

INTRODUÇÃO AO DESENVOLVIMENTO DE SOFTWARES PARA ANALISTAS DO COMPORTAMENTO



ABPMC

ASSOCIAÇÃO BRASILEIRA DE PSICOLOGIA
E MEDICINA COMPORTAMENTAL

Hernando Borges Neves Filho
Luiz Alexandre Barbosa de Freitas
Nicolau Chaud de Castro Quinta
(orgs.)

ORGANIZAÇÃO

Hernando Borges Neves Filho

Luiz Alexandre Barbosa de Freitas

Nicolau Chaud de Castro Quinta

Introdução ao desenvolvimento de softwares para analistas do comportamento

1ª Edição

Associação Brasileira de Psicologia e Medicina Comportamental, Campinas, São Paulo, Brasil.

ISBN: 978-85-65768-07-8



ABPMC

ASSOCIAÇÃO BRASILEIRA DE PSICOLOGIA
E MEDICINA COMPORTAMENTAL

E-book de distribuição digital gratuita.

2018

Associação Brasileira de Psicologia e Medicina Comportamental – ABPMC

Campinas, São Paulo, Brasil, 2018

Gestão 2017-2018

Presidente: Felipe Lustosa Leite

Vice-presidente: Liane Jorge de Souza Dahás

Primeira secretária: Renata Conceição Silva Pinheiro

Segunda secretária: Luana Flor Tavares Hamilton

Primeiro tesoureiro: Odilon Duarte Neto

Segundo tesoureiro: Bernardo Dutra Rodrigues

Capa: Nicolau Chaud de Castro Quinta

Diagramação, editoração e projeto gráfico: Hernando Borges Neves Filho

Supervisão editorial: Comissão editorial da ABPMC (Gestão 2017-2018)

Presidente da comissão editorial: Ângelo Augusto Silva Sampaio

Secretários da comissão editorial: Luiz Alexandre Barbosa de Freitas,

Hernando Borges Neves Filho e Paulo César Morales Mayer

Agência Brasileira do ISBN

ISBN 978-85-65768-07-8



I61

Introdução ao desenvolvimento de softwares para analistas do comportamento / Organizadores: Hernando Borges Neves Filho, Luiz Alexandre Barbosa de Freitas, Nicolau Chaud de Castro Quinta. -- São Paulo: Associação Brasileira de Psicologia e Medicina Comportamental – ABPMC, 2018.

187 p.

ISBN. 978-85-65768-07-8

1. Psicologia - Linguagem de programação (Computadores). 2. Software - Desenvolvimento. 4. Análise do comportamento. I. Neves Filho, Hernando Borges. II. Freitas, Luiz Alexandre Barbosa de. III. Quinta, Nicolau Chaud de Castro. IV. Associação Brasileira de Psicologia e Medicina Comportamental.

CDD 150

Catálogo elaborado pelo Sistema Integrado de Bibliotecas da UNIVASF.

PREFÁCIO

Este livro tem o objetivo de fornecer ferramentas aos estudantes, profissionais, professores e pesquisadores em Análise do Comportamento. Partimos do princípio de que o leitor não tem conhecimento prévio sobre programação e linguagens de programação. A proposta é começar do básico e mostrar como a programação pode ajudar nos seus estudos, no trabalho e na pesquisa. Não se trata de um livro completo, que vai te preparar para ser um programador profissional. Muito diferente disso. Esse livro foi pensado para te ajudar a dar os primeiros passos e daí seguir para o aprendizado de uma, ou mais, linguagens de programação que serão mais adequadas às suas necessidades. Ainda é comum as pessoas acharem que programação é para programadores profissionais, com formação específica e vinculados à grandes empresas, responsáveis por desenvolver programas de uso cotidiano, como editores de texto, aplicativos de celular e até mesmo no som do carro. Entretanto, pretendemos mostrar a você que este não é o único cenário possível. A programação é uma ferramenta muito poderosa e pode ser utilizada por qualquer um, basta desenvolver as habilidades certas.

H.B.N.F.

L.A.B.F.

N.C.C.Q.

Março de 2018

Autoras e autores

Amanda Calmon Nogueira da Gama Rodegheri - Possui graduação em Psicologia pelo Centro Universitário de Brasília (2011) e mestrado em Psicologia pela Universidade de Brasília (2017). Atualmente está como pesquisadora voluntária e aluna especial no mesmo programa. Tem experiência em Análise Experimental do Comportamento, com ênfase nas áreas de recaída e resistência à mudança.

Carlos Rafael Fernandes Picanço - Possui graduação (2010) em Psicologia e mestrado (2013) em Teoria e Pesquisa do Comportamento pela Universidade Federal do Pará. Tem experiência na área de Psicologia Experimental, atuando principalmente nos seguintes temas: pesquisa com animais não humanos (*Sapajus* spp), processos discriminativos básicos e comportamento verbal. Seus interesses de pesquisa atuais são sobre (a) visão computacional e suas aplicações a processos comportamentais básicos; (b) sistemas educacionais pautados em metodologias ativas e; (c) auto-gerenciamento e programas de ensino individualizado.

Carlos Eduardo Costa - Graduado em Psicologia pela Universidade Estadual de Londrina (1994). Mestrado (1997) e Doutorado (2004) em Psicologia Experimental pela Universidade de São Paulo. Realizou estágio de Pós-Doutorado (2015) junto ao Programa e Mestrado Profissional em Análise do Comportamento Aplicada, no Paradigma: Centro de Ciências e Tecnologia do Comportamento (SP). Professor Associado da Universidade Estadual de Londrina (UEL). Professor e orientador do curso de graduação em Psicologia e do Programa

de Pós-Graduação em Análise do Comportamento na UEL. Editor Associado do periódico Acta Comportamentalia (desde 01/2010) e Perspectivas em Análise do Comportamento (desde 07/2015). Foi coordenador do Programa de Pós-Graduação em Análise do Comportamento na UEL (06/2010 a 05/2013). Membro do Comitê Assessor de Ciências Humanas da Fundação Araucária (2017-2019). Pesquisador CNPq (PQ2, 2017-2019).

Felipe Lustosa Leite - Possui graduação em Psicologia pela Universidade de Fortaleza (2006) e Mestrado (2009) e Doutorado (2014) em Teoria e Pesquisa do Comportamento pela Universidade Federal do Pará. Atuou como professor substituto da Faculdade de Psicologia na Universidade Federal do Pará entre 2013-2014. Realizou estágio de pós-doutorado no Programa de Pós-graduação em Teoria e Pesquisa do Comportamento da Universidade Federal do Pará, trabalhando no Laboratório de Comportamento Social e Seleção Cultural entre Agosto de 2014 e Janeiro de 2015. Foi professor de Análise do Comportamento do Curso de Psicologia da Universidade de Fortaleza, e atualmente é editor-associado da Revista Perspectivas em Análise do Comportamento, co-fundador e Diretor de Pesquisa e Ensino da Imagine Tecnologia Comportamental, e presidente eleito da Associação Brasileira de Psicologia e Medicina Comportamental para a gestão 2017-2018.

Gerônimo Oliveira da Silva Filho - Graduando em Psicologia na Universidade de Fortaleza (UNIFOR). Tem experiência em programação. Trabalhou como acompanhante terapêutico na empresa Imagine Tecnologia Comportamental. Participa do grupo de pesquisa do Laboratório de Psicologia Experimental (Lapex) na Faculdade Ari de Sá.

Hernando Borges Neves Filho - Completou Bacharelado em Psicologia e Formação de Psicólogo pela Universidade Federal do Pará (UFPA). É mestre pelo Programa de Pós-Graduação em Teoria e Pesquisa do Comportamento da Universidade Federal do Pará (UFPA) e doutor pelo Programa de Pós-Graduação em Psicologia Experimental da Universidade de São Paulo (USP). Realizou doutorado-sanduíche na The University of Auckland (Nova Zelândia). Foi bolsista de Pós-doutorado (DOC-FIX/CAPES-FAPEG) na Pontifícia Universidade Católica de Goiás (PUC-GO) e atualmente é bolsista de Pós-doutorado (PNPD/CAPES) no Programa de Pós-Graduação em Teoria e Pesquisa do Comportamento (UFPA). Atualmente é também editor associado da Revista Brasileira de Terapia Comportamental e Cognitiva (RBTCC) e coordenador geral do Grupo de Estudos em Criatividade, Inovação e Comportamento (CRIACOM). Suas principais linhas de pesquisa são: a) criatividade, artes e comportamento novo, b) o papel de variáveis moduladoras da resolução de problemas em uma perspectiva comparativa, c) entendimento de causalidade e pensamento analógico, d) metodologias de ensino e mensuração de criatividade; e) desenvolvimento de jogos e aplicativos de educação e pesquisa; e, f) história da psicologia experimental.

Ítalo Siqueira de Castro Teixeira - Bacharel em Filosofia pela Universidade Estadual do Ceará (2012) e em Psicologia pela Universidade de Fortaleza (2014). Mestre em Ciências do Comportamento pela Universidade de Brasília (2017) e atualmente cursando doutorado no Programa de Pós-Graduação da Universidade de Brasília. Possui experiência na área de Análise

Experimental do Comportamento, com ênfase em pesquisa básica, nas áreas de escolha, recaída e resistência à mudança.

Julia Zanetti Rocca - É professora na Universidade Federal de Mato Grosso - Campus Universitário de Rondonópolis. Fez mestrado em Filosofia e doutorado em Psicologia, ambos na Universidade Federal de São Carlos. Trabalha na área de psicologia educacional, com ênfase em processos de aprendizagem de leitura e escrita. Tem experiência com ensino informatizado no programa "Aprendendo a Ler e Escrever em Pequenos Passos" (ALEPP) do Instituto Nacional de Ciência e Tecnologia sobre Comportamento, Cognição e Ensino (INCT - ECCE) e prestou assessoria a o Laboratório de Inovação em Computação (LINCE - UFSCar) na construção de programas de ensino.

Lucas Franco Carmona - Graduado em Psicologia pela Universidade Estadual de Londrina (2015). Atualmente é aluno de mestrado da Universidade de Brasília (UnB) em Análise Experimental do Comportamento, no programa de Ciências do Comportamento. Tem experiência em variabilidade, numerosidade e resistência à mudança.

Luiz Alexandre Barbosa de Freitas - Graduado em Psicologia pela Universidade Federal de São João del Rei (2006) e Mestre em Análise do Comportamento pela Universidade Estadual de Londrina (2009), atualmente é doutorando em Teoria e Pesquisa do Comportamento pela Universidade Federal do Pará. É professor na Universidade Federal de Mato Grosso desde 2011 onde leciona a disciplinas de Análise Experimental do Comportamento e Métodos de Investigação em Psicologia. Tem experiência na área de Psicologia, com ênfase em Análise do

Comportamento, atuando principalmente nos seguintes temas: Análise do Comportamento, Ensino de Análise do Comportamento e Transtorno do Espectro Autista.

Nicolau Chaud de Castro Quinta - É graduado (2004), mestre (2008) e doutorando em Psicologia pela Pontifícia Universidade Católica de Goiás. Tem experiência em análise comportamental clínica (psicoterapia), aplicada (intervenções focais), e experimental (pesquisa básica), com enfoque nos estudos de escolha com humanos e infra-humanos, desenvolvimento de habilidades básicas sociais, detecção de mentiras através do comportamento, e aplicações de design de jogos à Psicologia e ao ensino. Psicólogo clínico em consultório particular e professor de pós-graduação em Terapia Analítico-Comportamental.

Pedro Gabriel Fonteles Furtado - Graduado em Engenharia de Teleinformática pela Universidade Federal do Ceará (UFC) e Mestre em Engenharia da Informação pela Universidade de Hiroshima (Japão). Possui experiência como desenvolvedor de jogos. Atualmente faz doutorado no laboratório de Engenharia do Aprendizado na Universidade de Hiroshima (Japão).

Rafael Peres Macedo - Graduado em psicologia pela Universidade de Rio Verde-Go (2016). Pós-Graduação (mestrado) em andamento na Universidade de Brasília, possuindo experiência na área de Psicologia, com ênfase em Psicologia Experimental, atuando principalmente nos seguintes temas: recaída, ressurgência, metacontingência e transmissão cultural.

Raphael Moura Cardoso - Psicólogo e Mestre em Psicologia pela Pontifícia Universidade Católica de Goiás. Doutor em Psicologia Experimental pela Universidade de São Paulo. Atualmente é pesquisador vinculado ao Programa de Pós Graduação em Psicologia - PUC-Goiás. Coordena as atividades do Projeto Aplicativo Primeiros Passos, que investiga o uso de tecnologia física e de informação-comunicação (TIC) por crianças; o projeto também envolve o desenvolvimento de um aplicativo com tarefas cognitivas para crianças pré-escolares baseadas em tarefas adequadas para sujeitos não-verbais e utilizadas em pesquisas etológicas e de desenvolvimento infantil. Possui habilidades no método observacional e experimental do comportamento e cognição. Estuda Arduino e Python. O objetivo de sua pesquisa é investigar a ontogênese e a adaptabilidade dos processos cognitivos, preferencialmente em situações rotineiras e ecologicamente relevantes. Seu principal interesse está relacionado aos seguintes temas: Comunicação Não-Verbal e Linguagem; Inteligência e Criatividade; e Bases Cognitivas, Motoras e Perceptuais do Uso de Tecnologia.

Ricardo Fernandes Campos Junior - Possui graduação em Ciências Biológicas pela Universidade Federal de Mato Grosso (2007) e mestrado em Genética pela Universidade de São Paulo (2012). Durante o mestrado trabalhou com Computação Bayesiana Aproximada e análise de processos evolutivos usando Teoria da Coalescência. Tem mais de 5 anos de experiência em programação em R, e recentemente têm trabalhado no desenvolvimento de projetos utilizando Inteligência Artificial.

Apresentação

Ao longo da sua formação acadêmica, qualquer estudante de psicologia chega a conhecer o computador como uma ferramenta útil na criação de documentos, tabelas, figuras, ou slides para uma apresentação. Além de planilhas de cálculo, o uso de uma linguagem de programação é muito menos comum aos alunos, especialmente aos alunos de graduação. Obviamente, a importância de aprender tais linguagens depende do futuro campo de especialização do psicólogo e de seus focos de interesse. Hoje as principais aplicações da programação encontram-se no campo da pesquisa básica, mas profissionais também podem precisar de uma maior familiaridade com conceitos de computação, dado o uso cada vez mais frequente de aplicativos *mobile* especializados no seguimento de tratamentos clínicos ou de tarefas em casa.

No contexto mesmo da pesquisa ou de serviços profissionais, programas de computador são utilizados em pelo menos três momentos. Em primeiro lugar, estes programas servem para aplicar um procedimento experimental, algum tipo de treinamento ou um teste clínico. Na Análise do Comportamento, por exemplo, tentativas de *matching to sample* apresentadas na tela do computador (e portanto produzidas com uma linguagem de programação) são frequentemente usadas para ensinar discriminações entre estímulos. Em segundo lugar, programas de computador auxiliam na coleta de dados - por exemplo, qual palavra o sujeito escolheu em cada tentativa de *matching to sample* e com que latência. Finalmente, uma vez que os resultados foram obtidos, podem ser analisados com programas escritos numa linguagem como R, Matlab ou Python. Em cada caso, e cada vez que seja possível, o uso de uma linguagem de programação permite evitar erros humanos na aplicação dos procedimentos ou na coleta e análise de dados.

Obviamente, o psicólogo que precisa de programas de computador no contexto de sua pesquisa ou de serviços ao cliente pode pedir a ajuda de um programador profissional. Esta solução não requer conhecimentos de computação, mas vai enfrentar vários problemas, além de seu custo. Ainda que tenha uma guia escrita com detalhes sobre os requisitos da tarefa, o programador pode ter dificuldades em entender os objetivos do psicólogo, o que atrasa a escrita do programa. Uma vez o programa escrito e o aplicativo funcionando corretamente, o programador não antecipa necessariamente possibilidades futuras de extensão. Finalmente, o

programador disponível num dado momento costuma mudar de ocupação, interesses, e acessibilidade. Nesse caso, será particularmente difícil modificar o programa após sua escrita inicial, seja para corrigir erros de funcionamento que foram descobertas posteriormente, ou para ampliar o leque das opções oferecidas pelo programa.

Portanto, em muitos casos, escrever seus próprios programas é uma excelente alternativa a pedir a ajuda de um programador profissional. Obviamente, esta opção depende da complexidade do programa - acima de algum limiar de complexidade, saímos do campo de competência do psicólogo ou do analista do comportamento. Contudo, dentro de limites razoáveis, o psicólogo que sabe programar tem à sua disposição uma ferramenta poderosa e extremamente útil.

A pior barreira que enfrenta o estudante interessado em programar é a dificuldade inicial de aprendizagem de uma linguagem computacional. Este livro foi escrito pensando nesta dificuldade. Seu objetivo principal é pedagógico, ou seja, guiar o estudante nos seus primeiros passos como programador(a), clarificando conceitos básicos de computação, dando exemplos de linguagens de programação e mostrando concretamente como podem ser aplicados à pesquisa, ao desenvolvimento de serviços ao cliente e à análise de dados.

A parte inicial do livro agrupa três capítulos. O primeiro, por Hernando Borges Neves Filho, Luiz Alexandre Barbosa de Freitas e Nicolau Chaud de Castro Quinta, serve de introdução geral, situando o uso de computadores no seu contexto histórico e científico. O segundo capítulo, por Nicolau Chaud de Castro Quinta, inicia o leitor à programação com um exercício simples e segue com dicas concretas para aprender a programar, seja com este livro ou com outros recursos. No terceiro capítulo, Luiz Alexandre Barbosa de Freitas ressalta conceitos genéricos (variáveis, decisão, iteração, etc.) que são compartilhados pela maioria das linguagens usadas atualmente. É a presença de tais conceitos que explica porque a aprendizagem de uma segunda linguagem de programação costuma ser mais fácil do que a primeira.

A segunda parte do livro foca em exemplos concretos de linguagens de programação usados na pesquisa com humanos ou animais. O Capítulo 4, por Carlos Rafael Fernandes Picanço, trata do ambiente de programação Lazarus (baseado na linguagem Pascal). A principal vantagem deste ambiente é criar facilmente aplicativos gráficos com imagens, botões, e campos

de respostas a ser preenchidos. Além disso, programas compilados com Lazarus em diferentes plataformas funcionam nos sistemas operacionais Windows, GNU/Linux e Mac OS X. No Capítulo 5, Carlos Eduardo Costa introduz o leitor ao uso do Visual Basic.NET, uma plataforma semelhante ao Lazarus nas suas características principais, mas focada especificamente no sistema Windows. Este capítulo permite entender a facilidade com a qual aplicações podem ser desenvolvidas com este tipo de ferramenta. O Capítulo 6, por Ricardo Fernandes Campos Junior e Julia Zanetti Rocca, muda de tema abordando o uso da linguagem R na análise de dados. Os autores guiam o leitor na aplicação do R a um conjunto de dados simulados, das análises mais simples à gráficas e testes estatísticos mais complexos.

No Capítulo 7, Italo Siqueira de Castro Teixeira, Amanda Calmon-Rodegheri, Lucas Franco Carmona e Rafael Peres Macedo discutem a linguagem procedural MED PC, criada para controlar eventos experimentais em caixas de condicionamento operante e recolher as respostas emitidas pelo animal. Nem todos os estudantes terão a oportunidade de usar esta linguagem (que supõe o acesso a um laboratório animal equipado por Med Associates Inc.), mas o que têm, encontrarão neste capítulo uma guia útil para programar seus primeiros esquemas de reforçamento. No último capítulo desta seção, Gerônimo Oliveira da Silva Filho, Pedro Gabriel Fonteles Furtado, Felipe Lustosa Leite e Hernando Borges Neves Filho explicam o funcionamento da ferramenta Unity, que foca na criação de jogos em dispositivos *mobile* (entre outras plataformas). Os autores ilustram o uso da Unity com um projeto simples e que permite entender como os componentes básicos desta ferramenta interagem.

O livro conclui com um capítulo no qual Raphael Moura Cardoso, Hernando Borges Neves Filho e Luiz Alexandre Barbosa de Freitas esboçam sua visão do futuro da programação no ensino de graduação em psicologia. Ainda não sei se a visão (ou manifesto) dos autores terá o sucesso que esperam, mas já estou seguro de uma coisa: o livro que vocês vão ler é uma excelente porta de entrada ao mundo da programação. Aproveitem!

Sumário

PARTE I

Como e porque programar: primeiros passos e guia de leitura da obra

CAPÍTULO 1.....	1
Por que estudantes, profissionais e pesquisadores de psicologia deveriam aprender programação?	

Hernando Borges Neves Filho
Luiz Alexandre Barbosa de Freitas
Nicolau Chaud de Castro Quinta

CAPÍTULO 2.....	13
Como ler este livro – e se tornar um programador	

Nicolau Chaud de Castro Quinta

CAPÍTULO 3.....	18
A lógica de programação	

Luiz Alexandre Barbosa de Freitas

PARTE II

Introdução à algumas linguagens de programação e seus usos no
laboratório de Psicologia

CAPÍTULO 4.....	33
Introdução ao desenvolvimento de interfaces gráficas com Lazarus e Free Pascal	

Carlos Rafael Fernandes Picanço

CAPÍTULO 5.....	83
Visual Basic.NET	

Carlos Eduardo Costa

CAPÍTULO 6.....	101
Introdução à linguagem de programação R aplicada à pesquisa e intervenção comportamental	

Ricardo Fernandes Campos Junior
Julia Zanetti Rocca

CAPÍTULO 7.....	124
Introdução ao MED PC	

Ítalo Siqueira de Castro Teixeira
Amanda Calmon-Rodegheri
Lucas Franco Carmona
Rafael Peres Macedo

CAPÍTULO 8.....	138
Unity: Criando jogos e outras aplicações multi-plataforma	

Gerônimo Oliveira da Silva Filho
Pedro Gabriel Fonteles Furtado
Felipe Lustosa Leite
Hernando Borges Neves Filho

PARTE III

Conclusão e próximos passos

CAPÍTULO 9.....	156
Ensino e pesquisa no século XXI: Um manifesto pelo ensino de programação na graduação em Psicologia	

Raphael Moura Cardoso
Hernando Borges Neves Filho
Luiz Alexandre Barbosa de Freitas



Por que estudantes, profissionais e pesquisadores de psicologia deveriam aprender programação?

Hernando Borges Neves Filho |  |  |

Luiz Alexandre Barbosa de Freitas |  |  |

Nicolau Chaud de Castro Quinta |  |

O desenvolvimento das ciências, especialmente as de cunho empírico, pode sempre ser historicamente atrelado ao desenvolvimento de tecnologias, em especial tecnologias de observação, registro e mensuração (e.g. Burke, Bergman & Asimov, 1985). Do domínio do fogo, que permitiu o nascimento da química, ao entendimento dos princípios da aerodinâmica e o surgimento da aviação, até os computadores de mesa e as inteligências artificiais, ciência, e seu método de investigação sistemático, caminham lado a lado com o advento de equipamentos que auxiliam na implementação deste método (e. g. Almqvist, 2003; Anderson Jr, 1998; Boden, 2016). Assim, na história cumulativa do conhecimento, todo novo advento, todo novo implemento, literalmente abre novas portas para novas descobertas e maneiras de interagir com o mundo. Dito de outra forma, toda tecnologia possibilita a ocorrência de novos comportamentos (Manrique, 2011).

Na Psicologia, situada como uma ciência interessada no comportamento humano e não-humano, existem diversos exemplos de tecnologias que moldaram o desenvolvimento da área. De fato, metáforas do funcionamento da mente humana baseada em tecnologias de diferentes épocas são bastante comuns (Gentner & Grudin, 1985), desde teorias de

funcionamento mental baseado em metáforas hidráulicas (Lorenz, 1995), até a famosa metáfora do computador, que iguala a mente humana a um mecanismo de processamento de informação, similar a um computador (Gardner, 2003). Entretanto, a história da Psicologia é também atrelada a adventos tecnológicos que vão muito além de metáforas. Tomemos como exemplo o desenvolvimento de uma parcela específica da Psicologia: a Análise do Comportamento.

A Análise do Comportamento é uma especialidade da Psicologia que trata o comportamento dos organismos, humanos e não-humanos, partindo de pressupostos funcionalistas e selecionistas (Carvalho Neto, 2002). Sua origem histórica se dá com o trabalho de B. F. Skinner, Psicólogo norte americano que capitaneou essa forma de olhar para o comportamento (para um apanhado de como Skinner, foi moldando sua obra ao longo de sua carreira, conferir Laurenti, 2012). Na gênese e no desenvolvimento da Análise do Comportamento, pode-se encontrar um pequeno aparato eletromecânico, a câmara de condicionamento operante, que foi responsável por permitir o registro automatizado de uma das medidas privilegiadas da área, a frequência de respostas (Sakagami & Lattal, 2016).

A câmara de condicionamento operante, foi, em suas primeiras versões, uma engenhoca eletromecânica desenhada e construída por Skinner que tinha como objetivo ser um ambiente simplificado, que permitisse a um animal, o sujeito experimental, emitir um comportamento discreto que pudesse ser facilmente observado e quantificado (Sakagami & Lattal, 2016). Com esse desenho básico, foram criadas câmaras que permitiam ratos pressionar barras, e pombos bicar discos. Esses comportamentos eram registrados com o abrir e fechar de registros eletromecânicos, que produziam curvas acumuladas de frequência de resposta, que podiam ser analisadas em termos de taxas de respostas (número de respostas em uma unidade de tempo), a medida privilegiada por Skinner. Até antes de Skinner, diversas outras medidas comportamentais eram então priorizadas, como a latência e a duração de respostas (medidas fidedignamente por relógios e cronômetros), e foi com a câmara de condicionamento operante, e seu mecanismo de registro automatizado do número de respostas, que surgiu todo um corpo de conhecimento científico que tratou de identificar como variáveis de controle antecedente e consequente controlam a frequência de determinadas respostas. A câmara de condicionamento operante, foi,

por assim dizer, o “microscópio” que permitiu uma análise minuciosa de uma propriedade do comportamento até então difícil de ser observada e mensurada com precisão.

A câmara de condicionamento operante foi uma de diversas inovações atreladas ao estudo do comportamento a partir de medidas como a frequência de respostas. Ainda na Análise do Comportamento, e ainda na obra de Skinner, outro exemplo são as máquinas de ensinar, que nada mais eram do que máquinas eletromecânicas que apresentavam um conteúdo didático de forma programada, e consequenciavam de forma imediata as respostas de alunos e alunas frente a uma pergunta ou questionamento deste conteúdo (Skinner, 1972). Hoje em dia, câmaras de condicionamento operante dos mais variados fabricantes e modelos podem ser encontradas em laboratórios didáticos de cursos de graduação em Psicologia, e máquinas de ensinar, como as desenhadas e implementadas por Skinner, são consideradas precursores de equipamentos de ensino programado à distância (Ferster, 2014). Um ponto em comum no desenvolvimento e uso atual desses dois equipamentos é o advento do computador. Hoje, câmaras de condicionamento operante, em suas mais variadas funções, são controladas por computador, utilizando diferentes interfaces e programas. Da mesma maneira, o ensino à distância é, em geral, realizado via um computador com acesso à internet. A invenção do computador, como uma ferramenta tecnológica, trouxe novas maneiras de lidar com diversos outros equipamentos e artefatos tecnológicos, que passaram a gradualmente ser informatizados e automatizados (i.e. a ser controlado por um computador).

Um computador, como uma máquina de processamento de informação capaz de controlar diversos equipamentos, é hoje em dia um dos itens mais básicos de qualquer local de trabalho, incluindo aí laboratórios de pesquisa. Antes, tais computadores eram artefatos que apenas empresas e laboratórios de ponta poderiam adquirir e usar e, em geral, os primeiros modelos de computador ocupavam salas inteiras, e consumiam grandes quantidades de energia elétrica². Hoje em dia, temos computadores nas palmas de nossas mãos, controlando nossas agendas, contatos e nos guiando com mapas atualizados em tempo real com informações de tráfego e transporte público. Na prática de Psicólogos, aplicativos que auxiliam as mais diversas tarefas terapêuticas tem se tornado cada vez mais comuns, e seus usos e vantagens são, e devem ser, rotineiramente debatidos na área (uma lista de aplicativos terapeuticos atuais, utilizados em

diferentes contextos e reconhecidos pela *American Psychological Association*, pode ser encontrada em http://www.zurinstitute.com/mentalhealthapps_resources.html).

Na pesquisa de Psicologia e Neurociências em geral, todo o registro de dados automatizado hoje envolve o uso de um computador (ou dispositivo móvel de função similar, como *tablets* ou *smarthphones*). Em um cenário como esse, no qual o computador é uma ferramenta de trabalho, intervenção e pesquisa tão presente, entender e saber “dizer” ao computador o que ele deve fazer é uma habilidade cada vez mais básica vinculada à empregabilidade, como um dia foi a datilografia (e. g. Frey & Osborne, 2016). Entender e saber “dizer” a um computador o que fazer é entender, dominar, sua programação.

Com este cenário em vista, neste capítulo introdutório da obra será apresentado um breve histórico da programação, acompanhado do que é uma linguagem de programação e o que pode ser feito com ela (de exemplos triviais a exemplos de tecnologia de ponta). Ao final do capítulo, será retomado o argumento de por que Psicólogos, mais especificamente analistas do comportamento, em seus mais variados níveis de formação e atividade, tem muito a ganhar ao aprender a programar. Concluído isto, o capítulo encerra com uma sugestão de primeiros passos para aprender a programar.

O que é uma linguagem de programação?

A programação é, como seu nome já implica, desígnios de rotina. Uma máquina segue uma programação, um código pré-definido. Na computação, a programação é o que faz computadores desempenharem suas funções. A computação surgiu no século XIX com Charles Babbage e Ada Lovelace (Hollings, Martin & Rice, 2017). Babbage conceptualizou e testou os primeiros modelos de computadores mecânicos da história, e Lovelace foi a responsável por criar e programar o primeiro algoritmo a rodar em uma máquina de Babbage. Um algoritmo é um conjunto de regras bem definidos que levam à solução de um problema, etapa por etapa (i.e. uma programação). Tal empenho, na época, tinha como objetivo solucionar erros de cálculo que os instrumentos da época para estimativas de tempo (e.g. tabelas astronômicas, cartas de navegação, quadros de maré) apresentavam (Kingsley-Hughes, 2005). Desde então, outros aparelhos para

calcular foram criados, sempre para solucionar um problema ou para realizar tarefas que seriam difíceis (ou demoradas) demais para fazer manualmente.

As primeiras máquinas, que seguiram e aprimoraram as máquinas de Babbage, eram construídas para tarefas específicas e, portanto, tinham funções limitadas. Foi somente em 1946 que o cientista alemão Konrad Zuse criou a primeira linguagem de programação, a Plankalkül (Kingsley-Hughes, 2005). Depois disso outras linguagens foram surgindo pouco a pouco até chegarmos no momento atual, em que há uma diversidade de linguagens para atender a necessidades e plataformas diferentes. Uma linguagem de programação é um conjunto de sintaxes e comandos padronizados que geram instruções a um computador. Cada linguagem possui um conjunto de sintaxes e uma gramática própria (apesar de haver algumas sobreposições), existindo portanto linguagens mais econômicas e linguagens mais expansivas (Fourment & Gillings, 2008).

Qualquer dispositivo eletrônico, um computador, um robô ou um *smartphone*, executa suas funções através de programas (hoje também comumente chamados de aplicativos). Um programa é uma série de instruções, de algoritmos, que especificam as tarefas que o dispositivo deve desempenhar, ou os problemas que deve resolver. Colocado de forma simples, programar é dizer ao computador o que ele deve fazer. Assim, em vários sentidos, aprender a programar é como aprender uma nova língua, com a diferença de que máquinas são muito mais sensíveis e inflexíveis a pequenas variações no texto. No entanto, enquanto os idiomas possibilitam a comunicação entre indivíduos, as linguagens de programação possibilitam a comunicação entre um indivíduo e uma máquina. Assim, a programação é, em si mesma, o comportamento do programador. Ele programa, ou seja, escreve códigos em um linguagem que pode ser entendida pelo computador, e o computador executa as tarefas que foram pedidas. Essas tarefas podem variar muito, como veremos a seguir.

O que pode ser feito a partir da programação?

Ainda que programas de computador possam ser utilizados de forma complementar a praticamente qualquer atividade humana, a automação de certas tarefas é de especial interesse aos psicólogos e analistas do comportamento por permitir um controle e manipulação de variáveis bastante seguro. Atualmente boa parte das pesquisas em Análise Experimental do Comportamento utiliza *softwares* para coleta de dados, substituindo procedimentos tradicionais de escolha e consequenciação que tipicamente utilizam papel, cartolina e fichas coloridas por programas de computador.

Um programa pode ajudar a padronizar atividades e evitar erros que são gerados quando a atividade é executada por uma pessoa. Pense, por exemplo, que você pretende ensinar uma criança a combinar palavras com figuras que as representam. Sabemos que muitos analistas do comportamento têm trabalhado nesta área para desenvolver tecnologia comportamental que torne o ensino de leitura mais eficaz. Uma pessoa que vá realizar as sessões de ensino pode acabar se atrapalhando na hora de apresentar as palavras e figuras da tarefa, pois precisará randomizar as posições dos estímulos para evitar equívocos (e.g. escolha meramente baseada na posição dos estímulos). Um programa pode ser elaborado para apresentar os estímulos (palavras e figuras) na posição e na ordem corretas. Além disso, ele poderá fazer vários registros, como: erros, acertos, tempo da sessão e latência de respostas. Estes registros teriam que ser feitos manualmente caso não contássemos com um programa para nos ajudar e novamente estaríamos sujeitos a erros “humanos”. É importante destacar que programas também podem errar, pois não passam de instruções feitas por outros humanos. No entanto, um programa bem construído e testado, fará exatamente aquilo para quê foi elaborado.

Um programa de computador pode ajudar também a automatizar uma tarefa repetitiva. Por exemplo, você precisa gerar gráficos a partir de um conjunto grande de dados, cada conjunto em uma tabela diferente do excel. É possível automatizar isso escrevendo um programa que identifique os dados nas suas planilhas e construa o gráfico da maneira que você escolher. Da

mesma forma, programas também podem ser utilizados para analisar dados, principalmente de experimentos que utilizam vídeos como dados brutos. Um exemplo deste tipo de programa é o [BORIS](#) (*Behavioral Observation Research Interactive Software*), que registra e categoriza comportamentos previamente definidos em vídeos de performances de animais. O programa analisa um vídeo e registra todas as ocorrências, duração e outras medidas dos comportamentos-focos que ocorrem (e são reconhecidos pelo programa) em um arquivo de vídeo. Programas similares são também comumente utilizados em pesquisas que utilizam equipamento de rastreamento do olhar, como o [PyGaze](#).

Para além dos laboratórios de Psicologia, áreas interdisciplinares como as Neurociências têm sido a vanguarda do desenvolvimento tecnológico em pesquisa. As Neurociências são altamente dependentes de tecnologia de ponta, sempre em busca de maneiras mais refinadas de medir, registrar e acompanhar em tempo real respostas fisiológicas do sistema nervoso. Nas últimas décadas, diversos equipamentos e técnicas de mapeamento cerebral têm surgido e ganhado destaque, sempre contando com o suporte de poderosos computadores e instrumentos de imagem e registro informatizado. Uma tecnologia desta linha que tem grande potencial, são as chamadas interfaces cérebro-computador (*Brain-computer interface*, BCI). Em linhas gerais, uma interface cérebro-máquina é um programa de computador que registra, identifica e converte sinais eletrofisiológicos gerados pelo cérebro e converte estes sinais em ações e comportamentos de uma máquina (Wolpaw et al., 2000). Tecnologias deste tipo permitem literalmente controlar braços mecânicos e cursores em uma tela de computador apenas com o pensamento, ou mais especificamente com um ativamento cerebral específico. O que torna o ativamento cerebral em ação externa é a interface, é o programa (e estes são em grande medida os grandes avanços tecnológicos nesta área, desenvolver interfaces e programas cada vez mais refinados).

Saindo do laboratório e deste contexto de pesquisa, programas e aplicativos tem também ganhado destaque como ferramentas de ensino e auxílio terapêutico. No ensino, diversos programas tem o objetivo de gerenciar e facilitar a organização de material didático. Um exemplo deste tipo de programa é o [GEIC](#) (Gerenciador de Ensino Individualizado por Computador), que é um sistema capaz de gerar programas de ensino individualizados, para ser

aplicado em crianças, e que inclusive pode ser utilizado à distância (via *web*). No que tange o desenvolvimento e uso de aplicativos para fins não necessariamente acadêmicos, um grande mercado tem se formado em torno de aplicativos que prometem “aprimorar habilidades cognitivas”, ou “treinar a inteligência de crianças”. Em grande parte, estes programas e aplicativos carecem de um teste empírico que ateste sua eficácia, o que, entretanto, não impede que estes sejam comercializados e alcancem sucesso com o público (Murfin, 2013). No futuro, um profissional que possivelmente teria os pré-requisitos para avaliar e certificar programas e aplicativos com promessas que envolvem “aprimoramentos cognitivos” deste tipo, é o psicólogo. Em questões menos polêmicas, como mencionado anteriormente, existem ainda diversos aplicativos e programas desenvolvidos em universidades e por grupos de *experts* em alguns assuntos terapêuticos que auxiliam Psicólogos e outros profissionais da saúde em tarefas como automonitoramento e terapia ABA (Brady, 2011). Estes são alguns exemplos de uma área recente e que está em pleno desenvolvimento: os aplicativos de auxílio terapêutico. Psicólogos e analistas do comportamento que queiram contribuir com este desenvolvimento dependem de um pré-requisito básico: entender o mínimo de programação.

Por que aprender a programar?

A resposta é simples: para participar dos desenvolvimentos científicos e profissionais do século XXI. Aprender a programar de forma alguma vai substituir ou suplantará o conhecimento tradicional da grade curricular da Psicologia. Entretanto, aprender a programar é uma habilidade complementar que capacita o psicólogo a ter maior autonomia em suas pesquisas, desenvolvendo programas feitos sob medida para seu problema de pesquisa, assim como também capacita o psicólogo a uma inserção no mercado que hoje é permeada por aplicativos e tarefas informatizadas. Aprender a programar permite ao pesquisador liberdade e precisão na forma de conduzir sua pesquisa, e ao profissional oferece uma oportunidade de transformar seu serviço prestado em um produto, ou de automatizar seu serviço.

Por onde começar, ou, qual o primeiro passo?

Basta uma breve reflexão para se perceber que muito do que está ao nosso redor é “código”, é efeito de uma programação executada por um computador. Se você está lendo este livro, lançado (até o momento) somente em versão digital, você está usando um computador, que está rodando um programa que lê este tipo de arquivo. Da mesma maneira, os autores utilizaram diferentes processadores de textos (programas) para escrever este livro. Seu alarme, programado para lhe despertar amanhã no seu telefone é também gerenciado por um aplicativo, assim como as mensagens que você recebe o dia inteiro de familiares e amigos. Tudo isso foi planejado e programado por alguém para executar estas funções (e diferentes programas executam funções com maior ou menor precisão). A pergunta diante disso é: como essas pessoas, esses programadores e programadoras, conseguiram programar isso? A resposta, o leitor ou leitora já sabe bem, foi aprendendo a programar.

Aprender a programar requer o tempo e a dedicação que qualquer outra atividade requer. Da mesma maneira, é possível desenvolver essa habilidade em diferentes níveis, desde um nível de *hobbysta* até profissional qualificado. Na Psicologia brasileira, até o momento, ainda não há uma ênfase em sua base curricular voltada ao ensino da programação, nem na graduação e nem na pós-graduação, apesar da clara relevância deste tópico na formação do psicólogo. Assim, o primeiro passo é seguir o caminho autodidata. Por sorte, hoje há muito material disponível gratuitamente na *web* que se propõe a instigar, ensinar e desenvolver repertórios de programação em diferentes níveis. Textos, livros, tutoriais e videos de ensino de programação podem facilmente ser encontrados. Entretanto, tal riqueza é também desnorteante, na medida em que isto dificulta saber por onde começar. Assim, os capítulos seguintes pretendem ser um guia para o iniciante interessado na programação, um guia voltado para psicólogos, mais especificamente analistas do comportamento. Um guia que irá discutir os pressupostos básicos do que é programação, sua lógica, assim como os caminhos iniciais para se escolher uma linguagem e aprendê-la. Algumas linguagem de programação serão apresentadas em suas funções e pressupostos básicos, por profissionais e pesquisadores que tanto conhecem a linguagem, como

também trabalham ou têm conhecimento de tópicos de análise do comportamento. Neste sentido, cada um dos demais capítulos deste guia é um primeiro passo.

Referências

- Almqvist E. (2003) From Aristotle to the birth of modern chemistry. In: *History of Industrial Gases*. Springer, Boston, MA. doi: [10.1007/978-1-4615-0197-8_2](https://doi.org/10.1007/978-1-4615-0197-8_2)
- Anderson Jr, J. D. (1998). *A history of aerodynamics*. London: Cambridge University Press.
- Burke, J., Bergman, J. & Asimov, I. (1985). *The impact of science on society*. Washington: National Aeronautics and Space Administration (NASA) University Press of the Pacific. Recuperado de <https://history.nasa.gov/sp482.pdf>
- Boden, M. A. (2016). *AI: Its nature and future*. New York: Oxford University Press.
- Brady, L. J. (2011). *Apps for autism: A must-have resource of the special needs community*. Arlington: Future Horizons.
- Carvalho Neto, M. B. (2002). Análise do comportamento: behaviorismo radical, análise experimental do comportamento e análise aplicada do comportamento. *Interação em Psicologia*, 6, 13-18. Recuperado de http://www.cemp.com.br/arquivos/25932_65.pdf
- Ferster, B. (2014). *Teaching machines: Learning from the intersection of education and technology*. Baltimore: John Hopkins University Press.
- Fourment, M. & Gillings, M. R. (2008). A comparison of common programming languages used in bioinformatics. *BMC Bioinformatics*, 9, 82. doi: [10.1186/1471-2105-9-82](https://doi.org/10.1186/1471-2105-9-82)

- Frey, C. B. & Osborne, M. A. (2016). The future of employment: How susceptible are jobs to computerisations? *Technological Forecasting & Social Change*. doi: [10.1016/j.techfore.2016.08.019](https://doi.org/10.1016/j.techfore.2016.08.019)
- Gardner, H. (2003). *A nova ciência da mente*. São Paulo: EDUSP.
- Gentner, D. & Grudin, J. (1985). The evolution of mental metaphors in Psychology: A 90-year retrospective. *American Psychologist*, 40, 2. doi: [10.1037/0003-066X.40.2.181](https://doi.org/10.1037/0003-066X.40.2.181)
- Hollings, C., Martin, U. & Rice, A. (2017). The early mathematical education of Ada Lovelace. *Journal of the British Society for the History of Mathematics*, 32, 221-234. doi: [10.1080/17498430.2017.1325297](https://doi.org/10.1080/17498430.2017.1325297)
- Kingsley-Hughes, A., & Kingsley-Hughes, K. (2005). *Beginning programming*. New York: John Wiley & Sons.
- Laurenti, C. (2012). O lugar da análise do comportamento no debate científico contemporâneo. *Psicologia: Teoria e Pesquisa*, 28, 367-376. doi: [10.1590/S0102-37722012000300012](https://doi.org/10.1590/S0102-37722012000300012)
- Lorenz, K. (1995). *Os fundamentos da etologia*. São Paulo: Editora da UNESP.
- Manrique, T. P. (2011). Los instrumentos de laboratorio como medio de contacto de la operante. *Revista Iberoamericana de Psicología: Ciencia y Tecnología*, 4, 41-45. Recuperado de <http://revistas.iberoamericana.edu.co/index.php/ripsicologia/article/view/209>
- Murfin, M. (2013). Know your apps: An evidence-based approach to evaluation of mobile clinical applications. *The Journal of Physician Assistant Education*, 24, 38-40. doi: [10.1097/01367895-201324030-00008](https://doi.org/10.1097/01367895-201324030-00008)
- Sakagami, T. & Lattal, K. (2016). The other shoe: An early operant conditioning chamber for pigeons. *The Behavior Analyst*, 39, 25-39. doi: [10.1007/s40614-016-0055-8](https://doi.org/10.1007/s40614-016-0055-8)
- Skinner, B. F. (1982). *Tecnologia do ensino*. São Paulo: Herder.

Wolpaw, J. R. et al. (2000). Brain-computer interface technology: A review of the first international meeting. *IEEE Transactions of Rehabilitation Engineering*, 8, 164-173. doi: [10.1109/TRE.2000.847807](https://doi.org/10.1109/TRE.2000.847807)

2

Como ler este livro - e se tornar um programador

Nicolau Chaud de Castro Quinta |  |

Se você leu o capítulo anterior, a esta altura já deve estar convencido dos vários benefícios que um repertório de programação pode lhe trazer enquanto analista do comportamento e em outras áreas de sua vida. Duas perguntas que podem estar passando pela sua cabeça neste momento são “será que eu consigo?” e “por onde começar?”.

Para responder a primeira pergunta, vamos fazer um pequeno exercício. Vá para <https://www.codechef.com/ide>, uma plataforma online de programação que permite criar e executar códigos simples em diferentes linguagens. A lista de linguagens mostrará como padrão C++ 6.3 (Gcc 6.3). Por questão de simplicidade, mude para PYTH (Cpython 2.7.13). Logo abaixo você verá duas linhas de código, a segunda delas vazia, conforme a Figura 1.

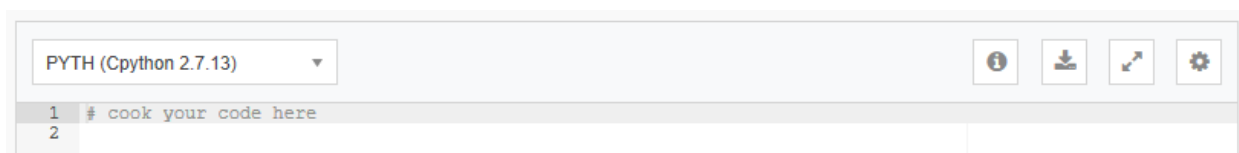


Figura 1. PYTH (Cpython 2.7.13) no website codechef.

Escreva o seguinte código nas linhas seguintes:

```
behavioristas = 30
psicanalistas = 25
humanistas = 20
psicologos = behavioristas + psicanalistas + humanistas
print(psicologos)
```

Abaixo da tela de códigos, você verá um botão *Run*. Clique nele. Após alguma espera (esse site trabalha com uma espécie de “fila” de submissões), você verá um espaço de *Output*

com o número 75. Pronto, você acabou de criar um pequeno programa que calcula o número total de psicólogos a partir da soma de psicólogos de diferentes abordagens!

Talvez você esteja pensando “não valeu, eu não criei o código, meramente copiei”. Assim como na aquisição de qualquer outro tipo de repertório, os estágios iniciais da aprendizagem em programação passam por muitos momentos de modelação. À medida que códigos do mesmo tipo vão sendo utilizados repetidas vezes, seu uso passa a dispensar qualquer tipo de cola ou consulta.

Ainda que seu primeiro código pareça bastante simples e intuitivo, ele tem especificidades importantes. Por exemplo, se você retirar o “s” de “humanistas” da penúltima linha, o programa não irá funcionar, e você receberá uma mensagem de erro. Ainda que uma pessoa consiga entender que “humanistas” e “humanista” se referem à mesma coisa, o computador não tem o mesmo entendimento, e é sensível a qualquer variação ou imprecisão em códigos.

Um atributo bastante importante para o programador é a curiosidade. Ainda que muito do que você vá fazer no início seja simplesmente seguir instruções e copiar modelos, a curiosidade em querer entender o “sentido” dessas linhas de código, explorar as possibilidades de variação, e conhecer as outras ferramentas de uma linguagem é o que efetivamente te fará aprender a programar. Se você é uma pessoa curiosa, poderá olhar para seu primeiro código e se perguntar “o que vai acontecer se eu mudar os valores de cada categoria?”, ou “será que a operação aceitaria uma quarta categoria de cognitivistas?”, ou “o cálculo funciona se eu trocar o operador matemático + por algum outro?”. A melhor forma de responder tais perguntas é testando. São testes desse tipo que te farão, em algum momento, deixar de ser um programador “ctrl+c ctrl+v” e te transformarão em um programador autônomo.

Antes de conhecer a fundo qualquer linguagem de programação, é fundamental compreender como funciona a lógica de programação. Por esse motivo, a leitura do capítulo 3 é indispensável. A lógica de programação é universal a todas as linguagens, e nesse capítulo você terá seu primeiro contato com os fundamentos da criação de códigos, além de compreender melhor conceitos importantes como “linguagem”, “lógica de programação” e “algoritmo”.

Os capítulos seguintes são específicos a diferentes linguagens de programação. Em vários aspectos, qualquer linguagem pode atingir qualquer objetivo, mudando apenas a forma como se

chega até lá. Diferentes linguagens atingem certos objetivos com mais facilidades que outras, o que pode influenciar na escolha pela linguagem de programação a ser adotada. Ainda que muitos programadores conheçam mais de uma linguagem, a grande maioria domina e se especializa em apenas uma. Um conhecimento aprofundado em diferentes linguagens de programação é não só custoso como desnecessário. Por esse motivo, antes de se aprofundar no seu conhecimento sobre programação, é necessário eleger uma linguagem na qual irá se especializar.

Caso você tenha disposição para isso, poderá ler todos os capítulos seguintes e conhecer um pouco sobre as diferentes linguagens e plataformas contidas neste livro: VB.NET, R, Unity, Lazarus/Free Pascal e Med PC. Você notará que muitas noções e códigos se repetem entre as linguagens, e isso será útil para fortalecer conceitos e diretrizes importantes no seu aprendizado. Ler todos os capítulos é também uma forma de tomar decisões mais seguras sobre qual linguagem escolher. Se, no entanto, você quer economizar tempo e começar diretamente pelo estudo de uma linguagem com maior probabilidade de fazer sentido para você, siga as orientações a seguir.

Para familiarizar o leitor com um elemento bastante corriqueiro na lógica de programação – as condicionais – criamos um algoritmo para lhe ajudar a escolher uma linguagem de programação. Esse algoritmo usará os termos **if** (se), **then** (então) e **else** (ou então), comumente usados em códigos de programação. Os itens numerados a seguir não devem ser lidos em sequência; comece pelo item 1 e prossiga para o item especificado.

- 1 > **if** você quer criar programas para automatizar procedimentos de coleta e registro de dados em câmaras de condicionamento para pesquisas em comportamento operante e respondente **then** vá para 11, **else** vá para 2.
- 2 > **if** você quer criar programas voltados para o registro, organização e análise de dados, capaz de trabalhar com um grande volume de dados e trabalhar de forma eficiente com análises matemáticas e representação visual de dados **then** vá para 10, **else** vá para 3.
- 3 > **if** para você é importante que seus aplicativos tenham suporte para mobile, ou seja, que rodem em tablets ou telefones celular **then** vá para 12, **else** vá para 4 *{caso precise apenas de programas que funcionem em computadores de mesa}*.

- 4 > **if** deseja criar jogos ou plataformas de coleta de dados com ênfase gráfica como programas de realidade virtual **then** vá para 12, **else** vá para 5.
- 5 > **if** quer trabalhar com uma linguagem open source, gratuita, com alto grau de compatibilidade entre diferentes plataformas (Windows, Linux, Mac) **then** vá para 9, **else** vá para 6 *{para conhecer uma alternativa de linguagem}*.
- 6 > **If** quer uma linguagem com maior compatibilidade a outros aplicativos da Microsoft (como o Excel) **then** vá para 8, **else** vá para 7.
- 7 > Aparentemente, nenhuma das linguagens deste livro parece atender suas necessidades. Você está seguro do que está procurando? Recomendamos a leitura integral do livro, pois conhecer com maior profundidade cada uma das linguagens apresentadas pode fornecer uma ideia melhor do que pode ser feito com cada uma delas. Há outras linguagens de programação disponíveis no mercado; continue lendo o capítulo e falaremos mais sobre elas.
- 8 > A linguagem mais indicada para atender suas necessidades é VB.NET. Conheça mais sobre ela no capítulo 5 deste livro.
- 9 > A linguagem mais indicada para atender suas necessidades é Lazarus/Free Pascal. Conheça mais sobre ela no capítulo 4 deste livro.
- 10 > A linguagem mais indicada para atender suas necessidades é R. Conheça mais sobre ela no capítulo 6 deste livro.
- 11 > A linguagem mais indicada para atender suas necessidades é Med-PC. Conheça mais sobre ela no capítulo 7 deste livro.
- 12 > A linguagem mais indicada para atender suas necessidades é Unity. Conheça mais sobre ela no capítulo 8 deste livro.

É importante mencionar que há diversas outras linguagens de programação no mercado não exploradas neste livro, como JAVA, Ruby, Python, C e PHP. Uma busca na internet permitirá que conheça mais sobre cada uma delas. Se você quer uma orientação sobre qual linguagem escolher através de um teste curto, rápido e interativo, visite <http://www.bestprogramminglanguagefor.me/>.

O objetivo deste livro não é fornecer subsídios suficientes para tornar o leitor um programador completamente equipado e preparado para criar programas e aplicativos. A leitura dos capítulos seguintes te guiará pelos seus primeiros passos na programação, ajudando você a compreender o que é a programação e se familiarizar com uma ou mais linguagens. Cada capítulo descreve um panorama geral da linguagem, as instruções de instalação, os elementos mais importantes do código, e contém um pequeno tutorial para criação de procedimentos importantes para analistas do comportamento. Ao final, são indicados links para leitura e aprofundamento. Aprender uma linguagem é um processo sem fim, e exige constante dedicação e aprimoramento. A boa notícia é que a internet dispõe de contingências muito favoráveis para esse caminho: para qualquer linguagem você encontrará facilmente apostilas, tutoriais, e comunidades de programadores dispostas a trocar informações e ajudar iniciantes.



A lógica de programação

Luiz Alexandre Barbosa de Freitas |  |  |

Toda a programação segue uma lógica, assim como outras tarefas do cotidiano, que podem ser decompostas em pequenas partes sequenciais. Podemos dizer que a lógica é o “coração” de um programa, ela é a organização do código a partir da qual o programa funciona. Não devemos nos enganar com a aparência de um programa, a parte gráfica, ou seja, aquela que está aparente para o usuário é apenas uma camada entre quem usa o programa e o que ele realmente faz. Por exemplo, ao clicar em um botão para colocar um trecho de texto em negrito, uma série de códigos que você não vê são ativados para produzir as alterações que você na sua tela. Este capítulo trata de alguns aspectos básicos de lógica que um programa deve ter para que faça tudo aquilo que você precisa.

O que é a lógica de um programa?

Agora um pouco mais detalhadamente, a lógica é a organização sequencial dos códigos de um programa para que um objetivo seja atingido. Este objetivo pode ser, por exemplo, a apresentação de estímulos visuais na tela do computador em tempos e durações específicas, o próprio controle de tempo do programa e várias outras coisas. A lógica é composta por códigos organizados em instruções, são estas instruções que dizem ao computador que ação deve ser realizada. Tomemos um exemplo do cotidiano que também requer uma ordenação lógica para que um objetivo seja alcançado. Se você pretende construir um avião de papel será preciso seguir uma sequência específica de passos para ter o produto final esperado. Você deverá dobrar o papel ao meio, no sentido vertical, reabrir a folha, dobrar as pontas de um dos lados até atingir a marca

feita pela primeira dobra e assim por diante. Se seguir todos os passos corretamente, seu avião ficará pronto e terá a aerodinâmica necessária para fazer um voo. Do contrário o objetivo não será alcançado, mas você pode desfazer o avião e verificar onde estava o erro que impediu seu avião de voar. Na programação isto também é possível (revisar a sua lógica em busca de erros e consertá-los), é chamado de *debugging* ou debugar, como é falado em português. Vamos agora analisar a lógica simples de um programa para somar dois números:

Objetivo: somar dois números

Passos:

- 1) inserir primeiro número
- 2) inserir segundo número
- 3) somar primeiro número com o segundo número
- 4) exibir o resultado da soma

Perceba que a instrução em passos precisa ter uma sequência específica para que o objetivo seja alcançado. Se alterarmos a ordem colocando o passo 4 antes do 3, por exemplo, nosso programa não funcionará corretamente, pois não haverá resultado para ser exibido sem que a soma tenha sido feita antes.

Avançando um pouco mais, um termo que tem se tornado mais comum nos últimos anos é o **algoritmo**. “Algoritmo na Ciência da Computação (Informática), está associada a um conjunto de regras e operações bem definidas e ordenadas, destinadas à solução de um problema, de uma classe de problemas, em um número finito de passos” (Manzano & Oliveira, 2011). Há algoritmos muito simples e outros muito complexos, dependendo da natureza do problema ou da classe de problemas que se quer resolver. Uma receita de pão caseiro pode ser considerada um algoritmo do cotidiano:

Objetivo: preparar um pão caseiro

Ingredientes

1 ovo

5 colheres de açúcar

1 colher de sopa de sal

400ml de água morna

20 g de fermento seco

9 colheres de sopa de óleo

1kg de farinha de trigo

Modo de preparo:

- 1 - colocar o fermento na água morna e deixar por 5 minutos
- 2 - acrescentar o açúcar, o sal, o ovo, o óleo
- 3 - acrescentar farinha de trigo aos poucos até a massa soltar das mãos
- 4 - sovar a massa até ficar macia
- 5 - deixar a massa descansar por 30 a 40 minutos
- 6 - dividir a massa em 3 partes, abrir com um rolo e enrolar como um rocambole
- 7 - colocar cada parte em uma forma untada com farinha de trigo
- 8 - colocar para assar em forno sem pré-aquecer por 50 minutos

Se alguém seguir este algoritmo ao final terá um pão caseiro, mas embora pareça simples, este algoritmo tem alguns problemas. Por exemplo, se considerarmos que a ação do fermento depende da temperatura ambiente e da umidade, a quantidade a ser utilizada deve variar para produzir o mesmo efeito, ou seja, o pão com as mesmas características planejadas por quem criou a receita. Então, analisando cuidadosamente, este algoritmo deveria conter uma condição antes de definir a quantidade de fermento. Se a temperatura ambiente estiver entre A e B e também se a umidade relativa do ar estiver entre X% e Y%, então a quantidade de fermento deverá ser W. Perceba que precisamos avaliar duas condições (temperatura ambiente e umidade do ar) para estabelecer a quantidade de um ingrediente. Um bom padeiro sabe disso, mesmo sem recorrer a um algoritmo formal.

Vamos a outro exemplo, as operações matemáticas. Quando queremos solucionar uma equação devemos seguir uma série de regras e elas podem ser consideradas algoritmos. Veja a ordem correta para executar as operações matemáticas:

- 1) Calcule primeiro os números que estão entre parênteses;
- 2) Calcule os números que contém expoentes;
- 3) Calcule as multiplicações e divisões seguindo da esquerda para a direita;
- 4) Por fim calcule as somas e subtrações também da esquerda para a direita.

Caso alguém não respeite esta ordem para executar as operações o resultado estará incorreto. Com os algoritmos nos computadores temos algo semelhante, a ordem em que as etapas do processo está organizada não estiver correto duas coisas podem ocorrer, seu programa não funcionará, ou funcionará de forma incorreta (produzindo, na nossa analogia com o cotidiano, um resultado errado ou um pão muito seco).

As variáveis, as constantes e seus dados

As linguagens de programação têm alguns elementos básicos com os quais trabalham. As variáveis são um conjunto destes elementos básicos. As variáveis são a maneira utilizada pelos programas de armazenar dados sobre o problema que será resolvido. O tipo de dado que será necessário para solucionar o problema pode variar e por isso há vários tipos diferentes que atendem a determinadas funções. Pode haver variações específicas sobre como cada linguagem considera uma variável, por isso trataremos aqui do que é mais comum entre todas as linguagens. De forma geral, as variáveis podem ser entendidas como representações dos valores que representam. O valor de uma variável pode ser alterado em diferentes momentos, sem que ela precise mudar de nome.

Algumas linguagens exigem que uma variável seja declarada, ou criada, antes de ser utilizada. É uma forma de dizer ao computador para reservar uma área da memória para ser usada pelo programa.

Na linguagem Pascal, por exemplo, é preciso utilizar a palavra-chave *var* para indicar que as variáveis serão criadas. Veja a seguir:

```

1> var
2> nome_da_variável : tipo_da_variável
3> idade : integer
4> nome : string

```

Na linha 1 temos a palavra-chave que indica que haverá declaração de variáveis; na linha 2 é apresentada a notação básica de como declarar uma variável; na linha 3 há a declaração de uma variável do tipo integer (números inteiros) e na linha 4 há a declaração de uma variável do tipo string (caracteres alfanuméricos).

Na linguagem Python a declaração de variáveis não precisa ocorrer explicitamente como no Pascal, basta atribuir um valor e a variável já está criada. Veja a seguir:

```

1> nome_da_variável = valor_da_variável
2> idade = 45
3> nome = marcos

```

Veja, na linha 1 está a notação básica de como declarar uma variável já contendo um valor e nas linhas 2 e 3 temos duas variáveis com valores diferentes.

Trataremos aqui de quatro tipos básicos de variáveis, apesar de haver outros. São elas de **alfanuméricos**, de **números inteiros**, de **números reais** e **lógicas**.

A variável de **alfanuméricos** (letras e números) pode conter um ou mais caracteres e armazena dados que são entendidos pelo computador como texto, mesmo podendo conter números. Por exemplo:

```

nome = marcos
idade = 45
cidade = manaus

```

Em cada uma das linhas há o nome da variável antes do sinal de igual (=) em azul e o valor atribuído à variável após o sinal de igual em verde. Isso significa que dentro da variável

`nome` o valor é `marcos`, da variável `idade` o valor é `45` e da `cidade` o valor é `manaus`. Estes valores ficarão guardados dentro das variáveis e podemos trabalhar com eles quando quisermos. Neste exemplo a variável `idade` é do tipo alfanumérico e seu valor não pode ser usado para operações matemáticas em razão do tipo de variável que está armazenando o dado.

Quando se quer trabalhar com números, as variáveis precisam numéricas. As variáveis numéricas mais simples são as de **números inteiros**, por exemplo 5, 38 e 1024. Veja a seguir as variáveis numéricas que armazenam as idades de cada pessoa.

```
laura = 25
maria = 18
carlos = 59
felipe = 39
```

No exemplo acima os nomes das pessoas são os nomes das variáveis e o valores numéricos em cada um podem ser utilizados em operações matemáticas. Suponha que nós queremos saber qual o valor da soma das idades das quatro pessoas. Então teríamos:

```
soma = laura + maria + carlos + felipe
soma = 141
```

Veja que para efetuar a soma nós criamos uma nova variável numérica chamada `soma`. Os nomes das variáveis pouco importam, desde que saibamos identificar posteriormente qual ela é e que sigamos as normas da linguagem que está sendo utilizada (as linguagens têm regras próprias para isto). Algumas linguagens inclusive permitem converter um tipo de variável em outro tipo, por exemplo, pode-se converter uma variável do tipo alfanumérico para o tipo números inteiros se os caracteres da primeira forem apenas números.

As variáveis de **números reais** são parecidas com as dos números inteiros, com o detalhe de que permitem trabalhar com valores decimais (por exemplo, 5.25; 14.9; 35.27). Como as linguagens geralmente são construídas tendo o inglês como idioma padrão, a notação numérica envolvendo casas decimais costuma utilizar um ponto ao invés de uma vírgula.

Por fim, há as **variáveis lógicas** que, por sua vez, armazenam dados lógicos como Verdadeiro e Falso. Veja o exemplo a seguir:

```
1> cadastrado = Verdadeiro
2> maioridade = Falso
3> habilitado = Falso
```

Na linha 1 o valor da variável cadastrado indica que é Verdadeiro, então, esse valor poderá ser útil para saber, por exemplo, se um usuário que tenta acessar o sistema já está cadastrado para isso. Na linha 2 o valor da variável maioridade indica que o valor é Falso, então, no nosso exemplo, o usuário ainda não atingiu a maioridade. Na linha 3 o valor Falso indica que o usuário não é habilitado. Todos esses valores poderiam ser úteis em um programa que deveria definir o tipo de produto pode ser oferecido a um usuário cadastrado.

Um outro tipo de elemento básico nos programas é a **constante**. As constantes são importantes pois, diferente das variáveis, seu valor não pode ser alterado no decorrer da execução de um programa. Por diversos motivos podemos querer que um valor seja mantido inalterado e para isso utilizamos uma constante. Podemos, por exemplo, querer manter constante o número de minutos de duração das sessões de um experimento e variar outros parâmetros como o intervalo do esquema de reforço e a quantidade de reforçador que será disponibilizado.

Operadores

Os operadores são símbolos ou expressões utilizadas para trabalhar com os elementos da programação, sejam eles constantes ou variáveis. Estes operadores podem ser de três tipos, aritméticos, relacionais e lógicos.

Os operadores **aritméticos** geralmente são úteis para obter resultados numéricos, embora haja outras possibilidades. Os mais básicos são o de adição (+), subtração (-), multiplicação (*), divisão (/) e exponenciação (** ou ^, dependendo da linguagem).

Veja o exemplo abaixo:

```

1> grupo_a = 13
2> grupo_b = 25
3> grupo_c = 8
4> total = grupo_a + grupo_b + grupo_c
5> media = total/3

```

Neste exemplo nós somamos os valores correspondentes a cada grupo na linha 4 utilizando o operador de soma (+) e na linha 5 o total foi dividido pelo número de grupos (3), neste caso uma constante, para se obter o valor médio. O sinal de igual (=) nessas linhas de código tem a função de atribuir um valor à variável que se encontra à esquerda do sinal.

Os operadores **relacionais** têm a função de comparar elementos (variáveis ou constantes). Os operadores mais comuns são maior que (>), maior ou igual a (>=), menor que (<), menor ou igual a (<=), igual a (= ou ==, a depender da linguagem) e diferente de (<> ou !=, a depender da linguagem). O resultado de se comparar dois elementos é um valor lógico, que pode ser VERDADEIRO ou FALSO. Veja:

```

1> grupo_a = 13
2> grupo_b = 25
3> grupo_c = 8
4> grupo_a > grupo_b
5> FALSO
6> grupo_b >= grupo_c
7> VERDADEIRO
8> grupo_c != grupo_a
9> VERDADEIRO
10> grupo_a == grupo_b
11> FALSO

```

As linhas 4, 6, 8 e 10 são como perguntas que estamos fazendo, nas quais queremos saber quais relações são verdadeiras e quais são falsas em relação às variáveis, e nas linhas 5, 7 e 11 temos as respectivas respostas.

Os operadores **lógicos** têm a função de combinar duas ou mais expressões para serem analisadas quanto à sua validade. Ou seja, novamente o resultado será um valor lógico de VERDADEIRO ou FALSO. Os operadores lógicos básicos são E (AND), OU (OR) e NÃO (NOT). Veja os exemplos a seguir:

```
1> grupo_a = 13
2> grupo_b = 25
3> grupo_c = 8
4> grupo_a > grupo_b AND grupo_b > grupo_c
5> FALSO
6> grupo_b >= grupo_c OR grupo_a == grupo_b
7> VERDADEIRO
```

Na linha 4, para que o resultado seja verdadeiro as duas expressões precisam ser verdadeiras, ou seja, grupo_a precisa ser maior do que grupo_b (o que é falso) E grupo_b precisa ser maior do que grupo_c (o que é verdadeiro). Caso as duas fossem falsas, o resultado também seria FALSO. Na linha 5, o operador lógico OU permite que o resultado seja verdadeiro se apenas uma das duas expressões for verdadeira. Assim, é verdadeiro que grupo_b é maior ou igual a grupo_c, mas é falso que grupo_a é igual a grupo_b. Considerando que o operador foi OU, então a condição verdadeira da primeira expressão já permite considerar o resultado como VERDADEIRO. Caso o operador usado tivesse sido E, o resultado seria FALSO.

Sobre o uso do operador NÃO, veja a seguir:

```
1> grupo_a = 13
2> grupo_b = 25
3> grupo_c = 8
4> NOT grupo_a > grupo_b
5> VERDADEIRO
6> NOT grupo_b >= grupo_c
7> FALSO
```

Na linha 4, a expressão em si é falsa, o valor de grupo_a (13) não é maior do que o valor de grupo_b (25), mas o uso do operador NOT fez com o que o valor lógico fosse invertido, retornando o valor VERDADEIRO. Algo semelhante ocorreu na linha 6, quando o NOT fez com que o resultado da expressão verdadeira grupo_b >= grupo_c fosse FALSO. Portanto, o operador NOT inverte os valores lógicos das expressões que acompanha.

Operações Lógicas

Muitas vezes em um programa será necessário estabelecer condições para seguir adiante. No exemplo do algoritmo para preparar um pão caseiro analisamos que seria indicado analisar as condições de temperatura e umidade do ar antes de decidir a quantidade de fermento a ser incluída na massa. Decisões como essas são comuns na lógica de qualquer programa e para isso utilizamos as operações lógicas ou estruturas de seleção. Assim, trabalhamos com condições do tipo SE(if) e ENTÃO(then). A forma como estas operações são escritas e os termos utilizados podem ser diferentes entre as linguagens, mas a lógica é a mesma. Para entender melhor, veja o exemplo a seguir.

Se a quantidade de acertos no teste foi abaixo de 90% a criança deve repetir o bloco de treino. Se a quantidade de acertos no teste foi igual ou maior que 90% a criança passa à fase seguinte. Em termos de lógica de programação poderíamos expressar estas condições assim:

```
1> tentativas = 10
2> acertos = 8
3> percentual_acertos = acertos / tentativas
4> criterio = 90 / 100
5>
6> if percentual_acertos < criterio then
7>     repete_bloco_de_treino
8>
9> if percentual_acertos >= criterio then
10>     inicia_fase_seguinte
```

Vamos analisar estas linhas de código. Na linha 1 foi definido quantas tentativas serão consideradas no teste. Na linha 2 o valor de “acertos” é definido com base no desempenho da criança no teste. Como será necessário saber o percentual de acertos, a variável na linha 3 é o cálculo disto. O valor da variável `percentual_acertos` vai depender de quantos “acertos” foram feitos e do número de tentativas. Utilizando os valores nas linhas 1 e 2, este valor será 0,8. É importante salientar que se alterarmos o número de tentativas ou o número de acertos, este percentual também poderá ser alterado. Na linha 4 o critério que será comparado com o percentual de acertos é calculado. O critério tem um valor constante (0.9) que é resultado da conta 90 dividido por 100. Na linhas 6 e 9 são definidas as condições. O programa deve analisar primeiro se `percentual_acertos` é menor que `critério`. Se, e somente se, isso for VERDADEIRO então (then) ele irá executar o comando `repete_bloco_de_treino` na linha 7. Caso não seja VERDADEIRO ele pulará a linha 9 e irá analisar a segunda condição. Se `percentual_acertos` do maior ou igual a `critério`, então (then) o programa executa `inicia_fase_seguinte`. As linhas 5 e 8, em branco, não executam nada e só foram incluídas para facilitar a leitura.

Outra maneira de funcionamento das operações lógicas é por meio da estrutura SE (if), ENTÃO (then) e SENÃO(else). Neste caso o SENÃO é acrescentado para o caso de nenhuma das condições (SE) anteriores ser atendida.

Vamos utilizar como exemplo os conceitos que um aluno pode obter em uma disciplina.

- Conceito A: nota entre 9 e 10
- Conceito B: nota entre 8 e 8,9
- Conceito C: nota entre 7 e 7,9
- Conceito D: nota entre 6 e 6,9
- Conceito E: nota abaixo de 6

Um algoritmo para representar esta classificação pode ser assim:

```

1> if nota >= 9 then
2>     conceito = "A"
3>
4> if nota >= 8 AND nota < 9 then
5>     conceito = "B"
6>
7> if nota >= 7 AND nota < 8 then
8>     conceito = "C"
9>
10> if nota >= 6 AND nota < 7 then
11>     conceito = "D"
12>
13> else
14>     conceito = "E"

```

Na lógica do exemplo acima, as linhas 1, 4, 7 e 10 determinam condições específicas e, caso nenhuma delas seja atendida, a linha 13 oferece uma alternativa. Na linha 1 apenas um expressão precisa ser verdadeira, a nota precisa ser maior ou igual a 9. Nas linhas 4, 7 e 10 é necessário que duas expressões sejam verdadeiras para atender ao critério. Na linha 13 nenhuma expressão é incluída após o operador “else” pois ele significa “caso nenhuma das condições anteriores seja atendida, então faça isso”.

Repetições ou iterações

Uma das coisas que torna os algoritmos tão úteis é a possibilidade de que realizem tarefas repetitivas muitas e muitas vezes, geralmente em um curto espaço de tempo. Por isso vamos concluir este capítulo introdutório sobre lógica de programação apresentando alguns comandos que permitem estas repetições. O comando ENQUANTO (while) - FAÇA(do) executa um bloco de outros comandos enquanto uma determinada condição for verdadeira. Veja no exemplo a seguir.

Eu quero criar um bloco de código que faça um contagem regressiva de um por um e mostre esta contagem na tela antes de passar para a parte seguinte do programa. A lógica pode ser a seguinte:

```
1> numero_atual = 10
2>
3> while numero_atual > 0 do
4>     imprima (numero_atual)
5>     numero_atual = numero_atual - 1
6>
```

Na linha 1 foi estabelecido o valor inicial para a contagem (10). Na linha 3 fica estabelecido que enquanto a condição `numero_atual` for maior do que 0 (zero) o bloco a seguir, com recuo de alguns espaços será executado, lemos da seguinte forma “enquanto `numero_atual` for maior do que zero faça”. Na linha 4 pedimos que o programa imprima na tela o valor de `numero_atual`, da primeira vez que isso ocorrer ele será dez. Na linha 5 atribuímos um novo valor à variável `numero_atual` que é igual a ela mesma menos 1. O trecho “`numero_atual - 1`” significa que o programa deverá pegar o valor de `numero_atual`, qualquer que seja ele, e subtrair uma unidade. Dito de outra maneira, a linha 5 atualiza o valor de `numero_atual` para uma unidade a menos, ou seja, aí é que está a contagem regressiva. Após fazer isto o programa retorna novamente para a linha 3 e confere se a condição permanece verdadeira, ou seja, se `numero_atual` continuar maior do que 0. Se for verdadeiro ele executa novamente as linhas 4 e 5. Isto se repete até que a condição não seja mais verdadeira, ou seja, que `numero_atual` seja igual a zero. O que o usuário do programa vê na tela é algo parecido com isto:

```

10
9
8
7
6
5
4
3
2
1

```

Outra maneira de criar repetições é utilizando os comandos FAÇA ATÉ (do until). Neste caso, os outros comandos contidos no bloco de código serão executados enquanto a condição for FALSA, quando ela se tornar verdadeira a repetição termina e o programa vai para as linhas seguinte. Podemos usar o mesmo exemplo da contagem anterior, veja como ficaria.

```

1> numero_atual = 10
2>
3> do until numero_atual = 0
4>     imprima (numero_atual)
5>     numero_atual = numero_atual - 1
6>

```

Cada vez que o bloco for executado (linhas 3 a 5) será subtraída uma unidade de numero_atua e o bloco vai ser repetido até que numero_atual seja igual a zero. Quando a condição se tornar VERDADEIRA o programa continuará da linha 6 em diante. O efeito para o usuário que observa o resultado na tela é o mesmo do exemplo com o comando ENQUANTO - FAÇA.

Chegamos ao final deste breve capítulo sobre lógica de programação. Obviamente há muitas outras possibilidades de comandos que podem ser empregados na lógica da programação

e, como afirmamos anteriormente, cada linguagem tem sua própria forma de fazer isso. Além disso, é comum combinar estes comandos para produzir resultados mais complexos.

O objetivo foi introduzir o iniciante nesta maneira estruturada de organizar uma tarefa em pequenos passos sequencialmente posicionados. Uma pesquisa rápida na internet com o termo lógica de programação retornará diversos materiais sobre o tema, incluindo livros, apostilas, vídeos e até cursos gratuitos em plataformas de ensino à distância. Vá em frente e estude mais a respeito, este conteúdo será útil para qualquer linguagem de programação com a qual você escolher trabalhar.

Referências

Manzano, J. A. N. G. & Oliveira, J. F. (2011). *Algoritmos. Lógica para Desenvolvimento de Programação de Computadores*. São Paulo: Ed. Érica.



Introdução ao desenvolvimento de interfaces gráficas com Lazarus e Free Pascal

Carlos Rafael Fernandes Picanço |  |  |

O objetivo deste capítulo é de:

- Informar o leitor ou leitora sobre a existência de diferentes dialetos derivados do Pascal.
- Situar o leitor ou leitora sobre o dialeto utilizado neste guia.
- Apresentar um recorte da comunidade de desenvolvedores Pascal.
- Apresentar um recorte básico do dialeto Free Pascal.
- Introduzir aspectos básicos do ambiente de programação Lazarus e Free Pascal.
- Introduzir o desenvolvimento de aplicações visuais nesse ambiente por meio de exemplos.

Os exemplos foram pensados para uma audiência de analistas do comportamento, especialmente aqueles lidando com participantes de pesquisa que devem interagir com uma interface gráfica. Por meio desses exemplos, o leitor ou leitora será guiado à resolução de problemas recorrentes:

- Como apresentar estímulos (antecedentes e consequentes)?
- Como esperar por respostas?
- Como rastrear e registrar tempo e frequência de estímulos e respostas?

Pré-requisitos

- Inglês: leitura e escrita instrumental.
- Conhecimento básico de informática: usar teclado e mouse.
- Conhecimento básico sobre o sistema operacional de escolha: como executar um programa?
- Conhecimento básico sobre interfaces gráficas comuns, por exemplo, busca por controles visuais: o que são janelas, o que é um menu superior, etc

Pascal - Breve histórico

A linguagem de programação Pascal, como originalmente arquitetada pelo professor Niklaus Wirth entre 1968 e 1971 (Jensen & Wirth, 1973) tinha como objetivo servir ao ensino introdutório de programação estruturada em suas aulas. Assim inicia a entrevista de Severance (2012) a Wirth. Embora Wirth também tenha ajudado a montar o sistema em grandes computadores de outras universidades, ele relata que a popularização de dialetos originários do Pascal só viria na década de 80 com o advento do microcomputador, de sistemas integrados de desenvolvimento e da redução de custo dos compiladores.

Quando comparado com linguagens como BASIC, Assembly, ALGOL e FORTRAN, o Pascal de Wirth possuía um melhor balanço entre boa legibilidade, modularidade e flexibilidade. Severance (2012), considera que a linguagem, por ser estruturada, era muito mais adequada para a construção de programas com qualidade de produção. Ainda assim, dando continuidade à entrevista, Wirth foi reconhecendo demandas não contempladas pelo Pascal, o que o levou a estendê-lo e reformulá-lo, criando outros dialetos derivados.

Mas aquela popularização ocorreria por meio de ainda outros dialetos, independentes, com suas próprias extensões e melhorias. Esses outros dialetos, em alguns casos, inclusive sendo adotados como dialeto padrão em cursos de introdução à programação. Como consequência, algumas gerações naquele período (70-80) aprenderam a pensar computacionalmente por meio

de um dialeto derivado ou original. É razoável considerar, portanto, que algumas gerações entraram no mercado de trabalho tendo um ou outro como primeira linguagem de programação.

Pascal - Padronização e Diversificação comercial

Não por acaso, com tal mão de obra disponível, bases de código milionárias foram escritas em dialetos do Pascal; por exemplo, como os primeiros sistemas operacionais da Apple Computers Inc. (1985). Por conta do crescente uso comercial de dialetos derivados do Pascal, o dialeto original foi padronizado (ISO 7185:1983), revisado (ISO 7185:1990) e expandido (ISO 10206:1990) com o objetivo de corrigir ambiguidades e assegurar a sua portabilidade. Atualmente, compiladores como o GNU Pascal (versão 3.4.x) e o Free Pascal (versão 3.x.x) oferecem, em algum nível, suporte a essas padronizações.

Uma linguagem sem ambiguidades e portátil permite que um programa, uma vez escrito, seja traduzido para a linguagem da máquina alvo independente do compilador utilizado. A despeito dos esforços, e embora sejam desejáveis de um ponto de vista técnico e prático, tais padronizações não figuram entre os dialetos mais populares derivados do Pascal. Outros dialetos (como o Delphi Pascal, Apple Pascal e Free Pascal) tornaram-se os “padrões de fato” no mercado.

Object Pascal - Um dialeto estendido

Ao longo das décadas de 70 e 80 houve uma popularização da chamada “programação orientada a objetos”. Surgia a necessidade de se estender a sintaxe do Pascal estruturado tornando-o mais permissivo ao novo estilo; originava-se então o Object Pascal. A orientação ao objeto tornava-se um modelo de referência para o planejamento e a programação de interfaces gráficas. Foi nesse contexto que interfaces de desenvolvimento integrado como o Turbo Pascal (Borland International Inc, 1984/1983 e sucessores, como o Delphi) e o Lazarus (lançado em 2001, ver [https://en.wikipedia.org/wiki/Lazarus_\(IDE\)](https://en.wikipedia.org/wiki/Lazarus_(IDE))) surgiram.

Free Pascal e Lazarus - um ambiente de desenvolvimento integrado

Os exemplos neste guia foram escritos por meio do ambiente integrado de desenvolvimento Lazarus (versão 1.8RC4; Lazarus IDE, 2017) e do compilador Free Pascal (Versão 3.0.4; Klämpfl et al, 2017). O ambiente contém recursos que reduzem a barreira de entrada na complexa cadeia que é o desenvolvimento de aplicações compiladas. Um compilador é um programa que traduz a sintaxe de alto nível (mais portátil), para uma linguagem de baixo nível (específica de uma máquina alvo). O produto final é um arquivo nativamente executável e que não demanda instalação.

Compilador (Free Pascal) e interface (Lazarus) são distribuídos por meio de licenças livres (GPL). Parte do ambiente também é licenciado (LGPL) de maneira a permitir a distribuição de aplicações comerciais com código fonte privado. O ambiente está disponível para sistemas operacionais como o OSX, Windows e baseados no kernel Linux (Debian, Ubuntu) e agrega uma grande comunidade de desenvolvedores independentes. Os principais meios de informação e comunicação nesse ecossistema são:

- A wiki: <http://wiki.freepascal.org/>
- O fórum: <http://forum.lazarus.freepascal.org/>
- As listas de emails:
 - Lazarus
 - Free Pascal
- Os sites oficiais:
 - Pacotes: <http://packages.lazarus-ide.org/>
 - Fundação: <https://foundation.freepascal.org/>
 - Lazarus: <http://lazarus-ide.org/>
 - Free Pascal: <https://www.freepascal.org/>
- O rastreador de bugs: <http://bugs.freepascal.org/>

Diversas coleções de unidades, componentes e pacotes reunidos nas chamadas

“bibliotecas” já vem pré-instalados. Eles permitem a execução de tarefas gerais de programação:

- Free Pascal Runtime Library (RTL).
- Free Pascal Component Library (FCL).
- Lazarus Component Library (LCL).

Outras coleções de terceiros também frequentemente utilizadas estão reunidas por meio do pacote “Gerenciador Online de Pacotes”. O pacote é distribuído juntamente com o Lazarus e pode ser instalado por meio do menu “Pacotes”, opção “Instalar pacotes”, item “OnlinePackageManager” na lista à direita.

Ao explorar o ecossistema por meio de buscadores online, seja em busca de ajuda, seja em busca de contribuições de terceiros, utilize palavras-chave como “free pascal guide”, “lazarus forum”, “component”, “package”, “componente”, “pacote” juntamente com os termos específicos de seu interesse.

Free Pascal e Lazarus - Instalação e configuração do ambiente de desenvolvimento.

Baixe os arquivos de instalação correspondentes para o seu sistema no sítio de hospedagem oficial (<https://sourceforge.net/projects/lazarus/files/>):

- GNU/Linux 64 bits
- GNU/Linux 32 bits
- Mac OS X 32bits
- Windows 64 bits
- Windows 32 bits

Em seguida execute o instalador (ou instaladores, se Linux e OSX). Caso seja solicitado, forneça os privilégios de administrador do sistema operacional ao instalador. O processo de instalação e configuração mínima é automático.

Este guia fará referência aos nomes dos controles da interface Lazarus tal como traduzidos para o português brasileiro, portanto recomenda-se a escolha deste idioma ao longo da

instalação. Opcionalmente, após a instalação, altere o idioma no menu superior Ferramentas->Opções da IDE->Ambiente->Geral->Idioma.

Lazarus - criando e executando uma aplicação

Ao executar o Lazarus pela primeira vez (por meio do comando “startlazarus” em sistemas Linux), uma aplicação (programa com uma interface gráfica) é criada automaticamente. Execute a aplicação pressionando F9 (Executar). Essa aplicação padrão é uma janela (um formulário) flutuante com funcionalidades básicas como fechar, minimizar, maximizar, restaurar, mover, redimensionar, entre outras. Essa janela também já vem preparada para receber eventos como aqueles produzidos por mouse e teclado (Figura 1).

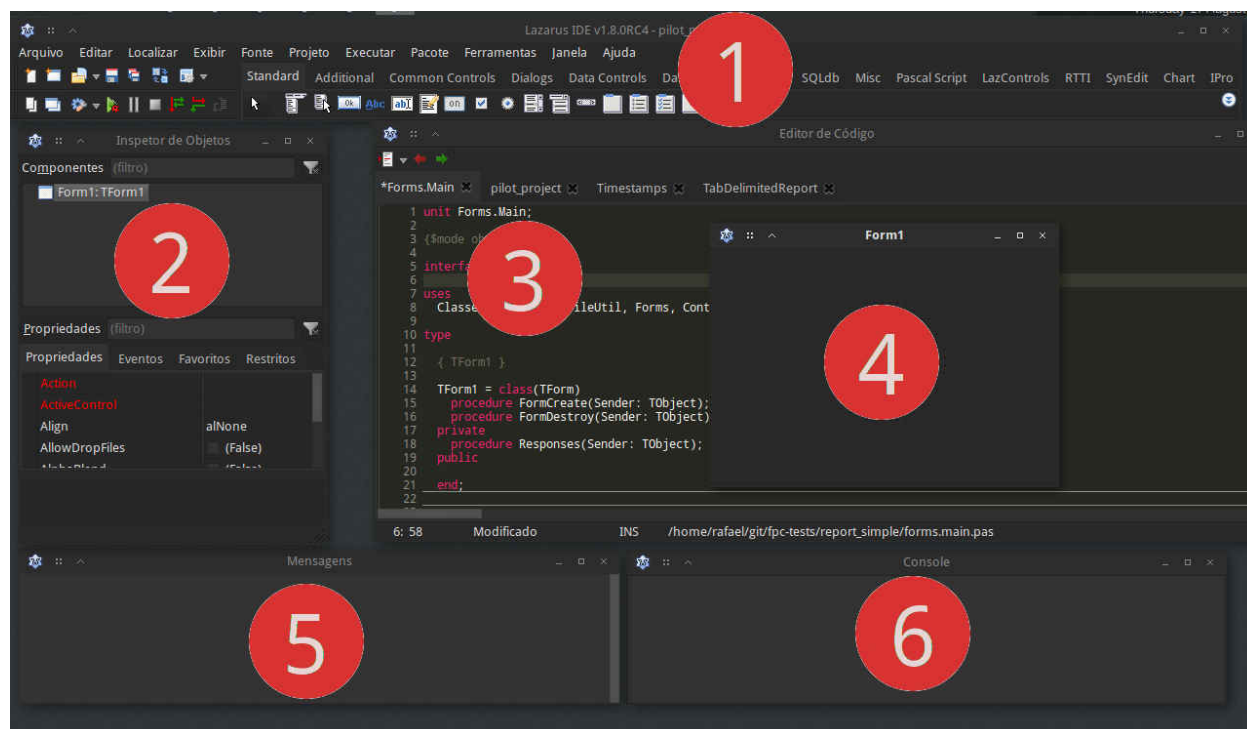


Figura 1. Janelas utilizadas neste guia. Legenda: 1, menu superior do lazarus; 2, inspetor de objetos; 3, editor de código; 4, janela principal da aplicação; 5, mensagens; 6, console.

Opcionalmente você pode trocar os esquemas de cores das janelas. Diversos esquemas de cores para as janelas estão disponíveis. Para mais informações, consulte o endereço:

<http://wiki.lazarus.freepascal.org/UserSuppliedSchemeSettings>.

Lazarus - encerrando uma aplicação

Para fechar a aplicação normalmente, utilize o comando correspondente na barra superior da janela. Para forçar o fechamento, selecione uma janela do Lazarus, por exemplo clicando sobre a janela “Editor de Código”, e pressione CTRL+F2 (Parar). O primeiro método permite avaliar a ocorrência de erros na cadeia de eventos de encerramento da aplicação. O segundo não produz a cadeia normal de eventos de encerramento e permite a interrupção da aplicação (travada), especialmente quando erros lógicos produzem espera infinita.

Lazarus - salvando o projeto de uma aplicação

Após fechar a aplicação, pressione CTRL+S e salve o projeto. Recomenda-se sempre renomear os nomes padrões para nomes que resumem a função do arquivo. O código fonte da aplicação padrão é composto por três arquivos principais. Dois arquivos (unit1.pas e unit1.lfm) compõe um “formulário” ou “janela” e o outro arquivo (project1.lpr) compõe um “projeto” ou “programa”. Adicionalmente, um arquivo de configuração do projeto (project1.lpi) e arquivos de recursos auxiliares também serão automaticamente criados ao salvar. Apenas o arquivo “unit1.pas” será editado diretamente. Renomeie esse arquivo para “Forms.Main.pas”. Renomeie o arquivo de configuração do projeto para “ProjetoPiloto.lpi” (o arquivo *.lpr será renomeado automaticamente). Como muitos arquivos estão envolvidos, recomenda-se reservar uma pasta para cada projeto.

Lazarus - depurando uma aplicação

Um depurador (*debugger*) é um programa que auxilia na detecção e correção de erros. Ao executar a aplicação padrão por meio do Lazarus (pressionando F9, por exemplo) informações que permitem melhor depuração são adicionadas ao executável. O depurador padrão utilizado é o GNU Debugger (GDB). Uma descrição detalhada de estratégias de depuração está fora do

escopo do presente guia. Entretanto, recomenda-se explorar duas delas:

- Observação em tempo real do conteúdo de variáveis por meio do menu Exibir->Janelas de depuração->Observadores;
- Adição de ponto de parada (Break Point) em linhas de código permite execução linha a linha por meio dos controles “Passar dentro” (F7) e “Passar sobre” (F8);

Para os objetivos deste guia, estratégias básicas de depuração serão apresentadas na sessão de exemplos por meio da janela “Console” (Figura 1, janela 6). Para exibi-la, pressione CTRL+ALT+O. Ao executar uma aplicação por meio do Lazarus, essa janela é a saída padrão do texto escrito por meio do construto básico da saída (WriteLn) apresentado nas sessões seguintes.

Free Pascal - Sintaxe básica

Nos tópicos seguintes, sempre que possível, os elementos da linguagem Free Pascal serão apresentados de maneira auto-explicativa nos campos de código por meio de “comentários”. Esta sessão foi planejada para ser um recurso de consulta e permitir assim como permitir a leitura corrida.

Comentários

Textos comentados são ignorados pelo compilador e permitem a documentação do funcionamento e significado de trechos do código.

Comente uma linha inteira usando duas barras no início da linha:

```
// Esta linha está comentada, pois inicia com duas barras.
```

É possível também inserir um comentário ao final da linha:

```
Este trecho não está comentado, // mas este está.
```

Comente diversas linhas por meio de chaves:

```
{
  Este trecho também está comentado,
  pois está entre chaves simples.
}
```

Comente um pedaço de texto dentro de uma linha por meio de chaves:

```
Este não, { este sim } este não.
```

Programa, Blocos, Início, Fim

Um programa pascal é um conjunto de blocos. Ele deve conter no mínimo um bloco de declaração de seu título e um bloco de comandos. Note que o ponto final demarca o final de um módulo. Outros comandos, blocos de comandos e declarações dentro de um módulo devem ser finalizados com ponto e vírgula:

```
program ProjetoPiloto    // declara o identificador, o título do programa
;                        // finaliza o bloco de declaração do nome
begin                  // inicia o bloco de comandos central
  WriteLn('Olá Mundo!'); // escreve o texto 'Olá Mundo!' e adiciona uma linha no
console
end.                   // finaliza o bloco de comandos central e o módulo
```

A linguagem não diferencia maiúsculas de minúsculas, portanto o seguinte programa é idêntico ao anterior:

```
PROGRAM projetopiloto;
BEGIN
  writeln('Olá Mundo!');
END.
```

A linguagem não é sensível à indentação (recuos, parágrafos, espaçamentos, etc.) por meio de caracteres não imprimíveis como o tab, espaço ou final de linha. Isto significa que o seguinte programa também é idêntico ao anterior:


```
program projetopiloto;begin WriteLn('Olá Mundo!');end.
```

Embora idêntico, diferentes convenções de indentação existem com o objetivo de melhorar a legibilidade do código. A linguagem permite que você crie sua própria convenção, mas recomenda-se o uso de convenções existentes.

Diretivas de compilação

Diretivas de compilação são instruções ao compilador (ao Free Pascal), não instruções do programa (neste contexto, o projeto piloto). Elas podem incluir elementos, assim como mudar o significado de elementos sintáticos de um dialeto. As seguintes diretivas especificam o dialeto utilizado neste guia. Ele corresponde ao dialeto da aplicação padrão do Lazarus (a interface gráfica):

```
// diretivas incluídas por padrão nos módulos criados pelo Lazarus
{$MODE ObjFPC}    // habilita a sintaxe de orientação ao objeto
{$H+}             // Torna o tipo String um apelido para o tipo AnsiString

// diretivas não incluídas por padrão nos arquivos, mas passadas por padrão pelo
Lazarus ao compilador
{$COOPERATORS ON} // habilita os operadores +=, -=, *= e /=
{$GOTO ON}        // habilita as palavras-chave label e goto
{$INLINE ON}      // habilita a declaração de procedimentos inlin
```

Unidades

Uma unidade é um módulo que permite o controle de sua visibilidade a outros módulos. Ela possui, necessariamente, um bloco público, visível a outros módulos, e um bloco privado, invisível a outros módulos. Identifique os blocos público e privado da unidade “Unit1” por meio dos comentários a seguir:

```

unit Unit1;           // inicia a unidade Unit1

interface             // inicia o bloco público da unidade

uses Unit2, Unit3;    // declara um bloco de uso com duas unidades

{
  As interfaces das unidades 2 e 3 são visíveis em toda a unidade 1,
  mas a interface da unidade 4 é visível apenas na implementação da unidade 1.
}

implementation        // final do bloco público e início do bloco privado da unidade

uses Unit4;           // a cláusula uses inicia um "bloco de uso" e permite usar
outras unidade

end.                  // final da unidade

```

Um módulo (programa ou unidade) pode ver a interface, mas não a implementação, de unidades em um bloco de uso por meio da cláusula “uses”. Identifique os blocos de uso na unidade Unit1 anterior por meio dos comentários. Se duas unidades diferentes declaram interfaces iguais, a interface da última unidade na lista é usada com o objetivo de evitar conflitos.

Atribuição, Variáveis, Constantes e Tipos

Uma variável é um identificador associado a um espaço reservado na memória do computador. Toda variável possui um tipo e precisa estar declarada em um bloco antes de ser usada. Identifique o bloco de declaração de variáveis por meio dos comentários a seguir:

```

var                                // inicia um bloco de declaração de variáveis
  b : boolean = true;             // declara b como um tipo boolean (booleano) inicializando-o
como true
  i : integer = -1;               // declara i como um tipo integer (número inteiro)
inicializando-o como -1
  s : string = 'Texto';          // declara s como um tipo string (texto) inicializando-o
como "Texto"
begin
  WriteLn(b);                     // converte o valor para texto e o mostra no console
  WriteLn(i);                     // converte o valor para texto e o mostra no console
  WriteLn(s);                     // mostra o texto no console
end.

```

Diferentes valores de um mesmo tipo podem ser atribuídos a uma variável no bloco de comandos. Constantes simples, diferentemente, podem ser declaradas, mas não podem receber atribuição de valores.

```

const                                // inicia um bloco de declaração de constantes
  tab = #9;                          // declara uma constante com um caractere não imprimível
(tab)
var
  b : boolean;                       // declara b como booleano
  i : integer;                       // declara i como inteiro
  s : string;                        // declara s como texto
begin
  b := false;                        // atribui false à variável b
  i := 10;                           // atribui 10 à variável i
  s := 'texto';                      // atribui "texto" à variável s
  WriteLn(b, tab, i, tab, s);        // imprime as variáveis no console usando o tab como
separador
  // tab := #32; não é possível
end.

```

Operadores

Operadores são símbolos reservados para operações comuns sobre variáveis (de tipos conhecidos). Consulte o guia de referência da linguagem para informações detalhadas sobre todos os operadores suportados (<https://www.freepascal.org/docs-html/ref/refse84.html>):

```
var
  b : boolean = false;
  i : integer = 10;
  s : string  = 'texto';
begin
  // Operações Booleanas
  b := not B;           // inverte o valor de B (para true)
  b := not B;           // inverte o valor de B (para false)
  b := 10 > 9;          // dez é maior do que nove? true
  b := 10 < 9;          // dez é menor do que nove? false
  b := 10 = 9;          // dez é igual à nove? false
  b := 10 <> 9;          // dez é diferente de 9? true

  // cuidado! a comparação entre texto diferencia maiúsculas e minúsculas
  b := s = 'texto';     // os textos são iguais? true
  b := s <> 'Texto';     // os textos são diferentes! true

  // Operações Aritméticas
  i := -i;              // inverte o sinal de i para negativo
  i := -i;              // inverte o sinal de i para positivo
  i := 10 + 10;          // soma entre dois inteiros
  i := 10 - 1;          // diferença entre dois inteiros
  i := 10 * 10;          // multiplicação entre dois inteiros
  i := 10 div 10;        // divisão entre dois inteiros
  i := 10 mod 3;         // resto da divisão entre dois inteiros

  // Operações com texto
  s := 'texto'+'texto'+'texto'; // concatenar texto
end.
```

Condições

Programas frequentemente realizam operações condicionalmente. Condições podem ser declaradas por meio de dois tipos de estruturas.

if ... then ... else

O primeiro tipo condicional permite testes booleanos e a bifurcação entre resultados verdadeiros e falsos:

```
var
  i : integer;
begin
  // se uma condição é verdadeira
  if True then
    begin
      i := 1; // então este comando será executado
    end
  else
    begin
      i := 0; // e este não
    end;

  // se uma condição é falsa
  if False then
    begin
      i := 1; // então este comando não será executado
    end
  else
    begin
      i := 0; // e este será
    end;
end.
```

“case ... of ... else ...”

O segundo tipo condicional permite testes sobre valores e texto e a bifurcação entre diversos resultados:

```
var
  i : integer = 0; // inicializa i com o valor 0
  s : string = 'a'; // inicializa s com o texto 'a'
begin
  // casos sobre valores
  case i of
    0 :           // caso i seja igual a 0
      i := 1;     // este comando será executado
    1 :           // caso i seja igual a 1
      i := 2;
    2 :
      i := 3;     // caso i seja igual a 2
  else
    i := -1;      // este comando será executado se nenhum dos casos especificados
  ocorrer
end;

// casos sobre texto
case s of
  'a'   : i := 0; // caso s seja 'a'
  'b'   : i := 1; // caso s seja 'b'
  'casa': i := 2; // caso s seja 'casa'
else
  i := -1;
end;
end.
```

Laços de repetição

Existem três tipos de laços de repetição. Dois deles permitem repetir um bloco de comandos “até que” ou “enquanto” uma condição for verdadeira. O outro permite repetir um

bloco de comandos de acordo com um intervalo de valores.

repeat ... until

Permite testar uma condição de saída após um bloco de comandos, ou seja, permite executar um bloco de comandos no mínimo uma vez e repeti-lo até que uma condição de saída seja verdadeira.

```
var
  i : integer;
begin
  i := 100;
  repeat
    i := i + 1;    // executa o bloco de comandos
  until i < 100;   // antes de testar a condição de saída
  // portanto i será igual a 101
end.
```

while ... do

Permite testar uma condição de saída antes de um bloco de comandos, ou seja, se a condição for falsa o bloco de comandos não executa nenhuma vez e, ao contrário, repetirá enquanto a condição for verdadeira.

```
var
  i : integer;
begin
  // repetir enquanto uma condição for verdadeira:
  i := 100;
  while i < 100 do // a condição de saída é executada primeiro
    begin          // portanto este bloco não será executado
      i := i + 1;
    end;

  i := 100;
  while i = 100 do // condição verdadeira, portanto
    i := i + 1;    // este bloco será executado uma vez
```

```

// condições de saída customizadas podem ser criadas
// o procedimento de saída de laço:
i := 0;
while True do      // execute
begin
    WriteLn(i)      // escreva o valor de i no console
    i := i + 1;     // e incremente i
    if i > 4 then    // se i maior do que 4 (condição de saída)
        Break;      // procedimento de saída de laço
    end;
// WriteLn produz -> 0, 1, 2, 3, 4

// também é possível pular blocos de comando dentro do bloco de repetições
i := 0;
while True do      // execute
begin
    if i < 4 then    // se menor do que 4
    begin
        WriteLn(i);
        i := i + 1; // incremente 1
        Continue;  // e continue do início
    end;
                        // se 5 ou maior
    i := i + 1;     // incremente 2
    WriteLn(i);
    Break;         // procedimento de saída do laço
end;
// WriteLn produz, o número quatro foi pulado -> 0, 1, 2, 3, 5
end.

```


for ... to ... do / for ... downto ... do

Permite repetir de acordo com um intervalo.

```
// do menor para o maior
for i := 0 to 9 do
  begin
    WriteLn(i); // 0, 1 .. 9
  end;

// do maior para o menor
for i := 9 downto 0 do
  begin
    WriteLn(i); // 9, 8 .. 0
  end;
end.
```

Vetores e Listas

Um vetor (array) é uma série de itens indexados. Cada item possui um índice e um tipo. Por padrão o primeiro item de um vetor possui índice 0. Vetores podem ser estáticos ou dinâmicos. Vetores estáticos possuem um tamanho fixo.

```
const
  space = #32;
var
  // declara e inicializa um vetor estático
  names : array [0..4] of string = ('joao', 'maria', 'rafael', 'thais', 'laura');

  // declara uma variável de tipo igual ao do vetor
  name : string;

  // apenas inteiros podem servir como índice de vetores
  i : integer;
```

```

begin
    // percorra os itens de um vetor sem se preocupar com seus índices
    for name in names do
        WriteLn(name);

    // percorra os itens de qualquer vetor por meio de seus índices
    for i:= Low(names) to High(names) do
        begin
            WriteLn(i, space, names[i]);
        end;
    end.

```

Vetores podem ser declarados como dinâmicos. Vetores dinâmicos possuem um tamanho variável.

```

var
    numbers : array of integer; // declara um vetor dinâmico
    number : integer;
begin
    SetLength(numbers, 2); // inicializa um vetor com 2 itens
    // Length(numbers);    // retorna o tamanho de um vetor, neste caso igual a 2
    // High(numbers);      // retorna o maior índice de um vetor, neste caso igual a 1
    // Low(numbers);       // retorna o menor índice de um vetor, neste caso igual a 0
    numbers[0] := 100;     // atribui um valor ao primeiro item do vetor
    numbers[1] := 200;     // atribui um valor ao segundo item do vetor

    SetLength(numbers, Length(numbers)+1); // expande o vetor, agora ele possui 3
itens
    numbers[2] := 300;

    SetLength(numbers, Length(numbers)-1); // reduz um vetor, agora ele possui 2 itens

    for number in numbers do WriteLn(number);
end.

```

Entretanto, em geral, não é recomendado usar um vetor de texto, mas sim uma lista de texto. Uma lista de texto (TStringList) é uma classe e classes serão apresentadas com mais detalhes nas seções seguintes. No momento, note que uma lista de texto é enumerável. Tipos enumeráveis podem ser percorridos como vetores, possibilitando o acesso a cada um de seus itens. Classes enumeráveis, portanto, podem ser percorridas como vetores.

```
uses Classes; // torna o tipo TStringList visível a este módulo

var
  Names : TStringList; // declara Names como do tipo TStringList
  name : string;
begin
  // Note que o caracter "ponto" (.) é utilizado
  // para acessar o conteúdo de classes e objetos

  // inicializa um objeto do tipo lista de texto (TStringList)
  Names := TStringList.Create;

  // atribui um texto delimitado à lista
  Names.DelimitedText := 'thais maria clara bárbara joana';

  // adiciona um item ao final da lista
  Names.Append('marcela');

  // percorre a lista escrevendo cada nome
  for name in Names do WriteLn(name);

  // libera a lista
  Names.Free;
end.
```

Procedimentos, Funções, Argumentos

Procedimentos e funções são estruturas que permitem a modularização e a reutilização de blocos de comandos. Por exemplo, ao invés de repetir diversas vezes os mesmos comandos, você pode declarar um procedimento e então usá-lo para executar os comandos. Considere os seguintes comandos:

```
var
  i : integer;
begin
  i := 1;
  WriteLn('-----');
  WriteLn('-   bloco de comandos   -');
  WriteLn('-----');
  WriteLn(i);

  i := 2;
  WriteLn('-----');
  WriteLn('-   bloco de comandos   -');
  WriteLn('-----');
  WriteLn(i);

  i := 3;
  WriteLn('-----');
  WriteLn('-   bloco de comandos   -');
  WriteLn('-----');
  WriteLn(i);

  i := 4;
  WriteLn('-----');
  WriteLn('-   bloco de comandos   -');
  WriteLn('-----');
  WriteLn(i);
end.
```

Uma alternativa para evitar repetições seria declarar um procedimento:

```
// declara o procedimento WriteBloc com o argumento ABlocNumber
procedure WriteBloc(ABlocNumber : integer);
begin
    WriteLn('-----');
    WriteLn('-    bloco de comandos    -');
    WriteLn('-----');
    WriteLn(ABlocNumber);
end;

var
    i : integer;

begin
    i := 1;
    WriteBloc(i); // chama o procedimento

    i := 2;
    WriteBloc(i);

    i := 3;
    WriteBloc(i);

    i := 4;
    WriteBloc(i);
end.
```

Um laço evitaria ainda mais repetições:

```
begin
    for i := 1 to 4 do WriteBloc(i);
end.
```

Todo procedimento ou função possui um identificador e uma assinatura com ou sem argumentos. Procedimentos podem ser declarados de diferentes maneiras no contexto de uma unidade, mas só é possível chamá-los de acordo com as regras de visibilidade da unidade.

```
unit Unit1;
{
    apenas a assinatura de procedimentos pode ser
    declarada na interface de uma unidade
}
interface

{ procedimentos possuem ou não argumentos de entrada em sua assinatura }

// declara o identificador PublicCommand como um procedimento sem argumentos:
procedure PublicCommand;

// declara um procedimento com um argumento:
procedure AnotherPublicCommand(AString : string);

// declara um procedimento com dois argumentos:
procedure YetAnotherCommand(AString1 : string; AInteger : integer);

{
    procedimentos declarados na interface
    devem ser redeclarados na implementação
}
implementation

// procedimentos declarados apenas na implementação
// não podem ser vistos por outras unidades usando esta unidade
procedure PrivateCommand;
begin

end;

procedure PublicCommand;
```

```

procedure NestedCommand;
begin
    // procedimentos declarados dentro de procedimentos, chamados aninhados,
    // são visíveis apenas em seu bloco de execução
end;

begin
    NestedCommand; // executa o comando aninhado deste procedimento
    PrivateCommand; // executa um comando privado da unidade
end;

procedure AnotherPublicCommand(AString: string);
    procedure NestedCommand;
    begin

    end;
begin
    NestedCommand; // executa o comando aninhado deste procedimento
    PrivateCommand; // executa um comando privado da unidade
end;

procedure YetAnotherCommand(AString1: string; AString2: string);
const
    // constantes podem ser locais
    LConst = 10;
var
    // variáveis também podem ser locais
    LInteger : integer;
    LBoolean : boolean;
    LString  : string;
    procedure LocalCommand;
    begin

    end;
begin

end;

end;

end.

```

Os argumentos de um procedimento podem receber prefixos que determinam como uma variável será passada ao procedimento. Um argumento sem prefixos é uma cópia da variável de entrada, isso significa que a cópia será modificada dentro do procedimento e a variável original não será modificada:

```
// declarando o procedimento AssignParameter
```

```
procedure AssignParameter(AValue : integer):
begin
    AValue := 20;
end;
```

```
{...}
```

```
// chamando o procedimento AssignParameter
```

```
var
    i : integer = 10;
begin
    AssignParameter(i);
    // note que i permanece igual a 10
end;
```

O prefixo “var” permite alterar a variável original de entrada:

```
// assinatura do procedimento Inc
procedure Inc(var AVariable: TOrdinal);
```

```
{...}
```

```
// chamando o procedimento Inc
```

```
var
    InputVariable : integer = 0;      // inicializa i com o valor inicial 0
begin
    Inc(InputVariable);               // incrementa i
    // InputVariable = 1
end;
```


O prefixo “out” permite alterar a variável original, mas ignora seu valor inicial.

```
// assinatura do procedimento WriteStr
procedure WriteStr(out OutputString: string; Args: Arguments);

{...}

// chamando o procedimento WriteStr
var
    OutputString : string;           // OutputString não possui um valor inicial, pois
    não foi inicializada
    i : integer = 50;
begin
    WriteStr(OutputString, i);        // converte i para texto e inicializa
    Outputstring com '50'
    // OutputString = '50'
end;
```

O prefixo “const” informa que a variável não será alterada:

```
// assinatura do procedimento ReadStr
procedure ReadStr(const S: string; Args: Arguments);

{...}

// chamando o procedimento ReadStr
var
    ConstantInput : string = '10 20 Texto';
    i1, i2 : integer;
    s : string;
begin
{
```

Importante

Argumentos do tipo "Arguments" são especiais.

Eles não podem ser redeclarados pelo programador, apenas usados por ele.

O programador pode incluir diversos argumentos de tipos conhecidos na posição de um argumento "Arguments".
O compilador fará as conversões necessárias se elas forem possíveis.

```

}
ReadStr(ConstantInput, i1, i2, s);

// ConstantInput = '10 20 Texto'
// i1 = 10
// i2 = 20
// s = 'Texto'
end;

```

Note que as funções Inc, WriteStr e ReadStr são funções da unidade System. Funções são exatamente como procedimentos, mas necessariamente retornam um resultado de um tipo específico. Use a variável "Result", automaticamente declarada, para retornar o resultado:

```

// declara uma função sem argumentos que retorna um booleano:
function GetBoolean : Boolean;
begin
    Result := true;
end;

// declara uma função que retorna um texto:
function GetString : string;
begin
    Result := 'texto';
end;

// declara uma função que retorna um valor inteiro:
function GetInteger : integer;
begin
    Result := 0;
end;

```

Conversões entre tipos frequentemente são realizadas por meio de funções. Funções comuns estão localizadas na unidade “SysUtils”.

```
uses SysUtils;           // unidade com funções de conversão

{...}

s := IntToStr(i);        // converte um inteiro para texto
i := StrToInt(s);        // converte um texto para inteiro
i := StrToIntDef(s, 0);   // converte um texto para inteiro, em caso de erro
                           retorna 0
b := StrToBoolDef(s, false) // converte um texto para booleano, em caso de erro
                           retorna false
```

Frisa-se que argumentos do tipo `Arguments` (comuns na unidade `System`) são exclusivos do compilador e não podem ser redeclarados. Se um número incerto de parâmetros de um mesmo tipo for necessário, use um vetor como argumento:

```
// declara o procedimento ManyStrings
procedure ManyStrings(AStrings : array of string);
var
  i : integer;
  s : string;
begin
  for i := Low(AStrings) to High(AStrings) do
    begin
      s := AStrings[i];
    end;
  end;

// chamando o procedimento ManyStrings

begin
  ManyStrings(['texto1', 'texto2', 'texto3']);
```

```

ManyStrings(['texto1', 'texto2']);
ManyStrings(['texto1']);
end;

```

Classes, Propriedades e Eventos

Variáveis, procedimentos e funções também permitem a construção de eventos, propriedades e classes de objetos. A arquitetura de eventos, propriedades e classes está fora do escopo do presente guia. Para informações detalhadas sobre arquitetura, procure por padrões de projeto (*Design Patterns*) nas ferramentas de busca, eles são, frequentemente, independentes de linguagens.

Ainda assim, é possível usar arquiteturas existentes ou apenas usar aspectos delas. Para isso, o objetivo no momento é de compreender um aspecto importante da arquitetura de programas orientados a objeto, sua sintaxe e como fazer uso de propriedades e eventos de classes existentes.

No contexto de programas orientados a objetos, eventos devem ser entendidos como um tipo de mensagem que um objeto pode enviar ou receber de outros objetos. Objetos são instâncias criadas por meio de classes. Classes são, literalmente, abstrações de coisas no mundo que possuem relações hierárquicas entre si. Essas abstrações tem, dentre outros, o objetivo de apreender o comportamento de coisas no mundo e tornar o programa intuitivo para aqueles que conhecem essas coisas no mundo. Por exemplo, considere uma lista de texto. O que normalmente se faz com uma lista de texto?

```

var
  list : TStringList;
begin
  // criar uma lista de texto
  // isto é, reservar um espaço na memória para ela
  list := TStringList.Create;

  // limpar o conteúdo da lista
  list.Clear;

```

```

// adicionar um texto ao final da lista
list.Append('texto 1');

// adicionar outro texto ao final da lista
list.Append('texto 2');

// alternar a posição de textos na lista
list.Exchange(0, 1);

// liberar a lista
list.Free;
end;

```

Objetos frequentemente possuem eventos associados a eles. Sintaticamente, um evento é um tipo que contém a assinatura de um método.

```

// declaração de um evento (TNotifyEvent)
// com um argumento (Sender)
// associado a um objeto (of object)
type TNotifyEvent = procedure(Sender : TObject) of object;

// "Sender" é o objeto que enviou a mensagem
// ou, em outras palavras, o objeto que disparou o evento

```

Eventos podem ser declarados como variáveis de uma classe e acessados diretamente ou por meio de propriedades. No pascal orientado a objetos, todas as classes possuem os métodos da classe TObject. Em um jargão técnico, todas as classes herdam os métodos de um ancestral comum que é o TObject. A seguir a classe TMyForm é declarada tendo como ancestral a classe TForm. A classe TForm abstrai o comportamento básico esperado de uma janela:

```

type

TMyForm = class(TForm)
private
    FNotifyEvent : TNotifyEvent;

```

```

    procedure SetSomeEvent(ANotifyEvent : FNotifyEvent);
    procedure EventNotification(Sender : TObject);
public
    property NotifyEvent : TNotifyEvent read FNotifyEvent write SetNotifyEvent;
end;

```

Existem dois operadores específicos para classes. O operador “is”, que permite testes booleanos, e o operador “as” que permitem atribuições:

```
implementation
```

```
{...}
```

```

procedure TMyForm.EventNotification(Sender : TObject):
var
    Form : TMyForm;
begin
    // testa se Sender herda de TMyForm
    if Sender is TMyForm then      // se sim então

        // o endereço de Sender como TMyForm é atribuído a Form
        Form := Sender as TMyForm;
end.

```

A aplicação padrão do Lazarus - o formato dos arquivos

Diversos elementos da sintaxe básica podem ser identificados na aplicação padrão do Lazarus. Para abrir o arquivo de projeto da aplicação; clique sobre a janela Editor de Código, pressione CTRL+O e selecione o arquivo projetopiloto.lpr. Esse arquivo possui a seguinte estrutura:

```

program ProjetoPiloto;           // projetopiloto.lpr

{$mode objfpc}{$H+}             // diretivas de compilação

uses                             // início do bloco de uso de unidades
    Interfaces,                 // uma interface específica para o sistema
torna-se disponível
    Forms,                     // torna a classe TForm visível
    Unit1                      // torna a variável Form1 visível
;                               // final do bloco de uso de unidades

{$R *.res}                      // inclui recursos auxiliares no arquivo
executável

begin                           // início do bloco de execução central do
programa
    RequireDerivedFormResource:=true; // produz um erro se uma janela for criada
sem recursos
    Application.Initialize;      // inicializa a interface, dentre outras
coisas...
    Application.CreateForm(TForm1, Form1); // cria o componente TForm1 atribuindo o
resultado à variável Form1
    Application.Run;             // carrega a janela principal (Form1) e o
laço (loop) de eventos
end.                             // final do bloco de execução central do
programa

```

Para os objetivos do presente guia, o arquivo de projeto será gerenciado automaticamente pelo Lazarus e o arquivo contendo o formulário principal será editado. Abra o arquivo Forms.Main.pas:

```

unit Forms.Main;                // título e início da unidade

{$mode objfpc}{$H+}            // diretivas de compilação

```


Exemplos

Os exemplos a seguir ilustram como resolver tarefas básicas relacionadas ao registro do comportamento e apresentação de eventos ambientais. Para isso, procedimentos e eventos simples serão implementados com o auxílio de recursos visuais da interface.

Exemplo 1. Registro tabulado de frequência e tempo

Alguns computadores pessoais permitem registrar eventos na escala de nanosegundos. Mas a escala de tempo do comportamento ao olho nú é bem mais lenta, e registros muito bem detalhados podem ser obtidos com granularidade máxima na escala de milisegundos. A granularidade do sistema de registro é sua frequência de amostragem. A amostragem deve, também, ocorrer de forma monotônica, isto é, não devem haver saltos irregulares de tempo no gerador das unidades de tempo a serem registradas.

Para obter um registro em milisegundos, implemente a unidade “Timestamps”. Crie uma nova unidade por meio do menu superior “Arquivo->Nova Unidade”:

```
unit Timestamps;

{$mode objfpc}{$H+}

interface

// essa função pode ser chamada muitas vezes
// por isso a directiva "inline" é declarada ao final
function Miliseconds(FirstTickCount : Cardinal) : string; inline;

implementation

uses SysUtils;

// um registro cumulativo de tempo deve tomar
// o primeiro registro como referência (FirstTickCount)
```

```

// o tipo cardinal só admite valores
// inteiros maiores ou iguais a zero
function Miliseconds(FirstTickCount : Cardinal) : string;
begin
    // a função GetTickCount64 retorna um tempo monotônico em milisegundos
    // a função IntToStr converte o valor para texto
    Result := IntToStr(GetTickCount64 - FirstTickCount);
end;

end.

```

Registros de texto tabulados além de permitirem a inspeção visual por meio de editores de texto simples, também permitem a automação da leitura dos dados para posterior tratamento e análise. Registros tabulados também são simples de serem implementados com o free pascal. Crie uma nova unidade e implemente um registrador tabulado da seguinte maneira:

```

unit TabDelimitedReport;

{$mode objfpc}{$H+}

interface

type

    { TTabDelimitedReport }

    TTabDelimitedReport = class
    private
        FFilename : string;
        FTextFile : TextFile;
        procedure SetFilename(AFilename: string);
    public
        procedure CloseFile;
        procedure NextFile;

```

```

    procedure WriteRow(Cols : array of string);
    property Filename : string read FFilename write SetFilename;
end;

var
    Report : TTabDelimitedReport;          // variável pública

implementation

uses SysUtils, LazFileUtils; // torna visível funções para o manuseio de arquivos

procedure TTabDelimitedReport.WriteRow(Cols: array of string);
const
    TAB = #9;
var
    i : Integer;
    LastColumn : Integer;
begin
    LastColumn := High(Cols);
    for i := 0 to LastColumn do           // percorre todos os itens
        if i < LastColumn then           // se antes do último item
            Write(FTextFile, Cols[i]+TAB) // escreve item e TAB
        else                             // se último escreve item e final de linha
            WriteLn(FTextFile, Cols[i]);
        Flush(FTextFile);                // salva as mudanças no disco rígido
    end;

procedure TTabDelimitedReport.SetFilename(AFilename: string);
var
    LFilePath, LExtension, LBaseName: string;
    i : Integer;
begin
        // retorna o caminho raiz do nome de arquivo
    LFilePath := ExtractFilePath(AFilename);

        // retorna apenas o nome base do arquivo

```

```

        // sem extensão e sem caminho
LBaseName := ExtractFileNameOnly(AFilename);

        // retorna a extensão do nome do arquivo
LExtension := ExtractFileExt(AFilename);

        // caso a extensão seja vazia ou .exe
        // a extensão torna-se '.txt'
case LExtension of
    '', '.exe' : LExtension:='.txt';
end;

        // nunca subscreva um arquivo já existente
        // se o arquivo existir, incremente seu nome
i := 0;
while FileExists(AFilename) do
    begin
        Inc(i);
        AFilename := LFilePath+LBaseName+'_data_'+Format('%3d', [i])+LExtension;
    end;

        // atribui um nome ao arquivo de texto
AssignFile(FTextFile, AFilename);
Rewrite(FTextFile);    // abre o arquivo de texto para escrita
FFilename:=AFilename;  // salva o nome do arquivo para uso posterior
end;

procedure TTabDelimitedReport.NextFile;
begin
    SetFilename(FFilename);          // abre um novo arquivo
end;

procedure TTabDelimitedReport.CloseFile;
begin
    System.Close(FTextFile);         // fecha o arquivo de texto
end;

```

```

initialization // antes de executar o programa, crie (a memória do) objeto
    Report := TTabDelimitedReport.Create;

finalization   // após finalizar o programa, libere (a memória do) objeto
    Report.Free;

end.

```

Em seguida, selecione o arquivo Forms.Main.pas (correspondente a janela principal) e use as unidades Timestamps e TabDelimitedReport na cláusula privada de uso de unidades:

```

implementation                                // campo privado da unidade

{$R *.lfm}

// torna visível a variável do relatório (Report)
// e a função Miliseconds
uses TabDelimitedReport, Timestamps;

end.

```

Para criar um arquivo de texto e o cabeçalho (“Tempo Categoria Evento”), utilize o evento de criação da janela principal:

- Selecione o arquivo Forms.Mains.pas
- Selecione a janela principal (Aperte F12)
- Clique duas vezes sobre o fundo da janela principal
- O procedimento padrão OnCreate será declarado automaticamente
- implemente o procedimento da seguinte maneira:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    // use a variável pública da unidade TabDelimitedReport
    // a propriedade Filename permite criar e inicializar
    // o arquivo de texto do relatório
    Report.Filename := Application.ExeName;

    // Application refere-se à variável
    // da unidade Forms. A propriedade ExeName
    // retorna o caminho completo do arquivo executável da aplicação

    // escreve o cabeçalho do programa
    Report.WriteRow(['Tempo', 'Categoria', 'Evento']);
end;

```

Ao final do programa, é necessário fechar o arquivo de texto. Para isso usaremos o evento de finalização da aplicação:

- Selecione Forms.Main.pas
- Alterne para a janela principal (Aperte F12)
- Clique sobre o fundo da janela principal
- Selecione a janela Inspetor de Objetos (Aperte F11)
- Selecione a aba Eventos
- Clique duas vezes sobre o campo em branco do evento OnDestroy:
- O procedimento será declarado automaticamente, implemente-o da seguinte maneira:

```

procedure TForm1.FormDestroy(Sender: TObject);
begin
    // fecha o arquivo de texto
    Report.CloseFile;
end;

```

Por meio de diversas chamadas ao procedimento WriteRow do objeto Report, um relatório em formato de texto simples com a seguinte estrutura é esperado:

Tempo	Categoria	Evento
0000	estimulo	S1
2000	resposta	R1
2500	estimulo	C1
5050	estimulo	S1
5500	resposta	R2
6000	resposta	R2
6100	resposta	R2
7000	resposta	R2
9000	resposta	R1
9500	estimulo	C1

Um relatório deve conter todas as informações de interesse. Nesse caso, duas respostas (R1 e R2) e dois estímulos (S1 e C1) devem ser registrados pelo programa. O programa está pronto para rastreá-los. Mas como detectar a ocorrência desses eventos?

Exemplo 2. Rastreamento de estímulos e respostas

Rastreadores de função e forma são requisitos para análises comportamentais que almejam alto poder preditivo. Por ser multideterminado, o comportamento demanda a análise conjunta de múltiplas fontes de dados. Na presente ocasião, por questões didáticas, a ênfase será na forma. Ao interagir com uma interface gráfica, dois tipos de eventos serão de especial interesse: respostas ao teclado e respostas ao mouse. Adicionalmente, estímulos, como mudanças na visibilidade de componentes da interface, serão rastreados por meio de um evento customizado. Crie uma nova unidade e implemente o evento da seguinte maneira:

```

unit Behavior.Events;

{$mode objfpc}{$H+}

interface

uses
  Classes;    // torna visível a classe TObject

type          // bloco de declaração de tipo
  // o procedimento recebe o nome de uma categoria, um evento
  // e o objeto que enviou a mensagem
  TBehavioralEvent = procedure(Sender: TObject; const Category: string; const
    Event:string) of object;

  // define as possíveis categorias como constantes simples
const
  BehavioralEvent = 'resposta';
  EnviromentEvent = 'estimulo';
  SystemEvent = 'virtual';

implementation

  // toda unidade precisa de um bloco de implementação
  // ainda que vazia

end.

```

A unidade com o evento comportamental deve ser usada em duas outras unidades. Primeiramente, a classe de estímulos TStimulus deve criada a partir da classe TImage. A classe TImage possui eventos de mouse e métodos para a apresentação de figuras. Crie uma nova unidade e implemente a classe TStimulus da seguinte maneira:


```

unit ExtCtrls.Stimulus;

{$mode objfpc}{$H+}

interface

uses
    ExtCtrls,          // torna visível a unidade TImage
    Behavior.Events;    // torna visível o evento TBehavioralEvent

type

    { TStimulus }

    TStimulus = class(TImage) // cria a classe TStimulus a partir da classe TImage
    private
        FOnVisibilityChange: TBehavioralEvent;
        procedure SetOnVisibilityChange(AValue: TBehavioralEvent);
    protected
        // a directiva override
        // permite customizar o procedimento
        // SetVisible da classe TImage
        // este procedimento é usado
        // para detectar a mudança de visibilidade
        // dos estímulos
        procedure SetVisible(Value: Boolean); override;
    public
        // para declarar o evento comportamental
        // escreva:
        // property OnVisibilityChange : TBehavioralEvent;
        // e em seguida aperte CTRL+SHIFT+C
        // a propriedade será declarada automaticamente
        property OnVisibilityChange : TBehavioralEvent read FOnVisibilityChange write
SetOnVisibilityChange;
    end;

```

implementation

```
{ TStimulus }
```

```
procedure TStimulus.SetOnVisibilityChange(AValue: TBehavioralEvent);
begin
    if FOnVisibilityChange=AValue then Exit;
    FOnVisibilityChange:=AValue;
end;
```

```
// implementação de eventos de estímulo
procedure TStimulus.SetVisible(Value: Boolean);
begin
    // primeiramente é necessário
    // chamar o procedimento SetVisible de TImage
    // isso é possível por meio do prefixo inherited
    inherited SetVisible(Value);
    // se um valor foi atribuído à propriedade então
    if Assigned(OnVisibilityChange) then
        if Value then // se visível
            // dispara o evento da propriedade como "Show"
            OnVisibilityChange(Self, EnviromentEvent, 'Show')
        else // se invisível
            // dispara o evento da propriedade como "Hide"
            OnVisibilityChange(Self, EnviromentEvent, 'Hide');
end;

end.
```

Em seguida os estímulos devem ser criados, configurados e apresentados na janela principal da aplicação. Os eventos associados à janela também devem ser implementados e configurados. Implemente-os da seguinte maneira:

```
unit Forms.Main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, StdCtrls,
  ExtCtrls.Stimulus; // torna visível a classe TStimulus

type

  { TForm1 }

  TForm1 = class(TForm)
    // declara um estímulo antecedente
    StimulusAntecedent : TStimulus;

    // declara um estímulo consequente
    StimulusConsequent : TStimulus;

    // declara o procedimento dos eventos de clique da janela
    procedure ComponentClick(Sender: TObject);

    // declara o procedimento dos eventos de teclado da janela
    procedure ComponentKeyPress(Sender: TObject; var Key: char);

    // declara o procedimento de criação da janela
    procedure FormCreate(Sender: TObject);

    // declara o procedimento de destruição da janela
```

```

    procedure FormDestroy(Sender: TObject);

    // declara o procedimento de registro de respostas e estímulos
    procedure RecordBehavior(Sender: TObject; const Category:string;
        const EventSufix: string); inline;
private
    // declara uma variável privada para o valor de início do registro
    FFirstTickcount : Cardinal;

    // declara um método de ajuda para a criação e configuração dos
    // estímulos
    procedure CreateStimulus(out AStimulus : TStimulus; AColor : TColor;
        ASize : integer = 300; ALeft : integer = 0; ATop: integer = 0);
public

end;

var
    Form1: TForm1;

implementation

// unidades usadas apenas na implementação
uses TabDelimitedReport, Timestamps, Behavior.Events;

{$R *.lfm}

{ TForm1 }

procedure TForm1.FormCreate(Sender: TObject);
begin
    // cria o estímulo antecedente com cor preta
    CreateStimulus(StimulusAntecedent, clBlack,150,0,0);

    // cria o estímulo consequente com cor azul

```

```

CreateStimulus(StimulusConsequent, clBlue,100,200,0);

// os estímulos são rastreados pelos seus
// respectivos nomes
StimulusAntecedent.Name:='Preto';
StimulusConsequent.Name:='Azul';

// cabeçalho do relatório
Report.Filename := Application.ExeName;
Report.WriteRow(['Tempo', 'Categoria', 'Evento']);

// valor de início do registro independente da hora local
FFirstTickCount := GetTickCount64;

// início de acordo com a data e hora local
RecordBehavior(Sender, SystemEvent, 'inicio:'+DateTimeToStr(Now));

// mostra o estímulo antecedente
StimulusAntecedent.Show;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    // registra o final de acordo com a hora local
    RecordBehavior(Sender, SystemEvent, 'final:'+DateTimeToStr(Now));

    // finaliza o relatório
    Report.CloseFile;
end;

procedure TForm1.RecordBehavior(Sender: TObject; const Category: string;
    const EventSufix: string);
var
    SenderName: string;
begin
    // o rastreamento ocorre por meio

```

```

// do nome dos objetos
if Sender is TComponent then
    SenderName := TComponent(Sender).Name
else
    SenderName := Sender.ClassName;

// registra uma linha no relatório
Report.WriteRow([
    Miliseconds(FFirstTickcount),
    Category,
    SenderName+#32+EventSufix
]);
end;

procedure TForm1.CreateStimulus(out AStimulus: TStimulus; AColor: TColor;
    ASize: integer; ALeft: integer; ATop: integer);
begin
    // note que AStimulus é um argumento de saída (out)

    // cria o estímulo na janela principal (self)
    AStimulus := TStimulus.Create(Self);

    // define o tamanho do estímulo
    // Left e Top possuem origem no canto superior esquerdo do monitor:
    // Left pixel horizontal
    // Top pixel vertical
    AStimulus.SetBounds(ALeft, ATop, ASize, ASize);

    // define o tamanho da figura do estímulo
    AStimulus.Picture.Bitmap.SetSize(ASize, ASize);

    // define a cor da figura do estímulo
    AStimulus.Picture.Bitmap.Canvas.Brush.Color := AColor;

    // desenha um retângulo preenchido na figura com a cor definida
    AStimulus.Picture.Bitmap.Canvas.Rectangle(0,0, ASize, ASize);

```

```

{***** MUITO IMPORTANTE *****)

// não esqueça de definir aonde o estímulo será desenhado
// isto é possível por meio da propriedade Parent
// aqui definimos a janela principal (Self) como o
// responsável por desenhar o estímulo

    AStimulus.Parent := Self;

{*****}

// define a visibilidade inicial do estímulo
AStimulus.Hide;

// atribui um valor às propriedades dos estímulos
// o operador @ deve ser usado na frente do
// procedimento correspondente ao evento
// da propriedade
// para lembrar a assinatura do evento
// segure CTRL e clique com o botão esquerdo na
// propriedade
AStimulus.OnVisibilityChange:=@RecordBehavior;
AStimulus.OnClick:=@ComponentClick;

// também seria possível carregar uma figura
// por meio do nome do arquivo da figura
// AStimulus.Picture.LoadFromFile(AFilename);
// AStimulus.Stretch := True;
end;

// o que acontece quando um componente é clicado?
procedure TForm1.ComponentClick(Sender: TObject);
begin

```

```

// registra o comportamento de clique
RecordBehavior(Sender, BehavioralEvent, 'Click');

// altera a visibilidade dos estímulos
// de acordo com os estímulos clicados
if Sender = StimulusAntecedent then
begin
    StimulusAntecedent.Hide;
    StimulusConsequent.Show;
end;
if Sender = StimulusConsequent then
begin
    StimulusConsequent.Hide;
    StimulusAntecedent.Show;
end;
end;

// o que acontece quando uma tecla é pressionada
// tendo um componente em foco?
procedure TForm1.ComponentKeyPress(Sender: TObject; var Key: char);
const
    SpaceKey = #32;
    DeleteKey = #127;
var
    Event : string = '';
begin
    case Key of
        SpaceKey : Event := '<32>';
        DeleteKey: Event := '<127>';
        #0..#31  : Event := '<NA>';
    end;
    RecordBehavior(Sender, BehavioralEvent, Event);
end;

end.

```


Por fim, configure os eventos de clique (OnClick como ComponentClick) e teclado (OnKeyPress como ComponentKeyPress) da janela principal:

- Selecione Forms.Main.pas
- Alterne para a janela principal (Aperte F12)
- Clique sobre o fundo da janela principal
- Selecione a janela Inspetor de Objetos (Aperte F11)
- Selecione a aba Eventos
- No evento OnClick, selecione o evento ComponentClick na lista:
- No evento OnKeyPress, selecione o evento ComponentKeyPress na lista

Execute a aplicação e confira os resultados. O código fonte dos exemplos apresentados e de outros exemplos podem ser conferidos no seguinte repositório:

- <https://github.com/cpicanco/free-pascal-prototypes>

Referências

- Apple Computer Inc (Org.). (1985). *Inside Macintosh* (1º ed, Vol. 1). Cupertino, CA: Addison-Wesley.
- Borland International Inc. (1984/1983). *Turbo Pascal Reference Manual* (2º ed). Scotts Valley, CA.
- Jensen, K., & Wirth, N. (1974). *PASCAL User Manual and Report* (Vol. 18). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: [10.1007/978-3-662-21554-8](https://doi.org/10.1007/978-3-662-21554-8)
- Klämpfl, F., Codère, C. E., Mantione, D., Maebe, J., Van Canneyt, M., Vreman, P. van de Voort, M. (2017). Free Pascal Compiler (Versão v3.0.4rc1) [Multi-plataforma]. Recuperado de <http://www.freepascal.org/>
- Lazarus IDE (Versão v1.8RC4) [Programa de computador]. (2017). Recuperado de <http://www.lazarus-ide.org>
- Severance, C. (2012). The Art of Teaching Computer Science: Niklaus Wirth. *IEEE Computer Society*, 45(7), 8–10. doi: [10.1109/MC.2012.245](https://doi.org/10.1109/MC.2012.245)



Visual Basic.NET

Carlos Eduardo Costa |  |  |

O Visual Basic.NET (VB.NET) é uma linguagem de programação visual, baseada na linguagem BASIC. O BASIC surgiu nos anos 60, e programar nesta linguagem era difícil e trabalhoso, para dizer o mínimo. Cada botão que aparecesse na tela (que não tinha o “apelo” visual de hoje em dia), cada parte do aplicativo (por exemplo, uma função “Salvar”, para salvar um arquivo em uma pasta) tinha de ser planejada e totalmente programada.

No início dos anos 90, surgiu o VB (ainda sem o .NET) que ofereceu a Interface Gráfica do Usuário (GUI) – por isso o “visual” se juntou ao BASIC. Mas o VB era muito mais do que o BASIC com uma interface visual. O VB oferecia recursos poderosos de programação, além da interface gráfica, tais como tratamento de eventos e acesso a API do Windows (*Application Programming Interface*, API). Com a API você podia usar recursos do Windows em suas aplicações. Por exemplo, em vez de programar toda a função para salvar os arquivos em uma pasta, você podia chamar um recurso da API do Windows que abria a janela “Salvar Como”, com todo o seu visual e funções já prontas.

Em 2000 surge a plataforma .NET que permitiu que os aplicativos baseados em Web pudessem ser distribuídos a uma variedade de dispositivos (telefones celulares e *tablets*) e a computadores de mesa ou *laptops*. Ou seja, aplicativos criados em linguagens de programação

incompatíveis podiam se comunicar uns com os outros com a plataforma .NET (Deitel, Deitel, & Nieto, 2004).

A escolha de uma linguagem de programação envolve diversos fatores (cf. Costa, 2006). Talvez o VB.NET seja bom para algumas pessoas pelos motivos A e B; para outras, pelo motivo C e para outras não seja a opção ideal. Uma das razões desse livro é justamente apresentar as várias linguagens de programação para que o leitor avalie, para seus propósitos e contexto, qual a melhor escolha.

Uma vantagem é que programar em VB.NET ajuda muito a escrever programas em VBA (*Visual Basic for Applications*), que é uma linguagem utilizada para escrever macros no Excel®. Geralmente eu programo um aplicativo para coleta de dados em VB.NET e, depois, exporto os resultados para o Excel® e programo uma análise preliminar desses dados em VBA, automatizando a tarefa de gerar gráficos e tabelas.

Aprender qualquer linguagem de programação traz desafios. Se você tem ou teve familiaridade com o VBA ou BASIC, talvez tenha mais facilidade com o VB.NET do que outras linguagens. Se você teve experiências com javascript, php ou action script, talvez prefira as linguagens C, C++ ou C#, pois as sintaxes são parecidas. Se teve experiência com PASCAL é provável que goste mais de programar em Delphi. Se você não teve nenhuma história de programação, eu acredito (por pura especulação) que o esforço e os desafios serão parecidos entre as diversas linguagens. Entretanto, isso é só uma suposição. A diferença na curva de aprendizagem em cada uma dessas linguagens pode ser um bom tema de pesquisa!

Obtendo, instalando e configurando o ambiente de programação do VB.NET

O VB.NET faz parte de um pacote do Visual Studio (VS.NET). O VS.NET é um conjunto completo de ferramentas de desenvolvimento de aplicativos para *desktop* e aplicativos móveis, entre outros, em linguagens Visual Basic, Visual C# e Visual C++ que usam todos o mesmo ambiente de desenvolvimento integrado (IDE) (Liuson, 2017). O VS.NET 2017 é a edição mais recente do ambiente de desenvolvimento integrado da Microsoft e possui a edição gratuita Community e as edições pagas Professional e Enterprise (Garret, 2017).

O VS.NET 2017 pode ser instalado em qualquer PC com Windows 7 SP1 ou superior (com as atualizações mais recentes do Windows): Home Premium, Professional, Enterprise ou Ultimate. É necessário um processador de 1,8 GHz ou mais rápido com núcleo duplo (Dual Core) ou superior; 4 GB ou mais de memória RAM; espaço em disco rígido entre 1 GB e 40 GB, dependendo dos recursos que você quiser instalar; placa de vídeo com suporte a uma resolução de exibição mínima de 720p (1280 por 720 pixels) – funcionando melhor com uma resolução de WXGA (1366 por 768 pixels) ou superior. Para mais detalhes dos requisitos do sistema, consulte a página: <https://www.visualstudio.com/pt-br/productinfo/vs2017-system-requirements-vs>.¹

Para baixar a versão gratuita do VS.NET (que contém o VB.NET) acesse <https://www.visualstudio.com/pt-br/downloads> e clique no botão [*Download* gratuito] no

¹ Acessado em 05/08/2017. Lembre-se que este endereço pode sofrer alterações. Qualquer dúvida, procure “requisitos do sistema para Visual Studio” em mecanismos de busca. Isso vale para todos os *links* deste capítulo.

retângulo *Visual Studio Community*. O arquivo executável (.exe) tem 1 MB, aproximadamente. Esse arquivo instalará um gerenciador para *download* e instalação do VS.NET, propriamente dito. Depois de baixado o arquivo dê um duplo *click* sobre ele. Aparecerá uma janela de aviso dizendo que, ao clicar em [Continuar], você concorda com os “Termos de Licença”. Clique no botão [Continuar]; siga as eventuais instruções na tela e aguarde. Veja um vídeo bem bacana sobre a instalação do VS.NET 2017 em https://www.youtube.com/watch?v=SH2HnOUq5_w.

O ambiente de desenvolvimento integrado (IDE)

Depois de instalar e clicar sobre o ícone do Visual Studio.NET em sua área de trabalho, uma janela semelhante à Figura 1 abrirá na tela do seu computador.

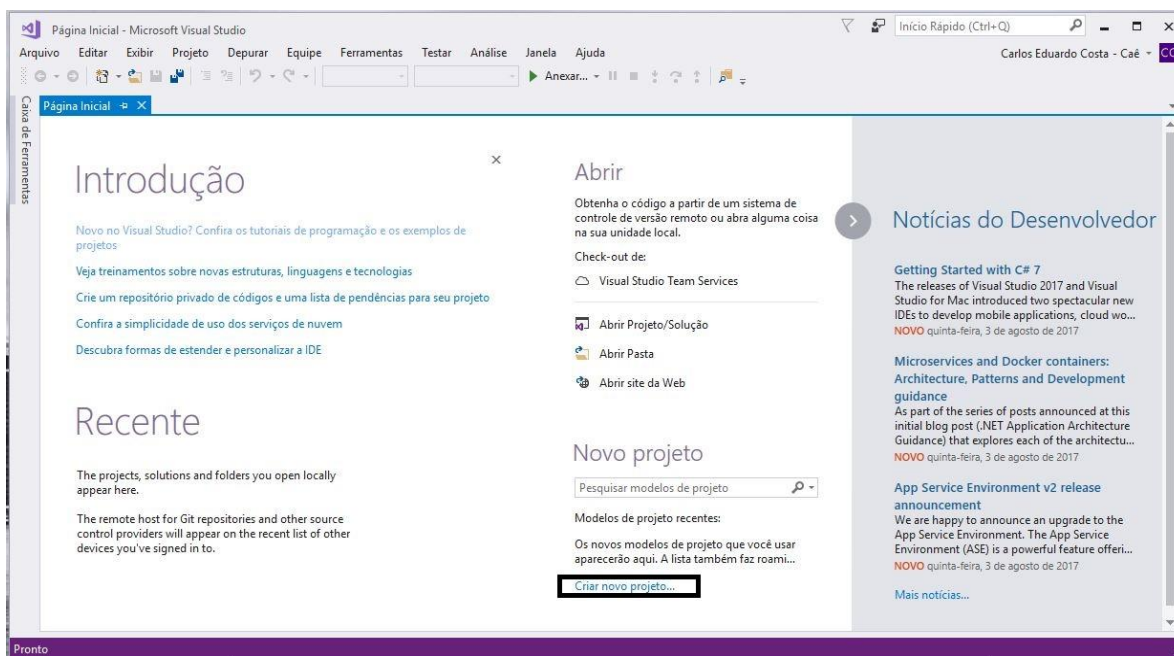


Figura 1. Página inicial do VS.NET.

Para que possamos ver a IDE do Visual Basic teremos de criar um novo projeto. No centro inferior da janela, marcado com um retângulo de bordas pretas na Figura 1, há a opção “Criar novo projeto...”. Ao clicar sobre ele a janela da Figura 2 aparecerá.

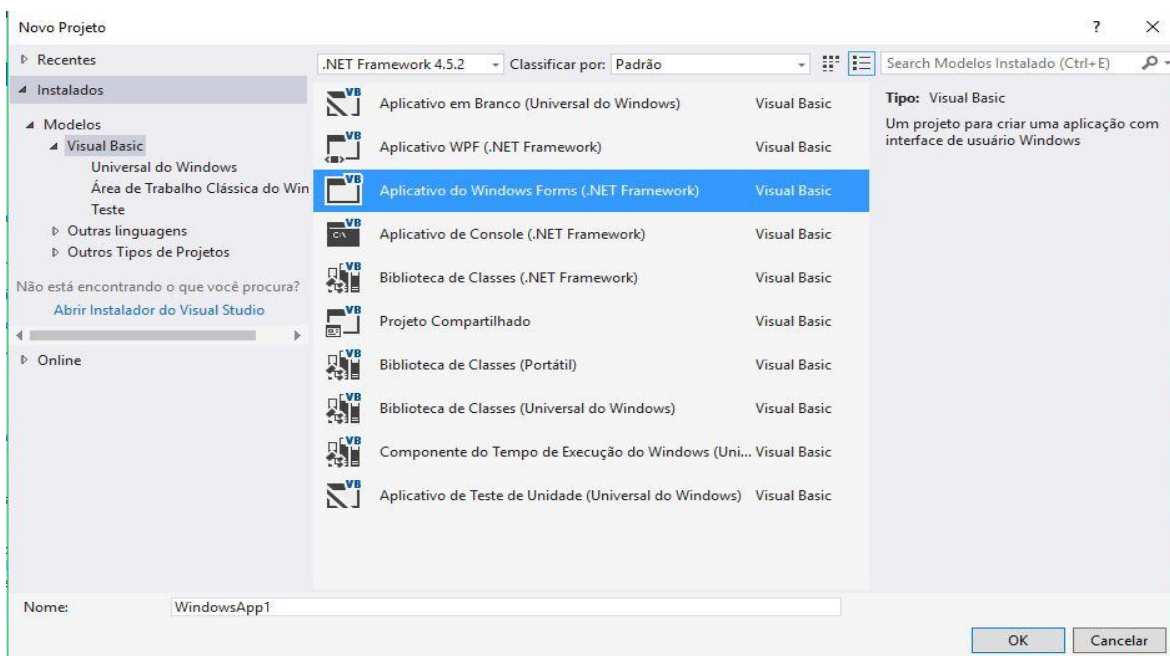


Figura 2. Janela de abertura de um Novo Projeto no VS.NET.

Vamos selecionar a terceira opção da caixa central, chamada “Aplicativo do Windows Forms (.NET Framework)” e substituir o nome do aplicativo de “WindowsApp1”, que aparece na parte inferior da janela, por “TESTE”. Depois clique em OK. Pronto! Chegamos à IDE! Uma janela padrão do Windows® já está criada, conforme você pode ver na Figura 3.

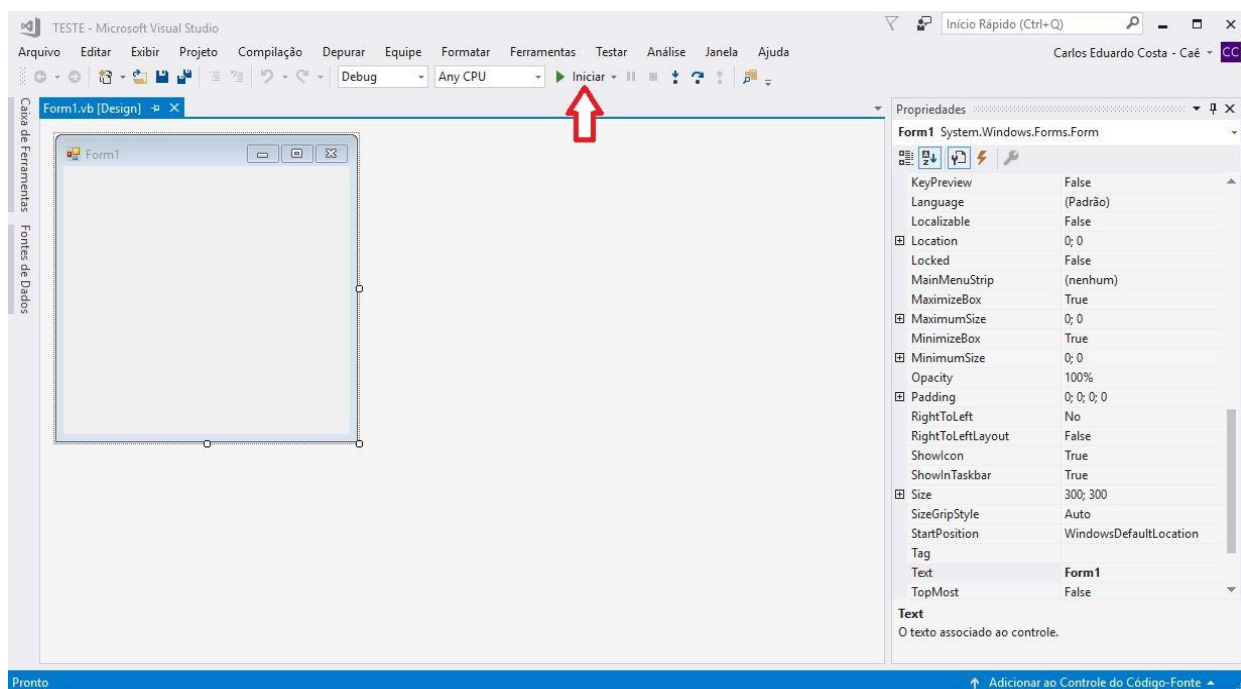


Figura 3. A IDE do aplicativo TESTE.

Repare que o nome do aplicativo (“TESTE”) aparece no canto superior esquerdo da janela, próximo ao logo do VS.NET. Vamos explorar um pouco o que foi criado. Clique no botão [Iniciar] que aparece na posição central, logo abaixo do menu (veja a seta vermelha na Figura 3).

Após alguns segundos aparecerá uma janela com o nome Form1 na tela do seu computador. Essa janela já é seu aplicativo inicial (ou, mais especificamente, uma parte do que será seu aplicativo final). Note que na parte superior dessa janela, do lado direito, há os três botões-padrões de formulários do Windows. Você pode minimizar a janela, expandi-la e fechá-la! Experimente! Ao clicar no botão com um “X” a janela será fechada e o aplicativo encerrado. Veja, você não teve de programar nada! Bastou incluir um formulário no seu projeto e ele já vem com essas funcionalidades!

Repare agora na Janela “Propriedades” que aparece do lado direito na Figura 3. A partir dela você pode configurar diversas propriedades do seu formulário. Vamos alterar apenas o nome do formulário de “Form1” para “Meu aplicativo”. Para fazer isso, vamos mudar o nome na propriedade “Text” do formulário. Veja que, na Figura 3, essa opção parece na penúltima posição e com o nome “**Form1**” em negrito. Clique com o cursor do *mouse* onde está “Form1”, apague, digite “Meu Aplicativo” e pressione [ENTER] no seu teclado. Ao fazer isso, veja que o nome do seu formulário, que aparece no lado esquerdo superior da janela (ver Figura 3), irá mudar para “Meu Aplicativo”.

Sintaxe básica: um pouco de programação

Vamos construir um aplicativo bem básico para que o leitor possa se familiarizar com a sintaxe (o jeito de escrever) um aplicativo em VB.NET. No canto esquerdo da tela há uma aba denominada “Caixa de Ferramentas”. Coloque o cursor do mouse sobre ela e pressione o botão esquerdo do mouse. A aba vai se expandir e aparecerão alguns títulos. Selecione “Controles Comuns” clicando sobre ele. Aparecerão vários controles que podem ser adicionados ao formulário. Todo o desenho de uma interface com o usuário em qualquer aplicativo construído em VB é composto de um formulário e os elementos relevantes para o aplicativo. Estes elementos são os blocos de construção dos aplicativos em Windows e são chamados de controles (Petroutsos, 2010). Veja a Figura 4.

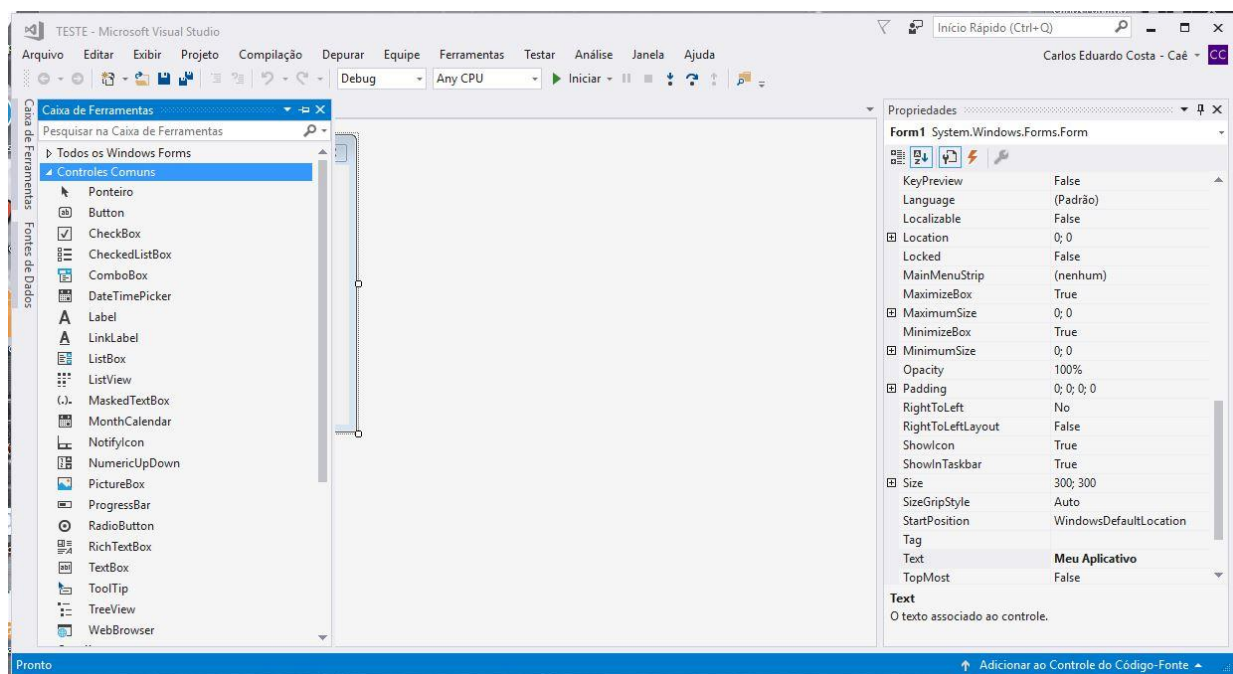


Figura 4. Aba da Caixa de Ferramentas selecionada.

Primeiro vamos inserir um botão no **Form**. Para isso, clique duas vezes sobre a opção “Button” (a segunda opção de “Controles Comuns” da “Caixa de Ferramentas”). Volte para o **Form** e veja que um botão foi inserido no canto superior esquerdo do **Form**. Arraste-o para o centro inferior do **Form**. Em seguida repita a tarefa selecionando o controle “Label” na lista de opções da “Caixa de Ferramentas”. Arraste o **Label** para a parte central superior do **Form**. Veja a Figura 5.

Temos um *layout* do nosso aplicativo. Mas ele não tem funcionalidades. Clique no botão [Iniciar] que aparece na posição central, logo abaixo do menu (veja localização deste botão na seta vermelha da Figura 3). Após alguns segundos a tela do aplicativo aparecerá em seu computador. Clique com o botão esquerdo do *mouse* sobre o botão [Button1] do “Meu

Aplicativo”. Embora você possa perceber que a parte visual esteja funcionando (o botão muda de cor quando você o aciona), nada mais acontece.

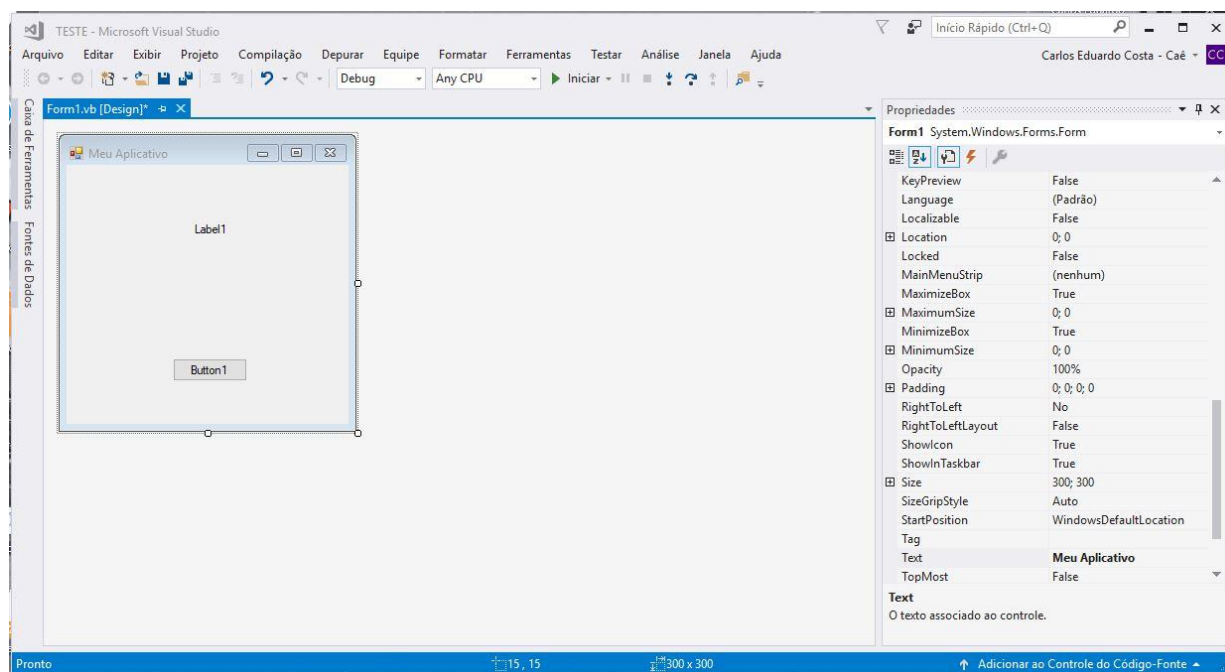


Figura 5. Layout do formulário “Meu Aplicativo” com um *button* e um *label* inseridos.

Então vamos à programação propriamente dita! Clique no botão fechar no seu aplicativo (aquele com um X, no canto superior direito, que fecha qualquer janela do Windows). Voltamos para a janela da Figura 5. Posicione o cursor sobre o botão [Button1] e dê dois cliques sobre ele. Aparecerá uma janela como da Figura 6.

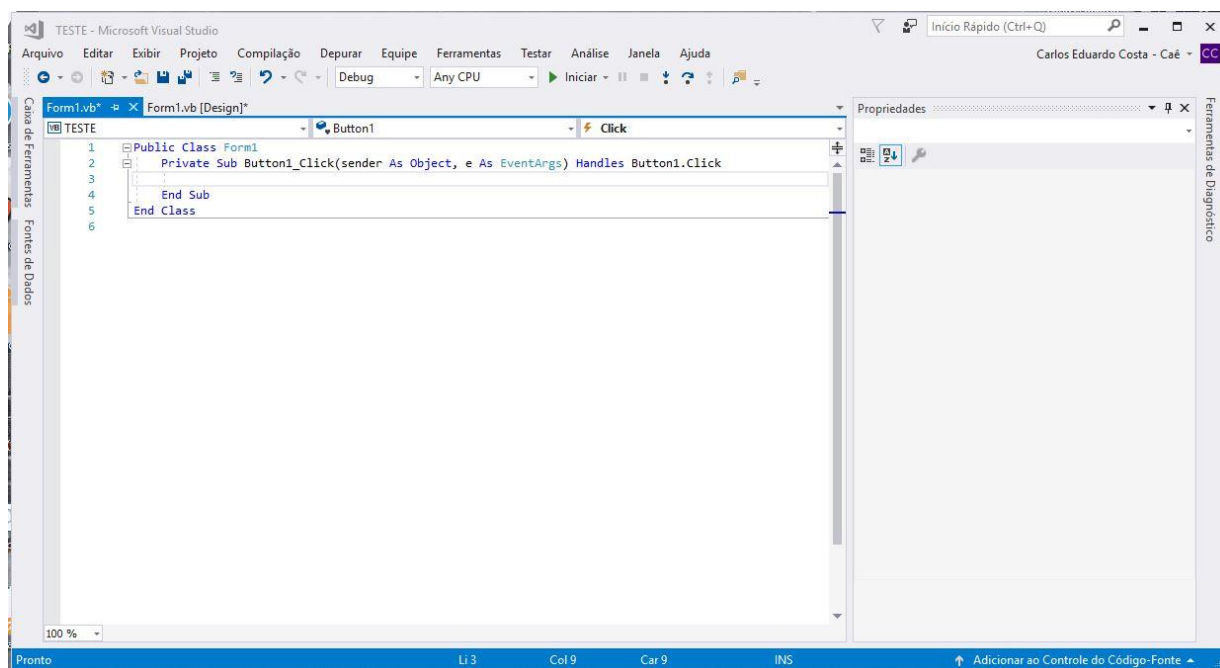


Figura 6. Janela na qual será escrito o código-fonte do seu aplicativo.

Note que uma nova aba aparece na parte superior esquerda. Agora temos a aba “Form1.vb” na qual será escrito o código-fonte do seu aplicativo e a aba “Form1.vb [Design]” na qual está a parte visual do seu aplicativo. Vamos colocar o cursor do mouse do lado direito da primeira linha e pressionar a tecla [ENTER]. Neste espaço, antes do comando “Private Sub...”, vamos declarar as variáveis que iremos usar. A Figura 7 exibe a janela com o código-fonte. Abaixo vamos explicar passo a passo.

Eu quero criar um contador que registre o número de vezes que o botão [Button1] será acionado (i.e., o número de respostas dadas). Para isso eu preciso criar uma variável que armazena esse número. Como se trata de um número inteiro, minha variável será uma *Integer* (é

um tipo de variável que armazena números inteiros no VB.NET)². Posso dar qualquer nome para minha variável. Mas, como qualquer aplicativo tem inúmeras variáveis é bom que você dê um nome que ajude você a lembrar que tipo de variável é (Integer, nesse caso) e o que faz (contador de respostas, neste caso). Portanto, chamarei minha variável de intContResp. A declaração da variável será:

```
Dim intContResp As Integer
```

A palavra “Dim” serve para declarar e alocar espaço de armazenamento para uma ou mais variáveis. A palavra “intContResp” foi o nome que dei para a minha variável. Eu poderia ter escolhido o nome “Clicadas”, “Respostas” ou qualquer outra. Eu prefiro (e aconselho) a iniciar o nome das variáveis pelo que elas armazenam de informação. Portanto, o nome da variável começou com “int” para ficar mais fácil de lembrar que essa variável armazena um número inteiro. “ContResp” porque é o que ela irá armazenar: “contagem de respostas”. “As Integer” significa que estou declarando (Dim) esta variável (intContResp) como uma variável do tipo Integer (i.e., uma variável que armazena números inteiros).

Quando o botão for acionado ele chamará o evento Button.Click. Então, temos de escrever no evento “Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click” (ver Figura 6) o que o aplicativo deve fazer.

² Você vai aprender mais sobre isso à medida que se aprofundar nos estudos de VB.NET. Lembre-se que o objetivo aqui é só mostrar o básico sem entrar em muitos detalhes. Você saberá mais sobre variáveis em VB.NET e como construir outros aplicativos em http://www.macoratti.net/vbn_bas1.htm.

```
intContResp = intContResp + 1
```

Agora, cada vez que o botão for acionado, o evento `Button.Click` é chamado e, então, ele pegará o valor da variável `intContResp` e somará mais 1 ao valor atual da variável. Na primeira vez será $0 + 1 = 1$; na segunda vez será $1 + 1 = 2$; na terceira vez que o botão for acionado ele contará $2 + 1 = 3$ e assim por diante.

O aplicativo já está contando o número de respostas, mas o que ele deve fazer com isso? Vamos “dizer” para o aplicativo (i.e., vamos programar para) exibir o resultado na `Label1`, que inserimos no nosso Form. Para isso, escreva a seguinte linha de código, abaixo da instrução para contar respostas:

```
Label1.Text = CStr(intContResp)
```

O comando da linha acima exibe o valor atual de “`intContResp`” no `Label1`. Todavia, uma *label* só aceita *strings* no “`.Text`” (letras ou caracteres em forma de letras), então, precisamos converter a variável do tipo `Integer` do `intContResp` em uma do tipo `String`³. Fazemos isso com o comando “`CStr`” (converte a variável, declarada entre parênteses, em *string*). A Figura 7 exibe as linhas de código digitadas no VB.NET.

³ Se eu tenho o número inteiro 1 e somo mais 1, eu teria o número inteiro 2 como resultado. Entretanto, se eu tenho um caractere (uma letra, um texto) “1” e somo mais “1” caractere, eu teria “11” como resultado. Neste último caso o programa entende que você quer juntar o primeiro caractere com o segundo (como se fosse juntar “c” + “a” = “ca”; “1” + “1” = “11”). Por isso, no exemplo, eu preciso somar números inteiros e, depois, transformá-los em *string* para exibir, como texto, na *label*.

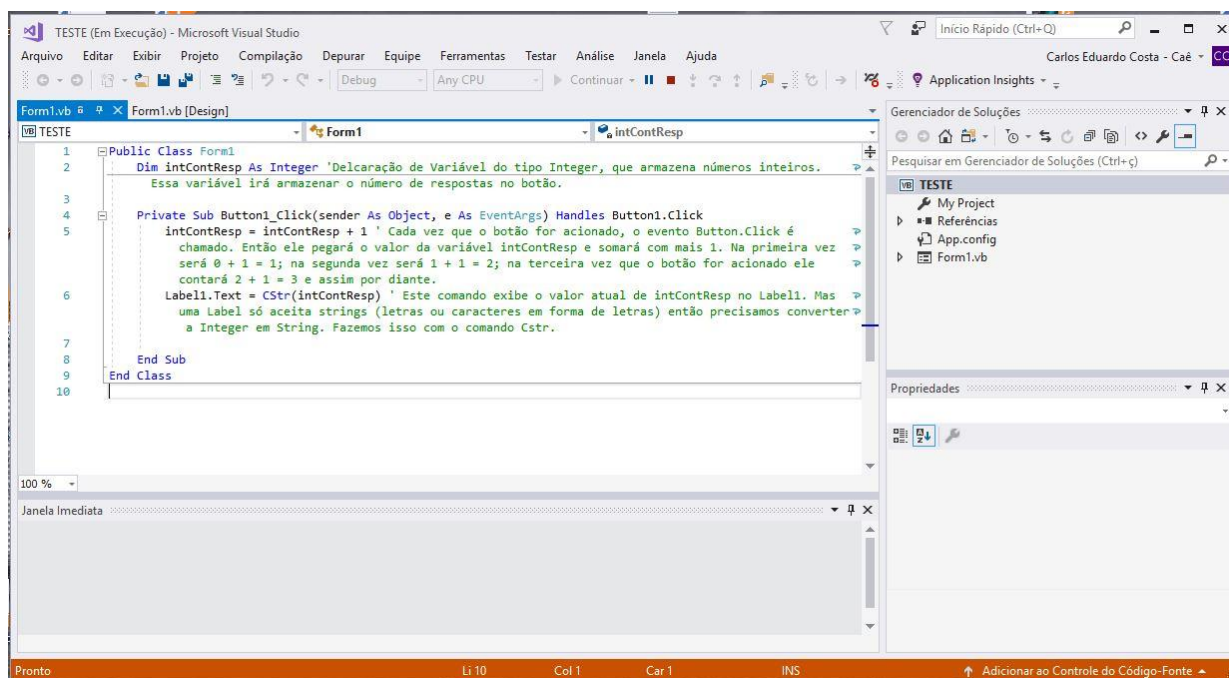


Figura 7. Linhas de código digitadas no VB.NET.

Note que, no exemplo que aparece na Figura 7, há diversos trechos escritos em letras verdes. Esses são comentários. Tudo o que você escreve no código-fonte depois de um apóstrofe (‘) é considerado como um comentário. Os comentários não são compilados pelo seu aplicativo. Mas eles são muito úteis! À medida que um aplicativo “cresce” vai se tornando mais difícil você lembrar o que faz cada trecho do código. Por isso, comentar seu código-fonte, à medida que você vai programando, é muito importante (Costa, 2006).

O aplicativo que construímos conta o número de pressões ao botão. Agora, vamos continuar a programação para que o aplicativo exiba um ponto para cada 10 pressões ao botão. Com isso estaremos programando um aplicativo que libera pontos em um programa de razão fixa (FR, do inglês *Fixed Ratio*) 10. Em um programa de FR o reforço ocorre após um número específico de respostas emitidas pelo organismo, independentemente do tempo gasto para

emiti-las (Ferster & Skinner, 1957). Em um FR 10 a consequência programada ocorre imediatamente após a 10ª resposta, contada desde o último reforço; depois da consequência liberada o contador volta a zero e recomeça a contagem.

Vamos acrescentar as linhas de código, abaixo, logo após o “Dim intContResp As integer”:

```
Dim intFRvalor As Integer = 10
```

```
Dim intPontos As Integer
```

A primeira declaração de variável acima (intFRvalor) armazenará o parâmetro do FR (i.e., o número de respostas exigido). A variável é iniciada, logo após sua declaração, com o valor 10. A segunda declaração de variável acima (intPontos) irá armazenar o número de pontos obtidos, que será exibido na Label1 do aplicativo.

Declarada estas variáveis, temos de alterar a parte do aplicativo que informa o que deve ser feito quando o botão [**Button1**] for acionado. Vamos acrescentar a seguinte linha de código, abaixo da linha de código “intContResp = intContResp + 1” no “Private Sub Button1_Click...”:

```
If intContResp = intFRvalor Then
```

Esta linha de código estabelece uma condição: se o número de respostas emitidas no botão [**Button1**] do aplicativo for igual a 10 (i.e., o valor do FR) então, algo será feito. O que o aplicativo deve fazer se esta condição for verdadeira? As três próximas linhas de código dizem ao aplicativo o que deve ser feito:

```

intPontos = intPontos + 1

Label1.Text = CStr(intPontos)

intContResp = 0

```

A primeira linha acrescenta um ponto ao contador de pontos (intPontos); a segunda linha exibe o valor de intPontos na Label1 do aplicativo e a terceira linha zera o contador de respostas novamente. Só uma coisa precisa ser acrescentada: o comando “End if” no final da condicional. A programação de um FR 10 está pronta! A Figura 8 exibe as linhas de código digitadas no VB.NET com as alterações para o FR 10. Acione o botão Iniciar na IDE do VB.NET (veja Figura 3) e teste seu aplicativo.

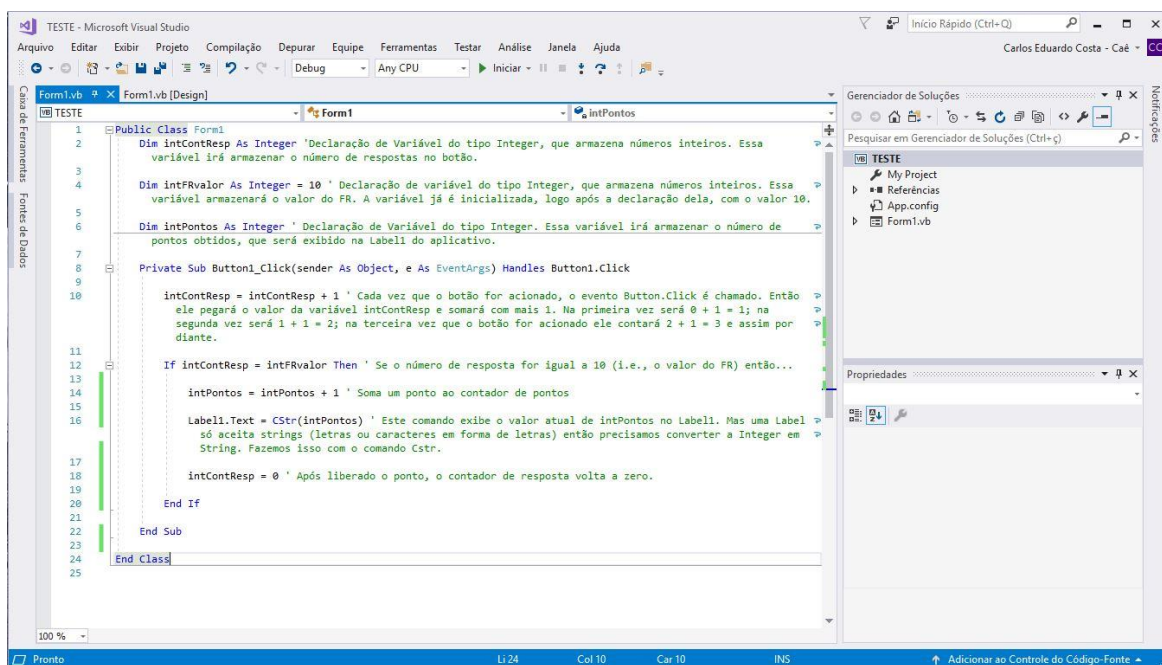


Figura 8. Linhas de código digitadas no VB.NET com a programação de um FR 10.

Claro que não é possível coletar dados com esse aplicativo. Seria necessário incluir os controles de tempo para registrar a duração da sessão e saber o momento em que cada evento (e.g., pressões ao botão, liberação de pontos etc.) ocorreu no tempo. Também seria necessário programar a saída de dados, i.e., como o aplicativo registraria cada evento para mostrar no final da sessão para que o pesquisador analise os dados. De qualquer modo, neste ponto, você deve ter uma noção bem melhor do que tinha antes de ler este capítulo, do que é e como programar em VB.NET.

Para aprender mais

Antes de indicar materiais com os quais é possível aprender mais sobre VB.NET, você deve explorar tudo que aprendeu aqui e pode avançar mais. Você pode explorar a janela “Propriedades” dos controles. Por exemplo, selecione o controle **Label1** no Form do seu aplicativo (ver Figura 5) e busque na janela “Propriedades” a propriedade “Fonts”. Ao clicar sobre essa propriedade note que aparece um pequeno botão com três pontos do lado direito da propriedade. Clique neste botão e abrirá uma janela para você selecionar o tipo de fonte, o estilo da fonte, o tamanho da fonte etc. Faça o mesmo com a propriedade “BackColor” e BorderStyle”. Veja que a medida que você altera as propriedades, visualmente o controle **Label1** é alterado no seu projeto. Uma das coisas mais importantes no aprendizado de uma linguagem de programação é você explorar os recursos do seu *software* de programação.

Você pode encontrar muita coisa boa na internet para aprender a programar em VB.NET. Também há ofertas de muitos livros em material impresso. Um bom início é começar com o

<https://blogs.msdn.microsoft.com/visualstudio/2017/03/07/announcing-visual-studio-2017-general-availability-and-more/>

Petroutsos, E. (2010). *Mastering Microsoft Visual Basic 2010*. Indianapolis: Wiley Publishing.



Introdução à linguagem de programação R aplicada à pesquisa e intervenção comportamental

Ricardo Fernandes Campos Junior |  |

Julia Zanetti Rocca |  |

O escopo da prática do analista do comportamento é bastante difícil de definir, uma vez que este profissional realiza intervenções em diversos contextos e com diferentes objetivos. Atualmente, o conjunto de propostas teóricas e de métodos é igualmente amplo, de modo que não é fácil delimitar pressupostos epistemológicos comuns para este profissional. Apesar disso, uma característica a respeito da qual provavelmente não haveria discordância é o fato de que seu trabalho deve se basear em dados. De modo que a coleta sistemática e análise constante de dados relevantes será parte integrante de praticamente qualquer intervenção (ver Vandenberghe, 2002).

Nesse sentido, a linguagem de programação R pode representar uma ferramenta importante para o trabalho na Análise do Comportamento. Isso porque trata-se de um ambiente de software livre especificamente criado para a análise de dados e sua representação gráfica. Ele foi constituído inicialmente no contexto científico, focalizando análise estatística e apresentação de gráficos para construção de artigos. Sua versão básica contém as funções⁴ necessárias para manipular os dados em uma base e aplicar os testes estatísticos mais frequentemente utilizados pela comunidade científica (R Core Team, 2016). Entretanto, por se tratar de um ambiente livre, os usuários podem constituir novas funções e agregar aquelas inicialmente disponíveis de modo que, atualmente, o uso do R não se restringe à academia.

⁴ Função: pedaço de código encapsulado em um objeto da classe “funções”. Cada função é criada com o objetivo de realizar uma tarefa específica. Uma função pode ou não ter argumentos de entrada, e pode ou não devolver um resultado, porém toda função realiza uma tarefa.

Por essas razões, o R permite a um analista do comportamento construir aplicativos do tipo Shiny⁵ (Chang et. al, 2017) que podem ser hospedados em servidores e utilizados em *tablets* ou celulares, de modo a possibilitar o registro de dados em situações reais. O protocolo de registros pode ser constituído de forma personalizada, contendo as informações necessárias para a prática de cada profissional. Sendo assim, durante o atendimento de uma criança com autismo, por exemplo, é possível registrar a frequência dos comportamentos alvo da intervenção na medida em que estes ocorrem. E, como esses aplicativos ficam disponíveis online, dados provenientes de mais de uma fonte ou observador podem ser salvos, mantendo a base de dados sempre atualizada.

Uma vez instalado, o R possui funções básicas que facilitam a manipulação, organização e indexação de dados, bem como permitem a seleção de subconjuntos dos mesmos a partir de diversos critérios. Também é possível realizar análises estatísticas variadas e representar os resultados em diferentes tipos de gráficos. Se determinados tipos de análise são realizados com frequência, estes podem ser programados em scripts⁶, de modo a permitir reaplicação em outros conjuntos de dados ou em subconjuntos dos dados iniciais. Desse modo, além da diversidade de recursos, é possível constituir funções personalizadas que realizem exatamente o que o profissional precisa.

Neste capítulo, será apresentado apenas um exemplo de utilização desses recursos. No caso, optamos por trabalhar com delineamento de sujeito único em sistema de linha de base múltipla entre comportamentos. Essas condições foram selecionadas uma vez que são típicas da área de Análise Aplicada do Comportamento e não são frequentemente utilizadas por outros profissionais. Na situação selecionada, assumimos que os dados foram registrados na situação da intervenção, por exemplo, durante os atendimentos de uma criança em uma clínica especializada, e digitados no formato de uma tabela simples, salva como arquivo .csv⁷.

⁵ Aplicativo Shiny: aplicativo web desenvolvido usando o pacote Shiny. Aplicativos web rodam em um servidor, e são utilizados pelos usuários por meio de um browser. Um exemplo de aplicativo Shiny pode ser visto no endereço: <https://shiny.rstudio.com/gallery/movie-explorer.html>

⁶ Script: sequência de comandos a serem executados pela linguagem que o interpreta. Assim como as funções, os scripts também são criados para executar tarefas. Enquanto as funções são normalmente criadas para executar uma única tarefa, scripts normalmente são escritos com o objetivo de executar várias tarefas em uma sequência específica.

⁷ Do inglês, *comma separetad values*, ou seja, valores separados por vírgula. Este formato de arquivo é universal, de modo que os dados podem ser salvos e abertos por outros programas de manipulação de dados Excel e

A proposta é que, ao final da leitura desse capítulo, o leitor consiga construir modelos de análises simples e recursos gráficos para situações de intervenção em sistema de linha de base múltipla. O exemplo trabalha com delineamento de sujeito único, mas poderia ser facilmente aplicado a situações com vários participantes ou com grupos. As instruções serão organizadas de forma a separar texto descritivo, os comandos utilizados na programação e os resultados verificados a partir destes. Para isto, as partes de textos do capítulo estarão com a fonte na cor preta, os comandos estarão em azul, e o resultado estará em vermelho.

Começando a trabalhar com o R - Instalação

O programa pode ser baixado no site <https://cran.r-project.org/> e a instalação é simples para qualquer sistema operacional. Recomenda-se também a instalação da IDE RStudio no site <https://www.rstudio.com/products/rstudio/download/>, a qual possui várias ferramentas para edição de scripts, apresentação de resultados e visualização de gráficos que facilitam o trabalho com o R. Para facilitar a compreensão do conteúdo, recomenda-se que o leitor instale o R e o RStudio e acompanhe as operações listadas.

Interface

A Figura 1 mostra a interface do RStudio quando ele é aberto pela primeira vez. As abas foram numeradas para que possamos descrevê-las separadamente. Cada uma dessas abas possui funções e botões próprios para lidar com suas devidas finalidades. A seguir faremos uma breve descrição de cada uma das abas:

1. *Console*: aqui os comandos serão executados. Tudo que for escrito no console será avaliado como um comando em R e executado.

2. *Environment*: esta aba mostra todos os objetos salvos pelo usuário. Ela é ainda organizada por tipo de objeto, podendo eles ser vetores⁸, matrizes ou tabelas, ou mesmo funções definidas pelo usuário.
3. *History*: um histórico dos comandos utilizados pelo usuário.
4. *Files*: mostra os arquivos presentes no atual diretório de trabalho.
5. *Plots*: aqui serão mostrados todos os gráficos produzidos pelo usuário.
6. *Packages*: mostra uma lista de pacotes, selecionando aqueles que estão atualmente carregados.
7. *Help*: apresenta a página de ajuda requisitada.
8. *Viewer*: mostra uma pré-visualização de objetos do tipo web produzidos pelo usuário.

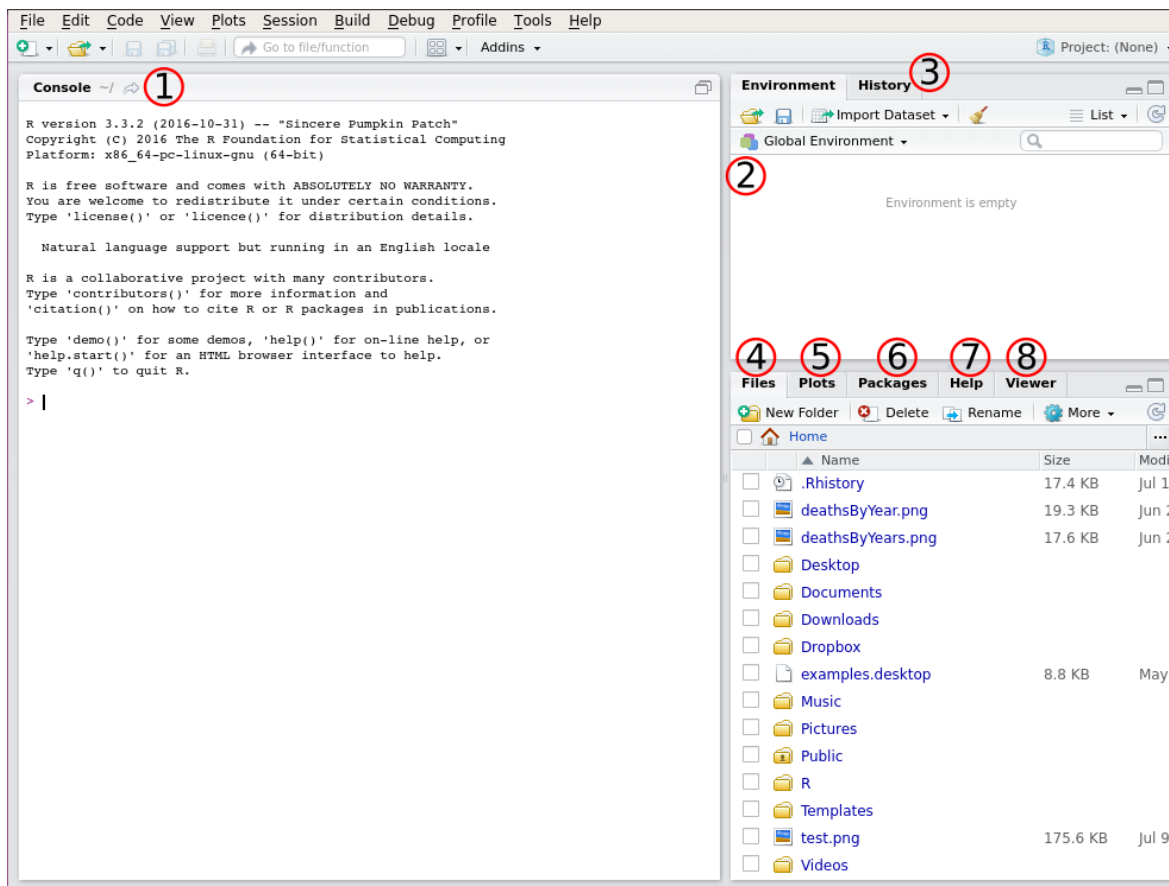


Figura 1. Visualização inicial da interface do RStudio.

⁸ Um vetor é uma sequência de elementos de dados que possuem o mesmo tipo ou classe. Ex: sequência de números, sequência de caracteres, sequência de nomes, etc.

Primeiros passos

Antes de iniciar o trabalho, crie uma pasta no seu computador na qual serão salvas a base de dados e os scripts que serão construídos. Essa pasta deve ser salva diretamente no `c:\`, caso você utilize Windows ou em `/home/usuário/` caso você utilize Linux. No presente exemplo, a pasta criada será denominada “dados” e estará salva no endereço “/home/ricardo/dados”.

Agora abra o RStudio para iniciar um novo script. Clique `File→New File → R Script`. Uma nova aba será aberta. Esta é a aba de edição, na qual os scripts podem ser escritos e editados para, posteriormente, serem avaliados e executados no console.

Usualmente, quando iniciamos um novo script, temos que modificar o diretório de trabalho para aquele que contém os arquivos que serão utilizados no projeto, ou seja, as tabelas de dados a serem analisados. O comando `setwd()` tem como função modificar o diretório de trabalho. É necessário, portanto, inserir a informação referente à localização do novo diretório de trabalho dentro dos parênteses. No presente caso, vamos modificar para a pasta `/home/ricardo/dados`⁹:

```
> setwd("/home/ricardo/dados")
```

O comando acima pode ser escrito na aba de script e passado para o console com o atalho `CTRL+ENTER`. Desta forma, a mudança de diretório é mantida no script, e o script todo pode ser re-executado quando necessário. Para executar os comandos contidos em todas as linhas da página de edição do script, basta clicar no botão *source* no canto superior direito da aba de edição de script recém criada no RStudio. Note que o comando digitado possui o símbolo ‘>’. Este é apenas o símbolo utilizado dentro do Rstudio para mostrar os códigos que foram digitados.

Para saber se o endereço do diretório foi, de fato, modificado, podemos usar o comando `getwd()`, seguido de `CTRL+ENTER`.

⁹ No *Windows*, as barras que separam entre pastas são inclinadas para a esquerda (`\`), mas o caminho para as mesmas dentro do R deve utilizar as barras inclinadas para a direita (`/`).


```
> getwd()
[1] "/home/ricardo/dados"
```

No caso da função `getwd()`, não é necessário inserir argumentos dentro dos parênteses, uma vez que ela visa apresentar o diretório atual, por isso é executada com os parênteses vazios.

Importante observar que, quando necessário, o argumento – no caso, o endereço do diretório – deve ser introduzido dentro dos parênteses **entre aspas**, uma vez que essa informação é externa ao R. Funções e objetos, por outro lado, podem ser introduzidas diretamente. Nesse sentido, para não ter que escrever o caminho do diretório sempre que for acessá-lo, é possível criar um objeto de tipo caractere que armazene esses dados. Após criado, este pode ser utilizado em substituição ao endereço do diretório dentro da função `setwd()` sem as aspas, como a seguir;

```
> meuCaminho = "/home/ricardo/dados"
> setwd(meuCaminho)
```

Conhecendo as funções, seus argumentos e modos de utilizá-los

O exemplo aqui apresentado é muito breve e apresenta apenas algumas poucas funções disponíveis no R. Para compreendê-las melhor, é importante consultar a ajuda. Cada função possui uma página de ajuda, a qual pode ser requisitada pelo comando `help(nome)`. A página de ajuda apresenta para que serve a função, quais são os argumentos necessários, além de fornecer exemplos reproduzíveis e informações detalhadas de cada um dos argumentos e valores resultantes da utilização da função.

Outras funções úteis na hora de pedir ajudar são `args(NomeDaFuncao)` e `example(NomeDaFuncao)`, as quais mostram os argumentos necessários/possíveis para cada função e exemplos para o seu uso, respectivamente.

```

> args(getwd)
function ()
NULL
> example(args)
args(ls)
function (name, pos = -1L, envir = as.environment(pos), all.names = FALSE, pattern,
  sorted = TRUE)10

```

Como mencionado, a função `getwd()` não possui argumento, por esta razão ela é apresentada sem entradas dentro dos parênteses de ‘function’. Por outro lado, a função `ls()`, a qual é usada para listar os objetos salvos na área de trabalho, possui 6 argumentos, os quais são separados por vírgula, ordenados, e nomeados, como mostrado pelo resultado de `example(args)`.

Todos os argumentos de funções no R possuem um nome próprio. Contudo, há duas formas de inserir argumentos em uma função no R: pelo nome ou de acordo com a ordem. No primeiro caso, é necessário conhecer o nome de cada argumento, e isso pode ser feito por meio da função `args()`, ou consultando a ajuda. Outra opção é inserir seus valores na ordem pré-determinada pela função, sem mencionar o nome. Nesse caso, a ordem também pode ser consultada via `args()` ou na ajuda. Caso quiséssemos, por exemplo, usar a função `setwd()` utilizando o nome do seu primeiro argumento, ao invés de indicá-lo pela ordem do mesmo como fizemos anteriormente, poderíamos fazê-lo da seguinte forma:

```

> setwd(dir=meuCaminho)

```

Gerando e manipulando a base de dados

No presente caso vamos gerar um conjunto de dados fictícios para permitir sua utilização pelas demais funções e a formulação do script para sua representação gráfica. Para isso, será utilizada a função `rnorm()`. Essa função gera n valores aleatórios com média m e desvio padrão s . A sintaxe básica desta é, portanto, `rnorm(n,m,s)`.

¹⁰ Para fins de simplicidade, é mostrado apenas um dos exemplos da função ‘args’. Usualmente são retornados vários exemplos para cada função requisitada.

A título de ilustração, iremos trabalhar com uma intervenção com o objetivo de ensinar três habilidades para a criança atendida utilizando determinado procedimento. Para garantir que o procedimento é a variável independente relevante para a mudança de comportamento, será realizado delineamento de linha de base múltipla entre comportamentos. Isso significa que o procedimento será introduzido em momentos diferentes da intervenção para cada uma das habilidades. Sendo assim, haverá uma fase inicial de registro do comportamento composta por dez sessões. A partir da verificação de estabilidade da taxa de respostas, o procedimento de ensino será introduzido para a habilidade 1 na sessão 10. A habilidade 2 será ensinada a partir da sessão 15 e a habilidade três a partir da sessão 20. Serão 30 sessões no total.

As habilidades serão medidas em relação à frequência de respostas em sessões de 50 minutos. A representação gráfica deverá apresentar o total de respostas sessão a sessão para cada habilidade em gráficos separados. O ponto em que a intervenção foi iniciada deverá ser marcado por uma reta nos três gráficos. Em princípio, se o procedimento for capaz de alterar a incidência dos comportamentos registrados, esta mudança será verificada por inspeção visual da curva no gráfico, sem necessidade de análises estatísticas adicionais. Se, para algum caso, a inspeção visual não for suficiente para decidir a respeito dos efeitos causados pela intervenção, será utilizada análise estatística.

Estamos, então, simulando o registro de dados de frequência para os três habilidades que serão ensinados à criança em trinta sessões. Sendo assim, precisaremos de trinta valores para cada habilidade, sendo que, a partir da intervenção, é esperado que a frequência dos comportamentos aumente. Desse modo, para a habilidade 1, haverá 10 valores da fase de linha de base, com média 3, e 20 para a fase de intervenção, mais altos, com média 9. Para a habilidade 2, haverá 15 valores correspondentes à fase de linha de base (média 5) e mais 15 para a fase de intervenção (média 7). E, finalmente, para a habilidade 3, haverá 20 valores na linha de base (média 2) e 10 na intervenção (média 6). A função `rnorm()` tem que gerar seis conjuntos de valores, sendo três para as habilidades 1, 2 e 3 na fase de linha de base e três para a fase de intervenção.

Deste modo, será necessário rodar `rnorm()` seis vezes, iremos então utilizar uma segunda função que irá aplicar n , m , e s repetidas vezes em `rnorm()`, e nos devolver todos os valores

gerados como uma lista. A função `mapply()` atende à essas necessidades para qualquer função `FUN` com os argumentos seus necessários. Esta função funciona como um fluxo, a qual vai aplicar uma função `FUN` repetidas vezes, utilizando os argumentos fornecidos a `mapply()` como argumentos para `FUN`, entregando à ela um valor de cada argumento de cada vez. Para o caso da função `rnorm()`, estaremos entregando 6 n , 6 m e 6 s . Então, no primeiro ciclo, `mapply()` irá aplicar o primeiro valor de n , a primeira média m e o primeiro desvio padrão s fornecidos em `rnorm()`. Dessa forma, `mapply()` irá rodar sequencialmente até que todos os argumentos fornecidos tenham sido utilizados. As linhas a seguir executam esta tarefa.

```
(1) > ns=c(10,20,15,15,20,10)
(2) > medias=c(3,9,5,7,2,6)
(3) > desvios=c(1,1,1,2,0.5,1)
(4) > frequencia = mapply(rnorm,ns,medias,desvios)
(5) > frequencia = unlist(frequencia)
(6) > frequencia = round(frequencia)
```

1 a 3: salvam respectivamente n , m e s cada qual em um objeto (`ns`, `medias` e `desvios`, respectivamente), além de demonstrar a função `c()`, a qual é utilizada para concatenar (juntar) valores e/ou objetos no R.

4: usa a função `mapply()` para aplicar `rnorm()` sequencialmente nos argumentos fornecidos e retorna uma lista, a qual possui o resultado separado para cada ciclo de `mapply()`. Convidamos o leitor a visualizar este resultado para compreender a estrutura do objeto de tipo lista antes de seguir com os demais comandos.

5: usa `unlist()` para concatenar os resultados em um objeto de tipo vetor. Convidamos o leitor a visualizar este resultado e comparar com o tipo anterior.

6: usa `round()` para arredondar os valores para valores inteiros.

O resultado dos comandos nos quais usamos `mapply()` é um vetor com 90 valores que representam a frequência de respostas das três habilidades que estão sendo ensinadas ao participante. Para transformar nossos valores em uma tabela com as informações do nome da habilidade e a condição do procedimento, iremos armazená-los e salvar no disco. Para isso,

vamos criar os objetos que possuam estas informações, e entregá-los à função `data.frame()` a qual vai transformá-los em tabela. Nas últimas duas linhas utilizaremos as funções `head()` e `tail()` para mostrar, respectivamente, o topo e o final da nossa tabela.

```
(1) > habilidades=c(rep("Mando1",30),rep("Mando2",30),rep("Mando3",30))
(2) > condicao=c(rep("Lbase",10), rep("Interv",20), rep("Lbase",15),
               rep("Interv",15), rep("Lbase",20), rep("Interv",10))
(3) > dados=data.frame(habilidades,frequencia,condicao)
```

```
(4) > head(dados)
```

	habilidades	frequencia	condicao
1	Mando1	3	LBase
2	Mando1	3	LBase
3	Mando1	3	LBase
4	Mando1	2	LBase
5	Mando1	3	LBase
6	Mando1	3	LBase

```
(5) > tail (dados)
```

	habilidades	frequencia	condição
85	Mando3	8	Interv
86	Mando3	6	Interv
87	Mando3	7	Interv
88	Mando3	6	Interv
89	Mando3	6	Interv
90	Mando3	5	Interv

As duas primeiras linhas usam a função `rep(obj,n)`, a qual é usada para repetir `obj` por `n` vezes. Estes são concatenados para formar os vetores necessários para cada coluna de nossa tabela. Além de poder visualizar nossa tabela com as funções mostradas, podemos utilizar a posição dos mesmos com o padrão `dados[linha,coluna]` tanto para visualizar quanto para alterar. Caso queira visualizar todos os objetos de alguma linha ou coluna específica, basta deixar em branco seu índice correspondente. Além disso, as colunas podem ser requisitadas pelos seus nomes. Rode os exemplos abaixo no seu computador e observe os resultados.

```

> dados[2,1]
> dados[2,1] = 4
> dados [2,1]
> dados[2,c(1,3)]
> dados[2,]
> dados[,3]
> dados[c(1,3,5),2:3]
> dados$habilidades
> dados$condicao
> dados$novaColuna = c(rep(1,45),rep(2,45))
> head(dados)
> dados = dados[,1:3]
> head(dados)

```

Outras formas de selecionar partes específicas dos dados serão usadas mais adiante neste capítulo.

Agora que os dados foram gerados, estes serão salvos no disco do computador, de forma que possam ser abertos no R como se estes tivessem sido coletados e salvos em um editor de planilhas, como o Excel. Nós, entretanto, utilizaremos um formato específico para salvar os dados, o formato ‘.csv’. Os comandos a seguir salva os dados no nosso diretório de trabalho atual, e nos mostra o arquivo salvo no computador.

```

> write.csv(dados,"dados.csv")
> dir()
[1] "dados.csv"

```

A função `write.csv()` é usada para salvar os dados e, neste exemplo, são usados os dois primeiros argumentos `x` e `file`, os quais indicam, respectivamente, o objeto a ser salvo e o caminho, juntamente com nome e extensão do arquivo a ser salvo. Como os argumentos usados são, respectivamente, o primeiro e o segundo argumentos da função, não é necessário usar seus nomes. Supondo que a posição destes argumentos fosse outra, teríamos que usar os seus nomes para fornecer estas informações, ou seja:

```
> write.csv(x=dados, file="dados.csv")
```

Abrindo e analisando os dados

Agora que os dados foram gerados e salvos no computador, eles podem ser abertos como qualquer outro conjunto de dados. Será utilizada a função `read.csv()` para abrir arquivos salvos no formato ‘csv’, como a seguir:

```
> read.csv("dados.csv")
```

Caso os dados estejam em outra pasta, poderíamos fornecer o caminho completo:

```
> read.csv("/home/ricardo/livro/dados.csv")
```

A fim de verificar se nossa tabela foi aberta corretamente, podemos utilizar algumas das funções já apresentadas anteriormente, ou seja, `head()` e `tail()`, além de outras funções que fornecem informações adicionais. A função `str()`, por exemplo, nos fornece informações sobre a quantidade de dados, e os tipos de vetores dentro de nossa tabela.

```
> str(dados)
'data.frame':90 obs. of 3 variables:
 $ habilidades: Factor w/ 3 levels "Mando1","Mando2",...: 1 1 1 1 1 1 1 1 1 1 ...
 $ frequencia : num 3 3 3 4 2 3 2 4 3 4 ...
 $ condicao : Factor w/ 2 levels "Interv","LBase": 2 2 2 2 2 2 2 2 2 2 ...
```

Note que estão especificados os tipos de vetores na frente dos nomes das colunas: `num` = numérico e `Factor` = fatores. Os fatores são tipos especiais, contudo para os fins deste capítulo eles podem ser usados como caracteres. As linhas a seguir transformam as colunas do tipo fatores para o tipo caracteres.

```
> dados$habilidades=as.character(dados$habilidades)
> dados$condicao=as.character(dados$condicao)
```

Após gerar nossos dados simulados, vamos utilizar algumas funções básicas do R para descrever os dados estatisticamente. Para este propósito, utilizaremos uma nova tabela contendo apenas a habilidade “Mando1”. O leitor pode, então, fazer o mesmo para as demais habilidades.

```
> mando1 = dados[dados[,1]=="Mando1",]
```

A linha acima cria um novo objeto com todas as linhas que contém “Mando1” na coluna 1, e seleciona todas as colunas. A forma de se ler é a seguinte “selecione do objeto ‘dados’ todas as linhas onde ‘dados[,1]’ forem iguais a ‘Mando1’, e todas as colunas deste objeto”. Observe que o símbolo de comparação lógica de igualdade em R é `==`, enquanto o símbolo `=` é reservado para atribuir um valor a um objeto. Com o novo objeto criado, vamos descrever os dados.

```
> summary(mando1[,2])
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
	2.0	4.0	8.0	6.8 9.0	10.0

```
> sd(mando1[,2])
```

2.771779

A função `summary()` nos fornece, respectivamente, valor mínimo, primeiro quartil, mediana, média, terceiro quartil e valor máximo. A função `sd()` no dá desvio padrão. Outras funções que podemos usar para descrever nossos dados são `mean()`, `median()`, `min()`, `max()`, `var()`, `sum()`, entre uma infinidade de outras.

Plotando os dados

Após descrever os dados, vamos usar algumas funcionalidades gráficas do R para criar gráficos com eles (Figura 2). Inicialmente, serão criados os gráficos de dispersão para a habilidade “Mando3”, como exemplo. Mais uma vez é sugerido ao leitor que reproduza os códigos a seguir para as demais habilidade.


```
(1) > plot(dados[,2][dados[,1]=="Mando3"], main="Habilidade: Mando3", xlab="Sessão",
ylab="Frequência")
(2) > linha=which(dados[,3][dados[,1]=="Mando3"]=="Interv")[1]-0.5
(3) > abline(v=linha,lty="dashed")
```

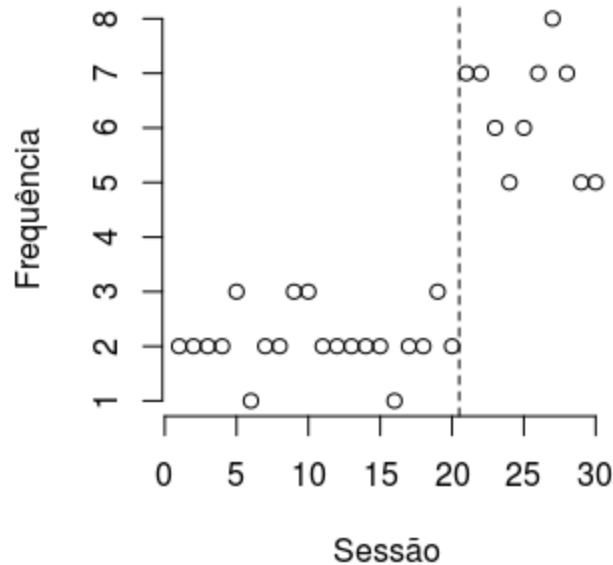


Figura 2. Gráfico de dispersão da habilidade ‘Mando3’.

A primeira linha apresenta a função `plot()`, que é a função básica de gráficos no R. Ela possui muitos argumentos, mas utilizamos apenas 4 nesta linha, os quais fornecem, respectivamente, os dados a serem plotados, o título, o rótulo do eixo x, e o rótulo do eixo y. É interessante que o leitor leia `help(plot)` e `help(par)`, a última função será usada mais adiante neste capítulo.

Na segunda linha, há vários comandos que nos ajudam a encontrar o ponto em que a intervenção começou. As partes dos mesmos serão explicadas a seguir:

‘`linha =`’ esta parte cria um novo objeto para armazenar o valor.

‘`which()`’ esta função nos retorna os índices dos valores de um vetor para os quais a afirmação lógica usada é verdadeira.

‘`dados[,3]`’ esta parte indica a partir de onde a parte seguinte irá fazer a seleção

‘`[dados[,1]=="Mando3"]`’ esta parte seleciona de `dados[,3]` apenas aquelas linhas que possuem ‘Mando3’ na coluna 1.

`'==Interv'` esta parte faz um teste lógico para todos os valores resultantes do teste anterior, e retorna TRUE (verdadeiro) para os casos em que a coluna 3 sejam de intervenção, de forma que a função `which()` possa identificá-los.

`'[1]-0.5'` essa parte foi introduzida para identificar onde será inserida a linha tracejada que separa linha de base e intervenção para cada habilidade. Isso pode ser feito pela seleção do primeiro valor do vetor de 10 valores resultantes dos códigos anteriores. A linha, entretanto, não deve passar em cima da sessão, mas entre estas, por isso, faz-se necessário diminuir 0,5 deste. Como nós sabemos que anterior à primeira sessão pós intervenção foi apresentado o vídeo, nós colocamos a linha entre as sessões com e sem intervenção.

Se após esta explicação o leitor ainda tiver dificuldade em compreender o que foi feito, rode as partes a seguir uma a uma e observe os resultados.

```
dados[,3]
dados[,3][dados[,1]=="Mando3"]
dados[,3][dados[,1]=="Mando3"]=="Interv"
which(dados[,3][dados[,1]=="Mando3"]=="Interv")
which(dados[,3][dados[,1]=="Mando3"]=="Interv")[1]
which(dados[,3][dados[,1]=="Mando3"]=="Interv")[1]-0.5
```

Finalmente, a função `abline()` na terceira linha usa o argumento `lty` para plotar uma linha de traços vertical, como comandado pelo argumento `v`.

Contudo, frequentemente analistas do comportamento utilizam gráficos de linhas (Figura 3). Para isto além de apresentar o ciclo `for()`, algumas funções úteis na manipulação de dados e argumentos adicionais são as funções `plot()` e `par()`.

```
(1) > par(mfrow=c(3,1),mar=c(2,2,1.5,0))
(2) > hab = unique(dados$habilidades)
(3) > tam = length(hab)
(4) > for(i in 1:tam)
(5) + {
(6) +   plot(dados$frequencia[dados$habilidades==hab[i]],type="l",bty="n",ylim =
      range(dados$frequencia))
(7) +   linha=which(dados[,3][dados[,1]==hab[i]]=="Interv")[1]-0.5
(8) +   abline(v=linha,lty="dotted",lwd=2)
(9) + }
```

(1) A função `par()` é usada para definir dois parâmetros da figura. O parâmetro `mfrow` diz, respectivamente, quantas linhas e quantas colunas serão criadas de área da figura. O parâmetro `mar` especifica a distância das margens da cada área gráfica, sendo elas: embaixo, à esquerda, em cima e à direita, respectivamente, ou seja: `mar=c(2,2,1.5,0)`. A presença do parâmetro `mar` fez com que os nomes dos eixos x e y fossem plotados fora da área da figura, portanto, não estão visíveis.

(2) A segunda linha usa a função `unique()`, a qual salva no objeto `hab` apenas os valores únicos do vetor `dados$habilidades`, ou seja, as três habilidades treinadas no nosso projeto. Convidamos o leitor a visualizar o conteúdo do objeto `hab` antes de prosseguir.

(3) A função `length()` é utilizada para pedir a quantidade de itens do objeto de tipo vetor `hab`, e salva em um novo objeto, `tam`. Note que para objetos que possuem mais de uma dimensão, como é o caso de nossa tabela dados, a qual possui duas dimensões (linha e coluna), a função usada para ver seu tamanho é `dim()`. Convidamos o leitor a visualizar o conteúdo do objeto `tam` antes de prosseguir.

(4) A quarta linha introduz o controle de fluxo `for()`. Todo o código dentro dele (linhas 5 a 9) será rodado várias vezes da forma como está, ao mesmo tempo em que o valor do objeto `i` se modifica conforme o fluxo avança. Para cada iteração de `for()`, o objeto `i` irá assumir um dos valores do vetor que está após o conector `in`, ou seja, cada um dos valores únicos no vetor `hab`, e o fluxo irá ocorrer até que `i` tenha assumido todos os valores. Convidamos o leitor a visualizar o conteúdo do vetor '`1:tam`' antes de prosseguir.

(5) e (9) As linhas 5 e 9 abrem e fecham o fluxo `for()`. Todo o código que estiver dentro das chaves do fluxo será rodado o número de vezes especificado após o conector `in`. Dessa forma, as linhas 5 a 9 serão executadas nessa ordem 3 vezes. Cada uma dessas vezes, `i` assumirá um valor diferente, e este valor será usado para mudar como o código dentro do fluxo será executado.

(6) A função `plot()` é utilizada para plotar os dados com os parâmetros `type`, `bty` e `ylim`, os quais especificam, respectivamente, o tipo de gráfico a ser plotado, o tipo de caixa do gráfico que será usada e os limites inferior e superior do eixo y. No argumento `ylim` é usada a função `range()`,

a qual fornece o menor e maior valor de um vetor. Ainda na linha 6, pode-se observar como o valor de *i* é usado para modificar o que será plotado. O código `dados$frequencia[dados$habilidades==hab[i]]` diz que serão plotados todos os valores de `dados$frequencia`, mas apenas aqueles cuja valor de `dados$habilidades` sejam iguais a `hab[i]`. Note que, para cada vez que o fluxo é executado, *i* irá assumir os valores 1, 2 ou 3, nesta ordem. Sendo assim, os valores de `dados$habilidades` selecionados serão ‘Mando1’, ‘Mando2’ e ‘Mando3’, cada qual em seu fluxo. Esta mesma lógica é usada na linha a seguir para selecionar a posição que mostra no gráfico a linha de intervenção.

(7) A linha 7 cria o objeto que armazena a posição da linha que separa a fase de linha de base da fase de intervenção utilizando a mesma lógica do gráfico anterior, porém usando o objeto do fluxo *i* para indicar qual habilidade está sendo plotada.

(8) A linha 8 também é usada da mesma forma que anteriormente, mas apresenta o argumento *lty* com o valor ‘*dotted*’, ou seja, linhas pontilhadas, e apresenta o argumento *lwd* para indicar a espessura da linha a ser plotada.

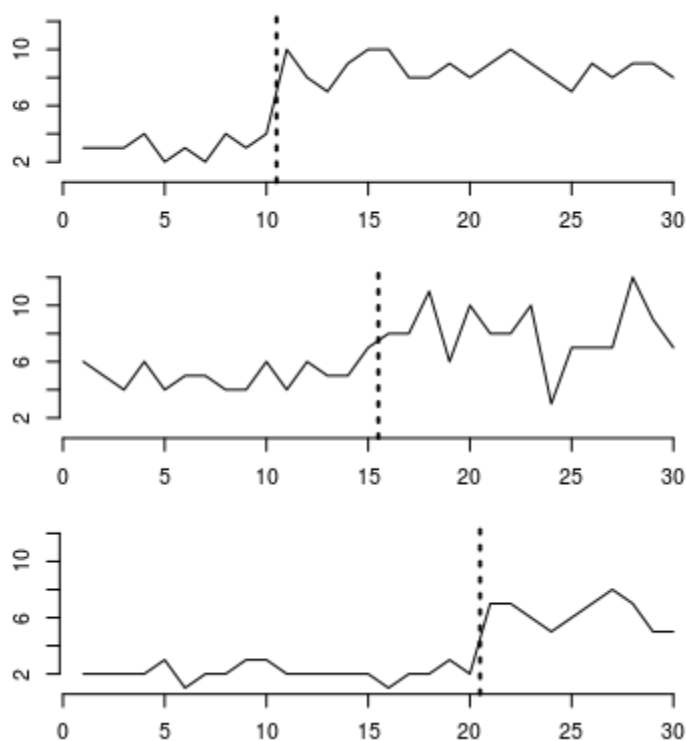


Figura 3. Gráfico de linhas das três habilidades desenvolvidas em delineamento de linha de base múltipla.

Conforme pode se observar na Figura 3, o desempenho da criança na habilidade ‘Mando2’ apresenta pouca variação média ao compararmos a fase de linha de base e a de intervenção. Por esta razão, as linhas de código a seguir irão utilizar alguns recursos gráficos e estatísticos para analisar essa diferença, começando pelo *boxplot()* (Figura 4). Para isso um novo objeto irá salvar apenas os valores de interesse, ou seja, aqueles da segunda habilidade, e este novo objeto será utilizado daqui em diante.

```
(1) > par(mfrow=c(1,1),bty="n")
(2) > mando2=dados[dados$habilidades=="Mando2",]
(3) > boxplot(mando2$frequencia~mando2$condicao)
```

Note que a linha 1 utiliza novamente o argumento *mfrow* da função *par()*. Isto porque, caso isto não fosse feito, o R continuaria plotando 3 gráficos por figura como havíamos comandado que o fizesse anteriormente. É também dentro de *par()* que chamamos novamente o argumento *bty*, ao contrário de chamá-la dentro de *plot()* ou de *boxplot()*. Outra coisa a se notar é a permanência das margens como havíamos especificado na Figura 3. Este padrão mostra a diferença entre as funções *par()* e as demais funções do tipo *plot()*: enquanto as alterações feitas pela primeira são permanentes até que o R seja fechado, as alterações feitas na segunda são usadas apenas uma única vez.

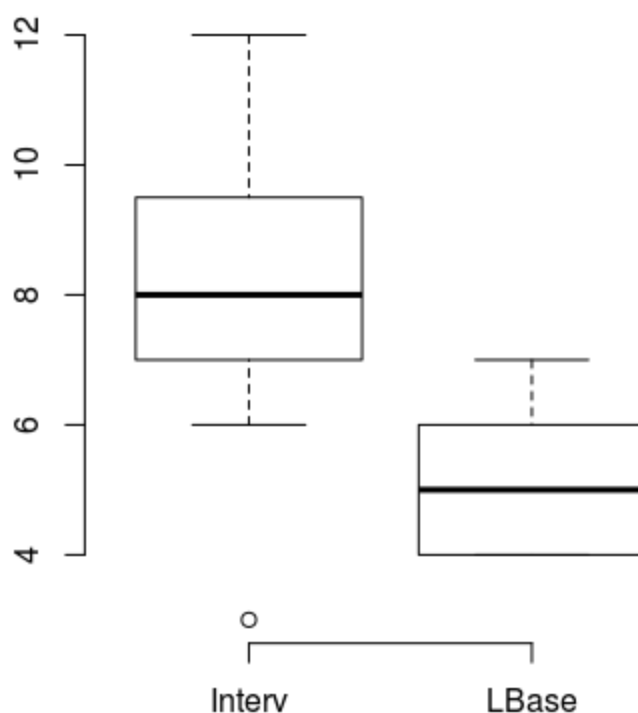


Figura 4. Boxplot da habilidade 'Mando2'.

Em seguida vamos resumir a distribuição dos dados por condição experimental utilizando a função `aggregate()` junto com a função `summary()`, a qual vai nos dar o resumo dos valores que observamos no boxplot. A função `aggregate()` separa os dados do argumento `x` em subconjuntos determinados pelo argumento `by`, o vetor que determina a separação dos subgrupos. A partir disso, especifica-se a função que será utilizada no argumento `FUN`. Nesse caso, iremos utilizar a função `summary()`, a qual devolve os valores de média, mediana e quartis, que são os mesmos apresentados no boxplot.

```
> aggregate(x=mando2$frequencia,by=list(mando2$condicao),FUN=summary)
  Group.1 x.Min.  x.1st  Qu. x.  Median x.Mean  x.3rd Qu.  x.Max.
1 Interv  2.000   5.000  6.000    6.267   7.500   10.000
2 LBase   3.000   4.500  5.000    5.067   6.000    7.000
```

Aplicando testes estatísticos

Os comandos a seguir irão aplicar análise de variância (ANOVA) para verificar se a diferença observada antes e depois da intervenção pode ser explicada pela intervenção. Note que foram usadas médias próximas, porém diferentes para as duas condições, sendo que para a condição após a intervenção o desvio padrão é mais alto. Como os dados gerados são aleatórios, o resultado da análise, assim como os dados apresentados nos gráficos, serão diferentes para cada vez que estes são gerados, sendo possível observarmos resultados significativos ou não significativos para diferentes conjuntos de dados gerados com os mesmos parâmetros. As linhas a seguir fazem o teste para o conjunto de dados da segunda habilidade, como feito anteriormente com o boxplot:

```
(1) > model=aov(mando2$frequencia~mando2$condicao)
(2) > summary(model)
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
mando2\$condicao	1	56.03	56.03	19.74	0.000127 ***
Residuals	28	79.47	2.84		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(1) Salva no objeto '*model*' o modelo estatístico criado pelo função *aov()*. Nesta função, o argumento é uma fórmula, na qual a primeira parte temos a variável dependente ou de resposta, a qual será explicada pela segunda parte, que é a variável independente ou variável explicativa. Estas partes são separadas pelo símbolo ~, assim como utilizado no boxplot.

(2) A segunda linha usa a função *summary* para pedir o resultado resumido do modelo de ANOVA. Este resultado é uma tabela que mostra o grau de liberdade (Df), a soma dos desvios quadrados (Sum Sq), a média dos desvios quadrados (Mean Sd), o valor do teste estatístico F (F value) e a probabilidade de significância associada ao valor de F (Pr>F). Para mais detalhes sobre o teste ou sobre o resultado, o leitor pode visualizar *help(aov)*.

Com o objetivo de verificar a quebra de alguma premissa a respeito da distribuição dos resíduos da análise, a função `plot()` pode ser aplicada diretamente ao nosso modelo com o comando `plot(model)`. Com isso, serão gerados alguns gráficos diagnósticos, cujos detalhes podem ser visualizados em `help(plot.lm)`. Podemos também acessar diretamente os resíduos do modelo com o comando `residuals(model)`, ou mesmo plotar os mesmos diretamente:

```
> plot(residuals(model))
```

Ainda com o objetivo diagnóstico sobre a distribuição dos dados, as funções `qqnorm()` e `qqplot()` podem ser utilizadas. Mais uma vez, convidamos o leitor a visualizar o `help()` dessas funções para mais detalhes sobre o resultado.

```
> qqnorm(mando2$frequencia)
> qqplot(mando2$frequencia[condicao=="Interv"], mando2$frequencia[condicao=="LBase"])
```

Leituras adicionais básicas e avançadas sobre R

A linguagem R é usada mundialmente por um público extremamente variado, desde biólogos, estatísticos, cientistas de dados, economistas, operadores da bolsa de valores, entre outros, e a crescente demanda de pessoas capazes de lidar com o aumento da quantidade de dados em todas as áreas têm feito com que este público cresça ainda mais. É provavelmente devido a este volume e diversidade da comunidade R que ela possui uma infinidade de pacotes desenvolvidos para atender às demandas de diversas áreas. Para instalar pacotes adicionais no R, basta utilizar a função `install.packages()` com o nome do pacote desejado como argumento. Para testar, convidamos o leitor a instalar o pacote ‘swirl’ (Kross et al., 2017), o qual é usado para ensinar a programar em R diretamente no console. Depois de instalado, o pacote pode ser carregado com o comando `require()`, como as seguir:

```
> install.packages("swirl")
> require("swirl")
```


Ainda devido ao volume e diversidade da comunidade R, existem diversas formas de conseguir informações e aprender mais sobre a linguagem. Usualmente, os próprios pacotes adicionais instalados possuem tutoriais próprios, conhecidos como “vignette”, que ensinam como utilizar as suas principais funções. Estes pacotes podem ser encontrados na página principal da comunidade em <https://cran.r-project.org/>. Abaixo serão listadas algumas fontes adicionais de informações para que o leitor possa aprofundar seus conhecimentos na linguagem.

1. www.stackoverflow.com: um fórum em que os membros podem postar perguntas para resolver problemas de programação ou mesmo pedir ajuda de como programar partes específicas de seu projeto. O *stackoverflow* já possui uma infinidade de postagens de conteúdo de erros, sendo fácil que o leitor possa resolver suas dúvidas sem mesmo ter que postá-las no site.
2. www.r-bloggers.com: este site reúne uma infinidade de bloggers, os quais postam novidades diariamente sobre pacotes, tutoriais, entre outros. R-bloggers é uma excelente fonte para aprender conteúdo avançado tanto no nível de programação, como no de análise de dados ou produção de gráficos.
3. <http://ecologia.ib.usp.br/bie5782/doku.php>: Site da disciplina de Introdução à Linguagem R do departamento de Ecologia da Universidade de São Paulo. O site reúne diversas apostilas, tutoriais, fórum, e materiais produzidos pelas turmas de anos anteriores da disciplina, e grande parte desse material é em português.
4. <https://www.coursera.org/learn/r-programming>: Site do curso de programação em R do Coursera, onde o aluno pode assistir aulas de graça. Em sua versão paga, o site ainda permite aos alunos fazer atividades e receber notas, além de obter certificado de conclusão.

Referências

- Chang, W.; Cheng, J.; Allaire, J. J.; Xie, Y. & McPherson. J. (2017). *Shiny: Web Application Framework for R*. R package version 1.0.3. Recuperado de: <https://CRAN.R-project.org/package=shiny>
- Kross, S.; Carchedi, N.; Bauer, B. & Grdina, G. (2017). *Swirl: Learn R, in R*. R package version 2.4.3. Recuperado de: <https://CRAN.R-project.org/package=swirl>
- R Core Team (2016). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. Recuperado de: <https://www.R-project.org/>
- Vandenberghe, L. (2002). A prática e as implicações da análise funcional. *Revista Brasileira de Terapia Comportamental e Cognitiva*, 4(1), 35-45. Recuperado de http://pepsic.bvsalud.org/scielo.php?script=sci_arttext&pid=S1517-55452002000100005&lng=pt&tlng=pt.



Introdução ao MED PC

Italo Siqueira de Castro Teixeira |  |

Amanda Calmon-Rodegheri |  |

Lucas Franco Carmona |  |

Rafael Peres Macedo |  |

O que é MED PC

A história de pesquisas em laboratório da Análise Experimental do Comportamento é marcada pelo uso de diversas ferramentas para programação e controle do ambiente experimental. O MED-PC® é um exemplo de ferramenta que foi desenvolvida para permitir o estabelecimento de controle e de registro de informações, de forma eficaz, dos vários eventos que ocorrem dentro de caixas de condicionamento operante. Trata-se de um *software*, criado pela *Med Associates Inc.*, que auxilia o desenvolvimento de contexto para coleta de dados relativos a diferentes parâmetros de uma pesquisa sobre o comportamento de organismos (e.g., respostas de pressão à barra, frequência da liberação de reforço, manipulação das condições de estímulos ambientais etc).

Atualmente na quinta versão, o *software* já recebeu algumas implementações em relação às versões anteriores, mas sempre mantendo funções importantes para pesquisadores ao: (a) conceder ferramentas para desenvolvimento de programas em uma linguagem nativa chamada *Med-State Notation* (MSN) e (b) disponibilizar ferramentas internas que criam uma ponte entre o programa construído nessa linguagem com o ambiente experimental dentro de caixas de condicionamento operante.

Os autores desse *software*, preocupados com o principal público a quem esse produto serviria, construíram uma plataforma de programação em que sua estrutura mescla características da notação clássica utilizada pela comunidade de analistas do comportamento, proposta por Mechner (1959), com a estrutura lógica e sintática da linguagem Pascal (Tatham & Zurn, 1989). Utilizando-se de uma forma mais simples de expressar comandos comparados àqueles gerados pela própria linguagem Pascal, a MSN permite a criação de contingências para estudos de condicionamento clássico e operante, mediante uma estrutura padrão que se utiliza da relação entre três estruturas básicas: “*input : output – transição*”. Cabe ao pesquisador montar um algoritmo com os comandos gerados nessa linguagem (descritos mais especificamente adiante neste capítulo).

Definição e características básicas - Entendendo mais sobre o software

O MED-PC® é um *software* que, conectado a diferentes componentes externos (i.e., caixas de condicionamento), proporciona o controle das informações geradas por esses componentes, seja com a finalidade de registro, seja com a finalidade de estabelecer os critérios das relações funcionais entre o acionamento desses mecanismos e diferentes variações ambientais permitidas pela aparelhagem. Desde as primeiras versões, o principal interesse dos criadores foi de otimizar a prática de experimentação em laboratórios, tendo como critério o desenvolvimento de uma ferramenta cujo caráter essencial fosse o baixo custo, tanto financeiro como de habilidades técnicas (Tatham & Zurn, 1989).

Combinado ao uso de caixas de condicionamento, a utilização do MED-PC® permite investigação dos mais variados fenômenos estudados em pesquisa básica do comportamento operante. A mesma empresa que produz o *software* também fornece as caixas de condicionamento que são conectadas a uma interface (externa), em que cada componente mecânico da caixa experimental (por exemplo a barra), lâmpadas de iluminação, comedouros e bebedouros, rodas de atividade etc), quando acionados, emitem pulsos elétricos que transitam por essa interface até serem identificados pelo *software*. Essa conexão entre caixa de condicionamento-interface-computador permite a comunicação necessária para que o

pesquisador avalie as configurações do ambiente experimental, análise o andamento de sessões e planeje, convenientemente, alterações. Dessa forma, o Med-PC®, enquanto um *software* de tradução e compilação, tanto a) transforma os algoritmos escritos na linguagem de programação em pulsos elétricos para acionamento de mecanismos dentro das caixas experimentais, como b) transforma os pulsos, quando provenientes da caixa de condicionamento, em dados que serão armazenados e utilizados como informações para o funcionamento das contingências programadas.

Uma representação visual da transmissão de informações por meio da interface é visualizável na tela do computador, no *layout* do Med-PC®. Nele é possível também realizar o controle de funções básicas, por exemplo: a) iniciar, alterar e finalizar, manualmente, a execução de programas; b) observar e acompanhar a execução de dois ou mais programas que podem funcionar concomitantemente; e a c) criação de macros, que são ferramentas pré-programadas para que as inicializações dos programas sejam automatizadas. Além dessas, existem funções para compilação de programas, ferramentas especializadas na criação e correção da linguagem de programação utilizada, dentre outras. O *software* também é acompanhado de estruturas que facilitam a instalação e a configuração adequada da conexão entre *software*, interface e os demais aparatos mecânicos.

A Med-State Notation (MSN)

A MSN é uma programação baseada na dinâmica de diagramas de transição, um conceito utilizado na engenharia de *softwares* que descreve o estabelecimento de regras na relação entre estados (*states*) de um programa. Além disso, a execução dos programas nessa linguagem ocorre numa ambientação multitarefa, em que as alterações dos dados são processadas em tempo real. Dessa forma, diferentes estados podem ocorrer simultaneamente, de tal maneira que é possível criar formas de controle e de mudança de variados elementos dentro de um programa.

Pode ser considerada uma linguagem mista, pois a tentativa de torná-la uma ferramenta de baixo custo demandava adaptações, o que ocasionou na utilização de características semelhantes a outras linguagens, mas com foco na otimização da aplicação. Assim, embora seja

composta de uma estrutura lógica muito semelhante à de outras linguagens, os desenvolvedores preocuparam-se em disponibilizar um ambiente de programação que priorizasse uma linguagem mais concisa e intuitiva.

Vantagens e Desvantagens dessa linguagem para Analistas do Comportamento

O Med-PC® é um *software* que apresenta uma função primordial ao trabalho de um pesquisador, qual seja, a de informatizar os ambientes de experimentação. Essa característica é, sem dúvidas, o ponto principal em relação ao qual a MSN, junto ao MED-PC®, apresenta-se como uma ferramenta vantajosa para o trabalho de qualquer analista do comportamento empenhado no estudo de diferentes fenômenos da nossa área, pois favorece o controle do ambiente experimental, que se repercute em aumento da garantia de validade interna das pesquisas realizadas.

Uma vez que o *software* opera em uma ambientação multitarefa, oportuniza-se a ocorrência simultânea de muitas pesquisas, em diferentes caixas experimentais, no mesmo dia e em um mesmo período, aumentando o poder de produção de conhecimento científico. Além disso, como tudo é planejado para ser controlado pelo computador, não há necessidade de dedicar uma atenção ponto a ponto para garantir controle dos eventos, pois será uma função a ser executada pelo próprio *software*.

A estrutura da linguagem disponível no software MED-PC® funciona numa relação entre eventos que utiliza a lógica “se...então”, um tipo de relação tão valiosa à comunidade científica e também semelhante a forma como os analistas do comportamento descrevem as relações entre os termos de uma contingência. Assim, o desenvolvimento da habilidade de programar em MSN é muito favorável para um cientista do comportamento, pois permite exercitar sua habilidade de planejador de contingências, uma vez que o MED-PC® concede um ambiente de programação com certa liberdade para criação de algoritmos.

Entretanto, mesmo facilitando a informatização da pesquisa, como vimos, o funcionamento do *software* requer uso não apenas de computador, mas de uma interface

mecânica para a trânsito de informações entre computador e caixas de condicionamento. Por se tratar de uma estrutura mecânica desenvolvida por uma empresa norte americana, a obtenção do produto pode não ser economicamente viável, considerando os custos de aquisição e de importação do produto. Além disso, a última versão do software está compatível apenas com versões atuais do Windows®, o que restringe a possibilidade de instalação e execução do produto.

Nível de dificuldade

Comparada a outros tipos de linguagem de programação (por exemplo, Visual Basic, Java etc) é possível classificar o grau de dificuldade de aprendizado dessa linguagem como fácil, uma vez que obedece a princípios semelhantes ao do planejamento de contingências, seguindo uma lógica se-então. Contudo, como o público alvo para o qual esse material está voltado (Analistas do Comportamento), em geral, não possui uma formação específica em linguagem de programação, faz-se necessárias habilidades para manuseio geral de *softwares* em computador, o que facilita o contato inicial com ferramentas disponíveis no produto.

Configurando o ambiente de programação

Após adquirir os direitos sobre o produto, o usuário receberá: um (a) CD-ROM contendo um instalador do programa, (b) um manual do usuário e (c) um manual de programação básica em linguagem MSN (ambos em inglês). A instalação pode ser facilmente conduzida com a utilização do CD-ROM que irá adicionar no computador: i) o *software* Med-PC®; ii) os *drivers* para funcionamento da placa PCI (instalada na parte interna do computador de coleta) que se conecta à interface; iii) ferramentas de importação dos dados para planilhas em arquivos Excel (*Med-PC to Excel*); e iv) uma ferramenta de tradução dos programas criados em MSN, o Trans IV, que funciona de forma semelhante a um bloco de notas (folha em branco com número

ilimitado de linhas), no qual os códigos devem ser escritos e compilados, gerando arquivos no formato padrão do *software* (.MPC).

No manual do usuário há um guia para instalação dos componentes da interface, contendo instruções para a instalação dos aparatos que estabelecem a conexão computador-interface-caixas de condicionamento. Já o manual de programação ilustra o uso de comandos básicos de programação em MSN. Composto de treze capítulos, o manual possui uma apresentação sistemática dos principais comandos, com definição e descrição das aplicações por meio de exemplos, e com exercícios ao final de cada exemplificação para que sejam treinadas e desenvolvidas as habilidades de programação em MSN.

Características da Sintaxe

Os programas escritos em MSN são formados por blocos de estados (*States*) chamados *State Sets*. Cada arquivo escrito pode conter até 32 *State Sets*, em que cada um desses grupos pode operar independentemente ou podem ser conectados mediante utilização de comandos específicos. O início de um *State Set* é representado pela escrita do comando “S.S.x,” em que *x* deve ser substituído por valores que vão de 1 a 32. Esses *State Sets* podem ser compostos por até 32 *States*, cuja representação escrita deve ser feita por “Sx,” logo abaixo da abertura de um *State Set*, sendo *x* valores que vão de 1 a 32.

Os *States* podem ser formados por um número ilimitado de *Statements* (relações entre eventos), compostos por comandos existentes na linguagem MSN, seguindo uma estrutura lógica padrão que contém três elementos básicos: a) *input*; b) *output* e c) transição. A escrita dessas relações entre eventos deve ser feita da seguinte forma “INPUT: OUTPUT ---> TRANSIÇÃO”. Observa-se que essa estrutura básica da sintaxe utilizada na MSN emula uma perspectiva causal de relação de dependência entre eventos em que: “Se algo acontecer : Então faça isso ---> Depois vá para ...”.

Para exemplificar, caso estivéssemos interessados em criar um programa para expor um determinado organismo a um esquema de razão fixa (FR), cujo critério para a produção de reforços fosse duas respostas, então a escrita deveria conter a seguinte estrutura (Figura 1):


```

S.S.1,
    S1,
    2#R1: On ^Bebedouro ---> S2

```

Figura 1. Esquema de FR em MED PC.

Nesse exemplo, apresenta-se o primeiro *State Set* do programa (S.S.1), juntamente ao primeiro *State* (S1) que contém o seguinte *Statement*: caso duas respostas ocorram no *operandum* (por exemplo, barra) de número 1 (#R1), então (:) ligue o bebedouro (On ^Bebedouro) e vá para o segundo *State* (---> S2).

Ao abrir um programa, todos os *State Sets* são lidos em paralelo (S.S.1 ao mesmo tempo em que S.S.2, S.S.3 etc), portanto, a ordem em que os *State Sets* aparecem no script não alteram a leitura de todos os comandos, já que a relação entre eles é estabelecida pelo programador. Dessa forma, os *States Sets* são compostos por uma sintaxe estruturada em blocos, uma inicialização múltipla de *State Sets* e uma dinâmica de transição. Essa composição permite o controle de cada aspecto do experimento, em que, por exemplo, o “S.S.1,” pode ser reservado para controlar a relação de contingência em vigor durante todo o experimento (e.g., a programação de um intervalo fixo – FI 30 s), o “S.S.2,” reservado para o controle das condições de estímulos dentro da sessão experimental (e.g., ligar ou desligar luz da caixa experimental), o “S.S.3,” utilizado para o registro das respostas e reforços emitidos durante toda a sessão e o “S.S.4,” para controlar o tempo da sessão experimental.

Deve-se ter claro que tanto letras maiúsculas e minúsculas como a existência de muitos espaços vazios entre as estruturas da sintaxe são irrelevantes para a execução dos comandos inseridos na relação de eventos de um *Statement*. O Med-PC® faz a leitura das linhas e salta parágrafos de forma rápida sem que ocorra nenhum prejuízo no desempenho do programa. Entretanto, é importante notar que alguns símbolos (e.g., *,%,\$ entre outros), que não exercem nenhuma função no programa, assim como as letras do alfabeto, quando dispostas fora de um comando (e.g., em forma de texto), podem resultar na falha de compilação do programa. Caso os pesquisadores queiram deixar algum comentário em texto sobre as fases do programa, a MSN permite que se utilize uma barra invertida (\) e, em seguida, o texto desejado. Dessa forma, após

a inclusão desse símbolo, qualquer informação inserida será ignorada pela leitura do *software*, tornando-se caracteres irrelevantes, tal como no exemplo abaixo (Figura 2):

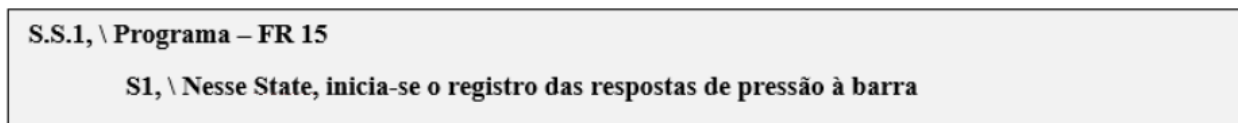


Figura 2. Exemplo de FR 15 em MED PC.

Antes de iniciar a escrita de um programa, o usuário pode renomear as sequências numéricas de cada estrutura da caixa experimental, declarando essas informações de substituição antes do primeiro *State Set*. Essa característica do software facilita o processo de programação, uma vez que o que seria antes tratado como um valor numérico, passará a ser tratado como uma palavra coerente com a situação experimental. Essa substituição pode vir em qualquer lugar no início do programa, utilizando o acento circunflexo “^” antes das palavras que irão substituir tais números. Assim, se o número do bebedouro da caixa experimental é 1 (um exemplo de configuração padrão), a substituição deste número por uma palavra deve conter a seguinte escrita “^Bebedouro = 1”. Com isso, em todos os locais do programa em que o termo “^Bebedouro” for utilizado, o *software* entenderá que algo deve ser feito em relação ao mecanismo enumerado por 1, ou seja, o bebedouro.

Além dessa renomeação, deve-se declarar também utilização de **variáveis** (as letras do alfabeto) que serão utilizadas no programa para que os diferentes comandos possam ser registrados durante a sessão. As variáveis integram grande parte dos Statements no programa e podem exercer diferentes funções. Em geral, elas guardam informações geradas durante a sessão, sendo, portanto, semelhante a uma “pasta de arquivos”. Por exemplo, algumas funções matemáticas básicas, como adição e subtração, são representadas por meio do aumento ou da diminuição do valor de uma **variável** em uma ou mais unidades numéricas. Entretanto, para que essa operação ocorra, é necessário dimensionar o uso dessa **variável**, declarando essa informação também de forma prévia, antes do primeiro *State Set*, utilizando o comando “DIM variável = número”, em que a **variável** deve ser aquela escolhida arbitrariamente pelo usuário e o número

deve especificar a quantidade de unidades numéricas permitidas para realizações de operações com essa mesma **variável** (e.g., DIM A = 100).

Alguns comandos básicos

Após serem feitas as configurações prévias para a construção do programa conforme descrito acima, a composição dos elementos dos *Statements* é feita mediante uso de diferentes comandos. Parte desses comandos será apresentado abaixo para exemplificar, na seção seguinte, alguns formatos de programas a serem desenvolvidas com a MSN.

“#”. Esse símbolo no programa é utilizado sempre como INPUT e significa a espera de algum evento que o segue. Em geral, os eventos utilizados neste comando são provenientes da caixa experimental (acionamento dos *operands*) ou proveniente de pulsos gerados virtualmente pelo *software* (pulsos emitidos pelo comando “Z”, que será explicado abaixo) Por exemplo (Figura 3):

#R1: ---> S4 (Se o organismo acionar o *operandum* número 1, vá para S4)

Figura 3. Exemplo de INPUT de espera.

“ADD e SUB”. Caso queiramos realizar operações matemáticas como subtração e adição, pode-se utilizar desses comandos para que unidades em uma determinada variável sejam alteradas com base no *Statement* escrito (Figura 4):

#R2: ADD A ---> S1 (Se acionamento do *operandum* 2, aumente em uma unidade à variável A e vá para S1)
1”: SUB T ---> S2 (Após um segundo, subtraia uma unidade da variável T e vá para S2)

Figura 4. Exemplo de ADD e SUB.

“SHOW”. Esse comando permite ao usuário ter um *feedback* em tempo real de todas as alterações dos valores das variáveis, apresentando-as, de forma ordenada, no *layout* do *software*, em um local especificado pelo próprio usuário, dentro da área reservada para a caixa de condicionamento em que o programa está sendo executado. A estrutura do comando SHOW

deve seguir o padrão “SHOW X, NOME ARBITRÁRIO, VARIÁVEL”, conforme o exemplo abaixo (Figura 5):

#R1: On ^Bebedouro; ADD A; SHOW 1, Resposta BD, A → S2 (Quando *operandum* número 1 for acionado, ative o bebedouro, aumente a variável A em uma unidade, mostre no espaço 1 do *layout* o valor atual dessa variável com o seguinte nome “Resposta BD”)

Figura 5. Exemplo de comando SHOW.

Esse *Statement* exemplifica o registro das respostas no *operandum* R1 (Resposta BD - barra direita) e o *feedback* imediato na tela do computador (adição de uma unidade à variável A), sempre que esse *operandum* for acionado pelo organismo. Nesse exemplo, o registro de cada pressão a barra é representado pela adição dessa unidade à variável A, mas é meramente ilustrativa, podendo ser utilizada qualquer letra do alfabeto para realizar tal registro, tal como indicado anteriormente. Deve-se observar também, que um *OUTPUT* pode ser composto por diferentes comandos, todos estes sempre sendo separados por ponto e vírgula (;). A execução dos outputs ocorre sequencialmente, ou seja, na ordem especificada pelo programador. Em contrapartida, a utilização de vírgulas (,) é feita para separar partes de um comando (como apresentado acima no comando *SHOW*).

“**Pulsos Z**”. A letra Z é a única letra do alfabeto que não funciona como variável para as demais operações realizadas no programa, sendo esta reservada para funcionar como transição entre *States* e entre *State Sets*. Ela pode ser utilizada no *Statement* como *INPUT* ou como *OUTPUT*, a depender do tipo de relação que o usuário queira estabelecer entre os estados do programa, como ilustrado no exemplo a seguir (Figura 6):

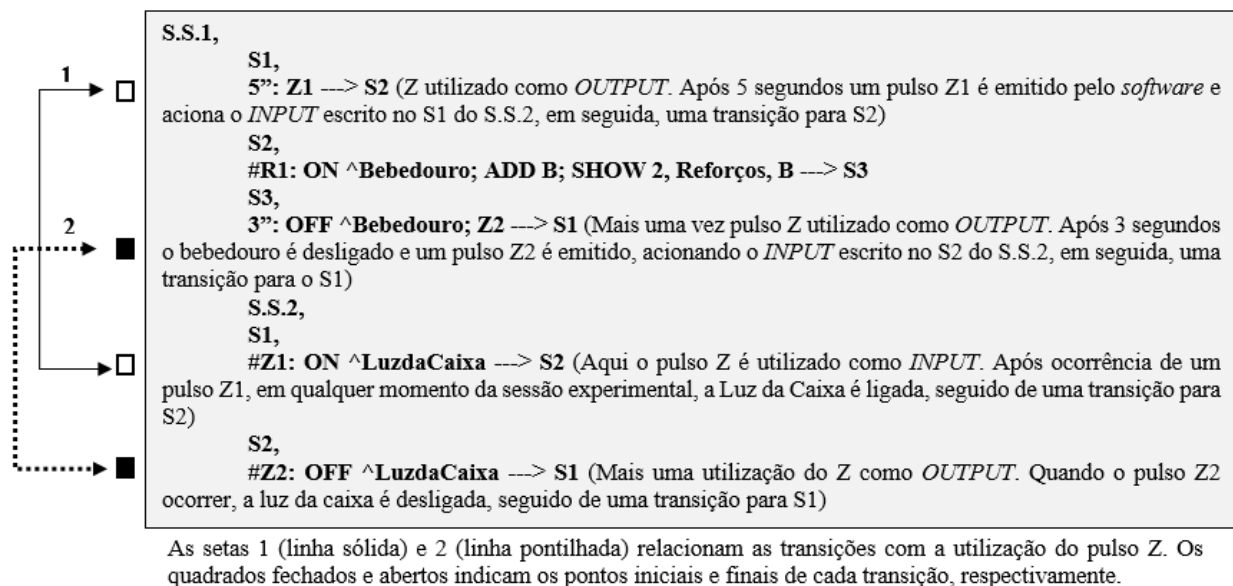


Figura 6. Exemplo de Pulsos Z.

“**IF**”. Este comando é utilizado no *OUTPUT* e serve para verificar, se um ou mais critérios foram atingidos. Para tanto, após a ocorrência de um determinado *INPUT* o programa identifica qual dos critérios foi atingido, designando uma transição para um *State Set* correspondente. A escrita desse comando deve seguir o padrão “IF variável = algum valor [@sim, @não]”, seguindo das condicionais que estabelecem a transição: “@sim: ---> Sx @não: ---> Sy”, em que x e y representam o número do *State Set* para o qual será realizada a transição, conforme o exemplo (Figura 7):

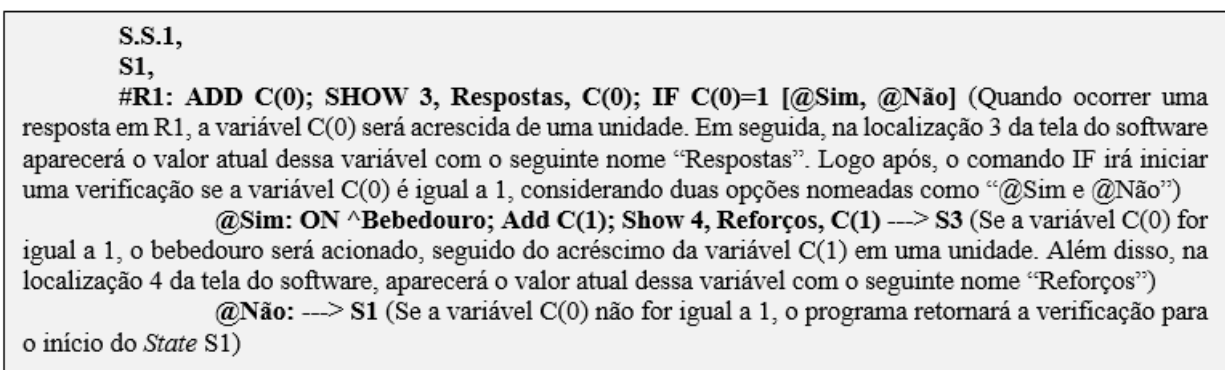


Figura 7. Exemplo de comando IF.

Nesse exemplo, o comando “IF” verifica se o valor da variável C(0) é igual a 1. Caso seja igual a 1, ou seja, o animal tenha emitido uma resposta, então o *software* irá liberar comando para acionar o bebedouro, seguindo de adição de uma unidade da variável C(1), que conta a quantidade de reforços produzidos, demonstrando-os na tela do computador pelo comando “SHOW”.

Exemplo de um programa cuja contingência em vigor é um esquema de FI 15 s

```

^LB1 = 0 \ Luz da barra 1
^Beb = 1 \ Bebedouro
DIM C = 10 \ Variável utilizada como contador de registros de respostas, reforços e tempo de
sessão

\C(1): Variável utilizada para registro do total de respostas emitidas
\C(2): Variável utilizada para registro do total de reforços produzidos
\C(3): Variável utilizada para registro do tempo total da sessão
S.S.1, \ Contador total de respostas durante a sessão
S1,
#R1: Add C(1) → S1
S.S.2, \ Contingência de FI 15 s propriamente
S1,
90": On ^LB1 → S2 \ Blackout de 90s para iniciar a sessão, logo após, liga a LB1
S2,
15": → S3
S3,
#R1: → S4 \ Programa aguarda emissão de uma resposta para produção do reforço após 15 s
S4,
0.1": On ^Beb; Off ^LB1; Add C(2) → S5 \ Reforço liberado, Desliga Luz da Barra 1, adição
\de uma unidade na variável C(2)
S5,
3": Off ^Beb, ^LB1 → S2 \ Após 3 s, recolhe o bebedouro, religa Luz da Barra
S.S.3, \ Contador de Tempo da Sessão Experimental – Contagem em Segundos
S1,
1": Add C(3) → SX
S.S.4, \ Controle dos critérios para término da sessão experimental: 100 reforços ou 60 minutos
S1,
0.1": If C(2) = 100 [@Sim; @Não] \ Se produzidos 100 reforços, encaminha-se para S3 (fim da
sessão)
        @Sim: → S3
        @Não: → S2
S2,
0.1": If C(3) = 3600 [@Sim; @Não] \ Se passados 60 minutos (3600 segundos), encaminha-se para
S3 (fim da sessão)
        @Sim: → S3
        @Não: → S1

```

S3,
0.1": Stopabortflush \ Comando que finaliza sessão experimental, salvando os dados registrados em um arquivo único.

Figura 8. Exemplo de FI 15 s.

O programa acima (Figura 8) se utiliza de alguns comandos demonstrados anteriormente. Em S1, encontra-se o *blackout* de 90 s. Após o qual, a luz da barra (LB1) será acionada. Após isso, em S2, o programa deve aguardar 15 s para avançar para S3. Transcorridos 15 s, o programa vai para S3 e nesse *State* uma resposta deve ocorrer para que o programa prossiga para S4. Ocorrida uma resposta, o reforço será liberado em S5 e o bebedouro será retornado à posição anterior, após 3 s em S6. Por fim, o programa irá para S2, dando início a um novo intervalo de 15 s do FI, até que sejam atingidos os critérios para término da sessão experimental.

Indicações para aprofundamento

Indica-se abaixo algumas fontes de aprofundamento no uso do MED-PC® e da linguagem MSN. O primeiro endereço é da página oficial da empresa *Med Associates Inc.* Nessa página é possível ter acesso não só a este produto, mas aos outros produtos oferecidos pela empresa para auxiliar desenvolvimento de pesquisas. O segundo endereço é de uma página desenvolvida por estudiosos da linguagem MSN, em que lá estão disponibilizados gratuitamente alguns exemplos de programação, com diferentes modelos de esquemas de reforçamento, que podem auxiliar o pesquisador que esteja iniciando no estudo dessa linguagem.

Endereço 1: <http://www.mednr.com/>

Endereço 2: <http://www.med-associates.com/>

Referências

Mechner, F. (1959). A notation system for the description of behavioral procedures. *Journal of Experimental Analysis of Behavior*, 2, 133-150. doi: [10.1901/jeab.1959.2-133](https://doi.org/10.1901/jeab.1959.2-133)

- Michael, J. & Shafer, E. (1995). State Notation for teaching about behavioral procedures. *The Behavior Analyst*, 18, 123-140. doi: [10.1007/BF03392698](https://doi.org/10.1007/BF03392698)
- Tatham, T.A. & Zurn, K.R. (1989). The MED-PC experimental apparatus programming system. *Behavior Research Methods, Instruments & Computers*, 21 (2), 294-302. doi: [10.3758/BF03205598](https://doi.org/10.3758/BF03205598)



Unity: Criando jogos e outras aplicações multi-plataforma

Gerônimo Oliveira da Silva Filho |  |

Pedro Gabriel Fonteles Furtado |  |

Felipe Lustosa Leite |  |  |

Hernando Borges Neves Filho |  |  |

Unity não é uma linguagem de programação, mas sim uma *game engine* disponível para Windows, Linux e Mac. *Game engines* são, de acordo, com Goldstone (2009), “As porcas e parafusos que ficam nos bastidores de todo jogo virtual”. De maneira mais objetiva, são softwares equipados de bibliotecas para facilitar a criação de jogos. A mensagem de marketing do Unity a descreve como uma *game engine* para todos os casos, tendo suporte tanto a 3D quanto a 2D e um grande leque de plataformas para qual o Unity consegue exportar (como por exemplo dispositivos mobile, videogames e aparelhos de realidade virtual). Unity também conta com comunidades ativas de usuários que compartilham suas experiências e dúvidas. Links para algumas dessas comunidades virtuais são encontradas no final do capítulo.

O Unity permite programar em duas linguagens: C# e Javascript. Sua versão de Javascript é, na verdade, uma variante chamada informalmente de Unityscript, não tendo todas as funcionalidades da linguagem original. O uso de C#, atualmente, é baseado na versão 6.0 da linguagem (Lian, 2017) . Mais detalhes sobre as duas linguagens fogem do escopo deste capítulo mas disponibilizaremos algumas fontes para informações mais detalhadas ao final.

Alguns jogos bastante populares, disponíveis em diferentes plataformas, foram desenvolvidos usando o Unity, como: Monument Valley, Kerbal Space Program, Temple Run Trilogy, Assassin’s Creed: Identity, Hearthstone: Heroes of Warcraft, entre outros. Apesar de todo funcionamento do Unity ser voltado para o desenvolvimento de jogos, outros tipos de aplicações

podem tirar vantagem dos recursos oferecidos pela *engine*. No contexto de pesquisa e desenvolvimento de tecnologia em Análise do Comportamento, o Unity pode se mostrar como uma boa ferramenta de desenvolvimento de softwares em vários segmentos. De maneira geral, o Unity pode oferecer uma valiosa contribuição no desenvolvimento de quaisquer aplicações que utilizem recursos como: gráficos 3D, simulações físicas, networking, suporte multimídia e capacidade multiplataforma. Em pesquisas que envolvem comportamento social e cultura, por exemplo, sua capacidade networking pode ser bastante útil. Quando se trata de desenvolver programas que envolvem realidade-virtual, como em pesquisas com VRET (Virtual Reality Exposure Therapy), o Unity é uma opção atraente. Esses e outros recursos podem ser muito úteis para um Analista do Comportamento e serão melhores explorados na próxima seção onde discutiremos as vantagens e desvantagens no uso do Unity.

Adquirindo o Unity

O Unity possui uma modalidade gratuita que contempla todas as suas funcionalidades e está disponível para qualquer usuário ou empresa que não obtenha mais de 100.000 dólares de receita bruta anual. A maioria das funcionalidades que discutiremos ao longo do capítulo estão disponíveis nesta versão gratuita da *engine*, quando houverem exceções sinalizaremos que o recurso é relativo a uma ferramenta criada por terceiros .

Vantagens e desvantagens

Apresentamos aqui uma lista de seis vantagens e duas desvantagens para ajudar o leitor ou leitora a decidir se o Unity é a escolha certa para o desenvolvimento de sua aplicação.

Vantagem 1: Suporte a 3D facilitado

Unity vem com suporte a gráficos 3D, incluindo um editor de cena. É muito rápido e simples adicionar um modelo 3D a Unity e é tudo feito sem escrever uma única linha de código.

Vários aspectos que seriam complicados de programar, como, por exemplo, sistemas de iluminação, já estão prontos para uso.

Vantagem 2: Suporte a realidade virtual

Unity tem suportes à vários aparelhos de realidade virtual. Na realidade virtual, o jogador veste um óculos e interage com um mundo virtual, como se estivesse dentro dele. De acordo com Haydu, Kochann e Borloti (2016), a tecnologia de realidade virtual parece trazer vantagens para as intervenções psicoterapêuticas de transtornos de ansiedade, como as fobias. Várias pesquisas vêm sendo desenvolvidas nessa área e o Unity se mostra como uma ótima opção para o desenvolvimento desse tipo de aplicação.

Vantagem 3: Multiplataforma facilitado

Unity tem suporte a computadores, dispositivos mobile, videogames (Playstation, Xbox, etc) e os já citados aparelhos de realidade virtual. Tirando otimizações e bugs, a ideia é que tudo que você criar funcione do mesmo jeito em todas as plataformas, simplificando bastante o tempo de desenvolvimento de *software* que envolva mais de uma dessas plataformas. No contexto de pesquisa, a possibilidade de um desenvolvimento simultâneo para uma multitude de dispositivos, em especial para dispositivos mobiles , pode diminuir os custos de desenvolvimento dos experimentos, ajudar na coleta de dados, além de aumentar bastante a flexibilidade do pesquisador de como utilizar o software desenvolvido.

Vantagem 4: Suporte a networking

Nesse contexto, networking quer dizer utilizar a internet para permitir que mais de um usuário acesse o mesmo aplicativo. Unity permite sincronizar diferentes objetos e informações entre os computadores dos jogadores, economizando bastante tempo de programação desse aspecto. Ao final do capítulo deixamos uma indicação de um tutorial a esse respeito. Isso abre

um leque grande para pesquisas em comportamento social e cultural. Já encontramos algumas pesquisas utilizando softwares para simular microssociedades em laboratório (e.g Camargo, 2014). Aplicações desse tipo podem se beneficiar dos recursos de networking oferecidos pelo Unity.

Vantagem 5: Prototipagem rápida

No desenvolvimento de software muitas vezes é interessante criar uma versão incompleta, somente com funcionalidades básicas (ou até mesmo funcionalidades que parecem funcionar, mas na verdade não são flexíveis o bastante para solucionar o problema), para ver se um software é viável ou não antes de se comprometer mais recursos. Chamamos isso de prototipagem (Rouse, 2005).

Unity oferece diversos suporte para criar jogos ou aplicativos multimídia rapidamente: rápido suporte a imagens, modelos 3D, partículas, editor de scene, simulação de física, sons, animação, tratamento de entrada, saída, etc. Essa rapidez no desenvolvimento reduz o tempo e o custo associados a criar protótipos.

Vantagem 6: Visual scripting (pago)

Visual scripting é uma forma de programar conectando blocos. Leigos costumam ter mais facilidade com esse tipo de programação. Unity tem várias maneiras de prover isso mas elas são todas pagas, disponibilizadas por terceiros. Unity planeja, no futuro, adicionar *visual scripting* a *engine* (Unity Technologies, 2015). Quando isso acontecer, a expectativa é que seja algo incluso em todas as versões.

Desvantagem 1: Inflexibilidade

Para programadores experientes, pode ser complicado de se adaptar a maneira como a Unity organiza seus códigos de *script* e ela não oferece outras maneiras organizar o código e os

objetos da engine. Enquanto que em muitas linguagens o programador consegue fazer tudo por código, em Unity é necessário utilizar também a interface, o que pode não ser intuitivo para quem já tem experiência com programação.

Desvantagem 2: Performance

Processamento de grande volume de dados estatísticos e outras coisas, se possível, não deveria se realiza pelo Unity utilizando linguagens de script. Unity é otimizado para jogos e não para executar código complexo que pode demorar bastante tempo para ser executado. Pesquisas em Análise do Comportamento geralmente não desenvolvem softwares que geram dados pesados o suficiente para impactar a performance da Unity, então talvez você não precise se preocupar com isso.

Conceitos básicos e interface

Nesta seção são introduzidos alguns conceitos essenciais para que se entenda como o Unity funciona (Goldberg, 2009). *Assets*, *Scenes*, *Components*, *GameObjects* e *Scripts* serão apresentados e discutidos. Optamos por usar ao longo do capítulo os nomes dos elementos básicos do Unity em inglês para facilitar sua identificação já que não há uma versão do software em português.

O conceito de **Scene** (Cena) é central em Unity. Uma *Scene* pode ser pensada como um arquivo que guarda uma fase, uma área específica ou um menu do jogo. Dividir seu jogo em muitas *scenes* pode ajudar na prototipagem, uma vez que você pode testar suas *scenes* individualmente e de forma rápida. Para que o jogador consiga visualizar o que está havendo, é necessário que na *scene* exista uma **Câmera**. A câmera é configurada pelo programador para mostrar ao usuário o que se deseja.

Chamamos os arquivos usados em um projeto de Unity de **Assets** (Recursos), e eles são armazenado em uma pasta com o mesmo nome.

Objetos em Unity são chamados de *GameObject*. Isso é um conceito diferente de programação orientada a objeto. Todo *GameObject* em uma *scene* tem posição, rotação e escalamento, isso diz respeito a um *Component* chamado *Transform*. Um **Component** (Componente) pode funcionar de várias formas, “Eles podem servir para criar comportamentos, definir aparência, e influenciar outros aspectos de uma função do objeto no jogo. Adicionando um *Component* a um *GameObject*, você pode imediatamente aplicar partes da *game engine* em seu objeto” (Goldberg, 2009, p.15). O *Transform* é um dos muitos componentes que já vem pré-definidos na *engine*.

Para um maior controle do funcionamento de seus *GameObjects* você pode programar scripts e adicioná-los como *components* nos *GameObjects*. **Scripts** são *assets* que contém a definição de uma classe em C# que é derivada de outra classe chamada de *MonoBehavior*. Todo Component, inclusive os predefinidos, são derivados de *MonoBehavior*.

Para entender o funcionamento da Unity é importante que você também esteja familiarizado com sua interface. A interface tem um papel central no desenvolvimento de aplicações quando usamos essa engine. Antes de prosseguirmos para um passo a passo de como desenvolver um aplicativo, vamos apresentar os elementos básicos da interface para que se torne mais fácil acompanhar nosso exemplo (Figura 1).

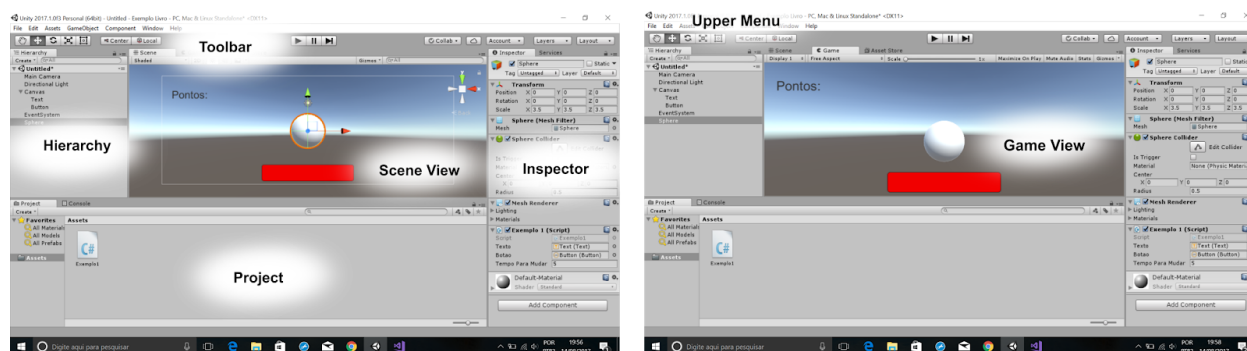


Figura 1. Principais elementos da interface do Unity em duas capturas de tela.

Como podemos ver na Figura 1, a interface da Unity é dividida em janelas com abas. Há algumas opções de layouts que você pode escolher. Dependendo da sua preferência e necessidade, você também pode mudar livremente as janelas e abas de local, agrupá-las,

desagrupa-las e fazer seu próprio layout. Neste capítulo, em nossas fotos utilizamos o *layout default*. Aconselhamos que por hora você faça o mesmo para facilitar sua execução do nosso exemplo mais a frente. Perceba que é possível identificar o nome de cada janela na aba que fica na parte superior esquerda de cada uma delas. Segue uma breve descrição das principais janelas da interface.

Scene View (Visualizador da cena)

O *Scene View* nos permite visualizar e editar nossa *scene*. É possível selecionar *GameObjects* no *Scene View*. As propriedades dos *GameObjects* selecionados irão aparecer no *Inspector* e poderão ser facilmente modificadas.

Game View (Visualizador do Jogo)

Como mencionamos, na Unity você precisa usar câmeras para mostrar sua *scene* para jogador. O *Game View* mostra a visão da(s) câmera(s) do seu jogo. Assim, através do *Game View* você pode visualizar o que o jogador verá quando ele estiver jogando. No *layout default* as janelas *Scene View* e *Game View* estão agrupadas e você pode visualizá-las clicando em suas respectivas abas.

Hierarchy (Hierarquia)

Na *Hierarchy* podemos ver uma representação textual de todos os *GameObjects* da *scene* e visualizar rapidamente como eles estão ligados entre si. Ao clicarmos em um *GameObject* na *Hierarchy* selecionamos ele na *scene* e também podemos acessar suas propriedades no *inspector*. Vale observar que apesar de todos os objetos que tem uma visualização na *scene* poderem ser encontrados na *Hierarchy*, nem todos os objetos que estão na *Hierarchy* tem uma visualização na *scene*. Esses objetos sem visualização tem informações de posição, rotação, etc, mas não são desenhados na tela, logo o jogador não sabe que eles existem.

Inspector (Inspetor)

Nos permite visualizar e modificar as propriedades de um objeto selecionado. No inspector é possível, por exemplo, mudar a cor de um botão, editar o conteúdo de um texto, modificar a localização de um *GameObject* na *scene* e também adicionar um *script* ao *GameObject*.

Project (Projeto)

Nos permite organizar os arquivos do projeto. Esses arquivos podem ser modelos 3D, músicas, sons, imagens, etc. Alguns arquivos selecionados aqui também podem ser modificados no inspetor.

Toolbar (Barra de ferramentas)

Nos dá acesso a algumas ferramentas essenciais. A esquerda temos as principais ferramentas para manipulação da *scene* e dos objetos na *scene* (Hand tool, Translate tool, Rotate tool, Scale tool) respectivamente. No centro temos os botões de play, pause e step que permitem começar um teste do jogo, pausar o jogo e encerrar um teste do jogo, respectivamente. À direita você pode acessar o Unity Cloud service e mudar o layout de sua interface.

Upper menu (Menu superior)

Permite diversas funções , como adicionar objetos, arquivos, scripts aos objetos, mudar as configurações do Unity e do jogo, etc.

Scene Gizmo (gizmo da cena)

O gizmo é um objeto interativo que mostra uma visualização do ângulo pelo qual a scene está sendo visualizada no *scene view*, como pode ser visto na Figura 2. Clicar no *scene gizmo* permite mudar rapidamente este ângulo.



Figura 2. O *scene gizmo* visualizado no *scene view*.

Exercício

Todos(as) aqueles(as) que já cursaram disciplinas básicas de AEC devem estar familiarizados com o processo comportamental a qual nos referimos por discriminação simples, que consiste, grosseiramente, em responder diferencialmente a um certo estímulo. Nos laboratório costumamos estabelecer um controle discriminativo entre uma luz e a resposta de pressão a barra. Quando a luz está acesa o rato recebe alimento ao pressionar a barra e quando a luz está apagada ele não recebe. Dizemos então que a luz acesa se torna um Estímulo Discriminativo ao passo que o rato passa a emitir consideravelmente mais respostas na presença da luz do que em sua ausência.

Escolhemos então simular em nossa aplicação um processo de discriminação simples, por meio de basicamente três elementos: um contador de pontos, uma esfera que muda de cor e um botão.

Não por acaso cada um dos três será responsável por um dos termos da tríplice contingência. A esfera, ou melhor, a cor da esfera, vai fornecer o contexto da resposta

(Sd/Sdelta). O botão será nosso operandum, é ele que vai permitir que o usuário emita uma resposta. Por fim, o contador de pontos será responsável por disponibilizar a consequência, uma vez que a resposta for emitida no contexto adequado.

Vamos construir um pequeno jogo onde uma esfera mudará de cor com uma certa periodicidade. Se o jogador apertar o botão enquanto a esfera estiver vermelha ele ganhará pontos a cada clique mas se ele apertar o botão e a esfera estiver branca, nada acontecerá. Então, vamos ao tutorial!

Usando Unity pela primeira vez

Se é a primeira vez que você está usando o Unity será necessário que você crie uma conta. É um processo simples que levará poucos minutos. Após sua conta ser ativada, basta clicar em New na aba Project e um novo projeto será criado. Para informações sobre o download e instalação do Unity disponibilizaremos um link ao final do capítulo.

Ajustando a visão

Com seu projeto criado e Unity aberto, primeiramente devemos ajustar a visão na scene para que possamos acompanhar visualmente nosso progresso. Para tanto, clique no **Scene Gizmo** com o botão direito do mouse e escolha a opção *Back*.

Criando o Objeto texto

Agora criaremos um objeto do tipo *Text* na interface do usuário. No **Upper Menu**, clique em GameObject > UI > Text. Observe que na **Hierarchy** apareceram dois novos itens : *Canvas* e *Text*.


Canvas é um espaço a parte do resto do jogo, onde ficam todos os elementos da Interface de Usuário (UI) que o jogador poderá interagir durante o jogo. Para tornar mais fácil a visualização do nosso exercício, optamos por renderizar o *Canvas* de frente para a câmera

principal de nossa scene. Para tanto selecione o item *Canvas* na **Hierarchy**. Agora no **Inspector** (Se você está usando o Unity pela primeira vez e a aba Services está sendo exibida, para visualizar o Inspector clique na aba com esse nome do lado direito da tela) procure o campo *Render Mode* e mude para a opção *Screen Play - Camera*. Ainda no Inspector, você deve clicar no pequeno círculo ao lado do campo *Render Camera*. Ao fazer isso será aberta uma janela, você deve selecionar a opção *Main Camera* e então fechar a janela.

Agora que seu *Canvas* pode ser visto próximo ao que vai ser exibido pela câmera, devemos trazer o objeto texto que você criou para frente da câmera. Para isso você deve selecionar o item *Text* na **Hierarchy**, dessa maneira você poderá ver as propriedades desse item no **Inspector**. Alterando os valores dos campos Pos (X,Y,Z) você pode modificar a localização desse objeto em cada uma das dimensões do espaço da *scene*. Para trazermos o objeto para o centro da *scene* podemos ajustar os valores dos três campos de Pos (X,Y, Z) para (0,0,0) ou clicar na engrenagem ao lado de *Rect Transform* e selecionar a opção *reset*.

Você já deve ser capaz de ver o objeto texto na *scene*. Agora procure a sessão *Text (script)* no inspector. Lá você vai encontrar um campo chamado *best fit*. Habilite o *checkbox* e como isso você verá o texto se expandir automaticamente.

Ainda na sessão *Text (script)* você vai encontrar o campo *Text*. Você deve ajustar esse campo de acordo com o texto inicial que você deseja que esse objeto apresente. No nosso caso, vamos escrever “Pontos: ”.

Agora você deve posicionar o texto onde ele deve ficar. Isso pode ser feito de duas maneiras: ou alterando os valores dos campos Pos(X,Y,Z) ou com o *mouse* diretamente na **Scene View**. Para posicionar com o mouse é só selecionar a Translate tool  na **Toolbar**, depois clicar no objeto que deseja mexer (no nosso caso, o texto) e puxar o eixo da dimensão em que você quer deslocar o objeto. Lembre que você pode ficar alternando entre as janelas **Scene View** e **Game View**. Na **Game View** você pode ver como seus objetos estão enquadrados pela câmera. Isso é importante na hora de escolher o lugar de um objeto, pois o usuário verá sua *scene* de acordo com o enquadramento de uma câmera. Em nosso exemplo, como o *canvas* está alinhado ao enquadramento da câmera, o enquadramento corresponderá a um contorno retangular que

você está vendo na *scene*. Para seu texto ficar exatamente na mesma posição da Figura 1, configure os campos Pos(X,Y,Z) com os seguintes valores (-270,90,0), respectivamente.


Criar botão

Vamos criar agora outro elemento da interface de usuário: um Botão. No **Upper Menu**, clique em GameObject > UI > Button .

Você criou um botão na *scene*. Agora vamos posicioná-lo onde ele deve ficar. Você pode mudar sua posição de modo similar ao que você fez com o texto, alterando o valor dos campos Pos (X,Y,Z) ou arrastando o botão na scene. Para seu botão ficar exatamente na mesma posição da Figura 1 configure os campos Pos(X,Y,Z) com os seguintes valores: (0,-110,0), respectivamente.

Com seu botão selecionado na **Hierarchy**, procure no **Inspector** o campo *Color*. Clique no retângulo branco do campo *color* para escolher a cor do seu botão. Escolha uma cor de sua preferência e depois feche a janela de seleção de cor. Para que seu botão fique da mesma cor do exemplo você pode copiar este código “FF0000FF “ no campo *Hex Color* da janela de seleção de cor.

Agora, para que nosso botão fique mais parecido com uma barra vamos deletar o texto do botão. Para tanto, basta expandir o item Button na **Hierarchy** clicando na setinha à esquerda do item. Com isso, o *GameObject Text* aparecerá abaixo do *GameObject Button*. Selecione o *Text* e aperte Delete no teclado. Perceba que agora o texto que estava no meio do botão desapareceu da scene.

Podemos também mudar as dimensões de nosso botão. Esse processo é similar ao deslocamento e ele pode ser feito tanto pelo ajuste de valores no *Inspector* como diretamente na *scene*. Para mudar pelo **Inspector**, você pode alterar os campos *Scale* (X,Y,Z). Cada um dos campos (X,Y,Z) diz respeito a uma das dimensões do espaço da scene. Você pode também selecionar a Scale Tool  na **Toolbar** e arrastar um dos eixos para redimensionar a dimensão correspondente, se você preferir modificar as dimensões na *scene*. Se você quiser que seu botão

tenha as mesmas dimensões da foto de exemplo você pode configurar os campos Scale (X,Y, Z) com os valores: (1.5, 1.5, 0), respectivamente.

Criando uma esfera

Chegamos ao último elemento de nossa scene: a Esfera. Diferente do Texto e do Botão ela não faz parte da interface do usuário, ela é um objeto 3D renderizado fora do *canvas*. Em termos simples, objetos 3D fazem parte da dimensão do jogo, onde ficam diversos outros elementos gráficos. No Canvas, só temos objetos de UI.

No **Upper Menu**, clique em GameObject > 3D > Sphere e a esfera será criada. Você pode redimensioná-lo exatamente da mesma maneira que fez com o Botão. Para que a esfera tenha as mesmas dimensões da foto do exemplo, configure os campos *Scale* (X,Y,Z) com os valores: (3.5, 3.5, 3.5), respectivamente.

É importante notar que a visão que você tem da esfera na **Scene View** é diferente da visão na Game View. Isso se dá pelo que discutimos antes sobre a esfera existir na dimensão do jogo enquanto que o Canvas existe numa dimensão separada. Eles são desenhados na câmera em momentos diferentes, então fique atento na hora de ajustar as dimensões da sua esfera.

Criando e adicionando um script

Vamos agora criar um *script* que vai servir para estabelecer o funcionamento dos elementos do nosso jogo e vamos adicionar esse script a nossa esfera. Para isso, selecione a esfera na **Hierarchy** e agora no **Inspector** clique em *Add Component*. Será aberta uma nova janela com uma lista de componentes que podem ser adicionados. Ao final da lista você vai encontrar o item *New Script*, clicando nele você poderá escolher o nome do *script* que você deseja criar. Em nosso exemplo optamos por “Exemplo1”. É importante que você nomeie seu *script* exatamente com esse nome, pois o código que você usará neste script constará com esse nome. Quando escolher o nome clique em *Create and Add*.

Escrever o script

Encontre o *script* que você acabou de criar em **Project** e clique nele duas vezes para abrir o programa de modificar *scripts*. Quando você clica, o Unity automaticamente vai abrir o programa apropriado se o programa já não estiver aberto. No Mac OS e no Windows, o Unity vai optar por instalar o Visual Studio, mas é possível utilizar o MonoDevelop. No Linux, a única opção seria o MonoDevelop.

Cole o código abaixo no espaço do código no programa que você estiver utilizando. Todas as linhas do código estão comentadas para que você possa entender o que cada linha faz. Após colar o código, clique em *Anexar ao Unity* na barra de ferramentas para que o arquivo seja atualizado no Unity.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class Exemplo1 : MonoBehaviour
{
    //Variável que vai receber a esfera
    private Renderer esfera;
    //Variável para o texto da interface que vai fazer a contagem de pontos
    public Text texto;
    //Botão
    public Button botao;
    // Variável que vai guardar quantos pontos o usuário tem
    private int pontos;
    //Tempo que a esfera fica em cada estado
    public float TempoParaMudar = 5f;
    //Variável que conta quanto tempo falta para mudar o estado da esfera
    private float tempo;

    //método do Unity chamado quando o jogo começa
```

```

public void Start()
{
    //atribuí a variável esfera a esfera em questão
    esfera = GetComponent<Renderer>();
    //avisa o botão para alertar quando o jogador apertar o botão
    botao.onClick.AddListener(BotaoApertar);
}

//método chamado quando o botão é apertado
private void BotaoApertar()
{
    //se a esfera estiver vermelha
    if (esfera.material.color == Color.red)
    {
        //jogador ganha um ponto
        pontos++;
        //muda o texto que mostra os pontos
        texto.text = "Pontos: " + pontos;
    }
}

//Método chamado a cada frame (60 vezes por segundo)
public void Update()
{
    //Muda a variável de tempo de acordo com o tempo mudado
    tempo = tempo - Time.deltaTime;
    //se o tempo chegar a zero
    if (tempo < 0)
    {
        //reseta o valor do tempo
        tempo = TempoParaMudar;

        //se a esfera estiver vermelha
        if (esfera.material.color == Color.red)
        {

```

```

        //muda a cor do material da esfera para branco
        esfera.material.color = Color.white;
    }
    //se a esfera estiver branca
    else
    {
        //muda a cor do material da esfera para vermelho
        esfera.material.color = Color.red;
    }
}
}
}

```

Conectando as variáveis

Agora você deve conectar o seu objeto texto e o botão ao *script* que está na esfera. Para tanto, selecione a esfera na hierarquia. Você encontrará agora no **Inspector** uma sessão com o nome *Exemplo 1 (Script)*. Nesta sessão, você poderá configurar todas as variáveis que você declarou como públicas no seu *script*. Declarar muitas variáveis como públicas normalmente é uma prática ruim em outras linguagens de programação, mas no Unity é algo necessário para que possamos visualizar os campos do *script* no editor. Dito isso, incluímos em "Fontes para Aprofundamento" um *link* da documentação de uma maneira de evitar tornar suas variáveis públicas e ainda assim visíveis na engine.

No nosso exemplo deixamos três variáveis públicas, **Texto, Botão e Tempo Para mudar**. Os campos Texto e Botão devem receber os objetos que você criou em sua *scene*. Você deve clicar no círculo ao lado direito do campo. Fazendo isso, será aberta uma nova janela onde você deve selecionar o objeto que deseja atribuir ao campo. Faça isso com ambos os campos. A variável "Tempo Para Mudar" diz respeito ao tempo em segundos que demora para a esfera mudar de cor. Ajustamos o valor inicial para 5 segundos, mas você pode mudá-lo como desejar. Com esses campos devidamente configurados aperte o play na toolbar para testar seu jogo.

Fontes para aprofundamento

1. Tutoriais oficiais (inglês) - <https://unity3d.com/learn/tutorials>
 - a. Tutorial oficial para networking
<https://unity3d.com/learn/tutorials/topics/multiplayer-networking>
2. API (application programming interface, interface de programação do aplicativo)
 - a. <https://docs.unity3d.com/ScriptReference/>
 - b. Maneira de evitar marcar variáveis como pública:
<https://docs.unity3d.com/ScriptReference/SerializeField.html>
 - c. Muito importante, contém todas as classes e funções do Unity
3. Tutoriais no Youtube
 - a. Pesquisar Unity no youtube revela muitos vídeos educativos, inclusive em português
4. Tutorial em português na SBGames
 - a. <http://www.sbgames.org/papers/sbgames09/computing/tutorialComputing2.pdf>
5. Tutorial da fábrica de jogos
 - a. <http://www.fabricadejogos.net/posts/tutorial-criando-um-jogo-de-plataforma-em-unity-parte-1/>
6. Comunidades virtuais
 - a. Em inglês
 - i. <https://forum.unity3d.com/>
 - ii. <https://www.reddit.com/r/Unity3D/>

Referências

- Haydu, V. B., Kochhann, J., & Borloti, E. (2016). Estratégias de terapias de exposição à realidade virtual: uma revisão discutida sob a ótica analítico-comportamental. *Psicologia Clínica*, 28(3), 15-34. Recuperado de http://pepsic.bvsalud.org/scielo.php?script=sci_arttext&pid=S0103-56652016000300002&lng=pt&nrm=iso

- Goldstone, W. (2009). *Unity game development essentials*. New York: Packt Publishing Ltd.
- Camargo, J. C. (2014). *Desenvolvimento Sustentável: Uma Análise Experimental do Comportamento de Extração de Recursos em Microsociedades de Laboratório*. (Dissertação de Mestrado). Programa de Pós-Graduação em Análise do Comportamento, Universidade Estadual de Londrina, Londrina, Brasil.
- Lian, A. (2017, 11 de julho). *Introducing Unity 2017 – Unity Blog*. Recuperado de <https://blogs.unity3d.com/en/2017/07/11/introducing-unity-2017/>
- Rouse, M. (2005, 12 de agosto). *What is prototype? - Definition from WhatIs.com*. Recuperado de <http://searchmanufacturingerp.techtarget.com/definition/prototype>
- Unity Technologies. (2015, 25 de junho). *Roadmap*. Recuperado de <https://unity3d.com/unity/roadmap>



Ensino e pesquisa no século XXI: Um manifesto pelo ensino de programação na graduação em Psicologia

Raphael Moura Cardoso |  |  |

Hernando Borges Neves Filho |  |  |

Luiz Alexandre Barbosa de Freitas |  |  |

Os primeiros cursos de graduação em Psicologia surgiram no Brasil na década de 1950, e o Conselho Federal de Psicologia, o primeiro código de ética do profissional de Psicologia, assim como as primeiras reuniões de sociedades científicas da área foram formalizadas pouco tempo depois, na década de 1970 (para um relato detalhado da história e pré-história da Psicologia no Brasil, conferir Lisboa & Barbosa, 2009; Soares, 2010). Nesse período, anterior à formalização da área no país, a Psicologia era tão somente uma disciplina auxiliar na prática de profissionais de áreas vinculadas à educação, saúde, medicina, direito entre outras. De 1970 até os dias de hoje, é possível observar uma crescente popularização da profissão de Psicólogo, medida pela abertura de centenas de cursos de graduação pelo Brasil.

Lisboa e Barbosa (2009), em um levantamento documental, observaram que até 2009, 396 cursos de Psicologia constavam no cadastro de instituições de ensino superior do Ministério da Educação (MEC), cursos esses majoritariamente presenciais, predominantemente de

universidades particulares, com fins lucrativos, com concentração maior no sudeste do país, com duração média de 10 semestres e carga horária de 4000 horas. Segundo o levantamento dos autores (Lisboa & Barbosa, 2009), a matriz curricular desses cursos tende a abordar tanto escolas clássicas como modernas de Psicologia, questões críticas, epistemológicas, filosóficas, científicas, éticas e relacionadas às práticas tanto tradicionais, como a clínica, como também práticas emergentes, como a psicologia do esporte. Em momento algum no levantamento dos autores, é mencionada alguma preocupação dessas matrizes curriculares com tecnologias, o impacto do advento de novas tecnologias na ciência e profissão, e muito menos qualquer vislumbre de preocupação com programação como um repertório básico de um psicólogo ou psicóloga.

Diante deste levantamento, e também da experiência dos autores em diversas universidades públicas e privadas do país, é possível afirmar com razoável certeza que hoje a programação não faz parte das matrizes curriculares de cursos de graduação no Brasil, seja como disciplina obrigatória, como eletiva, ou mesmo como curso livre. Um dos possíveis motivos para isso pode ser decorrente das matrizes terem sido formuladas ainda no século passado, em uma época na qual a programação ainda não era vista como um repertório básico e, mesmo no caso de currículos mais recentes, é possível supor que estes tenham se espelhado em currículos antigos e bem estabelecidos (entretanto, para que tal suposição seja confirmado, são necessários mais estudos de mapeamento do histórico das matrizes curriculares da Psicologia no Brasil). Diversas outras hipóteses desse tipo podem ser levantadas, e todas são legítimas perguntas de pesquisa histórica sobre a formação na área. Entretanto, por mais que as matrizes curriculares da Psicologia no Brasil estejam bem estabelecidas e razoavelmente padronizadas (Lisboa &

Barbosa, 2009), há sempre a possibilidade (e necessidade) de revisões e atualizações. Neste cenário, este capítulo final da obra pretende servir à função de um modesto manifesto em prol da inserção da programação nas matrizes curriculares dos cursos de graduação em Psicologia, partindo do pressuposto de que este é um repertório básico para a formação dos profissionais e pesquisadores da Psicologia no século XXI.

Para discutir isso, primeiro analisaremos alguns dados sobre empregabilidade em um futuro a curto e médio prazo, e qual o papel da programação nesse cenário. Na sequência, será apresentado e discutido o papel da Psicologia, como ciência e profissão, nessa nova conformação do mercado de trabalho. Feito isso, discute-se brevemente algumas das vantagens da programação na atividade do Psicólogo (expandindo o que foi brevemente exposto no Capítulo 1). Por fim, discutiremos como é possível inserir a programação no modelo curricular atual da Psicologia, e que habilidades e competências podem ser esperadas dos alunos e alunas nesse novo modelo. Na conclusão, será feito um pequeno exercício de futurologia ao elencar algumas das vantagens que a área terá ao ter seus profissionais e pesquisadores capacitados em programação, mesmo que seja em nível básico.

A importância da programação em um mundo cada vez mais informatizado

Diversos educadores, sociólogos, psicólogos e comentadores dessas áreas têm afirmado, já há algumas décadas, que a programação será (ou já é) uma das habilidades mais requisitadas e correlacionadas com a empregabilidade no século XXI (Balmant, 2017; Dishman, 2016;

Rushkoff, 2012; Shaaban, 2017; Vickery, 2016). De fato, tal relação já foi encontrada em diversos estudos (para uma revisão de aspectos relacionados com alta empregabilidade neste início de século, conferir Maree 2017), e algumas análises chegam até a supor que tal habilidade será de fato o esperado de um “*blue collar worker*”, i.e. uma habilidade tão básica e generalizada que será equivalente ao hoje chamado “trabalho manual”, que não requer formação específica (Thompson, 2017). Antecipando isso, diversas agências governamentais e não-governamentais têm fomentado e salientado o papel da programação para o desenvolvimento social e econômico de nossa sociedade neste começo de século. A UNESCO, por exemplo, descreve o papel da programação como peça fundamental da vida no século XXI:

“Aprender a programar é um processo de emancipação. Não se trata mais de treinar engenheiros, mas sim de como dar maneiras de todos os cidadãos criar, trabalhar, manusear informações e desenvolver um olhar crítico sobre a tecnologia, ao mesmo tempo em que se diverte com isso. Nesse contexto, ‘programar’ deve ser visto como uma nova forma de aprender, e não mais como um conjunto de ‘técnicas’. (UNESCO, 2014, tradução livre.)

A principal mensagem deste trecho da UNESCO, é que programar, como um conjunto de técnicas que cada vez mais é um pré-requisito básico de empregabilidade, requer, agora, um segundo passo, para além das meras técnicas propriamente ditas. Esse segundo passo é o que alguns autores e autoras chamam de “pensamento computacional” (Wing, 2008), que é,

basicamente uma maneira de olhar para o mundo como um universo de informações, e pensar como é possível ordená-lo, processá-lo e assim navegar adequadamente nele.

No decorrer da geração de nossos pais e avós, lidar com um computador e saber as operações básicas de seus sistemas operacionais tornaram-se habilidades básicas esperadas de qualquer profissional, desde a operação de um caixa de supermercado (hoje gerenciado por um sistema informatizado), até o planejamento de uma sessão de terapia genética (e.g. Yeats, Folts & Knapp, 2010). Hoje, tão logo crianças adquirem coordenação motora suficiente, estas são expostas a computadores, *tablets* e *smartphones*, na medida em que há disponibilidade desses dispositivos em seus lares e escolas (Haughton, Aiken & Cheevers, 2015). Assim, delineia-se um cenário em que saber manusear estes dispositivos é uma habilidade esperada de todas as pessoas, e assim, o próximo passo seria entender como estes computadores e dispositivos móveis funcionam. Saber o básico do funcionamento e como modificar estes dispositivos para diferentes tarefas (i.e. aplicar um “pensamento computacional”) é, portanto, parte necessária de diversas atividades profissionais, na medida em que mais e mais profissões inevitavelmente lidam (e são dependentes) de uma série de *softwares* e aplicativos. De maneira similar, é de se esperar que avanços e desenvolvimentos científico em diferentes áreas sejam também dependentes de profissionais (cientistas) que disponham de tecnologia e meios de modificar e aprimorar essas tecnologias para fins de suas pesquisas. Nestes dois casos, o repertório básico para estes fins é a programação. Um exemplo prático disso está demonstrado no ranking de 2017 dos cursos mais populares da Coursera (<https://blog.coursera.org/year-review-10-popular-courses-2017/>), uma das maiores plataformas de cursos online. Dos 10 mais procurados, cinco estão diretamente

relacionados à programação, desde cursos introdutórios a algoritmos até temas avançados como o *deep learning*.

Em um cenário no qual atividades triviais e profissionais como pedir um táxi, trabalhar como motorista particular autônomo, encomendar um jantar, organizar e despachar os pedidos de um restaurante, agendar uma consulta com um médico, e salvar o prontuário de um paciente; são todas mediadas por sistemas informatizados, como fica a atividade dos profissionais da psicologia?

O papel do Psicólogo em um mundo informatizado: A incorporação de tecnologias na ciência e profissão de Psicólogo

Os profissionais da Psicologia têm atuado em áreas muito diversas e os campos parecem se abrir cada vez mais. Tradicionalmente há áreas como clínica, escolar e trabalho e, mais recentemente, como emergentes a Psicologia do Esporte, do Trânsito, Jurídica, Hospitalar, Políticas Públicas, Psicologia em Desastres, apenas para citar algumas. Embora algumas tenham sua origem mais distante e outras mais próximas temporalmente, nos dias atuais, em todas elas os profissionais precisam utilizar recursos computacionais.

Tomemos como exemplo a atuação no contexto clínico, geralmente o motivo pelo qual os alunos ingressam nos cursos de Psicologia (Magalhães, Stralio, Keller e Gomes, 2001). O profissional deverá registrar seus atendimentos, arquivar estes registros e, eventualmente, utilizar uma agenda informatizada para marcar os horários. Para isso basta que este profissional tenha

habilidades para utilizar editores de texto e agendas *online*, algo muito comum atualmente. Há também a possibilidade de oferecer serviços psicológicos por meio de ferramentas de comunicação à distância (veja a Resolução CFP Nº 011/2012), e então precisará usar ferramentas um pouco mais sofisticadas. Mas, isso pode não ser suficiente em um futuro breve. O profissional pode dar um passo além e coletar mais informações sobre o trabalho que realiza. Pode criar um banco de dados sobre seus atendimentos. Por meio desse banco de dados ele poderá saber quais os pacientes mais assíduos, quais os dias da semana em que houve mais faltas no último ano e também conhecer outras características gerais e diagnósticas dos seus pacientes. Contudo, há mais a ser conhecido. Ele deverá manter registros de melhora dos seus pacientes e saber quanto tempo, em média, seus pacientes com depressão demoram até se sentirem melhor e quais as chances de terem episódios agudos. O mesmo é válido para outros tipos de demandas que surgem nos consultórios psicológicos. Dados como esses poderão torná-lo um profissional melhor e o auxiliarão a oferecer um serviço com mais qualidade. A reunião e análise desses dados, oriundos da prática de muitos profissionais, terá um impacto ainda maior sobre os serviços como um todo. Sem surpresa alguma, criar e gerenciar esse tipo de dado requer alguma programação, manuseio e modificação de *softwares*.

Outros exemplos de como a programação é útil, para não dizer essencial, à atuação do Psicólogo poderiam ser igualmente mencionados. Mas a questão aqui é simples, não há como ter profissionais de psicologia realmente preparados para atuar de forma relevante sem ensiná-los a programar.

Alguém poderia argumentar que o psicólogo não precisa saber programar para realizar qualquer das atividades já mencionadas. Ele poderia contratar um programador. Mas não é tão

simples quanto parece. As necessidades que nós, psicólogos, temos em relação à programação são muito específicas e, algumas vezes, difíceis de explicar a um programador que não entende nada de psicologia (e nem tem obrigação de saber).

Os detalhes dos softwares utilizados nas pesquisas psicológicas ilustram muito bem os problemas resultantes da dificuldade de comunicação entre psicólogos e programadores. Ao criar um programa para a pesquisa psicológica o programador deve saber com exatidão qual é o dado crítico que está sendo coletado e qual o nível de precisão desses dados. O controle temporal, por exemplo, deve ser o mais exato possível, pois diferenças mínimas poderão comprometer todo o estudo. Além disso, a forma como os cálculos são feitos, a partir dos dados já coletados, é crucial para a pesquisa científica. O *layout* do programa pode ser excelente, mas um pequeno erro na fórmula que é executada quando você clica no botão CALCULAR será desastroso.

A programação como uma habilidade e competência básica de um graduando ou graduanda de Psicologia

Esperamos que até aqui tenha sido clara a mensagem que desejamos transmitir - os cursos de Psicologia no Brasil devem incorporar urgentemente habilidades computacionais em seus Projetos Científicos Pedagógicos. A programação será cedo ou tarde um repertório básico de profissionais. Infelizmente, o ensino de programação e automação ainda é raro na educação fundamental oferecido pela rede pública de ensino do Brasil (ver Rosa & Azenha, 2015). No sistema privado, porém, a oferta de disciplinas com conteúdos de computação e robótica tem se tornado típico, sendo apresentada ainda nos primeiros anos escolares. Na medida em que

habilidades computacionais básicas, como a programação, serão requisitos mínimos de qualquer profissional está habilidade terá que ser ensinada na escola. Consequentemente, em alguns anos, os jovens entrarão no ensino superior já com estas habilidades, logo será uma habilidade que a Universidade potencializará também em seus estudantes. Esta é apenas uma questão de tempo.

O maior motivo de atenção neste momento, contudo, são os estudantes de Psicologia em formação ou que se formarão nos próximos anos. Vários destes estudantes não tiveram oportunidade de desenvolver habilidades computacionais durante as etapas anteriores ao ensino superior e o contato deles com a tecnologia foi geralmente limitado a uma experiência de consumidor. Quase sempre, a graduação em Psicologia, nos moldes atuais, não corrigirá essa deficiência ao longo do curso. A consequência por ignorar a importância das habilidades computacionais na formação em Psicologia são duas, em curto prazo: **1)** formação de um contingente enorme de jovens profissionais com pouco preparo para as exigências do mundo do trabalho no século XXI; e **2)** profissionais da Psicologia não preparados para reconhecer oportunidades e gerar inovações tecnológicas ou sociais.

Na pesquisa psicológica básica, a programação já é uma habilidade desejável para jovens pesquisadores e pesquisadoras. As pesquisas atuais exigem geralmente que se programem instrumentos de coleta de dados (e.g. *eye-tracking*), ou que se desenvolva um programa simples (e.g. em Python) ou realize análise de dados (e.g. em R). Em áreas como treinamento em organizações e educação, a gamificação já é reconhecida como uma ferramenta útil para aumentar o engajamento dos participantes com um programa de ensino e treino (Lineham, Kirman & Roche, 2015). O uso da tecnologia de realidade virtual pela psicoterapia se mostra promissor, por exemplo, no tratamento de fobias (Morina et al., 2015), transtornos de estresse

pós-traumático (Donat et al., 2017) e reabilitação cognitiva (Bohil, Alicea & Biocca, 2011). A aplicação destas tecnologias já ocorre nos grandes centros (incluindo brasileiros) e serão comuns em poucos anos. Portanto, não estamos a descrever um cenário distante no tempo, mas uma tendência que já está em curso.

Se psicólogos e psicólogas têm pouco preparo para lidar com as novas tecnologias, também serão praticamente incapazes de reconhecer novas oportunidades de atuação. O mercado de aplicativos móveis é bastante ilustrativo para fins de demonstração dessa deficiência na formação atual do profissional da Psicologia. Atualmente diversas empresas investem no desenvolvimento de aplicativos como forma de divulgar seu negócio, estreitar relacionamento com consumidores e oferecer seus serviços ou produtos ao maior número de pessoas. O investimento em mídias digitais móveis se tornou essencial para o mundo dos negócios. Apesar disto, as Empresas Juniores de Psicologia continuam oferecendo a tradicional carta de serviços: recrutamento, seleção e treinamento, e consultorias organizacionais. Enquanto isso, áreas como: aprendizagem móvel, desenvolvimento de mídias móveis, análise de dados, tecnologia e saúde, e gerenciamento de informações estão praticamente intocados pelos profissionais brasileiros de psicologia.

Afirmar que a Psicologia possui um vasto corpo teórico e empírico sobre o comportamento humano, e que esse conhecimento é útil no entendimento dos fatores que afetam o comportamento do usuário (e.g. realização de compra através sites, busca visual do usuário, tempo investido em uma página, imersão em games e etc) é óbvio e certamente pouco convincente. A Psicologia, como em outros ramos das ciências, gera conhecimento técnico que pode ser utilizado por várias especialidades nas mais diversas aplicações. A questão é demonstrar

se há atualmente demanda para psicólogos e psicólogos com conhecimento em tecnologia que justifique que um/a estudante de psicologia siga essa formação. Para isso, façamos um exercício de observação simples:

(1) Nas lojas de aplicativos são inúmeros os lançamentos com o rótulo educacional. Quem atesta? Quais as formas de explorar as *affordances* da mídia para promover ensino e aprendizagem? Em muitos casos, as empresas que desenvolvem estes aplicativos dizem que seu conteúdo contou com algum tipo de participação de psicólogos/as, ou que são baseados em alguma teoria psicológica. Verdade ou não, o fato é que a alusão à Psicologia feita por estas empresas demonstra como o aval destes profissionais agrega valor ao produto ofertado devido o impacto positivo sobre os consumidores;

(2) Você já deve ter visto ou ouvido falar sobre *Psycho Apps*. No mercado há diversos destes aplicativos que prometem uma série de benefícios aos seus usuários. Por exemplo, controle do stress, avaliação do estado de humor, ou até mesmo melhorar o desempenho cognitivo através de “exercícios para o cérebro” (por exemplo, o Lumosity®). A eficácia de alguns destes *apps* é questionável (por falta de pesquisa!), embora a maioria assegure que estejam apoiados por evidências científicas. A popularidade dos *Psycho Apps* indica uma demanda do público por aplicativos nessa área. Quem sabe uma oportunidade?

Acreditamos, portanto, que a área de tecnologia seja um terreno fértil para a atuação de profissionais da Psicologia e empreendedores. Contudo, consideramos que o maior impacto da

incorporação de práticas tecnológicas na formação em Psicologia seja o empoderamento de jovens profissionais em suas práticas e intervenções. Por exemplo, ao aprender desenvolver e programar uma página na internet, os profissionais da Psicologia tornarão capazes de criar canais de comunicação mais abrangentes com a sociedade seja para divulgar seus serviços, quanto para disseminar uma informação ou coletar dados de uma população. Também passarão a ser capazes de aperfeiçoar rotinas de trabalho e o gerenciamento de informações em instituições públicas ou privadas, por exemplo, através do monitoramento dos resultados de uma determinada ação social. Imagine que você poderá verificar, na sua prática profissional, o controle inibitório de uma determinada criança em sua clínica através do desempenho dela em uma tarefa simples de “GO/NO-GO” que você mesmo programou (para detalhes sobre o que é um procedimento de GO/NO-GO, conferir Perez, Campos & Debert, 2009). Desenvolver com competência o seu próprio *software* de medição e registro desse comportamento agregaria um valor imensurável à sua prática profissional, na medida em que o programa poderia ser modificado por você mesmo, em tempo hábil, de modo a se adaptar a novas demandas de clientes e situações diversas.

Por fim, ao aprender a programação, os profissionais poderão conquistar maior autonomia, pois não dependerão totalmente de um técnico de TI para realizar ações simples tais como programar uma página ou um banco de dados para solucionar um problema simples (por exemplo, em um órgão que assiste vítimas de violência, avise quando não houver retorno para um determinado caso atendido no prazo de 90 dias). As oportunidades são inúmeras, o limite é a criatividade e a capacidade de futuros profissionais da área em identificar problemas e informatizar a solução destes. O primeiro passo para isso é aprender o básico de programação.

Discussão e considerações finais

No passado, justificava-se a crença generalizada de que programação era habilidade restrita “ao pessoal da computação, engenharia e *nerds*”. Os computadores serviam a funções muito específicas e o uso destes dispositivos era circunscrito a contextos bem delimitados (e.g. o escritório, na indústria ou algumas instituições de ensino). Hoje, porém, os computadores se tornaram parte importante de nosso instrumental e, por isso, conhecer a tecnologia é crucial na formação de cidadãos empoderados.

Neste capítulo, focamos na “programação” como uma habilidade e competência básica do/da formando em Psicologia no século XXI. Obviamente, alguns dos exemplos de áreas de atuação que apresentamos exigirão habilidades computacionais mais sofisticadas. Acreditamos, porém, que a aprendizagem de programação seja o passo inicial para que o/a estudante de Psicologia adentre o universo da tecnologia de informação e comunicação. Todavia, o foco não deve ser a programação nesta ou naquela linguagem específica, mas capacitar os jovens estudantes na tradução de um problema de modo que sua solução possa ser auxiliada por computadores (i.e. pensamento computacional).

A inclusão de disciplinas voltadas à tecnologia nos cursos de graduação de Psicologia enfrentarão alguns obstáculos. A primeira é abandonar a posição que a Psicologia pouco tem a contribuir com a tecnologia. As Diretrizes Curriculares Nacionais para os cursos de graduação em Psicologia exemplificam bem este descompasso entre a formação acadêmica e as demandas atuais. Ao longo de todo o documento, a palavra tecnologia não é citada uma vez sequer, tampouco se reconhece a área tecnológica como uma possível ênfase a ser oferecida pelos

curso. No texto se reconhece que as competências básicas devem se apoiar em diversas habilidades, incluindo:

“VII - utilizar os recursos da matemática, da estatística e da informática para a análise e apresentação de dados e para a preparação das atividades profissionais em Psicologia”. (Ministério da Educação, Resolução CNE/CES nº5/2011, artigo 8º).

Ora, esta posição é justamente aquela que julgamos equivocada no ensino atual de Psicologia. Embora o texto afirme que “informática” deva ser utilizada como recurso, durante o curso em Psicologia, solicita-se ao estudante apenas as habilidades básicas como utilizar um editor de textos ou de planilhas, um pacote estatístico e pesquisa na internet. Estas eram certamente habilidades necessárias no início da década de 1990. Entretanto, atualmente utilizar “recursos da estatística e da informática” significa necessariamente programar e pensar de modo computacional. O texto também coloca Psicologia como uma área distante das áreas tecnológicas e ignora sua importância histórica na Inteligência Artificial, Robótica e Interação-Humano-Computador.

Outra dificuldade para o ensino de programação nos cursos de Psicologia tem um forte aspecto cultural e tem sido amplamente debatido. As áreas de tecnologia continuam a serem dominadas por homens (este livro infelizmente não foge à regra, por exemplo). Logicamente, a ausência de mulheres nessas áreas ocorre pela falta de estímulo nos períodos escolares. A Psicologia (no Brasil, um curso caracterizado pela grande presença feminina) poderá ajudar a combater este viés ao incluir programação em sua grade curricular, pois estimulará que mulheres

aprendam uma habilidade tecnológica e, desse modo, possam exercer melhor papéis de liderança e inovação.

Finalmente, outra dificuldade advirá da falta de professores e professoras com habilidades computacionais necessárias para o ensino de programação e promoção do pensamento computacional. Uma saída seria utilizar professores dos cursos de computação. No passado, tentativa similar foi adotada para o ensino de estatística na graduação de Psicologia, isto é, a disciplina foi ministrada por professores das áreas de ciências exatas e agrárias ou engenharias; o resultado: as disciplinas destinadas a estatística foram sendo eliminadas gradativamente da grade e os estudantes se queixam frequentemente que os exemplos dados em sala de aula não se relacionam com a Psicologia. Como dito anteriormente, o importante é compreender como um problema pode ser solucionado com o auxílio de máquinas. Para isso, faz-se necessário formar novos docentes psicólogos com estas habilidades e aproveitar aqueles que já possuem expertise em programação para lecionar essas aulas na graduação. Desse modo, não defendemos uma Computação aplicada à Psicologia, mas uma Psicologia & Computação como disciplina.

Uma das primeiras atividades que um iniciante na programação aprende é geralmente a execução da função *“print”* com a frase *“Hello, World!”*. De fato, a aprendizagem de programação abre um mundo novo de possibilidades e, por isso, a frase é muito apropriada neste contexto. Chegou a hora dos cursos de Psicologia permitirem que seus estudantes também conheçam esse mundo.

Referências

- Balmant, O. (2017, Outubro). Programação é o inglês do século XXI. *Estadão*. Recuperado de <http://educacao.estadao.com.br/noticias/geral,programacao-e-o-ingles-do-seculo-xxi,70002064295>
- Bohil, C. J., Alicea, B., & Biocca, F. A. (2011). Virtual reality in neuroscience research and therapy. *Nature Reviews Neuroscience*, 12, 752. doi: [10.1038/nrn3122](https://doi.org/10.1038/nrn3122)
- Conselho Federal de Psicologia (2012). Resolução nº 011/2012. Recuperado de http://site.cfp.org.br/wp-content/uploads/2012/07/Resoluxo_CFP_nx_011-12.pdf
- Dishman, L. (2016, Junho). Why coding is still the most important job skill of the future. *Fastcompany*. Recuperado de <https://www.fastcompany.com/3060883/why-coding-is-the-job-skill-of-the-future-for-everyone>
- Donat, J. C., Barbosa, M. E., Silva, G. R., & Kristensen, C. H. (2017). Virtual Reality Exposure Therapy for Posttraumatic Stress Disorder of bank employees: A case study with the virtual bank. *Contextos Clínicos*, 10, 23-32. doi: [10.4013/ctc.2017.101.02](https://doi.org/10.4013/ctc.2017.101.02)
- Haughton, C., Aiken, M. & Cheevers, C. (2015). Cyber babies: the impact of emerging technology on the developing infant. *Psychology Research*, 5, 504-518. doi: [10.17265/2159-5542/2015.09.002](https://doi.org/10.17265/2159-5542/2015.09.002)
- Lisboa, F. S. & Barbosa, A. J. G. (2009). Formação em Psicologia no Brasil: Um perfil dos cursos de graduação. *Psicologia: Ciência e Profissão*, 29, 718-737. doi: [10.1590/S1414-98932009000400006](https://doi.org/10.1590/S1414-98932009000400006)

- Linehan, C., Kirman, B., & Roche, B. (2015). Gamification as behavioral psychology. Em S. P. Waltz & S. Deterding (Eds.). *The gameful world: Approaches, issues, applications*. Massachusetts, MIT Press.
- Magalhães, M., Stralio, M., Keller, M., & Gomes, W. B. (2001). Eu quero ajudar as pessoas: a escolha vocacional da psicologia. *Psicologia: Ciência e Profissão*, 21, 10-27. Doi: [10.1590/S1414-98932001000200003](https://doi.org/10.1590/S1414-98932001000200003)
- Maree, K. (2017). *Psychology of career adaptability, employability and resilience*. New York: Springer.
- Ministério da Educação, Brasil (2011). Resolução CNE/CES 05 de 15 de março de 2011. Institui as Diretrizes Curriculares Nacionais estabelecendo normas para o projeto pedagógico complementar para a Formação de Professores de Psicologia. Recuperado de http://portal.mec.gov.br/index.php?option=com_docman&view=download&alias=7692-rces005-11-pdf&category_slug=marco-2011-pdf&Itemid=30192
- Morina, N., Ijntema, H., Meyerbröcker, K., & Emmelkamp, P. M. (2015). Can virtual reality exposure therapy gains be generalized to real-life? A meta-analysis of studies applying behavioral assessments. *Behaviour research and therapy*, 74, 18-24. Doi: [10.1016/j.brat.2015.08.01](https://doi.org/10.1016/j.brat.2015.08.01)
- Shaaban, A. M. (2017, Junho). Coding is the ultimate 21st-century skill set that every young girl should learn. *Their World: A Brighter Future for Every Child*. Recuperado de <http://theirworld.org/voices/code-clubs-teach-tanzanian-girls-technology-skills>
- Soares, A. R. (2010). A Psicologia no Brasil. *Psicologia: Ciência e Profissão*, 30, 8-41. doi: [10.1590/S1414-98932010000500002](https://doi.org/10.1590/S1414-98932010000500002)

- UNESCO (2014, 10 de julho). Learn by coding. Recuperado de <https://en.unesco.org/news/learn-coding>
- Rosa, F. R. & Azenha, G. S. (2015). *Mobile Learning in Brazil*. Columbia University, Center for Brazilian Studies, Zinnerama, São Paulo, SP. Recuperado de http://www.aprendizagem-movel.net.br/arquivos/Columbia_INGLES_SUMARIO.pdf
- Rushkoff, D. (2012, 13 de novembro). Code literacy: A 21st-century requirement. *Edutopia*. Recuperado de <https://www.edutopia.org/blog/code-literacy-21st-century-requirement-douglas-rushkoff>
- Thompson, C. (2017, 2 de Agosto). The next big blue-collar job is coding. *Wired Business*. Recuperado de <https://www.wired.com/2017/02/programming-is-the-new-blue-collar-job/>
- Vickery, E. (2016, Agosto). Coding and computer science: Necessary courses in 21st century school. *Partnership for 21st Century Learning*. Recuperado de <http://www.p21.org/news-events/p21blog/1984-coding-and-computer-science-necessary-courses-in-21st-century-schools>
- Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical transactions of the royal society of London A: mathematical, physical and engineering sciences*, 366(1881), 3717-3725.
- Yeatts, D. E., Folts, W. E. & Knapp, J. (2010). Older workers' adaptation to a changing workplace: Employment issues for the 21st century. *Educational Gerontology*, 26, 565-582. doi: [10.1080/03601270050133900](https://doi.org/10.1080/03601270050133900)