad Myers também explica que “linguagens de programação tradicionais [texto] não são consideradas de duas dimensões porque os compiladores ou interpretadores as processam como uma cadeia de caracteres unidimensional”.

Em seu trabalho Brad também alerta para a confusão existente entre os conceitos de *Programação Visual* e *Visualização do Aplicativo*. Pensando nesses termos como conjuntos, no primeiro temos os aplicativos com recursos visuais para criar um programa e entregá-lo à máquina a fim de que seja compilado ou interpretado. A *Visualização do Aplicativo* se trata de um conjunto mais amplo do a *Programação Visual*. Para compreendermos melhor essa classificação, é importante explicarmos o contexto no qual ela está inserida na Ciência da Computação. A PV é uma forma de *Visualização do Aplicativo* que, por sua vez, está contida num conjunto maior, denominado *Visualização de Sistemas*. Segundo Gračanin *et al.* (2005), a “Visualização de Sistemas (VS) é a área que investiga as abordagens e técnicas para representações gráficas estáticas ou dinâmicas de algoritmos, programa (código) e dados processados. A VS se preocupa, principalmente, com a análise de programas e seu desenvolvimento. O objetivo é melhorar nossa compreensão sobre o inerentemente invisível e intangível aplicativo [...]. O maior desafio é encontrar uma maneira efetiva de mapear os aspectos de um aplicativo em representações gráficas usando metáforas visuais”. Além da PV, existem outras formas de VS, tais como foram apontadas por Price *et al.* (1993) em sua

Figura 2.1: Imagem extraída do trabalho de *Glinert e Tanimoto (1984)*. A figura ilustra um algoritmo que está esperando a entrada de um valor para o cálculo de seu fatorial.

revisão da literatura. A figura 2.2 foi adaptada de seu texto e mostra os tipos de VS que foram encontrados por ele e demais autores na literatura daquela época. Ainda assim, na tese de doutorado de *Sorva (2012)*, o autor constatou que não surgiram trabalhos que fugissem dessa classificação.

Figura 2.2: Classificação de ferramentas de **Visualização de Sistemas**; imagem adaptada do trabalho de *Price et al. (1993)*. O tamanho de cada região não é relevante; para manter a imagem simples, nem toda intersecção está presente.

Na figura 2.2 podemos perceber que existem dois principais tipos de VS: *Visualização do Al-*

goritmo e *Visualização do Aplicativo*. Os recursos de *Visualização de Algoritmos* são utilizados principalmente para mostrar o resultado da execução de um algoritmo, passo a passo (exemplo: mostrar os estados de um vetor durante a execução de um algoritmo de ordenação; mostrar a aparência de uma árvore binária durante a execução de uma operação de adição ou remoção de nó). Sobre a *Visualização do Aplicativo*, nesse conjunto estão recursos visuais que permitem ao usuário enxergar o estado dos dados ou do código de forma dinâmica ou estática. Note que na figura 2.2 a *Programação Visual* está numa intersecção entre *Visualização estática do código* e *Visualização estática dos dados*. Na seção 2.5 traremos exemplos desses aplicativos.

Dentro do sistema de classificação de Price *et al.* (1993), o aplicativo *Alice* se encaixa na intersecção entre *Programação Visual* e *Visualização estática do código*, entretanto, também possui recursos de *Animação do código*. Ainda assim, podemos classificá-lo conforme o principal objetivo do aplicativo, que no caso do *Alice* é o ensino de programação, portanto o código-fonte é o foco durante a classificação. A mesma estratégia vale para o *iVProg*, que pertence ao conjunto de intersecção de *Programação Visual* com *Visualização estática do código*.

Apesar do objetivo ser de permitir que usuários não especialistas em programação consigam, até certo ponto, programar, a PV não surgiu com propósitos educacionais. Ainda assim, na literatura podemos encontrar muitos trabalhos com PV que vão no sentido de apoiar o processo de aprendizagem de programação; por exemplo os trabalhos de Carlisle *et al.* (2004a) e Kölling (2008a). Além disso, experimentos realizados com ambientes de programação como o *Alice*, nos dizem que além de eliminar as preocupações dos iniciantes com a sintaxe das linguagens de programação tradicionais, o uso do aplicativo deixou os aprendizes motivados durante o processo de aprendizagem (Adams, 2007).

Na seção seguinte, apresentamos os principais trabalhos relacionados à esta dissertação, pois foram os aplicativos que deram origem ao trabalho realizado aqui.

2.4 Programação Visual com o iVProg

A primeira versão do *iVProg* começou a ser desenvolvida em 2009 e foi trabalho de mestrado de Reginaldo Rideaki Kamiya (Kamiya, 2010). O aplicativo foi desenvolvido como um *Módulo de Aprendizagem Interativa* e foi construído a partir do *Alice*, criado na Universidade Carnegie Mellon. A seguir, descrevemos o conceito de *iMA* e o *Alice*. Por fim, apresentamos a primeira versão do *iVProg*.

2.4.1 Módulos de Aprendizagem Interativa

Um *Módulo de Aprendizagem Interativa*, cujo acrônimo é *iMA*, basicamente é um aplicativo com finalidade educacional, geralmente desenvolvido com a tecnologia *Java* ou *HTML+JavaScript+CSS* e que atende, por definição, a duas características essenciais:

1. poder ser carregado por navegadores *Web/Internet* (e rodar);
2. possuir um protocolo de comunicação com o servidor *Web*.

Historicamente, os *iMA* são desenvolvidos desde os anos 2000 no IME-USP, sob coordenação do Prof. Dr. Leônidas de Oliveira Brandão (orientador deste trabalho de mestrado). Os principais objetivos dessa família de aplicativos estão relacionados aos processos de ensino e de aprendizagem. Atualmente existem cinco (5) *iMA*: *iGeom* para o ensino de geometria (Isotani, 2005); *iGraf* usado no ensino de gráficos de funções (do Prado, 2008); *iComb* destinado ao ensino de análise combinatória (Eisenmann, 2009); *iCG* no ensino de fundamentos de programação de baixo nível (ICG, 2015); e o *iVProg* destinado ao ensino de fundamentos de programação de alto nível (Kamiya, 2010). Todos esses *iMA* foram desenvolvidos com o uso da tecnologia *Java* e funcionam como *Applets* (Oracle, 2015a).

No processo de ensino, os *iMA* visam melhorar o trabalho do professor. Ao funcionarem como *Applets*, os *iMA* podem ser executados em navegadores de internet desde que o usuário possua o

plugin da máquina virtual *Java* instalado. Assim, eles podem ser integrados à um *Sistema Gerenciador de Cursos* (SGC) como o *Modular Object Oriented Dynamic Learning Environment* (*Moodle*). Esse SGC possui o código livre e tem a maior comunidade de usuários entre os SGC, atingindo mais de 200 países com mais de 50 mil sítios registrados junto à comunidade oficial do sistema ([Moodle, 2015](#)).

A integração com o *Moodle* é possível graças a um módulo desenvolvido pelo grupo de pesquisa do professor Leônidas, chamado *iTarefa* ([Rodrigues, 2011](#)). Ao utilizar o *iTarefa*, um professor pode criar atividades interativas usando os *iMA* existentes. No geral, os *iMA* possuem um recurso de correção automática das atividades, o que poupa trabalho do professor em corrigir um número muito grande de tarefas e permite que se concentre em outras atividades (exemplo: ajuda aos aprendizes com esclarecimento de dúvidas). Além disso o *iTarefa*, combinado com os *iMA* e o *Moodle*, pode gerar relatórios sobre as atividades realizadas pelos aprendizes, centralizando, evidenciando e resumizando as informações relevantes a respeito do desempenho dos aprendizes e suas ações no sistema.

Além dos *iMA* visarem a melhoria do trabalho do professor, no processo de aprendizagem têm como objetivo prover aos aprendizes um ambiente instigante e motivador. Com grande interatividade, os *iMA* permitem que os aprendizes investiguem, experimentem e conjecturem a respeito dos objetos de estudo (geometria, gráficos de funções, combinatória, etc.). Além disso, a correção automática das atividades representa um grande benefício, pois permite que os aprendizes identifiquem rapidamente quais assuntos precisam estudar mais e com mais atenção ([Koedinger e Alevan, 2007](#)).

Com a motivação do ensino de programação de alto nível, a primeira versão do *iVProg* começou a ser desenvolvida a partir do *Alice*. Descrevemos o processo a seguir.

2.4.2 Aplicativo de Programação Visual Alice

O *Alice* é um ambiente de programação bastante completo e complexo. Como foi criado na universidade Carnegie Mellon, até a versão 2.4, ele está totalmente em inglês. Nele os aprendizes podem criar seus algoritmos visualizando o resultado numa animação em 3D de um mundo virtual.

Num primeiro momento, por volta do ano 2000, os algoritmos eram construídos no *Alice* por meio de comandos de texto, parecidos com o da linguagem *LOGO*, criada em 1967 por Seymour Papert ([LOGO, 2015](#)). Entretanto, no caso do *Alice*, os pesquisadores observaram que os usuários sem experiência em programação ou com pouca fluência digital tinham dificuldades para digitar os comandos ou utilizar o mouse para algumas tarefas ([Conway et al., 2000](#)). Esses fatos motivaram a adaptação do aplicativo para a PV.

Depois de implementarem a PV, o mundo virtual pode ser populado por seres e objetos. As propriedades dos objetos podem ser alteradas com o uso do *mouse*. Para isso, os usuários devem arrastá-los a partir de uma bandeja do aplicativo até a janela do mundo virtual. A figura 2.3 mostra essa situação em três estágios: na tela 1 o usuário clica numa casa de praia e começa a arrastá-la; na tela 2 o usuário já está com o objeto sobre a janela do mundo virtual e na tela 3 o usuário já soltou o objeto na posição desejada.

A construção dos algoritmos para controlar os entes visuais do *Alice* se dá de maneira análoga. Os aprendizes precisam arrastar os blocos de comando a partir de uma bandeja do aplicativo até a área destinada à construção dos algoritmos. A figura 2.4 mostra uma tela *Alice* com alguns blocos de comando utilizados para movimentar um cão. Abaixo do mundo virtual existe um campo no qual o usuário cria seu primeiro método, que é equivalente à função principal de algumas linguagens de programação (*Java* e *C*, por exemplo). Na área destinada ao algoritmo é possível declarar variáveis locais ou adicionar parâmetros. Além disso, o usuário pode criar subrotinas para melhorar a organização de seu código.

A execução do algoritmo gerado pelo aprendiz, o que provoca a animação do mundo virtual, acontece com o click no botão *Play* (executar, rodar). Dessa forma, as complexidades de um ambiente de animação tradicional, bem como o de programação, são ocultadas do aprendiz, permitindo que se concentre na construção do algoritmo e nas consequências de sua execução.

Figura 2.3: Três estágios para instanciar uma casa de praia no mundo virtual do Alice

Figura 2.4: Exemplo de construção de algoritmo no Alice para controle do comportamento de um ser no mundo virtual (cão).

Em termos de tecnologia, o *Alice* foi criado com a linguagem *Java* e, devido à sua complexidade e trabalho com elementos gráficos em 3D, o tamanho do aplicativo excedia os 140 Mb na versão usada por Rideaki. Dessa forma, não era viável integrá-lo aos navegadores de internet sem alguma intervenção para reduzir seu tamanho. Versões posteriores do *Alice* foram sendo incrementadas com mais pacotes gráficos e bibliotecas de áudio. A versão 2.4 possui mais de 430 Mb.

Há, ainda, a versão 3 do *Alice* que foi criada depois do lançamento da primeira versão do *iVProg*, enquanto esta dissertação de mestrado já estava em progresso. O instalador da versão 3 ultrapassa 1 Gb e também não pode ser integrado aos navegadores de internet devido ao seu tamanho. A interface gráfica foi remodelada, porém algumas características foram mantidas para atender aos antigos usuários. Uma funcionalidade que não existia nas versões anteriores e que chama a atenção do ponto de vista pedagógico é a transcrição do algoritmo criado na linguagem visual para a linguagem *Java*. A figura 2.5 mostra a interface do *Alice* 3; a tela 1 exhibe a tela equivalente a tela de mesmo número na figura 2.3 e a tela 2 exhibe o recurso mencionado de transcrição para a linguagem *Java*. Todas as versões do *Alice*, e mais fortemente a versão 3 graças ao recurso de transcrição para a linguagem *Java*, tem como foco o ensino da Programação Orientada a Objetos (Dann *et al.*, 2012).

Figura 2.5: Telas do *Alice* versão 3.

A seguir, descrevemos a relação entre a primeira versão do *iVProg* e o *Alice*.

2.4.3 Desenvolvimento da primeira versão do *iVProg*

Como dissemos anteriormente, o primeiro *iVProg* foi desenvolvido a partir do aplicativo *Alice*, onde esse último utiliza recursos de *Programação Visual* e *Visualização do Programa*. O *iVProg* tem como foco o estudo do paradigma de programação estruturada e conceitos essenciais a construção de algoritmos nesse paradigma. Dessa forma, muitas adaptações foram feitas para torná-lo um *iMA* com as características desejadas.

A definição de *iMA* nos diz que o aplicativo precisa funcionar dentro de um navegador *Web*, implementando o protocolo que permite receber a atividade e envia a solução do aluno. Para isso, funcionar como um *Applet* e ter o protocolo de comunicação específico dessa família é suficiente. Tendo em vista a definição, Reginaldo Rideaki empregou seus esforços para realizar esse projeto. Durante a adaptação do *Alice* para transformá-lo num *iMA* as principais alterações foram as seguintes:

- **Remoção total dos pacotes de animação em 3D:** um *iMA* deve funcionar nos navegadores de internet. No entanto, para o *iVProg* a animação provocaria um excesso de carga (em megabytes) que inviabilizaria seu carregamento no navegador. A remoção desses pacotes implicou numa redução significativa no tamanho da aplicação: de aproximadamente 140 Mb para aproximadamente 1,4 Mb; ou seja, apenas 1% do total da aplicação *Alice*.
- **Mudança do paradigma de programação:** com a remoção dos recursos de animação, muito da interface precisou ser adaptado. Dessa forma, foram retirados os itens de menu que

faziam referência ao mundo virtual, tanto os componentes de instanciação de entes quanto os blocos de comando destinados ao controle deles. Além disso, a tela de execução do mundo virtual foi adaptada para funcionar como um console, para o qual a saída de dados foi redirecionada.

- **Internacionalização do aplicativo:** o *iVProg* foi construído a partir de uma versão do *Alice* anterior à 2.4. A versão que serviu como base estava totalmente em inglês. Dessa forma, um processo de internacionalização foi necessário. A internacionalização de software é um processo de desenvolvimento no qual se realiza uma adaptação do aplicativo para permitir que ele possa ser localizado em outras culturas. Nesse caso, o país alvo seria o Brasil. De toda forma, durante esse processo, todas as estruturas de linguagem do *iVProg* (palavras do menu, frases de aviso, etc.) foram concentradas num único arquivo. Assim, para mudar o idioma do aplicativo basta selecionar o arquivo do qual as estruturas referentes à linguagem são carregadas. A primeira versão do *iVProg* pode ser utilizada em dois idiomas: português e inglês.
- **Adaptação para carregamento em navegadores da internet:** por definição, se o *iVProg* estava sendo construído como um *iMA*, ele precisaria ser carregado nos navegadores. O *Alice* não previa isso em seu projeto, portanto Rideaki, além de reduzir o tamanho do aplicativo, precisou criar os métodos de carregamento como *Applet* para o *iVProg*. Em termos práticos, na linguagem *Java*, para que um aplicativo seja carregado como *Applet* sua classe principal precisa ser uma subclasse de `java.applet.Applet`. No caso do *iVProg*, como há uma interface gráfica, ele estende a classe `javax.swing.JApplet` que é uma classe especial de *Applet*.
- **Implementação do protocolo de comunicação dos *iMA*:** o protocolo de comunicação se trata de uma *Application Programming Interface* (API) e um conjunto de constantes que o *iMA* deve possuir. Essa API permitirá sua comunicação com o *iTarefa* e, conseqüentemente, com o SGC *Moodle*. No momento em que esta dissertação estava sendo escrita, o protocolo de comunicação era composto de 29 constantes e três métodos. As constantes fornecem informações para a interpretação da atividade entregue pelo *iMA* ao SGC. Os métodos permitem que a aplicação devolva a atividade em formato de texto para o SGC, devolva a correção da atividade ou um pacote compactado com a atividade.

Com essas alterações, o *iVProg* já era um *iMA*. A tela 1 na figura 2.6 apresenta a tela principal do *iVProg*, a tela 2 contém um algoritmo em execução, com o console e a janela de entrada de dados em vista. Os conceitos de programação que podem ser abrangidos nessa versão do *iVProg* são: variáveis (numérica, texto e booleana), expressões algébricas e booleanas, estrutura de seleção do tipo “if/else”, laços de repetição do tipo “for” e “while”, leitura e escrita de dados e recursão. Além de um experimento observado conduzido por Rideaki, descrito em (Kamiya, 2010), o *iVProg* também foi utilizado em um curso de introdução à programação para apoiar o processo de aprendizagem numa abordagem mista de ensino. Na seção seguinte, explicaremos sobre esses casos de uso.

2.4.4 Casos de uso do iVProg

O *iVProg* foi utilizado com turmas de licenciatura em Matemática nos primeiros semestres de 2010 (curso II) e 2011 (curso III). Os cursos tiveram duração de 18 semanas e foram ministrados de maneira mista, com atividades presenciais e atividades semanais entregues pelo *Moodle*. Por ensino misto, queremos dizer que os fizeram atividades presencialmente, em laboratório, e atividades realizadas à distância (Djenic *et al.*, 2011). As aulas deram-se num laboratório de informática com dois aprendizes por computador. Para efeitos de comparação, apresentaremos os dados de um curso I ministrado em 2005 com uma abordagem de programação não visual, o *iCG* (ICG, 2015). Nos

Figura 2.6: Telas da primeira versão do iVProg

três cursos a linguagem *C* foi utilizada, sendo que no curso I e II foi utilizada a partir da semana 6 e no curso III foi utilizada desde o começo.

A ideia era verificar se o aplicativo poderia melhorar a compreensão da resolução de problemas computacionais ao eliminar a preocupação com a sintaxe de uma linguagem de programação tradicional. Além disso, era desejável verificar como a utilização da programação visual poderia impactar no aprendizado de uma linguagem tal como a linguagem *C*.

Até a 6ª semana do curso I utilizou-se apenas o *iCG* no laboratório. As demais semanas deram-se em sala de aula e os exercícios foram resolvidos em *C*, em casa. No curso II, até a 6ª semana utilizou-se apenas o *iVProg* e depois o ambiente tradicional de programação em *C*, sempre no laboratório. No curso III, a linguagem *C* e o *iVProg* foram utilizados juntos, desde o começo, resolvendo primeiro no *iVProg* e depois em *C*. No total, no curso I foram resolvidos 5 exercícios em laboratório e 3 exercícios bastante complexos, que exigiam modelagem, em casa. No curso II e III foram resolvidos 20 exercícios usando o *iVProg* e 33 exercícios usando a linguagem *C*, em sua maioria no laboratório. Existiam mais exercícios em *C* porque o *iVProg* ainda não manipula arquivos ou vetores. As avaliações dos cursos foram baseadas nos exercícios resolvidos (*C* e não-*C*) e nas provas 1 (P1) e 2 (P2). A tabela 2.1 exhibe os dados referentes a frequência dos aprendizes nos cursos, notas nas P1 e P2 e notas nas listas de exercícios *C* e não-*C* (não-*C* significa usando o *iCG* para o curso I e o *iVProg* para os cursos II e III).

	Curso I (2005)		Curso II (2010)		Curso III (2011)	
	Média	Variância	Média	Variância	Média	Variância
Frequência	56,48%	0,0109	72,92%	0,072	71,02%	0,061
P1	3,28	8,3	5,59	8,7	5,85	5,64
P2	4,33	17,02	3,47	4,92	6,36	7,29
Não-C	7	20,5	9,26	0,35	9,59	0,65
Linguagem C	4,28	18,7	8,51	0,43	9,08	0,74
Média final	4,06	10,44	4,59	6,31	5,24	6,19

Tabela 2.1: Comparação do desempenho entre os anos de 2005, 2010 e 2011 em cursos de programação introdutória

Pode-se observar um interessante aumento na frequência as aulas dos aprendizes nos cursos II e III em relação ao curso I. Além disso as médias em provas e exercícios *C* e não-*C* foram maiores quando o *iVProg* foi utilizado, inclusive as médias finais. Esses números nos permite conjecturar que o *iVProg* teve impacto positivo na motivação e no aprendizado dos aprendizes durante os cursos II e III, incluindo em como resolver os problemas utilizando uma linguagem de programação

tradicional, como a linguagem *C*.

Os resultados apresentados acima motivaram a continuação dos trabalhos com a PV e o próprio *iVProg*. Na seção seguinte, apresentamos os trabalhos relacionados encontrados na literatura, os quais utilizamos para angariar contribuições para esta dissertação.

2.5 Trabalhos relacionados

Muitos pesquisadores dedicam esforços para desenvolver métodos e ferramentas educacionais que apoiem os processos de ensino e aprendizagem de algoritmos e programação. No livro *Guide To Teaching Computer Science: An Activity-Based Approach* (Hazzan *et al.*, 2011) por exemplo, os autores fizeram uma revisão sobre os principais métodos de ensino que são baseados em: aplicativos educacionais disponíveis, diferentes formas de organização da sala de aula e acompanhamento no desenvolvimento de projetos de vários tipos. Apesar de todos esses tópicos conterem um material interessante e importante para o ensino de programação, o foco deste trabalho é voltado para os aplicativos educacionais úteis a essa área.

Nesta seção, em particular, apresentamos os trabalhos relacionados ao tema desta dissertação de mestrado. A busca foi realizada em 3 principais fontes de trabalhos científicos: a biblioteca digital da *Association for Computing Machinery* (ACM), a biblioteca digital do *Institute of Electrical and Electronics Engineers* (IEEE) e os Anais e Workshops da Sociedade Brasileira de Informática na Educação (SBIE). Na ACM e no IEEE foi possível usar a seguinte string de busca: “*visual programming*” AND (tool OR environment) AND (teach* OR learn*) AND (algorithm* OR programming). Já nos anais do SBIE a busca precisou ser feita manualmente. Nos três acervos a busca por trabalhos relacionados foi feita de 2004 à 2014, ainda assim, quando encontrávamos referências relevantes nós as adicionávamos ao conjunto de trabalhos relacionados. A consequência disso é que, eventualmente, podem aparecer trabalhos que estão situados fora do intervalo de 2004 à 2014.

A partir desses motores e da busca manual na base de dados do SBIE, encontramos uma lista de trabalhos relacionados que diferem em muitos aspectos. São exemplos: o mecanismo de interação, a forma de construção dos algoritmos, a tecnologia utilizada na implementação dos aplicativos, etc. Dado o grande número de abordagens para aplicativos nessa área, mencionamos na seção 2.3 que tanto Price *et al.* (1993) quanto Sorva (2012) consideram que os trabalhos da literatura a respeito de recursos visuais empregados na atividade de programação podem ser organizados dentro dos conjuntos descritos na figura 2.2. Nesta revisão focaremos o conjunto referente à intersecção da *Programação Visual* com a *Visualização estática do código*, portanto os critérios visam filtrar os trabalhos e obter como resultado as ferramentas pertencentes a esse conjunto. Afinal, é o conjunto ao qual pertence tanto o *Alice* (criação do algoritmo) como o *iVProg*; consequentemente, também é a qual pertence o *iVProg2*. A seguir, descrevemos o processo de busca e apresentamos os critérios utilizados.

A busca nos anais do SBIE foi um pouco mais trabalhosa, pois é um evento nacional promovido pela comunidade de Informática na Educação bastante amplo e não possui ainda um motor de busca sofisticado como ACM ou IEEE. Dessa forma, o primeiro critério de inclusão dos trabalhos foi ter como assunto o ensino de programação para iniciantes, o que deixou 77 artigos no total. Dos 77, um segundo filtro foi aplicado para buscarmos aqueles trabalhos que apresentavam, ou propunham, uma ferramenta destinada ao ensino de programação para iniciantes que utilizava recursos visuais, colocando-as no conjunto de trabalhos relacionados à *Visualização de Sistemas*. Dentre eles, havia aqueles cuja motivação era a robótica e outros cuja motivação era a animação de entes ou objetos virtuais, assim como o *Alice*. No final, deixamos apenas os trabalhos que possuíam a *Visualização do Aplicativo* como principal recurso, restando 3 artigos: o de Barbosa *et al.* (2009), Medeiros *et al.* (2014) e Miranda *et al.* (2007).

Já nos acervos internacionais, encontramos 78 artigos relacionados por meio do motor de busca da IEEE e o impressionante número de 634 artigos na ACM. Infelizmente, depois da análise, percebemos que a ACM retornava mais de uma vez o mesmo artigo. Ainda assim, depois da busca nesses acervos, foram lidos os títulos e os resumos dos trabalhos, dessa forma foi possível separar

Construção do Algoritmo	Trabalhos relacionados
Texto	Jeliot (Sajaniemi e Kuittinen, 2003), Jype (Helminen e Malmi, 2010), ViLLE (Kaila <i>et al.</i> , 2010), Sem nome 1 (Korsh <i>et al.</i> , 1998), jGRASP (Cross <i>et al.</i> , 2004), TRAKLA2 (Rajala <i>et al.</i> , 2010), Matrix Pro (Karavirta <i>et al.</i> , 2004), JHAVÉ (Naps, 2005), OOP-Anim (Esteves e Mendes, 2004), DRUIDS (Whale, 1994), ANIMAL (Okamoto <i>et al.</i> , 2009), Splaf (de Jesus, 2011), Portugol 3.x (Manso <i>et al.</i> , 2010), Gidget (Lee e Ko, 2011), Greenfoot (Kölling, 2008b), Karel (Pattis, 1981), Jeroo (Dorn e Sanders, 2003), PlayLogo 3D (Paliokas <i>et al.</i> , 2011)
Fluxograma	SIVIL (Materson e Meyer, 2001), RAPTOR (Carlisle <i>et al.</i> , 2004b), Progranimate (Scott <i>et al.</i> , 2008), iLogic (Chun e Ryoo, 2010), Kara (KARA, 2015) ProGuide (Areias e Mendes, 2007), B# (Greyling <i>et al.</i> , 2006), H-SICAS (Marcelino <i>et al.</i> , 2008), Sem nome 2 (Charntaweechun e Wangsiripitak, 2006)
Ícones	Kodu (MacLaurin, 2011), ProgrameFácil (Miranda <i>et al.</i> , 2007), NXT-G LEGO (Kelly, 2010)
Blocos de Comandos	Alice (Conway <i>et al.</i> , 2000), Scratch (Utting <i>et al.</i> , 2010), BYOB (Federici, 2011), EToys (Freudenberg <i>et al.</i> , 2009), AgentSheets (Repenning e Ioannidou, 2004), pyGoGo-Blocos (Barbosa <i>et al.</i> , 2009), Game Development Framework (Medeiros <i>et al.</i> , 2014), Visual Script Editor (Msiska e van Zijl, 2012)

Tabela 2.2: Tabela com trabalhos relacionados.

aqueles que de fato possuíam ferramentas com algum recurso de *Visualização de Sistemas* destinado ao ensino de programação para iniciantes. Algumas vezes, em ambas as fontes, encontramos trabalhos que utilizavam uma aplicação desenvolvida por terceiros, sendo apenas um relato de uso. Nesses casos, se a aplicação estivesse dentro do escopo definido por nós, então essa ferramenta era incluída no conjunto de trabalhos relacionados. No final da análise, na busca pelos trabalhos de *Visualização do Aplicativo*, obtivemos subconjuntos de 14 trabalhos na ACM mais 2 buscas externas e 17 trabalhos mais 2 buscas externas no IEEE.

Até esse momento obtivemos os trabalhos relacionados à *Visualização do Aplicativo*. A partir dessa seleção, analisamos os projetos com mais cuidado e filtramos os trabalhos para separá-los pelos mecanismos de criação dos algoritmos. O resultado dessa análise é apresentado na tabela 2.2.

É importante observar que não estamos ignorando a significância dos outros tipos de recursos visuais pertencentes ao conjunto de ferramentas de *Visualização de Sistema*, estamos apenas filtrando para podermos analisar os trabalhos com o maior grau de proximidade a proposta do *iVProg2*. Dessa forma, a categoria dos aplicativos que mais nos interessa, presentes na tabela 2.2 é a de **Blocos de Comandos**. Analisaremos os trabalhos a seguir:

- **Alice:** o aplicativo *Alice* foi descrito na seção 2.4.2. Ainda assim, reforçamos aqui com a descrição presente no próprio sítio da internet (Alice, 2015): “O *Alice* permite a aprendizagem dos conceitos fundamentais de programação num contexto de criação de filmes animados e *videogames* simples. No *Alice*, objetos 3D (por exemplo: pessoas ou veículos) podem popular um mundo virtual e os aprendizes criam um programa para animar esses objetos. Na interface interativa do *Alice*, os aprendizes arrastam e soltam blocos gráficos para criar o programa, onde esses blocos correspondem às instruções de códigos tradicionais como da linguagem *Java* e *C++*. O *Alice* permite que visualizem imediatamente como as animações programa-

das reagem, viabilizando a compreensão da relação entre as instruções da programação e o comportamento dos objetos em sua animação.” - tradução livre.

- **Scratch:** essa ferramenta foi concebida no *Massachusetts Institute of Technology*, o conhecido MIT. Seu uso se dá principalmente no ensino fundamental, no ensino de conceitos matemáticos e de computação. Quanto a interação, o *Scratch* é parecido com o *Alice*, pois os usuários utilizam o *mouse* para arrastar os comandos a partir de uma bandeja do aplicativo até o campo destinado a criação do algoritmo. O *Scratch* permite a criação de animações em 2D, jogos simples, músicas, histórias interativas, etc. Em seu sítio da internet ([Scratch, 2015](#)), além de assistir às animações exemplo, a edição dos algoritmos está disponível, porém é condicionada ao plugin *Flash Player*¹). A figura 2.7 é uma imagem da interface da ferramenta em uso no navegador da internet.

Figura 2.7: Tela do *Scratch* no navegador da internet com auxílio da máquina virtual *Flash Player*.

- **BYOB:** o aplicativo de programação *BYOB* (acrônimo de *Build You Own Blocks*) foi criado a partir do *Scratch*, descrito anteriormente. Segundo os autores, foi um dos primeiros aplicativos derivados dele. O *BYOB* foi criado para o ensino de programação estruturada a fim de atender uma demanda percebida pela academia. Assim como o *Scratch*, que pode ser utilizado em navegadores da internet, o *BYOB* seguiu o mesmo caminho. Há uma versão desse aplicativo que se chama *Snap! Build Your Own Blocks* e foi desenvolvido com a tecnologia HTML5, permitindo trivialmente sua execução em dispositivos móveis. A figura 2.8 apresenta a tela inicial do *Snap!*, executando num navegador de Internet.
- **EToys:** foi construído sobre a máquina *Squeak*² (*Smalltalk*). É destinado principalmente a crianças e tem o objetivo de apoiar o ensino de conceitos básicos de matemática e computação. Para sua utilização em computadores existem algumas opções: descarregar uma versão específica para o sistema operacional do usuário ou uma versão *All-in-one*, que pode ser executada em qualquer sistema operacional sem a necessidade de instalação. A diferença para os aplicativos anteriores é que no *EToys* a criança tem a possibilidade de criar qualquer desenho dentro do aplicativo e atribuir os comportamentos ao próprio desenho. A figura 2.9 exibe a tela do aplicativo executando um exemplo no qual um comportamento de movimentação é atribuído a um objeto (carro).

¹O *Flash Player* é uma máquina virtual que permite a execução de arquivos no formato SWF.

²O *Squeak* é uma implementação de código livre de *Smalltalk*. Pode ser encontrado em <http://www.squeak.org/>.

Figura 2.8: Tela do *Snap! Build Your Own Blocks* no navegador da internet.

Figura 2.9: *EToys* na Área de Trabalho (em tela cheia).

- **AgentSheets:** é um aplicativo de PV que aplica parcialmente o paradigma de planilhas eletrônicas para controlar imagens, usando a ideia de linhas e colunas para definir suas posições dentro de uma região. O *AgentSheets* foi construído com a intenção de permitir a realização de simulações e criação de jogos por usuários que não dominam a programação. A figura 2.10 mostra a tela do aplicativo em execução.
- **pyGoGoBlocos:** esse aplicativo foi desenvolvido para criar algoritmos que são executados na placa *GoGo Board* ³. É uma ferramenta dependente de uma série de pacotes relacionados à linguagem de programação *Python*, o que torna sua instalação complicada para usuários

³A *GoGo Board* é uma placa de baixo custo destinada ao ensino de Robótica. Mais informações podem ser obtidas no endereço: <http://www.gogoboard.org/pt-br>

Figura 2.10: *AgentSheets* na Área de Trabalho. Na janela **a)** temos a área destinada à criação do algoritmo que controla o comportamento dos agentes; **b)** é a janela contém as condições possíveis de serem observadas; por fim, **c)** é a janela que contém as ações que o agente pode executar.

pouco experientes. Quanto a interação e criação do algoritmo, lembra os aplicativos *Alice* e *Scratch*, pois as instruções estão no formato de blocos de comandos encaixantes. A figura 2.11 exibe uma tela do aplicativo.

Figura 2.11: *PyGoGoBlocos* na Área de Trabalho.

Fonte: <https://master.dl.sourceforge.net/project/br-gogo/DOC>.

- **Game Development Framework:** o artigo propõe um ambiente de desenvolvimento de jogos voltados para o ensino de algoritmos e programação. Esse ambiente teria sido construído durante um trabalho de mestrado (Medeiros, 2014). Infelizmente só conseguimos acesso ao artigo, não conseguimos a dissertação ou a ferramenta. Entretanto, segundo os autores, o ambiente faz uso da biblioteca *Blockly*. Essa biblioteca facilita a criação de editores de PV, pois utiliza a tecnologia do *HTML5* para fornecer um ambiente de edição com blocos que representam as instruções das linguagens de programação. A figura 2.12 apresenta um exemplo contido no sítio de divulgação da biblioteca (Blockly, 2015).

Figura 2.12: *Blockly* no navegador de internet.

- **Visual Script Editor (VSE):** este aplicativo foi construído com a biblioteca *OpenBlocks* (Roque, 2007). O VSE é utilizado para a criação de *scripts* em *LUA*. Os *scripts* são, então, exportados pelo aplicativo para um motor de jogos em 3D, que executa a animação de acordo com o que fora definido pelos blocos visuais. A figura 2.13, retirada do artigo de Msiska e van Zijl (2012), traz a imagem do aplicativo.

Os trabalhos apresentados anteriormente são os que possuem maior similaridade com a proposta do *iVProg2*. Podemos perceber que todos eles utilizam, além da PV, algum recurso de *Animação do Algoritmo*. No entanto, com o objetivo de reduzir o peso do aplicativo em *Java* e manter a filosofia da primeira versão do *iVProg*, o *iVProg2* ainda não conterá recursos de *Animação do Algoritmo*.

Dos trabalhos relacionados, alguns trouxeram contribuições diretas e bastante significativas, como o *Alice* e o *Scratch*. Quanto ao *Alice*, esse foi aplicativo de PV que deu origem a primeira versão do *iVProg*, assim, sua maior contribuição foi o próprio código-fonte criado na universidade Carnegie Mellon e os relatos de uso do sistema presentes na literatura que inspiraram a continuação dos trabalhos com PV realizados em nosso grupo de pesquisa. São exemplos de trabalhos que nos motivam: (Adams, 2007), (Dann *et al.*, 2012), (Kelleher *et al.*, 2007) e (Cooper *et al.*, 2000). As contribuições do *Scratch* também vão na direção dos relatos de seu uso nos ambientes de ensino, principalmente no que diz respeito a percepção dos aprendizes quanto a Ciência da Computação e a desconstrução de estereótipos relacionados à essa área (Armoni *et al.*, 2015). Tal como o *Alice*, o *Scratch* também colabora com a diminuição das dificuldades relacionadas à sintaxe de linguagens de programação textuais (Lewis, 2010).

A contribuição do *BYOB* para este trabalho se dá pela confirmação da necessidade de investirmos esforços em aplicativos que utilizam a PV no ensino de programação estruturada, dado que a

Figura 2.13: Tela do aplicativo VSE, obtida no artigo de *Msiska e van Zijl* (2012).

primeira versão do *iVProg* teve início em 2009. Além dessa contribuição, ficamos interessados pela abordagem que utiliza *HTML5* como solução tecnológica para promover maior portabilidade do aplicativo.

A contribuição do *EToys* e *AgentSheets* se dá pelos estudos realizados com crianças, nos mostrando que mesmo um público bastante jovem pode lidar com blocos de comando. Quanto a complexidade do raciocínio computacional, já na década de 1970 do século XX, Seymour Papert nos mostrou que crianças da fase escolar equivalente ao ensino fundamental são capazes de criar programas interessantes com a linguagem *LOGO* *Solomon e Papert* (1976). Quanto as contribuições do *Game Development Framework* e *Visual Script Editor*, vem principalmente do conhecimento sobre as bibliotecas que utilizaram durante a construção de seus ambientes. Entretanto, a previsão de uso dessas bibliotecas está nos trabalhos futuros a esta dissertação.

O capítulo seguinte apresenta o aplicativo *iVProg2* e o curso experimental realizado. Nele, também explicamos os métodos aplicados durante o desenvolvimento do trabalho.

Capítulo 3

O aplicativo *iVProg2*

Neste mestrado demos continuidade aos trabalhos com o *iVProg*, desenvolvido anteriormente por Kamiya (2010). Continuamos a investir nossos esforços nesse aplicativo de PV devido aos bons resultados verificados durante seu uso (2.4.4). Decidimos pela reconstrução do aplicativo devido a possibilidade de criá-lo como primeiro produto da *Linha de Produto de Software*, descrita na subseção 3.2.1, para melhorar a qualidade do código-fonte e facilitar futuros desenvolvimentos sobre o *iVProg2*¹.

A segunda versão do aplicativo *iVProg* é uma das principais contribuições deste trabalho. A figura 3.1 mostra o *iVProg2* durante a execução. Suas funcionalidades são aquelas presentes na primeira versão do aplicativo mais uma lista de funcionalidades complementares. Na próxima seção (3.1) apresentamos uma comparação entre as versões. Na seção seguinte (3.2), explicamos sobre o processo de seu desenvolvimento e as técnicas empregadas. A seção 3.4, traz as considerações finais sobre o desenvolvimento do aplicativo.

Figura 3.1: Tela do aplicativo *iVProg2* na área de trabalho. O algoritmo ao fundo calcula o maior entre dois números inseridos pelo usuário e o imprime no console cinza.

¹ O aplicativo pode ser descarregado em: <http://www.matematica.br/ivprog2/iVProg2.jar>.

3.1 Descrição das funcionalidades

O desenvolvimento do *iVProg2* foi motivado por diversos fatores. Dentre eles, estava a criação de novas funcionalidades que seriam difíceis de serem implementadas na versão anterior devido a falta de documentação.

Decorre da definição de *iMA* duas funcionalidades obrigatórias: funcionar em navegadores *Web* (como é o caso dos Applets) e possuir o protocolo de comunicação dessa família de aplicativos. Veremos na seção seguinte que o uso da Linha de Produto de Software para *iMA* já fornece essas funcionalidades prontas para o desenvolvedor. Nesse caso, restam as funcionalidades implícitas desses aplicativos que são listadas por [Dalmton \(2012\)](#) como decorrentes de: (i) ferramentas de autoria de atividades pelo professor; (ii) interatividade nas atividades; e (iii) correção automática da solução do aprendiz.

As três funcionalidades implícitas são dependentes do domínio da aplicação. A primeira versão do *iVProg* possuía apenas a funcionalidade (ii). Entendemos como interatividade no aplicativo a construção do algoritmo e a facilidade em testá-lo sempre que necessário, clicando apenas em um botão (*Rodar*). Ambos os mecanismos foram mantidos no *iVProg2*, com pequenas alterações. A instanciação de uma instrução no *iVProg* se dava de maneira análoga ao *Alice*, arrastando os comandos a partir de uma bandeja do aplicativo. No *iVProg2*, o usuário cria o algoritmo a partir de um menu de contexto com as opções de comandos. Essa alteração se deu principalmente porque métodos de interação com *pointing-and-click* agredem menos a fisiologia dos usuários e tem maior precisão durante a iteração em comparação ao método de *drag-and-drop* ([Inkpen, 2001](#)). Ainda assim, mativemos a técnica de *drag-and-drop* funcionando junto com *pointing-and-click* durante a reorganização do algoritmo. A intenção é não permitir que usuários acostumados com o *Alice* e com primeira versão do *iVProg* se sintam prejudicados de alguma forma. A figura 3.2 apresenta a interface gráfica do *iVProg2* com o menu de contexto aberto.

Figura 3.2: Tela do aplicativo *iVProg2* na área de trabalho. Em evidência está o menu de contexto.

As funcionalidades (i) e (ii) foram implementadas no *iVProg2*. De certo modo, são inter-relacionadas, pois a correção automática depende da definição do gabarito, que, por sua vez, é gerado durante a autoria pelo professor. No caso do *iVProg2*, o usuário, enquanto professor, define os testes aos quais os algoritmos dos aprendizes serão submetidos. Para cada teste deve-se definir

o valor em % da nota daquele teste, os valores de entrada de dados (com seus tipos) e os valores de saída esperados (com seus tipos). A heurística para realizar a avaliação, então, executa o algoritmo do aprendiz uma vez para cada teste, fornecendo os valores de entrada que o professor definiu e comparando os valores de saída do algoritmo com os valores esperados.

Além das funcionalidades implícitas dos *iMA*, existem funcionalidades específicas do domínio do *iVProg* que não estavam presentes na sua primeira versão, mas que foram implementadas durante este trabalho de mestrado. Elas são: criação de vetores, criação de matrizes e visualização do algoritmo na linguagem de programação *C*. Como nós desenvolvemos um interpretador para o *iVProg2*, essas funcionalidades foram criadas no momento de reconstrução do aplicativo. Na seção seguinte explicaremos como se deu essa reconstrução.

3.2 Desenvolvimento do iVProg2

Durante o processo de desenvolvimento empregamos algumas técnicas da *Engenharia de Software* e também de *Desenvolvimento Ágil*. Da *Engenharia de Software* herdamos do trabalho de Dalmon (2012) a *Linha de Produto de Software* sobre a qual construímos o aplicativo de PV. Do *Desenvolvimento Ágil* utilizamos o *Test-Driven Development* (*Desenvolvimento Guiado por Testes*) na construção do interpretador para algoritmos. As seções seguintes explicam sobre essas técnicas e como foram empregadas neste trabalho.

3.2.1 Linha de Produto de Software para iMA

Segundo Dalmon (2012), a “*Linha de Produto de Software* (LPS) pode ser definida como um conjunto de técnicas e métodos com o objetivo de facilitar e aprimorar o processo de desenvolvimento de famílias de aplicativos similares” (Dalmon, 2012)(p. 15). No contexto de nosso grupo de pesquisa, os aplicativos similares são os *iMA* e a técnica de implementação escolhida foi a criação de um *arcabouço de aplicação*. Um *arcabouço de aplicação* pode ser visto como um aplicativo inacabado, que possui pontos a serem especializados a fim de se obter comportamentos específicos do aplicativo que se deseja criar a partir dele (Guerra, 2010). Dessa forma, no intuito de criarmos o *iVProg2*, utilizamos o arcabouço de aplicação para *iMA* e o estendemos, especializando-o para o ensino de programação.

A tecnologia utilizada no desenvolvimento do arcabouço foi *Java*, permitindo que atendessem à definição de *iMA* (2.4.1). Assim, no contexto de trabalho desenvolvido neste mestrado, o arcabouço entra como um conjunto de código-fonte na linguagem *Java* que contribuiu com funcionalidades comuns aos *iMA*. Uma consequência do reuso de código do arcabouço, característica desse método de implementação de LPS, é o reuso de arquitetura. Além disso, o arcabouço vem acompanhado de um manual de utilização que contém um passo a passo para a criação de novos *iMA*, contribuindo, portanto, com o reuso de processo de desenvolvimento.

A arquitetura da LPS é organizada por componentes, que estão listados a seguir:

- i) *Obrigatórios*: componentes invariáveis a todos os *iMA*.
 - *BaseUserInterface* - Esse componente é responsável por instanciar a interface gráfica específica de domínio e serve de base para essa interface. Além disso, provê os botões com controles comuns a todos os *iMA*.
 - *SystemControl* - É no controlador do sistema onde todos os componentes são instanciados. Além disso, também garante a comunicação entre os outros componentes.
 - *AssignmentControl* - Gerencia tudo que se refere às atividades realizadas pelos aprendizes e professores. São exemplos de funcionalidades contidas neste componente: criação atividade, correção de atividade, gravação em arquivo (quando em uso na área de trabalho), etc.

- *Communication* - É o componente responsável pela comunicação do *iMA* com os meios externos, seja o sistema operacional ou um SGC. Além disso, encapsula o protocolo de comunicação padrão dessa família de aplicativos que é utilizado ao funcionar como um Applet dentro de um SGC.

ii) *Opcionais*:

- *OperationModule* e *AssignmentModule* - Basicamente, esses componentes são compostos por interfaces (Oracle, 2015b) da linguagem *Java* que o desenvolvedor pode implementar durante a criação de novos módulos. Se o módulo agir como uma operação no domínio (por exemplo: esconder objetos selecionados) deve conter a implementação de *OperationModule*. No caso de ser uma operação sobre uma atividade (por exemplo: recomeçar atividade) deve ser implementação de *AssignmentModule*.

iii) *Específicos de domínio*: os componentes específicos de domínio são aqueles que o desenvolvedor utilizará para especializar o *iMA* que está sendo criado. No caso do arcabouço para *iMA*, existem dois componentes que devem ser estendidos.

- *DomainModel* - O modelo de domínio é o componente no qual definimos os objetos, operações e regras de negócio para o domínio do *iMA*.
- *DomainUserInterface* - É um componente composto por classes abstratas que devem ser estendidas para que sejam criadas a interface específica de domínio e a interface de autoria de atividades. Nesse componente também são criadas as representações visuais dos objetos de domínio.

Desses componentes, com os *obrigatórios* nós não precisamos nos preocupar, pois têm origem nas funcionalidades comuns dos *iMA* e estão implementados no arcabouço de aplicação. Não foram implementados *Plugins* para o *iVProg2*, portanto não utilizamos os componentes *OperationModule* ou *AssignmentModule*. Dessa forma, restam os componentes que dependem diretamente do domínio da aplicação. Esses componentes foram estendidos para que o *iVProg2* pudesse existir. Na seção seguinte descrevemos as implicações do uso do arcabouço durante a implementação do aplicativo.

3.2.2 Projeto e implementação do *iVProg2*

O arcabouço possui um manual de utilização (LInE, 2015) para guiar o desenvolvedor de *iMA* durante o processo de implementação. Uma das orientações do manual é um passo-a-passo que explica a ordem de desenvolvimento dos componentes específicos do domínio da aplicação (*DomainModel* e *DomainUserInterface*). Nos próximos parágrafos descrevemos o processo de acordo com as orientações seguidas.

Os primeiros passos do método de desenvolvimento dizem respeito a listagem dos objetos específicos do domínio e as operações de criação e manipulação desses objetos. Os *objetos de domínio* fazem parte da camada de dados da aplicação, então suas instâncias e a relação entre elas representa o estado da atividade. No caso deste trabalho o domínio é o da *Programação Estruturada*, portanto, os *objetos de domínio* são aqueles utilizados durante a criação de um algoritmo nesse paradigma. A seguir apresentamos a lista dos objetos criados para o *iVProg2*:

- *Contexto*: objeto que encapsula uma tabela do tipo chave-valor que armazena os valores mais básicos do interpretador - por exemplo, um número inteiro. A chave é o identificador único do objeto de domínio *Valor*, o valor armazenado é uma primitiva da linguagem *Java* (*int*, *double*, *boolean* ou *String*).
- *Valor*: encapsula um identificador único para ser utilizado com o *Contexto*. Pode representar um número inteiro, número real, valor booleano ou cadeia de caracteres.
- *Variáveis*: número inteiro, número real, valor lógico (*booleano*), cadeia de caracteres.

- *Vetores*: contém valores (*Valor*) de um único tipo.
- *Matrizes*: contém valores (*Valor*) de um único tipo.
- *Operações aritméticas*: *adição*, *subtração*, *multiplicação* e *divisão*.
- *Operações booleanas*: *e*, *ou* e *negação*.
- *Comparações*: *diferente*, *menor ou igual*, *maior ou igual*, *menor*, *maior*.
- *Operações sobre cadeia de caracteres*: *concatenação*, *quebra de string a partir de índice*, *busca na cadeia de caracteres*.
- *Linha de Atribuição*.
- *Seleção*: instrução *if/else*.
- *Repetição*: instruções equivalentes ao *while* e *for*.
- *Função*.
- *Função recursiva*.
- *Devolução de valores*: equivalente ao *return* da linguagem *C*.
- *Referências* para variáveis, vetores, matrizes e funções.
- *Leitura e Escrita* de dados.

Em termos práticos de implementação, os objetos listados estendem uma classe do arcabouço de aplicação. À saber, é a classe `ilm.framework.assignment.model.DomainObject`. A próxima etapa consistiu na criação da lista de *ações de domínio*.

As *ações de domínio* estão relacionadas à instanciação dos objetos, alterações em seus estados e composição das relações entre eles. São exemplos de *ações de domínio*: a *Criação de Variável*, a *Mudança do Valor Inicial da Variável*, a *Mudança do Tipo da Variável*, etc. Essas ações são extensões da classe `ilm.framework.assignment.model.DomainAction`. Cada ação encapsula regras que são definidas numa classe que estende `ilm.framework.domain.DomainModel`, que é o *modelo de domínio*. Depois de listados os *objetos de domínio* e as *ações de domínio*, deve-se implementá-los.

Nessa fase da implementação, o manual nos orienta a utilizar a técnica de *Desenvolvimento Ágil* de TDD (Beck, 2003). Essa técnica promove a implementação em pequenos ciclos de criação de testes automatizados, escrita do código e refatoração. A utilização do TDD pode diminuir a quantidade de erros durante a programação (Maximilien e Williams, 2003), garantir um código mais limpo e deixa o desenvolvedor mais seguro do trabalho realizado (Beck, 2003). Assim, para cada *objeto de domínio* e *ação de domínio*, criamos testes automatizados no ambiente *Eclipse*² com o apoio do *JUnit*³. No total, foram criados 115 testes de unidade.

Depois de implementados, os *objetos de domínio* e as *ações de domínio* nos permitem executar o código gerado pela PV, pois a relação entre os *objetos de domínio* compõe o interpretador do código. Dado o domínio do aplicativo, o funcionamento do interpretador deve ser similar a uma linguagem de programação do paradigma considerado. Para que essa necessidade fosse atendida, modelamos as relações com a utilização do padrão de projeto *Composite* (Gamma et al., 1994). Em *Engenharia de Software*, um padrão de projeto é uma solução repetível para um problema recorrente durante a criação de sistemas (Gamma et al., 1994). O padrão utilizado permite a composição de objetos em estrutura de árvore para representar hierarquias de parte-todo. A figura 3.3 ilustra a hierarquia em

² O *Eclipse* é um *Ambiente de Desenvolvimento Integrado* (IDE) que possibilita o uso de várias tecnologias, incluindo o *Java*. O sítio do projeto é: <http://www.eclipse.org>.

³ O *JUnit* é um arcabouço para a escrita de testes automatizados. O sítio do projeto é: <http://junit.org>.

UML desse padrão. A vantagem está no tratamento uniforme que pode ser dado para estruturas simples (folha) e uma estrutura composta (composição), pois ambas implementam a interface de componente.

Figura 3.3: Padrão Composite. A imagem foi adaptada a partir do livro de *Gamma et al.* (1994).

Esse padrão foi utilizado no interpretador para a criação de expressões aritméticas e booleanas e, também, na representação do código-fonte. No caso das expressões aritméticas, por exemplo, temos as operações básicas de adição, subtração, multiplicação e divisão. Essas operações podem ser compostas de outras operações, de valores, de variáveis ou da combinação dessas. Dessa forma, uma operação aritmética pode ser vista como uma composição de expressões, sendo ela mesma uma expressão, o que nos permite criar estruturas como a ilustrada da figura 3.4. Um raciocínio análogo funciona para as expressões booleanas.

Figura 3.4: Ilustração da estrutura de uma expressão aritmética que pode ser construída no interpretador.

Além da utilização nesse caso, o padrão também foi empregado na construção dos *objetos de domínio* que representam o código-fonte, ou seja, na representação das instruções. Nesse caso, as estruturas que correspondem à folha são os comandos: leitura de dados, escrita de dados, comando de devolução e linha de atribuição de valores. As instruções que correspondem à composição são: laço de repetição *while*, laço de repetição *for*, *função* e seleção *if/else*. A figura 3.5 representa a organização desses elementos conforme o padrão de projeto.

Para estar em conformidade com o arbouço, uma pequena adaptação foi feita no padrão *Composite*. Em vez de ser uma interface, *DataObject*, que corresponde ao *Component* do diagrama presente na figura 3.3, é uma classe abstrata. Dessa forma, ela estende a classe *DomainObject* do arcabouço, tornando todas as instâncias da hierarquia abaixo dela *objetos de domínio*.

Em seguida, a orientação do manual é para que sejam criadas as classes *SystemFactory* e *IlmProtocol*. A primeira classe é responsável por informar quais são as classes usadas pela parte específica do *iMA*. A classe *IlmProtocol* define a especificação dos *iMA*, os métodos públicos consultados pelo SGC, os seus parâmetros de configuração e o formato de arquivo utilizado. A implementação des-

Figura 3.5: Ilustração da estrutura de uma expressão aritmética que pode ser construída no interpretador.

As classes é bastante técnica, a documentação do processo de desenvolvimento se dá pelo próprio código-fonte. Nesse ponto do desenvolvimento, o *iVProg2* ainda não poderia ser instanciado, uma vez que sua interface não existia. Os passos seguintes dizem respeito a criação da interface gráfica específica de domínio.

A construção da interface gráfica se deu na ordem aproximada das necessidades básicas para o funcionamento de um algoritmo no *iVProg2* e a implementação foi gradual. Além disso, visamos sempre a componentização dos elementos gráficos para que a criação de novos blocos de comandos ou alterações nos existentes fossem simplificadas. O resultado da implementação foi exibido na figura 3.1.

Depois da implementação do aplicativo *iVProg2*, nós o utilizamos num curso experimental a fim de compará-lo com um ambiente de programação textual. No capítulo 4 descrevemos esse curso e apresentamos os resultados. Na seção seguinte, apresentamos o protótipo em *HTML5* que foi desenvolvido. As considerações finais a respeito do desenvolvimento do *iVProg2*, são apresentadas na seção 3.4.

3.3 *iVProg2* em *HTML5*

Durante a realização deste mestrado, o *Netscape Plug-in API* (NPAPI), que possibilita a execução de *Applets* e outros *plugins* nos navegadores de internet, foi abolido do navegador *Chrome* (Blog, 2013) e depois também foi removido *Mozilla Firefox*, dois dos mais utilizados navegadores (Vrbanec et al., 2013). Esse fato e as complicações encontradas por usuários durante a instalação e configuração das máquinas virtuais do *Java* motivou nosso grupo de pesquisa a buscar alternativas a essa tecnologia. Atualmente a mais viável aparenta ser o *HTML5* junto de *Javascript*, sendo compatíveis com a grande maioria dos navegadores.

A criação do protótipo foi realizada com o auxílio de Tulio Faria que foi bolsista TT do grupo de pesquisa. O protótipo foi construído com a utilização de arcabouços de código livre em *Javascript* e *HTML*, sendo eles o *AngularJS*⁴ e o *Twitter Bootstrap*⁵.

As funcionalidades do *iVProg2* em *HTML5* que foram implementadas são análogas às de um *iMA Java*. Entretanto, não funciona como um *Applet*. Para executar no *Moodle*, uma pequena alteração foi feita no pacote *iTarefa*, de forma a possibilitar uso de *iframe* (W3C, 2015). Dessa forma, em vez de carregar no navegador como um *Applet*, a aplicação é carregada como uma página *HTML* dentro do *iframe*. Assim, as funcionalidades obrigatórias dos *iMA* foram implementadas.

⁴É um arcabouço que viabiliza a arquitetura de model-view-controller para *Web*. Sítio: <https://angularjs.org/>

⁵É um arcabouço destinado a construção de projetos responsivos para *Web* e dispositivos móveis. Utiliza *HTML*, *CSS* e *Javascript*. O sítio é <http://getbootstrap.com/>.

Descrevemos na subseção 3.2.2 que existem funcionalidades implícitas que precisam ser implementadas. São as funcionalidades referentes a autoria de exercícios, atividade interativa e correção automática, que também desenvolvidas nessa versão do aplicativo. Entretanto, nem todas os recursos providos pelo arcabouço de aplicação estão presentes no protótipo. Um recurso importante que ainda não está presente é o *Desfazer/Refazer*, que prejudica a usabilidade do projeto. A figura 3.6 traz uma imagem do *iVProg2* em *HTML5* executando no *Moodle*.

Figura 3.6: Captura de Tela com o *iVProg HTML5* com exercício para imprimir maior inteiro, integrado ao *Moodle* via *iTarefa*.

A execução do código gerado por meio da PV é possível graças à conversão do algoritmo para uma cadeia de caracteres na linguagem *Javascript*. Depois da conversão, a cadeia é entregue a um método que consegue executar comandos na própria linguagem de *Javascript*.

Dado que a aplicação estava implementada, a menos de algumas funcionalidades, planejamos a realização de um estudo comparativo entre as tecnologias *Java* e *HTML5*. Entretanto, assim como explicaremos na seção 4.4.1, o *quorum* do curso não foi suficiente para a conclusão desse estudo.

Na seção seguinte, apresentamos as considerações finais sobre o desenvolvimento da aplicação na linguagem *Java*, realizando uma comparação entre as versões desenvolvidas com essa tecnologia.

3.4 Considerações sobre o processo de desenvolvimento

Neste capítulo apresentamos o processo de desenvolvimento do aplicativo *iVProg2* a partir da LPS para *iMA*. A motivação para a reconstrução do sistema se deu principalmente para melhorar a qualidade do código-fonte, pois o da primeira versão era de difícil manutenção e não havia

documentação. Além disso, estávamos motivados a criar o primeiro produto da LPS, que caracteriza uma das contribuições deste trabalho ao dar continuidade aos trabalhos de [Dalmon \(2012\)](#).

Para melhorar o processo de desenvolvimento relacionado ao *iVProg2*, durante a reconstrução o código foi documentado utilizando-se o recurso Javadocs ([Oracle, 2015c](#)). Além disso, também analisamos as métricas de código-fonte do aplicativo e comparamos com o arcabouço e com a primeira versão do *iVProg*. Os resultados são exibidos na tabela 3.1.

Métricas	Valor Médio Arcabouço	Valor Máximo Arcabouço	Valor Médio <i>iVProg2</i>	Valor Máximo <i>iVProg2</i>	<i>iVProg</i> versão 1	<i>iVProg</i> versão 1
Métodos ponderados por classe	9,78	43,0	16,4	160	13,53	383
Profundidade da Árvore de Heranças	2,90	7,0	3,74	7	4	10
Número de Filhos	0,37	6,0	0,57	16	0,69	17
Acoplamento entre Classes	2,52	11,0	3,17	13	7,32	44
Falta de Coesão entre Métodos	0,32	1,0	0,41	1,33	0,193	1,5

Tabela 3.1: Comparação entre as métricas do arcabouço, *iVProg* e *iVProg2*.

Segundo o trabalho de [Chidamber e Kemerer \(1994\)](#), as métricas coletadas têm significância para a manutenção do aplicativo. Os valores das métricas do arcabouço podem ser considerados entre bons e médios. Justificam-se os valores mais altos para a métrica “Métodos ponderados por classe” devido a complexidade da instanciação exemplo em comparação aos dois aplicativos de PV. Entretanto, o valor máximo para essa métrica no *iVProg2* é menor do que a metade para o *iVProg*. De maneira geral, os valores das métricas do *iVProg2* se mostram mais próximos dos valores do arcabouço do que o *iVProg*. Além disso, o número de classes do *iVProg2* é expressivamente menor do que do *iVProg*, sendo, respectivamente, 152 classes e 609 classes. Dessa forma, podemos considerar que houve melhora na qualidade do código do *iVProg* para o *iVProg2*.

Capítulo 4

O curso experimental

Neste capítulo descrevemos o experimento que realizamos a fim de comparar a *carga de trabalho* incorrida aos aprendizes devido ao tipo de ambiente de programação utilizado. De forma mais específica, gostaríamos de comparar o uso do *iVProg2* com um ambiente de programação textual similar em termos de tecnologia e recursos. Os requisitos do sistema de programação textual são: funcionar no SGC *Moodle*, ter correção automática de exercícios e depender do *Java*. Na seção seguinte, explicamos a relação do protocolo *NASA-TLX* com este trabalho. Em seguida, explicamos como se deu a preparação do curso (4.3). Na seção ??, explicamos sobre o protocolo utilizado para a mensuração da carga de trabalho e, em seguida, explicamos sobre o desenvolvimento de um módulo do protocolo para o *Moodle*.

4.1 Protocolo *NASA-TLX*

O protocolo *NASA-TLX* foi criado para mensurar a *carga de trabalho* incorrida sobre um sujeito que realiza uma tarefa ou opera alguma interface complexa. Na época, segundo os autores, a motivação para realizar os estudos era a falta de pesquisas e cuidados teóricos relacionados à operação dessas interfaces. Além disso, os autores também tinham a intenção de criar um banco de dados com uma grande variedade de exemplos de aplicações do protocolo para que as pessoas pudessem compreender os princípios gerais e observar as relações providas pelos resultados (Hart e Staveland, 1988).

Em 2006, um mapeamento das áreas em que o *NASA-TLX* vinha sendo utilizado foi realizado (Hart, 2006). Constatou-se que o protocolo vem sendo adaptado e utilizado na análise de interfaces de controle de tráfego aéreo, cabines de aviões (civis e militares), usinas de energia, controle de veículos terrestres, veículos espaciais, controles de unidades robóticas, entre outros.

Apesar do grande uso nas indústrias e operações de interfaces complexas, também podemos encontrar sua aplicação relacionada à avaliação de software educacionais, como no caso da tese de doutorado de Santos (2009). No trabalho de Santos, o protocolo foi utilizado para avaliar a inserção de um recurso computacional num ambiente de educação à distância, comparando a carga de trabalho de dois grupos (experimental e de controle). Tais aferições são importantes nos processos educacionais para validar os benefícios promovidos pelo aplicativo (Santos, 2009), pois se a carga de trabalho do grupo experimental for menor do que do grupo de controle, então há benefício real.

No caso desta dissertação de mestrado, o protocolo foi utilizado para medir a carga de trabalho dos aprendizes de programação durante a utilização de dois ambientes distintos. Ao comparar os resultados do protocolo, a intenção era a de testar a hipótese de que a utilização de PV tem menor demanda mental em relação à utilização de programação textual.

Basicamente, o protocolo é aplicado logo após a realização de uma tarefa e é composto de duas etapas. Numa delas, é realizado o preenchimento de seis escalas numéricas limitadas entre 0 e 100. As escalas que compõem o protocolo são descritas a seguir:

- **Demanda Mental (DM):** Quanto das atividades mental e perceptual foram exigidas (por exemplo: pensar, decidir, calcular, lembrar, observar, procurar, etc.)? A tarefa foi fácil ou difícil, simples ou complexa?
- **Demanda Física (DF):** Quanto esforço físico foi exigido durante a atividade (por exemplo: clicar, digitar, empurrar, puxar, controlar, ativar, etc.)? A tarefa foi rápida ou morosa, leve ou pesada?
- **Demanda Temporal (DT):** Quanta pressão em relação ao tempo você sentiu no ritmo de execução da tarefa? O ritmo era lento e vagaroso ou rápido e frenético?
- **Desempenho (D):** Quão bem sucedido você se considera ao realizar os objetivos da tarefa? Ficou satisfeito com o seu desempenho no cumprimento da tarefa?
- **Esforço (E):** O quão árduo você teve de trabalhar (mental e fisicamente) para alcançar seu nível de desempenho?
- **Frustração (F):** Quão inseguro, desencorajado, irritado, estressado ou aborrecido versus seguro, encorajado, satisfeito, relaxado ou complacente você se sentiu durante a tarefa?

No intuito de aplicar o protocolo dentro do ambiente *Moodle* numa situação de educação à distância, criamos um módulo para o SGC que permitia a coleta desses dados. A implementação foi realizada a partir de um código em *HTML* gentilmente cedido por Keith Vertanen, da *University Montana Tech*. Apesar do *NASA-TLX* no sítio de Keith estar em *HTML* e *Javascript*, não há um recurso para armazenamento das respostas ou identificação dos usuários. Dessa forma, a implementação de um módulo de atividades do *Moodle* permitiu que fossem armazenados tanto os valores atribuídos às escalas do protocolo quanto a identificação dos aprendizes. Ademais, como o protocolo precisa ser aplicado logo após a realização de uma tarefa, a proximidade entre as atividades e o protocolo cria uma configuração mais confortável para o preenchimento. A figura 4.1 ilustra a proximidade com o bloco de atividades.

Figura 4.1: Captura da tela do módulo *NASA-TLX* no *Moodle* - proximidade entre atividades e protocolo

A figura 4.2 ilustra a situação em que essas escalas são preenchidas. As seis escalas são disponibilizadas numa única página, na qual os usuários devem clicar sobre a região retangular que julgarem representar sua percepção em relação a demanda daquela escala durante a realização da atividade. Ao clicar sobre a região, ela se tornará cinza, demarcando a faixa indicada pelo usuário.

Na etapa seguinte é feita uma escolha pareada que engloba cada uma das escalas, para decidir entre cada par, qual escala foi percebida como de maior influência durante a realização da tarefa. Como são seis escalas, temos 15 pares possíveis. As escolhas entre os quinze pares gera um sistema de ponderação, de forma que cada escala recebe um peso que pode variar entre $\frac{0}{15}$ e $\frac{5}{15}$, dependendo da escolha do usuário que está preenchendo o protocolo. Finalmente, a partir dos valores atribuídos às escalas e seus respectivos pesos, é possível calcular um valor de carga de trabalho geral realizando a soma dos produtos entre cada escala e seus respectivos pesos. A figura 4.3 ilustra a situação em que ocorre uma das 15 telas de escolha entre os pares de escalas.

PÁGINA INICIAL / MEUS CURSOS / MOOC / PROG2014-T2 / 2. SELEÇÃO / NASA TLX - COMO FORAM OS DESAFIOS?

Nasa TLX - Como foram os desafios?

Responda ao questionário

Questionário sobre a tarefa - Parte 1

Clique em cada escala no ponto que melhor representa sua experiência durante a tarefa

Demanda mental
Baixa Alta
Quanto das atividades mental e perceptual foram exigidas (por exemplo: pensar, decidir, calcular, lembrar, observar, procurar, etc)? A tarefa foi fácil ou difícil, simples ou complexa, rigorosa ou tolerante?

Demanda física
Baixa Alta
Quanto esforço físico foi exigido (por exemplo: empurrar, puxar, virar, controlar, ativar, etc)? A tarefa foi fácil ou exigente, morosa ou rápida, leve ou pesada?

Demanda de tempo
Baixa Alta
Quanta pressão em relação ao tempo você sentiu no ritmo de execução da tarefa? O ritmo era lento e vagaroso ou rápido e frenético?

Desempenho
Boa Ruim
Quão bem sucedido você se considera ao realizar os objetivos da tarefa? Ficou satisfeito com o seu desempenho no cumprimento dessas metas?

Esforço
Baixo Alto
O quão duro você teve de trabalhar o (mental e fisicamente) para alcançar o seu nível de desempenho?

Frustração
Baixa Alta
Quão inseguro, desencorajado, irritado, estressado e aborrecido versus seguro, gratificado, satisfeito, relaxado e complacente você se sentiu durante a tarefa?

Continuar >>

Figura 4.2: Captura da primeira tela do protocolo *NASA-TLX*: atribuição de valores para as escalas individuais.

A primeira implementação do módulo se destina apenas à versão 2.6.1 (Build: 20140113) do *Moodle*. Nessa versão, armazena-se numa tabela criada para o protocolo os valores atribuídos para cada uma das escalas e os pesos obtidos a partir das escolhas entre os pares de escalas. Esses dados podem ser analisados estatisticamente a partir de sua exportação para o formato CSV (Comma-separated values), que pode ser lido por programas de planilhas eletrônicas.

Depois de implementarmos o módulo, nós o integramos ao sistema *Moodle*, organizamos o material do curso e demos início ao experimento. Na seção seguinte descrevemos o sistema de programação textual que foi utilizado para comparação com o *iVProg2*.

4.2 Virtual Programming Lab - VPL

O VPL é um módulo oficial para o *Moodle* desenvolvido na *Universidad de las Palmas Gran Canaria*, na Espanha. Ele permite a construção de algoritmos em diversas linguagens de programação. Foi contruído com a tecnologia *Java* e funciona como um *Applet*. Para ser utilizado neste experimento, tivemos que realizar a tradução módulo para o português, caracterizando mais uma contribuição desta pesquisa.

O VPL possui um editor de texto e foi utilizado com a linguagem de programação *C*. A partir da extensão do arquivo, esse editor identifica o tipo de linguagem utilizada, inclusive, colorindo de modo diferente as palavras reservadas (destaque/*highlight*). A figura 4.4 é uma captura de tela de exercício via *Moodle/VPL*, com o algoritmo de um dos alunos, tendo passado nos dois testes (nota máxima).

Os exercícios podem ser executados com apenas um clique no botão *Executar*. Ao clicar nesse botão, o código é enviado ao servidor para ser compilado e, se não existirem erros de sintaxe, executado. A correção automática dos exercícios segue um sistema parecido com o do *iVProg2*,

PÁGINA INICIAL / MEUS CURSOS / MOOC / PROG2014-T2 / 2. SELEÇÃO / NASA TLX - COMO FORAM OS DESAFIOS?

Nasa TLX - Como foram os desafios?

Responda ao questionário

Questionário da tarefa - Parte 2

Clique no fator que representa o fator de maior contribuição para a carga de trabalho da tarefa.

Demanda de tempo Quanta pressão em relação ao tempo você sentiu no ritmo de execução da tarefa?
O ritmo era lento e vagaroso ou rápido e frenético?

ou

Demanda mental Quanto das atividades mental e perceptual foram exigidas (por exemplo: pensar, decidir, calcular, lembrar, observar, procurar, etc)? A tarefa foi fácil ou difícil, simples ou complexa, rigorosa ou tolerante?

Figura 4.3: Captura da segunda tela do protocolo *NASA-TLX*: uma das telas destinadas à escolha entre pares para estabelecer os pesos de cada escala

pois o professor deve definir testes com valores de entradas e saídas esperadas.

Um inconveniente desse módulo é que, por questões de segurança, o VPL exige que a execução ocorra dentro de um sistema de proteção (*jail system*) que, necessariamente, deve ser instalado num servidor diferente daquele onde se encontra o *Moodle*. Dessa forma, para que tudo funcione são necessários dois servidores.

Na seção seguinte explicamos sobre a configuração do experimento.

4.3 Sobre o experimento

Para realizarmos a comparação entre os ambientes de programação precisávamos de um cenário onde os recursos computacionais de *Programação Visual* e *Programação Textual* pudessem ser comparados. Dessa forma, elaboramos um curso experimental.

O curso de Introdução à Programação foi de curta duração (4 semanas) e tinha a intenção de introduzir os primeiros conceitos de algoritmos e programação; desde o conceito de variáveis até a construção de laços de repetição.

A fase de inscrição foi completamente aberta ao público, não requerendo qualquer documento e também não foi restrita a qualquer instituição ou nível de ensino. Todas as inscrições foram realizadas por meio do preenchimento de formulário na internet e confirmadas a partir de um e-mail válido. As informações requeridas no processo de inscrição eram: nome completo, e-mail, instituição ou instituto de origem, número USP quando existente e também eram perguntados se já possuíam alguma experiência em programação. Além disso, todas as pessoas registradas foram notificadas e estavam cientes de que era um curso de cunho experimental e, ao participarem, estariam concordando que os dados das atividades, formulários e demais poderiam ser utilizados para fins acadêmicos. Essa notificação, além de presente no sistema de registro, era enviada para o e-mail dos participantes assim que se cadastravam no sistema. Entretanto, a propaganda foi realizada por um curto período de tempo (4 semanas) e se deu principalmente na Universidade de

Figura 4.4: Captura de tela com VPL no Moodle.

São Paulo. Ainda assim, conseguimos realizar a propaganda em duas outras instituições de ensino. No fim das inscrições, 54% dos inscritos eram da Universidade de São Paulo e 46% de outras instituições de ensino.

A divulgação para o curso foi realizada em 3 diferentes canais: apresentação oral de até 5 minutos em 15 salas de ingressantes na Universidade de São Paulo, distribuição de panfletos na mesma universidade e, também, foi divulgado um convite nas redes sociais de dois membros do grupo de pesquisa ao qual o autor desta dissertação está vinculado.

O curso teve 217 voluntários inscritos de diversas instituições de ensino. No entanto, a maioria deles era formada por aprendizes da escola de engenharia da Universidade de São Paulo. O motivo se deve a realização de 11 das 15 apresentações orais a respeito do curso nessa escola; além disso, também foi onde distribuímos os panfletos impressos.

Basicamente, dividimos os inscritos em 3 grupos: um grupo experimental com *iVProg2 Java* (G1), um grupo de controle com o VPL (G2) e um grupo experimental com *iVProg* em *HTML5* (G3). Dado que o propósito deste experimento foi avaliar a carga de trabalho mental do uso do modelo de PV versus o uso do modelo de programação textual, nós preparamos ambientes de aprendizagem equivalentes para todos os grupos. Dessa forma, os inscritos não foram alocados de forma aleatória. Ao invés disso, tentamos balanceá-los usando o seguinte critério: (a) balanceamos o número de aprendizes por instituição em cada grupo, (b) para cada instituição balanceamos o número de aprendizes que possuíam experiência com programação em cada grupo e (c) para cada instituição, também balanceamos o número de aprendizes que não possuíam experiência em programação em cada grupo. Seguindo este protocolo, pudemos verificar a taxa de permanência e

participação para os grupos de aprendizes com diferentes perfis.

No intuito de evitar problemas com a instalação da máquina virtual *Java*, o que evitaria a execução dos *Applets* tanto do *iVProg Java* quanto do VPL, criamos tutoriais em vídeo explicando como preparar seus computadores com as ferramentas necessárias para acompanhar as aulas. Os aprendizes receberam as credenciais para acessar o ambiente *Moodle*, já com acesso aos tutoriais, uma semana antes de iniciarmos as atividades do curso.

O material instrucional foi desenvolvido de forma independente do paradigma de programação envolvido e consistia de quatro módulos, cada um contendo um bloco de atividades seguidas do protocolo *NASA-TLX* para a avaliação da carga de trabalho mental empregada na realização dessas atividades. Além disso, todos os módulos possuíam um canal de comunicação para o esclarecimento de dúvidas. Para algumas explicações conceituais a respeito dos exercícios, um modelo de fluxograma foi introduzido (exemplo: figura 4.5). Dessa forma, poderíamos evitar entrar em detalhes do ambiente de programação ao explicarmos o funcionamento de, por exemplo, um algoritmo com seleção. Quando uma menção ao ambiente era inevitável, produzíamos imagens de exemplo em cada ambiente, gerando materiais distintos para cada grupo.

Figura 4.5: Fluxograma utilizado para explicação de algoritmo de seleção.

Os conteúdos teóricos e práticos organizados nos módulos envolviam os conceitos de algoritmos, variáveis e seus tipos, entrada e saída de dados, expressões aritméticas e booleanas, seleção e construções de laços de repetição. A figura 4.6 mostra uma tela com o ambiente Moodle preparado para o curso. A seguir, descrevemos cada um dos módulos:

- **Módulo 1 “Algoritmos”:** esse módulo era composto da definição de algoritmos e explicações de conceitos básicos para começarem a programar. Esses conceitos eram: variáveis e seus tipos, entrada e saída de dados, expressões booleanas, expressões aritméticas e atividades com atribuição de valores às variáveis a partir da entrada de dados.
- **Módulo 2 “Seleção”:** o módulo de seleção começava com comentários a respeito das atividades do módulo anterior, então definíamos a estrutura de seleção e explicávamos o seu uso por meio de exemplos. Por fim, havia um bloco de atividades relacionadas à esse conceito.
- **Módulo 3 “Laços de repetição”:** mais uma vez, iniciamos o módulo com a resolução das atividades anteriores por meio de textos explicativos. Depois, apresentamos os conceitos dos laços do tipo “enquanto”, “para” e “repita”, mostramos exemplos de cada um e propusemos atividades relacionadas a esses conceitos.
- **Módulo 4 “Fechamento”:** esse era o último módulo. Nele, também discutimos as atividades propostas no módulo anterior. Entretanto, nenhum novo conceito foi apresentado. Ao invés disso, havia um número de atividades maior e que também eram mais complexas. Essas atividades deveriam ser realizadas tanto no sistema no qual cada grupo estava habituado quanto de forma discursiva, em linguagem natural.

Na seção seguinte apresentamos os dados referentes a carga de trabalho obtidos durante o curso.

Figura 4.6: Parte de captura de tela do ambiente no qual se deu o curso. Em destaque na imagem está o módulo “Algoritmos”

4.4 Resultados do curso experimental

Depois de prepararmos o conteúdo do curso e o método para coleta de dados referentes à carga de trabalho, o curso experimental teve início. Os dados obtidos com o protocolo foram separados e preparados para a análise estatística. As escalas do protocolo evidenciam a percepção dos estudantes em relação à carga de trabalho incorrida durante a realização das atividades. As escolhas pareadas entre as escalas (demanda mental, demanda física, etc.) fornecem um peso para cada uma delas. Lembramos que, ainda, seria possível obter um valor geral a partir da somatória dos produtos de cada escala por seu respectivo peso. Quanto maior o valor geral, maior é a carga de trabalho incorrida durante a realização de uma atividade. Porém, decidimos analisar essas escalas, com suas ponderações, separadamente. De toda forma, é importante observar qual das escalas teve maior impacto durante a realização das atividades. Nos esforçamos para que a variável que exerceria maior impacto no estudo fosse apenas o ambiente de programação utilizado. Na subseção seguinte realizamos a análise dos voluntários inscritos.

4.4.1 Análise dos inscritos

Apesar da quantidade de inscritos superar os 200 voluntários, infelizmente, 74 deles não confirmaram a inscrição. Além disso, dos usuários que confirmaram a inscrição uma grande quantidade

Grupo	Sistema	Com experiência	Sem experiência	Total
G1	<i>iVProg</i>	31	41	72
G2	VPL	31	41	72

Tabela 4.1: Distribuição dos voluntários nos grupos G1 e G2

Grupo	Sistema	Com experiência	Sem experiência	Total
G1	<i>iVProg</i>	9	16	25
G2	VPL	7	14	21

Tabela 4.2: Número de aprendizes do G1 e G2 que não acessaram o sistema

Grupo	Sistema	Com experiência	Sem experiência	Total
G1	<i>iVProg</i>	2	3	6
G2	VPL	2	8	10

Tabela 4.3: Número de aprendizes que acessaram o sistema na última semana do curso

acessou o sistema e não realizou atividades. Os grupos experimentais foram os mais prejudicados. Além disso, o G3 teve pouquíssimos aprendizes participando do final do curso e preenchendo o protocolo *NASA-TLX*. Outro fator importante é que as funcionalidades do *iVProg2* em *HTML5* ainda não tinham todas sido mapeadas do *iVProg2 Java*. Dessa forma, limitamos a análise dos dados às turmas do *iVProg2 Java* (G1) e VPL (G2). Essa decisão impediu que fizéssemos qualquer comparação significativa em termos de influência da tecnologia (*Java versus HTML5*) para as versões do *iVProg2*. No entanto, ainda pudemos conduzir uma análise na direção de avaliar o objetivo desta dissertação, pois conseguimos comparar G1 e G2 em termos de *Programação Visual* e *Programação Textual*. A tabela 4.1 exibe a distribuição dos voluntários em termos de experiência prévia com programação nesses dois grupos. Já a tabela 4.2 traz a informação da quantidade de aprendizes que não acessou o sistema nesses grupos e sua experiência prévia com programação.

O baixo índice de participação pode ser explicado principalmente por três fatores: o período do curso, a origem dos estudantes inscritos e, também, o fato de serem voluntários. Além disso, a última semana de nosso curso online coincidiu com o final de semestre da USP, o que provavelmente explica o número baixíssimo de estudantes que acessaram o sistema na última semana de curso. Isso é exibido na tabela 4.3, entretanto, o número de respondentes do protocolo *NASA-TLX* é ainda menor do que o número de acessos ao sistema.

Dado que não entrevistamos os estudantes, não é possível explicar exatamente o motivo da baixa adesão dos voluntários. No entanto, a hipótese do fato de serem voluntários e a ausência de emissão de certificados no curso estão de acordo com um problema comum enfrentado em cursos abertos online, incluindo os da categoria *Massive Open Online Course* (MOOC) (Seaton *et al.*, 2014).

4.4.2 Análise dos dados do *NASA-TLX*

Para a análise estatística dos dados providos pelo protocolo *NASA-TLX*, aplicamos um teste não-paramétrico. A escolha dessa abordagem se justifica pelo pequeno número de respondentes do protocolo e na aparente distribuição assimétrica dos dados. Ao utilizar um teste não-paramétrico a análise sofrerá um menor impacto devido a pequena quantidade de respondentes e a eventuais *outliers*. Considerando os valores de uma mesma escala nos dois grupos, nós aplicamos o teste estatístico de *Wilcoxon-Mann-Whitney* (WMW) para analisar o impacto do ambiente de programação utilizado (Sprenst e Smeeton, 2007, Capítulo 6 - pp. 151). Sucintamente, o teste se trata da definição de postos baseados nos valores das amostras. Quanto maior o valor do dado coletado, maior é

o seu posto. Assim, é possível encontrar qual dos dois grupos tem os maiores postos para uma dada escala do *NASA-TLX*, com um certo nível de significância, nos permitindo inferir sobre a população da qual a amostra foi obtida. Durante a análise de nossos dados, usamos o intervalo de confiança $\alpha = 0.05$. A análise dos postos permite a caracterização da distribuição das amostras, identificando se são equilibradas ou se, e como, são desequilibradas. A seguir, definimos as hipóteses nula e alternativa consideradas para essa análise.

- H_0 : distribuição_{G1} = distribuição_{G2}

Significa que as distribuições dos postos são equilibradas, portanto não há diferença estatisticamente significativa entre os grupos G1 e G2. Dessa forma, H_0 nos diz que a carga de trabalho é equilibrada para os usuários do *iVProg2* e do VPL.

- H_1 : distribuição_{G1} < distribuição_{G2} Significa que as distribuições dos postos não são equilibradas e, além disso, os postos de menor valor estão no G1 (*iVProg2*). Isso nos permitiria dizer que a carga de trabalho incorrida aos usuários do *iVProg2* é menor do que a carga incorrida aos usuários do G2 (VPL).

O protocolo *NASA-TLX* permitiu observações interessantes sobre o uso dos sistemas *iVProg2* e VPL durante a execução das atividades do curso. O protocolo forneceu dados sobre: a demanda mental (DM), demanda física (DF), demanda temporal (DT), desempenho (D), esforço (E) e frustração (F), de acordo com o ponto de vista do aprendiz. Como dissemos anteriormente, ele foi aplicado após cada bloco de atividades com *iVProg2* (no G1) ou com C (no G2). Nossa intenção era identificar qual das escalas tinha maior diferença, na percepção dos estudantes, durante a realização das atividades.

Na figura 4.7 estão presentes dois gráficos do tipo *boxplot* referentes ao primeiro bloco de atividades. Nessa figura podemos observar que no G1, a escala com maior impacto foi a referente ao esforço (E), cuja mediana vale 8. No grupo G2, o esforço foi menor, no entanto a demanda mental (DM) e temporal (DT) foram observadas como significativamente maiores. A DM tem mediana de 4 no G1 e 8 no G2. Entretanto, os valores máximos dessas duas escalas no G2 estão abaixo de 25, enquanto no G1 podemos encontrar valores para DM entre 25 e 30. Para confirmar essas observações realizadas a partir do *boxplot*, usamos o teste WMW, que está representado na tabela 4.4. Nessa tabela os valores para p foram calculados a partir das amostras com as escalas ponderadas de G1 e G2 com as hipóteses: H_0 : distribuição_{G1} = distribuição_{G2} e H_1 : distribuição_{G1} < distribuição_{G2}, com uma única exceção, que é a escala desempenho (D), para a qual consideramos H_1 : distribuição_{G1} > distribuição_{G2}. O motivo para existir essa exceção é a oposição dos valores da escala, pois quanto maiores os valores, pior era o desempenho percebido pelos voluntários.

	E	F	DM	D	DF	DT
p -value	0.6409	0.6676	0.1167	0.3002	0.8272	0.1132

Tabela 4.4: Teste WMW aplicado em cada uma das escalas ponderadas de G1 e G2 no primeiro bloco de atividades

Para cada valor de p , podemos rejeitar a hipótese nula (H_0) e aceitar H_1 se $p \leq 0.05$. Concluímos, então, que não há evidência suficiente para rejeitar a hipótese nula. Entretanto, vale ressaltar que os valores p da tabela 4.4 para demanda mental e temporal são bem menores em relação às outras escalas. Os dados coletados a partir das atividades dos estudantes estavam de acordo com a percepção acima. O número de tentativas para resolver os problemas no G2, usando C+VPL (programação textual), foi de até 15 vezes e não era incomum encontrar um número de tentativas maior do que 5. No G1, grupo que usou o *iVProg2*, o número máximo de tentativas foi 4 (apenas 1 estudante); mais do que isso, a situação mais comum era de que o estudante submetia a resposta correta na primeira tentativa.

Figura 4.7: Comparação entre boxplots gerados a partir dos dados do G1 (22 respostas) e G2 (12 respostas) para o primeiro bloco de atividades

A figura 4.8 apresenta a mediana e distribuição dos dados referentes às escalas do segundo bloco de atividades, presente no módulo 2 do curso. Novamente, a demanda mental (DM) é menor no G1 (6.33), do que no G2 (11.33). Na tabela 4.5 é apresentado o resultado do teste WMW para cada uma das escalas. Mais uma vez, não há evidências suficientes para rejeitar a hipótese nula. Entretanto, vale destacar que os valores de p para as escalas de esforço (E) e demanda mental (DM) são menores em comparação aos outros valores de outras escalas. Isso é confirmado na figura 4.8 ao observarmos que as medianas do esforço (E) e demanda mental (DM) são menores no G1 em comparação ao G2, sendo respectivamente, 4.835 contra 9.165 e 6.33 contra 11.33.

Figura 4.8: Comparação entre boxplots gerados a partir dos dados do G1 (6 respostas) e G2 (6 respostas) para o segundo bloco de atividades

Em relação ao número de submissões no G2 ($C+VPL$), encontramos um valor consideravelmente maior do que no G1 ($iVProg2$). No G2, o número máximo de tentativas para resolver os problemas foi de 12 e era comum encontrar situações na qual os estudantes que fizeram as tarefas precisavam de mais de 5 tentativas. No G1, entretanto, o número máximo de tentativas foi 4 e a

	E	F	DM	D	DF	DT
<i>p</i> -value	0.0744	0.8935	0.1473	0.2071	0.2023	0.532

Tabela 4.5: *Teste WMW aplicado em cada uma das escalas ponderadas de G1 e G2 no segundo bloco de atividades*

situação mais comum era de estudantes enviando a resposta correta na primeira tentativa.

Na figura 4.9 apresentamos os resultados das escalas para o terceiro bloco de atividades, presentes no módulo 3. Nesse bloco, destaca-se a escala do esforço (E) percebido pelos estudantes do G1 em relação ao esforço percebido pelos estudantes do G2, cujos valores das medianas são, respectivamente, 6 e 13.33. Mais uma vez, contruímos a tabela 4.6 aplicando o método WMW nas escalas ponderadas dos dois grupos.

Figura 4.9: *Comparação entre boxplots gerados a partir dos dados do G1 (5 respostas) e G2 (7 respostas) para o terceiro bloco de atividades*

	E	F	DM	D	DF	DT
<i>p</i> -value	0.1452	0.6028	0.4353	0.7435	0.9642	0.6028

Tabela 4.6: *Teste WMW aplicado em cada uma das escalas ponderadas de G1 e G2 no terceiro bloco de atividades*

A análise dos dados referentes ao terceiro bloco de atividades nos permitiu observar que a mediana da demanda mental (DM) era de 15 no G1 e de 16 no G2, uma pequena diferença comparada às situações anteriores. Em relação ao esforço (E), no G1 a mediana vale 6 e no G2 vale 13.33. A respeito do número de submissões no G1, tivemos poucos estudantes realizando as atividades, entretanto as submissões estavam corretas em apenas 1 tentativa. No G2, o número de submissões foi ligeiramente maior; no entanto, o número de tentativas atingiu o valor máximo de 17.

Assim como mencionamos anteriormente, o número de submissões para o módulo 4 não foi suficiente para podermos realizar alguma análise comparativa. Portanto, esses foram os dados coletados a partir do protocolo *NASA-TLX*. A seguir, apresentamos e discutimos algumas informações coletadas a partir de um questionário que enviamos através da internet para os voluntários.

4.4.3 Análise do questionário

O protocolo *NASA-TLX* nos permitiu observar que durante a realização das atividades alguns usuários se percebiam frustrados. No intuito de compreender esse fenômeno, a baixa quantidade de voluntários realizando o curso e, além disso, coletar dados qualitativos a respeito do curso experimental, construímos um questionário na ferramenta *Google Docs* e enviamos para os e-mails de todos os inscritos. As questões contidas no questionário eram as seguintes:

- Q1) Você conseguiu acessar o sistema do curso de Introdução à Programação?
- Q2) Se você acessou, em qual turma você foi registrado?
- Q3) Se você fez as atividades e leu o material instrucional, como poderíamos melhorar o ambiente e o material?
- Q4) Qual a sua opinião a respeito da ferramenta utilizada para criar os algoritmos?
- Q5) Se você não acessou o sistema do curso ou desistiu de terminar o módulo I, poderia compartilhar o motivo conosco? Se sim, preencha o campo abaixo nos contando o motivo.

No total, obtivemos 27 respostas. Dentre elas, apenas 3 eram de usuários que não acessaram o sistema. Desses três usuários, estávamos especialmente interessados nas respostas à questão Q5, que listamos a seguir:

- “Conseguí entrar na primeira semana, mas perdi minha senha e não consigo recuperar”
- “Não consegui acessar o sistema, não recebi nenhum email, com senha ou informações de acesso ao sistema”
- “Não recebi o E-mail com dados para acessar, enviei um email a vocês que responderam prontamente, mas não tive tempo para acessar o módulo!”

A partir dessas respostas percebemos que o sistema não conseguiu atingir as caixas de e-mail de todos os voluntários ou os e-mails caíram em suas caixas de lixo eletrônico. No caso do voluntário que não recebeu o e-mail no início do curso, mas pediu o acesso posteriormente, acabou por nos alertar dessa falha e, além disso, nos mostrou que a participação no curso não era uma de suas prioridades.

No geral, as respostas dos participantes mostravam que estavam satisfeitos com o curso, com a metodologia e com as ferramentas que usaram. Mesmo com os tutoriais para a instalação da máquina virtual Java, alguns voluntários relataram problemas para executar o *Applet* tanto do *iVProg2* quanto do VPL. Nós recebemos e-mails de 3 participantes pedindo ajuda na instalação do *Java* e na configuração do nível de segurança para permitir a execução de *Applets* no navegador. Adicionalmente, outro participante escreveu como resposta no questionário que “...Apesar de um leve problema em relação à configuração do java no início, ela funciona muito bem...”.

Recebemos, também, muitos elogios em relação ao curso e metodologia “Fiquei bastante intrigado com a ferramenta, a forma como ela provê entradas para a aplicação sem argumentos em *main* e após isso lê seletivamente as saídas é muito prática para a correção dos exercícios, além disso o fato de haver feedback instantâneo foi bastante útil.”. Outro comentário positivo foi: “Eu acredito que a ferramenta é útil ao ensino inicial de algoritmos...”. Entretanto, um dos estudantes escreveu: “A idéia da ferramenta como metodo de ensino da logica de programacao é boa, só que gasta muito tempo para fazer o codigo (declarar variaveis toda hora, etc)...”.

Em relação ao conteúdo do curso, recebemos algumas sugestões como esta: “Simulações prontas de vários algoritmos (verificar paridade, contar dígitos, etc.) poderiam ser apresentadas junto ao seu conteúdo teórico. Isso facilitaria o aprendizado e a valorização dessas técnicas a usuários sem base matemática sólida.”.

4.5 Considerações finais

Dado que o curso adotou uma abordagem online com voluntários e, além disso, não oferecemos qualquer tipo de certificação, nós observamos uma redução consistente no número de aprendizes no decorrer do curso.

Dos mais de 200 inscritos, o último bloco de atividades teve apenas 8 resoluções de atividades, considerando os dois grupos (G1 e G2). Acreditamos que isso pode ser explicado pela liberdade do modelo do curso e o fato de os aprendizes serem, em sua maioria, estudantes que acabaram de ingressar na universidade. Com o aumento das atividades na universidade, a participação caiu drasticamente.

Outro problema comum dos cursos online é a relação entre a participação dos estudantes e a dificuldade da atividade, que é inversamente proporcional. Um exemplo claro dessa relação é observado no trabalho de [Seaton et al. \(2014\)](#) que analisou um *Massive Open Online Course* (MOOC) sobre circuitos e componentes eletrônicos promovido pelo MIT. O autor constatou que a taxa de estudantes que nunca acessaram o sistema do curso era de 29% (46000 de 154000 nunca apareceram). Isso nos faz pensar que uma taxa próxima de 30% seja uma parte intrínseca desse modelo de educação à distância. Além disso, no mesmo trabalho, o autor afirma que em cursos com certificação os aprendizes são mais dedicados e a taxa de abandono cai. O nosso curso não ofereceu certificação devido a sua característica experimental.

Apesar do quorum reduzido no nosso curso online, o protocolo *NASA-TLX* indicou que a *Programação Visual* parece ser uma boa opção para a introdução de conceitos básicos da programação.

O capítulo seguinte sintetiza o trabalho realizado e apresenta as conclusões.

Capítulo 5

Conclusão e Trabalhos Futuros

Neste capítulo sintetizamos as conclusões resultantes do processo de implementação e do curso experimental aplicado. Além disso, apresentamos as sugestões de trabalhos futuros.

5.1 Conclusões

Neste trabalho de mestrado, construímos um aplicativo de *Programação Visual* a partir de uma *Linha de Produto de Software* tendo em vista um aplicativo anterior. Desse aplicativo, mapeamos funcionalidades e o estendemos. Assim, demos continuidade a duas dissertações de mestrado anteriores a esta, que foram os trabalhos de Dalmon (2012) e Kamiya (2010).

Na seção 1.3, definimos o objetivo geral como “reconstruir e usar o aplicativo de programação visual *iVProg* e comparar sua carga de trabalho em relação a programação textual”. De fato, o *iVProg*, construído a partir do *Alice* por Reginaldo Rideaki, foi reconstruído, dando origem ao *iVProg2*. Tal reconstrução, se deu sobre o arcabouço de aplicação desenvolvido por Danilo Dalmon. Além disso, nós utilizamos o *iVProg2* num curso aberto e online, de caráter experimental, sobre conceitos básicos de programação. Com os resultados obtidos no curso, o aplicativo mostrou-se uma alternativa interessante para introduzirmos os conceitos básicos de programação aos aprendizes, principalmente pelos indícios que levantamos a respeito da demanda mental e esforço serem menores em comparação a uma linguagem textual. Assim, no sentido da reconstrução do aplicativo e comparação da carga de trabalho entre a programação visual e a programação textual, consideramos que o objetivo foi atingido.

Sobre os objetivos específicos, apresentamos, a seguir, a discussão a seu respeito:

- *Criar material instrucional para o curso de Introdução à Programação*: um material que abrange vídeos instrucionais, atividades interativas e textos para referência teórica foram criados. Esse material pode ser facilmente distribuído, alterado ou reutilizado, pois está disponível sob licença *Creative Commons* 3.0 na internet. Além disso, está sendo atualizado para a aplicação de uma nova edição do curso experimental que deverá ocorrer no segundo semestre de 2015. Dessa forma, consideramos esse objetivo como cumprido devido a existência do material e sua criação ter sido feita sob supervisão do orientador desta dissertação de mestrado.
- *Criar tutoriais de instalação da máquina virtual Java nos ambientes Linux e Windows*: com a finalidade de evitar grandes complicações na instalação da máquina virtual *Java* durante a realização do curso experimental, criamos tutoriais em vídeo mostrando todos os passos: como verificar se já possui o plugin do *Java* instalado, fazer o download adequado para a arquitetura do processador e como instalar e testar a máquina virtual *Java*. Novos vídeos foram produzidos no decorrer do trabalho devido a atualizações da máquina virtual pela empresa que a mantém. Esses tutoriais estão disponíveis gratuitamente num sítio de compartilhamento de vídeos pela internet.

- *Criar um interpretador para executar o algoritmo criado no iVProg2*: durante a reconstrução da ferramenta *iVProg*, foram necessários o planejamento e construção de um interpretador da linguagem visual para a execução dos algoritmos construídos no aplicativo. Esse interpretador é suficientemente flexível para ser estendido ou utilizado em qualquer outra aplicação *Java*. Sem esse interpretador, nos veríamos obrigados a utilizar um pacote de terceiros que estava compilado e compactado no formato “.jar” (*Beanshell*¹). O uso de um pacote de terceiros diminuiria a flexibilidade do código do *iVProg2*, pois não poderíamos estendê-lo facilmente, o que iria contra a proposta da reconstrução do aplicativo apresentada no objetivo geral. Apesar da linguagem *Java*, a partir de sua versão 1.6, possuir o próprio interpretador, o grupo de pesquisa ao qual o autor é vinculado sempre optou por construir os *iMA* compatíveis com a versão 1.4. Justifica-se essa decisão por termos como objetivo atingir o maior número de usuários possíveis sem obrigá-los a atualizarem suas máquinas virtuais.
- *Criar um módulo do protocolo NASA-TLX para Moodle*: O protocolo *NASA-TLX* é um conjunto de questões com a finalidade de mensuração indireta da carga de trabalho durante a realização de uma tarefa. Na seção 4.1, descrevemos o protocolo e como o utilizamos no projeto. Como não existia uma forma adequada de aplicar o protocolo dentro do ambiente *Moodle*, criamos um módulo para aplicá-lo num cenário de Educação à Distância. Dessa forma, o método para a aferição e a própria ferramenta de aferição adaptada por nós, podem ser utilizadas por outros grupos de pesquisa em diferentes contextos, caso precisem dela dentro do *Moodle*.

De forma geral, acreditamos ter cumprido com os objetivos geral e específicos propostos na seção 1.3. Ainda assim, existem pontos delicados que necessitam de mais discussão. Na seção seguinte, discutimos os desafios e as percepções que tivemos durante a realização do trabalho.

5.2 Considerações finais

Nesta pesquisa desenvolvemos um aplicativo de *Programação Visual* e o utilizamos num curso experimental. Dessa forma, discutiremos esses dois aspectos, desenvolvimento e utilização, e apresentamos os pontos que necessitam de mais reflexão e mais trabalho.

5.2.1 Sobre o desenvolvimento do aplicativo

Podemos afirmar que a contribuição do trabalho de Dalmon (2012) foi significativa, pois a LPS, de fato, ajudou no desenvolvimento do *iVProg2* promovendo o reuso tanto do código-fonte quanto da arquitetura do aplicativo. Apesar do uso de um arcabouço deixar o desenvolvimento menos flexível e do projeto ser desafiador do ponto de vista do programador que o realizou, o programador considera que o processo de desenvolvimento foi bem guiado e o uso da LPS trouxe segurança e confiança ao trabalho que estava sendo realizado.

A proposta de reconstrução do sistema existiu por percebermos o potencial da PV a partir da primeira versão do *iVProg*. Dessa forma, motivados pelos resultados do trabalho de Kamiya (2010), decidimos reconstruir o *iVProg* sobre a LPS para *iMA*. A intenção era deixar o código documentado, facilitando a manutenção e a extensão. Entretanto, é difícil julgarmos se atingimos essa meta, pois o programador que estendeu as funcionalidades do *iVProg*, gerando o *iVProg2*, foi o próprio desenvolvedor do *iVProg2*. Dessa forma, a avaliação desse aspecto ainda está incompleta.

Por fim, o desenvolvimento do aplicativo se deu a partir da experiência do grupo de pesquisa em interfaces interativas voltadas à Educação. Entretanto, sabemos da importância de testes de usabilidade e outros métodos de validação e criação de interfaces, porém não conseguimos realizá-los durante o processo de desenvolvimento do *iVProg2*. Pretendemos, assim, realizar esses testes de usabilidade buscando melhorias na interação, principalmente no que concerne a criação de

¹O *Beanshell* permite a execução de uma *String* na linguagem *Java* dentro de um programa em *Java*. Seria algo similar ao método `eval()` do *Javascript*. O pacote está disponível em: <https://code.google.com/p/beanshell2/>

algoritmos mais longos. Assim como discutimos na seção 4.4.3, os usuários se percebiam mais frustrados na medida em que o algoritmo ficava mais longo. Assim, um processo iterativo ainda será realizado para estudarmos e melhorarmos o *iVProg2*.

5.2.2 Sobre os resultados do curso

O maior desafio encontrado durante a realização deste trabalho se deu durante a aplicação do curso experimental. A dependência de voluntários nos deixou ansiosos ao verificar que a falta de um certificado e a vida pessoal dessas pessoas poderia influenciar tanto na quantidade de dados que poderíamos coletar, principalmente devido à grande demanda de tempo que o curso as incorria. A pequena quantidade de dados compromete a pesquisa no sentido de impedir afirmações mais fortes, entretanto, os indícios levantados a respeito da carga de trabalho e a metodologia utilizada nos motiva a continuar a pesquisa para conseguirmos mais dados a ponto de realizarmos inferências estatísticas mais significativas.

5.2.3 Sugestões de trabalhos futuros

Durante este trabalho de mestrado realizamos esforços para reconstruir e estudar um ambiente de programação que diminuísse a carga de trabalho incorrida aos aprendizes, permitindo que se concentrassem nos conceitos e técnicas de programação. Entretanto, outros estudos são necessários para compreendermos bem os efeitos do ambiente de programação. Além disso, temos a intenção de melhorar a interação e estender o sistema *iVProg2*, com novas funcionalidades e novos sistemas de avaliação. Outro aspecto importante que foi percebido durante o seu uso no curso experimental, é que a tecnologia Java tem apresentado alguns problemas para os usuários menos experientes. Discutiremos esses aspectos a seguir.

Como dissemos anteriormente, a quantidade de dados que conseguimos coletar dos voluntários não nos permite fazer afirmações fortes a respeito da comparação entre os ambientes, ainda assim, conseguimos indícios de que a programação visual pode ser utilizada como instrumento inicial do processo de aprendizagem de programação. Sugerimos, assim, como um trabalho futuro um estudo mais detalhado, num ambiente mais controlado, a respeito das diferenças entre as programações visual e textual. Durante nosso experimento, o último bloco, do qual não obtivemos dados para comparação, iria nos fornecer subsídios para analisar se os aprendizes sem experiência tanto do grupo experimental quanto do de controle, conseguiriam elaborar algoritmos equivalentes e escrevê-los na língua materna. Assim, poderíamos comparar os resultados dos processos de aprendizagem em ambientes de programação diferentes. Essa seria a última análise do experimento. Dessa forma, reforçamos que esse trabalho precisa ser continuado.

Quanto ao desenvolvimento do sistema *iVProg2*, sua avaliação automática é realizada baseando-se em testes que o professor cria no aplicativo. Gostaríamos, ainda, de permitir uma análise de equivalência de algoritmos e expressões. Essa funcionalidade seria útil para o próximo passo que pretendemos dar com o *iVProg2*, ainda na tecnologia Java, que é a inserção de um Sistema Tutor Inteligente. Entretanto, percebemos que a tecnologia Java está sendo prejudicada por algumas decisões da empresa que a mantém. Esse fato motivou a criação do protótipo do *iVProg2* na tecnologia *HTML5*, para a qual pretendemos estudar como migrar os *iMA* e como adaptar o arcabouço de aplicação. Assim, este trabalho conversa e dá continuidade ao apontamento de trabalho futuro presente em Dalmon (2012), quanto a evolução da LPS. Finalmente, deixamos algumas perguntas de pesquisa que surgiram durante a reflexão sobre esta dissertação de mestrado:

- Existem diferenças nos algoritmos criados por aprendizes que usaram aplicativos de PV no início do processo de aprendizagem para aqueles que não usaram? Se existem, quais são elas? São diferenças que implicam em resultados positivos ou negativos?
- Como melhorar a criação de algoritmos do *iVProg* no caso de algoritmos mais longos?
- Como migrar um arcabouço de aplicação da linguagem Java para uma tecnologia tal como o *HTML5*? O que pode ser aproveitado e o que precisará de adaptações?

Referências Bibliográficas

- Adams (2007)** Joel C. Adams. Alice, middle schoolers & the imaginary worlds camps. *SIGCSE Bull.*, 39(1):307–311. ISSN 0097-8418. doi: 10.1145/1227504.1227418. URL <http://doi.acm.org/10.1145/1227504.1227418>. Citado na pág. 14, 25
- Aguilar (2008)** L.J. Aguilar. *Fundamentos de Programação - 3.ed.: Algoritmos, estruturas de dados e objetos*. ISBN 9788580550146. Citado na pág. 11
- Alice (2015)** Alice. Alice. <http://www.alice.org/index.php>, Janeiro 2015. Último acesso em 12/01/2015. Citado na pág. 21
- Areias e Mendes (2007)** Cristiana Areias e António Mendes. A tool to help students to develop programming skills. Em *Proceedings of the 2007 International Conference on Computer Systems and Technologies*, CompSysTech '07, páginas 89:1–89:7, New York, NY, USA. ACM. ISBN 978-954-9641-50-9. doi: 10.1145/1330598.1330692. URL <http://doi.acm.org/10.1145/1330598.1330692>. Citado na pág. 21
- Armoni et al. (2015)** Michal Armoni, Orni Meerbaum-Salant e Mordechai Ben-Ari. From scratch to “real” programming. *Trans. Comput. Educ.*, 14(4):25:1–25:15. ISSN 1946-6226. doi: 10.1145/2677087. URL <http://doi.acm.org/10.1145/2677087>. Citado na pág. 25
- Baranauskas (1994)** M. C. C. Baranauskas. Procedimento, função, objeto ou lógica? linguagens de programação vistas pelos seus paradigmas. Em José Armando Valente, editor, *Computadores e Conhecimento - Repensando a Educação*, páginas 45–63. Citado na pág. 9, 10
- Barbosa et al. (2009)** Marcelo R. G. Barbosa, Felipe A. Silva, Victor M. de A. Oliveira, Valéria D. Feltrim, Luiz G. B. Mirisola, Paulo C. Gonçalves, Josué J. G. Ramos e Lucas T. Alves. Implementação de compilador e ambiente de programação icônica para a linguagem logo em um ambiente de robótica pedagógica de baixo custo. *Simpósio Brasileiro de Informática na Educação - XX SBIE*. Citado na pág. 20, 21
- Beaubouef e Mason (2005)** Theresa Beaubouef e John Mason. Why the high attrition rate for computer science students: Some thoughts and observations. *SIGCSE Bull.*, 37(2):103–106. ISSN 0097-8418. doi: 10.1145/1083431.1083474. URL <http://doi.acm.org/10.1145/1083431.1083474>. Citado na pág. 2
- Beck (2003)** K. Beck. *Test-driven Development: By Example*. A Kent Beck signature book. Addison-Wesley. ISBN 9780321146533. URL https://books.google.com.br/books?id=gFgnde_vwMAC. Citado na pág. 31
- Blinkstein (2008)** P. Blinkstein. O pensamento computacional e a reinvenção do computador na educação. http://www.blinkstein.com/paulo/documents/online/ol_pensamento_computacional.html, Janeiro 2008. Último acesso em 10/01/2015. Citado na pág. 4
- Blockly (2015)** Blockly. Blockly. <https://developers.google.com/blockly/>, Janeiro 2015. Último acesso em 12/01/2015. Citado na pág. 25

- Blog (2013)** Chromium Blog. Saying goodbye to our old friend npapi. <http://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html>, Janeiro 2013. Último acesso em 12/01/2015. Citado na pág. 33
- Böhm e Jacopini (1966)** Corrado Böhm e Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371. ISSN 0001-0782. doi: 10.1145/355592.365646. URL <http://doi.acm.org/10.1145/355592.365646>. Citado na pág. 11
- BRASSCOM (2013)** BRASSCOM. Brasil pula para a 4ª posição no ranking mundial de ti. <http://www.brasscom.org.br/brasscom/Portugues/detNoticia.php?codArea=2&codCategoria=26&codNoticia=402>, Março 2013. Último acesso em 10/01/2015. Citado na pág. 5
- Carlisle et al. (2004a)** Martin C. Carlisle, Terry A. Wilson, Jeffrey W. Humphries e Steven M. Hadfield. Raptor: Introducing programming to non-majors with flowcharts. *J. Comput. Sci. Coll.*, 19(4):52–60. ISSN 1937-4771. URL <http://dl.acm.org/citation.cfm?id=1050231.1050238>. Citado na pág. 14
- Carlisle et al. (2004b)** Martin C. Carlisle, Terry A. Wilson, Jeffrey W. Humphries e Steven M. Hadfield. Raptor: Introducing programming to non-majors with flowcharts. *J. Comput. Sci. Coll.*, 19(4):52–60. ISSN 1937-4771. URL <http://dl.acm.org/citation.cfm?id=1050231.1050238>. Citado na pág. 21
- Charntaweekhun e Wangsiripitak (2006)** K. Charntaweekhun e S. Wangsiripitak. Visual programming using flowchart. Em *Communications and Information Technologies, 2006. ISCIT '06. International Symposium on*, páginas 1062–1065. doi: 10.1109/ISCIT.2006.339940. Citado na pág. 21
- Chidamber e Kemerer (1994)** Shyam R Chidamber e Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493. Citado na pág. 35
- Chun e Ryoo (2010)** Seok-Ju Chun e Jungwoo Ryoo. Development and application of a web-based programming learning system with led display kits. Em *Proceedings of the 41st ACM Technical Symposium on Computer Science Education, SIGCSE '10*, páginas 310–314, New York, NY, USA. ACM. ISBN 978-1-4503-0006-3. doi: 10.1145/1734263.1734369. URL <http://doi.acm.org/10.1145/1734263.1734369>. Citado na pág. 21
- Conway et al. (2000)** Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove e Kevin Christiansen. Alice: Lessons learned from building a 3d system for novices. Em *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '00*, páginas 486–493, New York, NY, USA. ACM. ISBN 1-58113-216-6. doi: 10.1145/332040.332481. URL <http://doi.acm.org/10.1145/332040.332481>. Citado na pág. 15, 21
- Cooper et al. (2000)** Stephen Cooper, Wanda Dann e Randy Pausch. Alice: A 3-d tool for introductory programming concepts. Em *Proceedings of the Fifth Annual CCSC Northeastern Conference on The Journal of Computing in Small Colleges, CCSC '00*, páginas 107–116, USA. Consortium for Computing Sciences in Colleges. URL <http://dl.acm.org/citation.cfm?id=364132.364161>. Citado na pág. 25
- Cross et al. (2004)** J.H. Cross, D. Hendrix e D.A. Umphress. Jgrasp: an integrated development environment with visualizations for teaching java in cs1, cs2, and beyond. Em *Frontiers in Education, 2004. FIE 2004. 34th Annual*, páginas 1466–1467. doi: 10.1109/FIE.2004.1408803. Citado na pág. 21
- Curricula (2013)** Computing Curricula. Computer science curricula 2013, the joint task force on computing curricula. Citado na pág. 9

- Curricula (2001)** Computing Curricula. Computer science, final report, the joint task force on computing curricula. *IEEE Computer Society and Association for Computing Machinery, IEEE Computer Society*. Citado na pág. 9
- Dalmon (2012)** Danilo Leite Dalmon. Uma linha de produto de software para módulos de aprendizagem interativa, julho 2012. Citado na pág. 4, 6, 27, 29, 35, 51, 52, 53
- Dann et al. (2012)** Wanda Dann, Dennis Cosgrove, Don Slater, Dave Culyba e Steve Cooper. Mediated transfer: Alice 3 to java. Em *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education, SIGCSE '12*, páginas 141–146, New York, NY, USA. ACM. ISBN 978-1-4503-1098-7. doi: 10.1145/2157136.2157180. URL <http://doi.acm.org/10.1145/2157136.2157180>. Citado na pág. 17, 25
- de Jesus (2011)** Emanuel de Jesus. Teaching computer programming with structured programming language and flowcharts. Em *Proceedings of the 2011 Workshop on Open Source and Design of Communication, OSDOC '11*, páginas 45–48, New York, NY, USA. ACM. ISBN 978-1-4503-0873-1. doi: 10.1145/2016716.2016729. URL <http://doi.acm.org/10.1145/2016716.2016729>. Citado na pág. 21
- Denny et al. (2011)** Paul Denny, Andrew Luxton-Reilly, Ewan Tempero e Jacob Hendrickx. Understanding the syntax barrier for novices. Em *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education, ITiCSE '11*, páginas 208–212, New York, NY, USA. ACM. ISBN 978-1-4503-0697-3. doi: 10.1145/1999747.1999807. URL <http://doi.acm.org/10.1145/1999747.1999807>. Citado na pág. 3
- Digital (2012)** Convergência Digital. Mercado de software nacional vai crescer 400% em 10 anos. <http://convergenciadigital.uol.com.br/cgi/cgilua.exe/sys/start.htm?infoid=32006&sid=5>, Outubro 2012. Último acesso em 10/01/2015. Citado na pág. 5
- Dijkstra (1979)** E. Dijkstra. Classics in software engineering. chapter Go to Statement Considered Harmful, páginas 27–33. Yourdon Press, Upper Saddle River, NJ, USA. ISBN 0-917072-14-6. Citado na pág. 11
- Dijkstra (2015a)** E. Dijkstra. A case against the go to statement. <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF>, 2015a. Último acesso em 11/01/2015. Citado na pág. 11
- Dijkstra (2015b)** E. W. Dijkstra. On the cruelty of really teaching computer science. <https://www.cs.utexas.edu/~EWD/transcriptions/EWD10xx/EWD1036.html>, Janeiro 2015b. Último acesso em 10/01/2015. Citado na pág. 2
- Djenic et al. (2011)** S. Djenic, R. Krneta e J. Mitic. Blended learning of programming in the internet age. *Education, IEEE Transactions on*, 54(2):247–254. ISSN 0018-9359. doi: 10.1109/TE.2010.2050066. Citado na pág. 18
- do Prado (2008)** Reginaldo do Prado. igraf : Uma proposta de sistema para ensino de função via web, 2008. Citado na pág. 14
- Dorn e Sanders (2003)** B. Dorn e D. Sanders. Using jeroo to introduce object-oriented programming. Em *Frontiers in Education, 2003. FIE 2003 33rd Annual*, volume 1, páginas T4C–22–7 Vol.1. doi: 10.1109/FIE.2003.1263372. Citado na pág. 21
- Eisenmann (2009)** Alexandre Luís Kundrát Eisenmann. icomb: Um sistema para o ensino e aprendizagem de combinatória em ambiente web, 2009. Citado na pág. 14
- Esteves e Mendes (2004)** M. Esteves e A.J. Mendes. A simulation tool to help learning of object oriented programming basics. Em *Frontiers in Education, 2004. FIE 2004. 34th Annual*, páginas F4C–7–12 Vol. 2. doi: 10.1109/FIE.2004.1408649. Citado na pág. 21

- Federici (2011)** Stefano Federici. A minimal, extensible, drag-and-drop implementation of the c programming language. Em *Proceedings of the 2011 Conference on Information Technology Education*, SIGITE '11, páginas 191–196, New York, NY, USA. ACM. ISBN 978-1-4503-1017-8. doi: 10.1145/2047594.2047646. URL <http://doi.acm.org/10.1145/2047594.2047646>. Citado na pág. 21
- Fincher e Utting (2010)** Sally Fincher e Ian Utting. Machines for thinking. *Trans. Comput. Educ.*, 10(4):13:1–13:7. Citado na pág. 1
- Freudenberg et al. (2009)** B. Freudenberg, Y. Ohshima e S. Wallace. Etoys for one laptop per child. Em *Creating, Connecting and Collaborating through Computing, 2009. C5 '09. Seventh International Conference on*, páginas 57–64. doi: 10.1109/C5.2009.9. Citado na pág. 21
- Gamma et al. (1994)** E. Gamma, R. Helm, R. Johnson e J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education. ISBN 9780321700698. Citado na pág. xi, 31, 32
- Glinert e Tanimoto (1984)** E.P. Glinert e S.L. Tanimoto. Pict: An interactive graphical programming environment. *Computer*, 17(11):7–25. ISSN 0018-9162. doi: 10.1109/MC.1984.1658997. Citado na pág. xi, 12, 13
- Gračanin et al. (2005)** Denis Gračanin, Krešimir Matković e Mohamed Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering*, 1(2):221–230. Citado na pág. 12
- Greyling et al. (2006)** J.H. Greyling, C.B. Cilliers e A.P. Calitz. B#: The development and assessment of an iconic programming tool for novice programmers. Em *Information Technology Based Higher Education and Training, 2006. ITHET '06. 7th International Conference on*, páginas 367–375. doi: 10.1109/ITHET.2006.339788. Citado na pág. 21
- Guerra (2010)** Eduardo Martins Guerra. *A Conceptual Model for Metadata-based Frameworks*. Tese de Doutorado, Instituto Tecnológico de Aeronáutica. Citado na pág. 29
- Hart (2006)** Sandra G. Hart. Nasa-task load index (nasa-tlx); 20 years later. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 50(9):904–908. doi: 10.1177/154193120605000909. URL <http://pro.sagepub.com/content/50/9/904.abstract>. Citado na pág. 37
- Hart e Staveland (1988)** Sandra G. Hart e Lowell E. Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. Em Peter A. Hancock e Najmedin Meshkati, editors, *Human Mental Workload*, volume 52 of *Advances in Psychology*, páginas 139 – 183. North-Holland. doi: [http://dx.doi.org/10.1016/S0166-4115\(08\)62386-9](http://dx.doi.org/10.1016/S0166-4115(08)62386-9). URL <http://www.sciencedirect.com/science/article/pii/S0166411508623869>. Citado na pág. 37
- Hazzan et al. (2011)** O. Hazzan, R. Lapidot e N. Ragonis. *Guide To Teaching Computer Science: An Activity-Based Approach*. Springer. Citado na pág. 9, 20
- Helminen e Malmi (2010)** Juha Helminen e Lauri Malmi. Jype - a program visualization and programming exercise tool for python. Em *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, páginas 153–162, New York, NY, USA. ACM. ISBN 978-1-4503-0028-5. doi: 10.1145/1879211.1879234. URL <http://doi.acm.org/10.1145/1879211.1879234>. Citado na pág. 21
- ICG (2015)** ICG. Stanford encyclopedia of philosophy. <http://plato.stanford.edu/entries/computing-history/>, Janeiro 2015. Último acesso em 10/01/2015. Citado na pág. 14, 18
- IDC (2014)** IDC. Estudo da idc brasil aponta crescimento de 8,7% do mercado de serviços de ti. <http://br.idclatin.com/releases/news.aspx?id=1757>, Novembro 2014. Último acesso em 10/01/2015. Citado na pág. 5

- Inkpen (2001)** Kori M. Inkpen. Drag-and-drop versus point-and-click mouse interaction styles for children. *ACM Trans. Comput.-Hum. Interact.*, 8(1):1–33. ISSN 1073-0516. doi: 10.1145/371127.371146. URL <http://doi.acm.org/10.1145/371127.371146>. Citado na pág. 27
- Isotani (2005)** Seiji Isotani. Desenvolvimento de ferramentas no igeom: utilizando a geometria dinâmica no ensino presencial e a distância, 2005. Citado na pág. 14
- Jadud (2006)** Matthew C. Jadud. Methods and tools for exploring novice compilation behaviour. Em *Proceedings of the second international workshop on Computing education research*, ICER '06, páginas 73–84, New York, NY, USA. ACM. ISBN 1-59593-494-4. doi: 10.1145/1151588.1151600. URL <http://doi.acm.org/10.1145/1151588.1151600>. Citado na pág. 4
- Kaila et al. (2010)** Erkki Kaila, Teemu Rajala, Mikko-Jussi Laakso e Tapio Salakoski. Effects of course-long use of a program visualization tool. Em *Proceedings of the Twelfth Australasian Conference on Computing Education - Volume 103*, ACE '10, páginas 97–106, Darlinghurst, Australia, Australia. Australian Computer Society, Inc. ISBN 978-1-920682-84-2. URL <http://dl.acm.org/citation.cfm?id=1862219.1862234>. Citado na pág. 21
- Kamiya (2010)** Reginaldo Rideaki Kamiya. ivprog: um sistema visual para ensino/aprendizagem de programação via web, 2010. Citado na pág. 4, 14, 18, 27, 51, 52
- KARA (2015)** KARA. Estudo da idc brasil aponta crescimento de 8,7% do mercado de serviços de ti. <http://www.swisseduc.ch/informatik/karatojava/javakara/>, 2015. Último acesso em 10/01/2015. Citado na pág. 21
- Karavirta et al. (2004)** V. Karavirta, A. Korhonen, L. Malmi e K. Stalnacke. Matrixpro - a tool for demonstrating data structures and algorithms ex tempore. Em *Advanced Learning Technologies, 2004. Proceedings. IEEE International Conference on*, páginas 892–893. doi: 10.1109/ICALT.2004.1357707. Citado na pág. 21
- Kelleher et al. (2007)** Caitlin Kelleher, Randy Pausch e Sara Kiesler. Storytelling alicie motivates middle school girls to learn computer programming. Em *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, páginas 1455–1464, New York, NY, USA. ACM. ISBN 978-1-59593-593-9. doi: 10.1145/1240624.1240844. URL <http://doi.acm.org/10.1145/1240624.1240844>. Citado na pág. 25
- Kelly (2010)** James Floyd Kelly. *LEGO MINDSTORMS NXT-G Programming Guide*. Apress, Berkely, CA, USA, 2nd ed. ISBN 1430229764, 9781430229766. Citado na pág. 21
- Kinnunen e Malmi (2006)** Päivi Kinnunen e Lauri Malmi. Why students drop out cs1 course? Em *Proceedings of the second international workshop on Computing education research*, ICER '06, páginas 97–108, New York, NY, USA. ACM. ISBN 1-59593-494-4. Citado na pág. 5
- Koedinger e Aleven (2007)** Kenneth R. Koedinger e Vincent Aleven. Exploring the assistance dilemma in experiments with cognitive tutors. *Educational Psychology Review*, 19(3): 239–264. ISSN 1040-726X. doi: 10.1007/s10648-007-9049-0. URL <http://dx.doi.org/10.1007/s10648-007-9049-0>. Citado na pág. 15
- Kölling (2008a)** Michael Kölling. Greenfoot: A highly graphical ide for learning object-oriented programming. Em *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, páginas 327–327, New York, NY, USA. ACM. ISBN 978-1-60558-078-4. doi: 10.1145/1384271.1384370. URL <http://doi.acm.org/10.1145/1384271.1384370>. Citado na pág. 14
- Kölling (2008b)** Michael Kölling. Greenfoot: A highly graphical ide for learning object-oriented programming. Em *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, páginas 327–327, New York, NY, USA. ACM. ISBN

- 978-1-60558-078-4. doi: 10.1145/1384271.1384370. URL <http://doi.acm.org/10.1145/1384271.1384370>. Citado na pág. 21
- Korsh et al. (1998)** J.F. Korsh, Jr. LaFollette, P.S. e R. Sangwan. Animating programs and students in the laboratory. Em *Frontiers in Education Conference, 1998. FIE '98. 28th Annual*, volume 3, páginas 1139–1144 vol.3. doi: 10.1109/FIE.1998.738589. Citado na pág. 21
- Kuhn (2001)** T.S. Kuhn. *A estrutura das revoluções científicas*. Coleção Debates. Perspectiva. ISBN 9788527301114. Citado na pág. 9
- Lahtinen et al. (2005)** Essi Lahtinen, Kirsti Ala-Mutka e Hannu-Matti Järvinen. A study of the difficulties of novice programmers. Em *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, ITiCSE'05, páginas 14–18, New York, NY, USA. ACM. Citado na pág. 3
- Lee e Ko (2011)** Michael J. Lee e Andrew J. Ko. Personifying programming tool feedback improves novice programmers' learning. Em *Proceedings of the Seventh International Workshop on Computing Education Research*, ICER '11, páginas 109–116, New York, NY, USA. ACM. ISBN 978-1-4503-0829-8. doi: 10.1145/2016911.2016934. URL <http://doi.acm.org/10.1145/2016911.2016934>. Citado na pág. 21
- Lewis (2010)** Colleen M. Lewis. How programming environment shapes perception, learning and goals: Logo vs. scratch. Em *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, páginas 346–350, New York, NY, USA. ACM. ISBN 978-1-4503-0006-3. doi: 10.1145/1734263.1734383. URL <http://doi.acm.org/10.1145/1734263.1734383>. Citado na pág. 25
- LInE (2015)** LInE. Manual de utilização do arcabouço de ima. <http://ccsl.ime.usp.br/redmine/projects/ima/files>, Janeiro 2015. Último acesso em 12/01/2015. Citado na pág. 30
- LOGO (2015)** LOGO. What is logo? <http://el.media.mit.edu/logo-foundation/logo/index.html>, Janeiro 2015. Último acesso em 10/01/2015. Citado na pág. 15
- MacLaurin (2011)** Matthew B. MacLaurin. The design of kodu: A tiny visual programming language for children on the xbox 360. Em *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, páginas 241–246, New York, NY, USA. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926413. URL <http://doi.acm.org/10.1145/1926385.1926413>. Citado na pág. 21
- Manso et al. (2010)** A. Manso, C.G. Marques e P. Dias. Portugol ide v3.x: A new environment to teach and learn computer programming. Em *Education Engineering (EDUCON), 2010 IEEE*, páginas 1007–1010. doi: 10.1109/EDUCON.2010.5492469. Citado na pág. 21
- Marcelino et al. (2008)** M. Marcelino, T. Mihaylov e A. Mendes. H-sicas, a handheld algorithm animation and simulation tool to support initial programming learning. Em *Frontiers in Education Conference, 2008. FIE 2008. 38th Annual*, páginas T4A–7–T4A–12. doi: 10.1109/FIE.2008.4720530. Citado na pág. 21
- Materson e Meyer (2001)** Timothy F. Materson e R. Mark Meyer. Sivil: A true visual programming language for students. Em *Proceedings of the Sixth Annual CCSC Northeastern Conference on The Journal of Computing in Small Colleges*, CCSC '01, páginas 74–86, USA. Consortium for Computing Sciences in Colleges. URL <http://dl.acm.org/citation.cfm?id=378593.378678>. Citado na pág. 21
- Maximilien e Williams (2003)** E.M. Maximilien e L. Williams. Assessing test-driven development at ibm. Em *Software Engineering, 2003. Proceedings. 25th International Conference on*, páginas 564–569. doi: 10.1109/ICSE.2003.1201238. Citado na pág. 31

- Medeiros et al. (2014)** Tainá Medeiros, Pedrina Brasil e Eduardo Aranha. Um framework para criação de jogos voltados para o ensino de lógica de programação. *Simpósio Brasileiro de Informática na Educação - XXV SBIE*. Citado na pág. 20, 21
- Medeiros (2014)** Tainá Jesus Medeiros. Um framework para criação de jogos voltados para o ensino de lógica de programação, 2014. Citado na pág. 25
- Miranda et al. (2007)** L.C. Miranda, F.F. Sampaio e J.A.S. Borges. Programefácil: Ambiente de programação visual para o kit de robótica educacional robofácil. *XVIII Simpósio Brasileiro de Informática na Educação*. Citado na pág. 20, 21
- Moodle (2015)** Moodle. Estatísticas sobre o moodle. <https://moodle.net/stats/?lang=pt-br>, Janeiro 2015. Último acesso em 10/01/2015. Citado na pág. 15
- Msiska e van Zijl (2012)** Mwawi F. Msiska e Lynette van Zijl. From visual scripting to lua. Em *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference, SAICSIT '12*, páginas 94–99, New York, NY, USA. ACM. ISBN 978-1-4503-1308-7. doi: 10.1145/2389836.2389848. URL <http://doi.acm.org/10.1145/2389836.2389848>. Citado na pág. xi, 21, 25, 26
- Myers (1990)** Brad A. Myers. Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput.*, 1(1):97–123. ISSN 1045-926X. doi: 10.1016/S1045-926X(05)80036-9. URL [http://dx.doi.org/10.1016/S1045-926X\(05\)80036-9](http://dx.doi.org/10.1016/S1045-926X(05)80036-9). Citado na pág. 12
- Naps (2005)** T.L. Naps. Jhave: supporting algorithm visualization. *Computer Graphics and Applications, IEEE*, 25(5):49–55. ISSN 0272-1716. doi: 10.1109/MCG.2005.110. Citado na pág. 21
- Nienaltowski et al. (2008)** Marie-Hélène Nienaltowski, Michela Pedroni e Bertrand Meyer. Compiler error messages: what can help novices? Em *Proceedings of the 39th SIGCSE technical symposium on Computer science education, SIGCSE '08*, páginas 168–172, New York, NY, USA. ACM. ISBN 978-1-59593-799-5. doi: 10.1145/1352135.1352192. URL <http://doi.acm.org/10.1145/1352135.1352192>. Citado na pág. 4
- NSTC (2013)** NSTC. Federal science, technology, engineering, and mathematics (stem) education 5-year strategic plan: A report from the committee on stem education national science and technology council. http://www.whitehouse.gov/sites/default/files/microsites/ostp/stem_stratplan_2013.pdf, Maio 2013. Último acesso em 10/01/2015. Citado na pág. 5
- Okamoto et al. (2009)** Y. Okamoto, Dinh Dong Phuong, F. Harada e H. Shimakawa. Programming skill transfer from experienced to novices with animated diagrams. Em *Intelligent Networking and Collaborative Systems, 2009. INCOS '09. International Conference on*, páginas 53–60. doi: 10.1109/INCOS.2009.27. Citado na pág. 21
- Oracle (2015a)** Oracle. Applets. <http://www.oracle.com/technetwork/java/applets-137637.html>, Janeiro 2015a. Último acesso em 10/01/2015. Citado na pág. 14
- Oracle (2015b)** Oracle. What is an interface? <https://docs.oracle.com/javase/tutorial/java/concepts/interface.html>, Janeiro 2015b. Último acesso em 12/01/2015. Citado na pág. 30
- Oracle (2015c)** Oracle. Javadoc tool. <http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html>, Janeiro 2015c. Último acesso em 12/01/2015. Citado na pág. 35
- Pair (1990)** C. Pair. Programming, programming languages and programming methods. Em R. Samurçay J.-M. Hoc, T.R.G. Green e D.J. Gilmore, editors, *Psychology of Programming*, páginas 9–19. Citado na pág. 1

- Paliokas et al. (2011)** Ioannis Paliokas, C. Arapidis e M. Mpimpitsos. Playlogo 3d: A 3d interactive video game for early programming education: Let logo be a game. Em *Games and Virtual Worlds for Serious Applications (VS-GAMES), 2011 Third International Conference on*, páginas 24–31. doi: 10.1109/VS-GAMES.2011.10. Citado na pág. 21
- Pattis (1981)** Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st ed. ISBN 0471089281. Citado na pág. 21
- Pears et al. (2007)** Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin e James Paterson. A survey of literature on the teaching of introductory programming. *SIGCSE Bull.*, 39(4):204–223. ISSN 0097-8418. doi: 10.1145/1345375.1345441. URL <http://doi.acm.org/10.1145/1345375.1345441>. Citado na pág. 9
- Price et al. (1993)** Blaine A. Price, Ronald M. Baecker e Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages & Computing*, 4(3):211 – 266. ISSN 1045-926X. doi: <http://dx.doi.org/10.1006/jvlc.1993.1015>. Citado na pág. xi, 12, 13, 14, 20
- Rajala et al. (2010)** T. Rajala, T. Salakoski, E. Kaila e M. Laakso. How does collaboration affect algorithm learning? a case study using trakla2 algorithm visualization tool. Em *Education Technology and Computer (ICETC), 2010 2nd International Conference on*, volume 3, páginas V3–504–V3–508. doi: 10.1109/ICETC.2010.5529489. Citado na pág. 21
- Repenning e Ioannidou (2004)** Alexander Repenning e Andri Ioannidou. Agent-based end-user development. *Commun. ACM*, 47(9):43–46. ISSN 0001-0782. doi: 10.1145/1015864.1015887. URL <http://doi.acm.org/10.1145/1015864.1015887>. Citado na pág. 21
- Robins et al. (2003)** Anthony Robins, Janet Rountree e Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172. Citado na pág. 2
- Robins et al. (2006)** Anthony Robins, Patricia Haden e Sandy Garner. Problem distributions in a cs1 course. Em *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, páginas 165–173, Darlinghurst, Australia, Australia. Australian Computer Society, Inc. Citado na pág. 3
- Rodrigues (2011)** Patrícia Alves Rodrigues. itarefa: componente moodle para incorporar módulos de aprendizagem interativa em cursos web, 2011. Citado na pág. 15
- Roque (2007)** Ricarose Vallarta Roque. *OpenBlocks: an extendable framework for graphical block programming systems*. Tese de Doutorado, Massachusetts Institute of Technology. Citado na pág. 25
- Sajaniemi e Kuittinen (2003)** Jorma Sajaniemi e Marja Kuittinen. Program animation based on the roles of variables. Em *Proceedings of the 2003 ACM Symposium on Software Visualization, SoftVis '03*, páginas 7–ff, New York, NY, USA. ACM. ISBN 1-58113-642-0. doi: 10.1145/774833.774835. URL <http://doi.acm.org/10.1145/774833.774835>. Citado na pág. 21
- Santos (2009)** Leila Maria Araújo Santos. *A inserção de um agente conversacional animado em um ambiente virtual de aprendizagem a partir da teoria da carga cognitiva*. Tese de Doutorado, Universidade Federal do Rio Grande do Sul, Centro de Estudos Interdisciplinares em Novas Tecnologias da Educação. URL <http://hdl.handle.net/10183/19017>. Citado na pág. 37
- Scott et al. (2008)** A. Scott, M. Watkins e D. McPhee. E-learning for novice programmers; a dynamic visualisation and problem solving tool. Em *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, páginas 1–6. doi: 10.1109/ICTTA.2008.4529966. Citado na pág. 21

- Scratch (2015)** Scratch. Scratch. <https://scratch.mit.edu/>, Janeiro 2015. Último acesso em 12/01/2015. Citado na pág. 22
- Seaton et al. (2014)** Daniel T. Seaton, Yoav Bergner, Isaac Chuang, Piotr Mitros e David E. Pritchard. Who does what in a massive open online course? *Commun. ACM*, 57(4):58–65. ISSN 0001-0782. doi: 10.1145/2500876. URL <http://doi.acm.org/10.1145/2500876>. Citado na pág. 44, 49
- SEP (2015)** SEP. Stanford encyclopedia of philosophy. <http://plato.stanford.edu/entries/computing-history/>, Janeiro 2015. Último acesso em 10/01/2015. Citado na pág. 1
- Simon (2011)** Simon. Assignment and sequence: why some students can't recognise a simple swap. Em *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, páginas 10–15, New York, NY, USA. ACM. ISBN 978-1-4503-1052-9. doi: 10.1145/2094131.2094134. URL <http://doi.acm.org/10.1145/2094131.2094134>. Citado na pág. 5
- Sleeman (1986)** Derek Sleeman. The challenges of teaching computer programming. *Commun. ACM*, 29(9):840–841. ISSN 0001-0782. Citado na pág. 2
- Solomon e Papert (1976)** Cynthia J. Solomon e Seymour Papert. A case study of a young child doing turtle graphics in logo. Em *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, AFIPS '76, páginas 1049–1056, New York, NY, USA. ACM. doi: 10.1145/1499799.1499945. Citado na pág. 26
- Sorva (2012)** Juha Sorva. *Visual program simulation in introductory programming education*. Tese de Doutorado, Aalto University, Department of Computer Science and Engineering. URL <https://aaltodoc.aalto.fi/handle/123456789/3534>. Citado na pág. 13, 20
- Souza et al. (2012)** Clair Teresinha Souza, Caroline da Silva e Rosana Maria Gessinger. Um estudo sobre evasão no ensino superior do brasil nos últimos dez anos. Trabalho de Conclusão de Curso, <https://revistas.utp.ac.pa/index.php/clabes/article/download/868/895>, 1 2012. Citado na pág. 2
- Sprent e Smeeton (2007)** P. Sprent e N.C. Smeeton. *Applied Nonparametric Statistical Methods, Fourth Edition*. Chapman & Hall/CRC Texts in Statistical Science. CRC Press. ISBN 9781439894019. URL <https://books.google.com.br/books?id=xxbSBQAAQBAJ>. Citado na pág. 44
- Swaine (2014)** M. R. Swaine. Encyclopedia britannica. <http://global.britannica.com/EBchecked/topic/183842/ENIAC>, 2014. Último acesso em 10/01/2015. Citado na pág. 1
- Toole e Kim (1999)** B. A. Toole e E. E. Kim. Ada and the first computer. Em *Scientific American*, páginas 76–81. Citado na pág. 1
- Tucker e Noonan (2009)** A. Tucker e R. Noonan. *Linguagens de Programação - 2.ed.: Princípios e Paradigmas*. ISBN 9788563308566. Citado na pág. 9
- TutorialsPoint (2015)** TutorialsPoint. Pascal - goto statement. http://www.tutorialspoint.com/pascal/pascal_goto_statement.htm, 2015. Último acesso em 10/01/2015. Citado na pág. 11
- Utting et al. (2010)** Ian Utting, Stephen Cooper, Michael Kölling, John Maloney e Mitchel Resnick. Alice, greenfoot, and scratch – a discussion. *Trans. Comput. Educ.*, 10(4):17:1–17:11. ISSN 1946-6226. doi: 10.1145/1868358.1868364. URL <http://doi.acm.org/10.1145/1868358.1868364>. Citado na pág. 21
- Vrbanec et al. (2013)** Tedo Vrbanec, Nenad Kirić e Matija Varga. The evolution of web browser architecture. Em *International Virtual Conference on Advanced Scientific Results*, páginas 472–480, New York, NY, USA. ACM. URL https://bib.irb.hr/datoteka/636660.2013_SCIECONF_Vrbanec-Kiric-Varga_The_evolution_of_web_browser_architecture.pdf. Citado na pág. 33

- W3C (2015)** W3C. Frames. <http://www.w3.org/TR/html401/present/frames.html>, Janeiro 2015. Último acesso em 12/01/2015. Citado na pág. 33
- Wexelblat (2014)** R.L. Wexelblat. *History of Programming Languages*. Elsevier Science. ISBN 9781483266169. Citado na pág. 10
- Whale (1994)** G. Whale. Druids: tools for understanding data structures and algorithms. Em *Multi-Media Engineering Education Proceedings, 1994., IEEE First International Conference on*, páginas 403–407. doi: 10.1109/MMEE.1994.383183. Citado na pág. 21