UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO DE FILOSOFIA E CIÊNCIAS HUMANAS
PROGRAMA DE PÓS-GRADUAÇÃO EM ANTROPOLOGIA SOCIAL

Victor Vieira Paulo

Cognição, ambiente material e relações sociais no desenvolvimento de programas computacionais

Florianópolis
2021

Victor Vieira Paulo

Cognição, ambiente material e relações sociais no desenvolvimento de programas computacionais

Dissertação submetida ao Programa de Pós-graduação em Antropologia Social da Universidade Federal de Santa Catarina para a obtenção do título de mestre em Antropologia Social.
Orientador: Prof. Dr. Gabriel Coutinho Barbosa.

Florianópolis
2021

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

Paulo, Victor Vieira
Cognição, ambiente material e relações sociais no desenvolvimento de programas computacionais / Victor Vieira Paulo; orientador, Gabriel Coutinho Barbosa, 2021.
107 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro de Filosofia e Ciências Humanas, Programa de Pós-Graduação em Antropologia Social, Florianópolis, 2021.

Inclui referências.

1. Programação. 2. Cognição. 3. Cibernética. 4. Estigmergia. I. Barbosa, Gabriel Coutinho. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Antropologia Social. III. Título.

Victor Vieira Paulo

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Dr. José Antonio Kelly Luciani,
Universidade Federal de Santa Catarina

Prof. Dr. Francisco Antônio Pereira Fialho,
Universidade Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestre em Antropologia Social.

Documento assinado digitalmente
Viviane Vedana
Data: 14/10/2021 13:02:13-0300
CPF: 946.958.570-49
Verifique as assinaturas em https://v.ufsc.br

Profª. Dra. Viviane Vedana
Coordenador do Programa de Pós-Graduação

Documento assinado digitalmente
Gabriel Coutinho Barbosa
Data: 14/10/2021 14:54:09-0300
CPF: 170.862.698-08
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Gabriel Coutinho Barbosa
Orientador

Florianópolis, 2021.

RESUMO

Este estudo investiga os aspectos cognitivos da elaboração de programas computacionais, com ênfase nas inflexões que os artefatos materiais e as relações sociais produzem sobre tal prática. A prática de programação é examinada sob a luz da teoria cibernética da cognição, a partir da qual é discutido o papel que os artefatos materiais desempenham neste ofício. Através da observação etnográfica do desenvolvimento de um programa computacional de código aberto, é explanado como esta atividade se baseia na coordenação por estigmergia. Esta análise é seguida por uma discussão sobre a relação entre a coordenação e o ambiente tecnológico no qual a atividade é realizada.

Palavras-chave: Programação, Cognição, Cibernética, Estigmergia.

ABSTRACT

This study investigates the cognitive aspects related to the elaboration of computer programs, emphasizing the inflections that material artifacts and social relations produce on such practice. The practice of programming is examined under the light of the cybernetic theory of cognition, from which is discussed the role that material artifacts play in this craft. Through ethnographic observation of the development of one open-source software, it's highlighted how this activity relies on stigmergy as a mode of coordination. This analysis is followed by a discussion about the relation between coordination and the technological environment in which the practice is carried out.

Keywords: Programming, Cognition, Cybernetics, Stigmergy.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Esta dissertação decorre da confluência de dois interesses que têm balizado minha trajetória acadêmica e profissional. O primeiro deles centra-se na questão da influência que o mundo material exerce sobre a mente. O segundo é a curiosidade pela computação.

O interesse por questões cognitivas remonta à minha pesquisa de conclusão de curso de graduação, dedicada a análise de uma modalidade de navegação praticada nas Ilhas Marshall. Ao longo desta investigação, descobri que as formas de navegação tradicionalmente praticadas na Oceania foram amplamente estudadas pela antropologia cognitiva. Esta bibliografia me levou ao trabalho de Edwin Hutchins, que desenvolveu sua teoria da cognição distribuída a partir de etnografia realizada entre a tripulação de uma embarcação militar. Neste mesmo período as leituras da graduação me puseram em contato com as ideias do ciberneticista Gregory Bateson, autor que exerceu grande influência sobre o trabalho de Hutchins. Estes dois autores despertaram meu interesse para as questões cognitivas, e sobretudo para uma hipótese bastante radical sobre a relação entre o mundo material e a mente, que enxerga a cognição como processos que não se encerram no corpo biológico, mas se espraiam também pelo ambiente externo ao organismo.

Minha curiosidade pela computação surgiu na mesma época. Enquanto escrevia meu trabalho de conclusão de curso eu trabalhava em uma empresa de desenvolvimento de software. Nos períodos não ocupados pela escrita do TCC (ou enquanto fugia dela) comecei a aprender a criar programas computacionais. O aprendizado em momentos de procrastinação se tornou ocupação profissional, e entre a graduação e o início do mestrado trabalhei como programador. Após o segundo ano de mestrado, com a elaboração da dissertação já em andamento, voltei a trabalhar na área de desenvolvimento de software.

À medida que estes interesses díspares se conectaram, passei a enxergar a programação como uma atividade cuja natureza cognitiva é indissociável dos artefatos tecnológicos através dos quais ela é praticada. Desta perspectiva surgiu o conjunto de questões que guia a presente pesquisa.

1.1 JUSTIFICATIVA

As questões cognitivas são fundacionais na antropologia. Desde o tempo dos precursores da disciplina os antropólogos têm se deparado com questões como: Que influência o contexto sociocultural exerce sobre as ideais, formas de pensamento e maneiras de perceber o mundo? O que há de universal e o que há de culturalmente particular nas diferentes formas pelas quais os humanos pensam? Em que grau os fenômenos da vida sociocultural podem ser atribuídos a fatores psíquicos, e em que medida a observação destes fenômenos pode revelar algo sobre o funcionamento da mente? A lista de questões é tão ampla quanto a lista de autores que se dedicaram a elas. De Durkheim a Evans-Prichard, de Lévi-Strauss a Lévy-Bruhl, De Leroi-Gourhan a Boas, remontar a história destas questões exigiria remontar boa parte da história da Antropologia.

1.1.1 Cognição na prática

As abordagens antropológicas de estudo da mente são variadas. Nesta dissertação foi realizada a opção de estudar os processos cognitivos através da observação e análise de uma prática concreta, realizada por indivíduos e coletivos através de ambientes tecnológicos complexos. O objetivo desta análise é verificar se, e de quais formas, os processos cognitivos em questão são influenciados pelo contexto material e pelas relações entre os participantes.

Muitas investigações cognitivas na antropologia centram-se em torno da indagação “O que pensam as pessoas? ”. Tal abordagem, que frequentemente se volta para o estudo de representações mentais compartilhadas, é exemplificada pelos estudos da antropologia social e cultural que mobilizam conceitos como cosmologia, visão de mundo, sistema de pensamento, crença, entre outros. Entretanto, talvez o exemplo mais bem-acabado deste tipo de abordagem venha do subcampo da disciplina que se convencionou chamar de antropologia cognitiva. Influenciada tanto pela antropologia quanto pelas Ciências Cognitivas, campo científico erigido justamente em torno da ideia de que a operação da mente deve ser analisada a partir de um nível representacional (GARDNER, 1996, p.53-54), a antropologia cognitiva não apenas reconhece a existência de elementos ideacionais na vida sociocultural,

mas considera que a cultura como um todo é inerentemente ideacional (HUTCHINS, 1995, p.354). O conceito de cultura cunhado por Ward Goodenough (1957, apud HUTCHINS, 1995, p.354, tradução nossa)[1], fundacional para este ramo da antropologia, atesta este ponto ao postular que cultura é "[...] o que quer alguém tenha que saber para se comportar apropriadamente em qualquer um dos papéis assumidos por qualquer membro de uma sociedade". Ao pensar a cultura nestes termos, a antropologia cognitiva se coloca como uma disciplina que investiga a natureza do conhecimento cultural, bem como os processos mentais que interagem com este conhecimento (D'ANDRADE, 1995, p. XIII-XIV).

Muitas investigações alcançaram resultados significativos a partir desta abordagem. No entanto, o foco em como as pessoas pensam parece vir frequentemente carregado de uma negligência sobre aquilo que elas fazem. Isto é, a agenda que estuda a cognição a partir dos aspectos ideacionais da vida sociocultural costuma deixar de lado o fato de que os processos cognitivos estão presentes nas práticas cotidianas das pessoas, e que essas práticas, por sua vez, são informadas pelos contextos socioculturais nos quais decorrem (HUTCHINS, 1995, p. XII-XII). Assim, o enfoque na prática constitui uma outra margem de estudo para a interação entre a cognição, sociedade e cultura.

O estudo da cognição através da prática tem sido reivindicado de uma forma ou de outra por vários autores da antropologia contemporânea - os estudos sobre cognição e percepção de Tim Ingold (2010) constituem um exemplo evidente, assim como a antropologia da Ciência e Tecnologia conforme praticada por autores como Bruno Latour (1997). Entretanto, no âmbito desta dissertação, as principais influências teóricas advêm da Teoria da Cognição distribuída, elaborada pelo antropólogo cognitivista Edwin Hutchins (1995), e a abordagem cibernética da cognição de Gregory Bateson (1972). O capítulo 1 se dedica a tratar em mais profundidade estes referenciais teóricos.

[1] No original: "whatever it is one must know in order to behave appropriately in any of the roles assumed by any member of a society".

1.1.2 Mente, matéria e sociedade

A agenda de estudos de cognição na antropologia sempre teve como um de seus eixos a investigação dos efeitos recíprocos entre os fenômenos socioculturais e a mente. Quando esta questão é abordada pela via ideacional, os problemas que se colocam dizem respeito ao caráter sociocultural dos conhecimentos mobilizados pelos indivíduos, e pela forma como esses conhecimentos são transmitidos entre as pessoas. Entretanto, quando visto sobre o prisma da prática, abre-se a possibilidade de analisar os possíveis efeitos cognitivos das relações sociais entre os praticantes. Este ponto é especialmente relevante quando consideramos que na sua vida cotidiana as pessoas frequentemente realizam colaborativamente diversas práticas que envolvem processos cognitivos de naturezas e complexidades variadas. Situações de colaboração em atividades de raciocínio, memorização, resolução de problemas, etc. são verificáveis da sala de aula até os ofícios profissionais, da vida doméstica até o debate público. Tais situações oferecem interessantes objetos de estudo para aqueles que desejam analisar a interação entre as dinâmicas socioculturais e o funcionamento da mente.

Entretanto, a análise das práticas cognitivas oferece ainda uma outra via para tratar deste problema. Uma delas, que busco percorrer nesta dissertação, diz respeito à interação entre os processos cognitivos e o contexto material em que eles ocorrem. Autores de diversas áreas do conhecimento têm investigado essa questão, e os resultados levam a crer que o mundo material desempenha um papel crucial em diversos processos cognitivos. Evidências demonstram que as pessoas frequentemente, e de forma ativa, mobilizam estruturas do mundo externo ao corpo para desempenhar papéis mnemônicos, auxiliar a canalização da atenção (CLARK, 2008, p.13, 21, 45, 63-64), ou mesmo transformar os processos cognitivos necessários para realizar uma dada tarefa (HUTCHINS, 1995, p.170-171). Assim, os elementos materiais (sejam eles características do ambiente circundante, artefatos de diversas naturezas, ou mesmo símbolos inscritos em suportes físicos), encontram-se profundamente imbricados nos processos cognitivos humanos, e as pessoas os manipulam e combinam de diversas formas para resolver os problemas que se apresentam cotidianamente (CLARK, 2008, p.13).

Mas há mais do que a exploração oportunística dos potenciais cognitivos de estruturas materiais pré-existentes. Os ambientes materiais nos quais os humanos vivem são construídos por eles mesmos, e tal construção frequentemente produz (seja por intenção ou como efeito colateral) meios materiais capazes de suportar, facilitar ou propiciar a realização de processos cognitivos. Como afirma Edwin Hutchins (1995, p.XVI, tradução nossa), "*Os humanos criam seus poderes cognitivos ao criar os ambientes em que eles exercem esses poderes*"[2]. O filósofo da cognição Andy Clark (2008, p.61-62) cunhou o termo "construção de nicho cognitivo" para se referir a esse processo, num paralelo com a atividade de construção de nicho ecológico através do qual animais escolhem e modificam o ambiente no qual vivem para que este se adeque ao animal. Na definição deste autor, a construção de nicho cognitivo é:

> [...] o processo pelo qual animais constroem estruturas físicas que transformam espaços de problema de maneiras que ajudam (ou por vezes impedem) pensar e raciocinar sobre algum domínio ou domínios alvo. Estas estruturas físicas combinam-se com práticas culturalmente transmitidas apropriadas para melhorar a resolução de problemas e, nos casos mais dramáticos, tornam possível formas inteiramente novas de pensamento e razão. (CLARK, 2008, p.62-63, tradução nossa[3])

A própria definição revela que a construção de nichos cognitivos está ubiquamente presente na vida humana. Um exemplo básico de construção de nicho cognitivo é verificado quando uma pessoa organiza seu ambiente de trabalho para facilitar determinadas atividades intelectuais. No entanto, a profundidade dos efeitos cognitivos deste fenômeno só pode ser plenamente apreciada quando consideramos que os seres humanos não constroem apenas seus ambientes individuais, mas também elaboram estruturas materiais de longa duração, e ao fazê-lo modificam os nichos cognitivos das gerações subsequentes, oferecendo (ou negando) a elas formas de pensar, perceber, apreender e resolver problemas. Algumas teses inclusive postulam que essa capacidade tipicamente humana de modificação constante e radical do ambiente teria tido impactos sobre a evolução biológica da espécie, sendo

---

[2] No original: *"Humans create their cognitive powers by creating the environments in which they exercise those powers"* (Edwin Hutchins (1995, p. XVI).

[3] No original: "[...] *the process by which animals build physical structures that transform problem spaces in ways that aid (or sometimes impede) thinking and reasoning about some target domain or domains. These physical structures combine with appropriate culturally transmitted practices to enhance problem solving and, in the most dramatic cases, to make possible whole new forms of thought and reason*."

um dos fatores que contribuíram para a notável plasticidade neural humana (CLARK, 2008, p.66-67).

Do ponto de vista da antropologia, o crucial é que os processos pelos quais os seres humanos constroem os ambientes materiais que habitam são justamente as práticas socioculturais. Portanto, o estudo das complexas redes causais pelas quais os processos cognitivos imbricados nas práticas são influenciados pelos ambientes, e por sua vez incidem sobre potenciais cognitivos destes ambientes ao modificar sua composição material, constitui uma outra via para o estudo da interação entre a cognição e o mundo sociocultural. Percorrer esta via envolve um estudo das práticas cognitivas em seu contexto natural de ocorrência, de forma que a complexidade das interações entre os ambientes e as práticas possa ser apreciada e escrutinada. A antropologia, por sua vez, está singularmente equipada para empreender esse estudo, pois sua ênfase na investigação da vida social em contexto lhe legou métodos e perspectivas que podem ser empregados para o estudo da vida cognitiva em contexto.

1.1.3 A prática

Este texto objetiva tratar das questões levantadas através da análise da prática de programação, e em particular da observação do desenvolvimento do software de código aberto chamado rustc, compilador da linguagem de programação Rust.

A elaboração de programas computacionais é uma atividade de manifesto caráter cognitivo. Além disso, o que é crucial sob o ponto de vista desta pesquisa é o fato de que esta atividade é indissociável do ferramental através do qual ela é praticada. Desde o hardware do computador - sem o qual não haveria o que programar - até aos complexos softwares mobilizados pelos programadores para escrever, analisar, testar e monitorar seus programas computacionais, a elaboração de software é uma prática perpassada de pé a ponta por artefatos materiais. Os próprios programadores dedicam parte significativa do seu tempo ao estudo, domínio e elaboração das ferramentas de sua prática, criando programas cuja razão de ser é auxiliar na confecção de outros programas. Como veremos mais tarde, o próprio rustc é um exemplo desta modalidade ferramental de software. Esta preponderância das ferramentas na elaboração de softwares justifica uma das hipóteses que baliza este

texto segundo a qual tais artefatos podem exercer influências relevantes sobre os processos cognitivos envolvidos na arte da programação.

A temática da programação oferece também uma margem para o estudo da dimensão das relações sociais. Nos dias atuais o desenvolvimento de qualquer programa computacional razoavelmente complexo exige ampla colaboração entre programadores. O caráter coletivo dessa prática suscita uma série de questões sobre como os programadores organizam e coordenam seu esforço cooperativo. Como veremos nos capítulos posteriores, tais fatores organizacionais têm efeitos importantes sobre os aspectos cognitivos da prática.

Para além do interesse geral colocado pela prática de programação, outros fatores particulares guiaram a seleção do rustc como projeto de desenvolvimento de software a ser analisado. A opção de analisar um processo de desenvolvimento de código aberto constitui uma das principais premissas desta pesquisa, e acarreta desdobramentos importantes para sua concepção teórica e metodológica.

1.1.3.1 Softwares de código aberto

Para tratar deste ponto são necessárias algumas definições e contextualizações. Um programa computacional é definido pelo seu código fonte, que consiste em uma sequência de instruções computacionais expressas em uma dada linguagem de programação. É comum que programas usados cotidianamente em computadores pessoais sejam obtidos na forma de um pacote fechado, contendo uma versão que pode ser executada, mas não modificada, já que o código fonte está ausente. Este código é propriedade exclusiva da pessoa ou organização que detém os direitos de propriedade intelectual sobre o programa, e apenas ela pode modificar o seu funcionamento. O desenvolvimento de programas de código aberto surge como uma contraposição direta a esta forma de criação e distribuição de software. Distribuído sob licenças de propriedade que permitem sua livre apropriação, modificação e mesmo redistribuição, um programa deste tipo tende a constituir um esforço coletivo: qualquer pessoa com as competências necessárias pode efetuar modificações ou correções no software e enviá-las aos distribuidores originais para integração no projeto principal, ou até mesmo distribuí-las como uma versão alternativa do programa em questão.

A história do compartilhamento do código fonte de programas computacionais é relativamente longa no âmbito da tecnologia da informação, e tem se desenrolado em conjunto com a evolução da Internet. O surgimento do projeto do sistema operacional Linux constitui um marco fundamental nesta história. Iniciado no começo da década de 90, os primeiros anos deste projeto ocorreram num contexto de massificação do acesso à Internet em computadores pessoais. O desenvolvimento do sistema operacional foi iniciado pelo engenheiro de software finlandês Linus Torvalds, que em 1991 disponibilizou o código fonte de seu programa em um canal de comunicação na Internet, buscando atrair o engajamento de outros programadores no projeto. A aposta foi bem-sucedida, e Linus passou a liderar uma crescente comunidade de usuários e colaboradores, organizada através de uma metodologia de desenvolvimento inovadora. Enquanto o modelo de desenvolvimento tradicional vigente no período previa a realização do trabalho por um time restrito de programadores, a comunidade do Linux permitia que qualquer um enviasse correções e novas funcionalidades. Cabia a Torvalds, e posteriormente a uma equipe de desenvolvedores escolhidos por ele, aceitar ou rejeitar as contribuições recebidas e guiar os rumos do desenvolvimento. O desenvolvimento do sistema operacional foi enormemente bem-sucedido, passando rapidamente a competir com outros programas do mesmo tipo comercializados à época. Esse sucesso é atribuído em grande parte à constituição de uma metodologia de desenvolvimento capaz de tirar proveito da potencialidade de colaboração em larga escala facultada pela Internet (RAYMOND, 2000).

A criação de um sistema operacional de qualidade provou que o desenvolvimento de programas de código aberto através da Internet configurava uma metodologia adequada para a realização de uma das mais complicadas tarefas da engenharia de software (RAYMOND, 2000). Diante de tal sucesso, não demorou para que tal modelo de desenvolvimento passasse a ser comparado favoravelmente a várias das práticas consolidadas na indústria de programas computacionais (TUOMI, 2001). A partir daí o desenvolvimento colaborativo de programas de código aberto através da Internet se consolidou. Tal processo teve como um de seus efeitos a emergência de um ecossistema de plataformas online especialmente desenhadas para suportar esta prática.

Já há algum tempo pesquisadores da antropologia e da sociologia vem estudando os programas computacionais de código aberto a partir de várias perspectivas. O fenômeno dos softwares de código aberto tem sido encarado como um movimento social (EVANGELISTA, 2010; 2013; MURILLO, 2009; KARANOVIC, 2012; COLEMAN, 2009), examinado nos termos da cultura e dos princípios ideológicos compartilhados pelos seus participantes, bem como das práticas políticas daí decorrentes. Particularmente prevalente é a abordagem desse movimento como uma cultura da dádiva (BERGQUIST & LJUNGBERG, 2001; APGAUA, 2004).

A presente pesquisa segue um caminho diferente, interessando-se pelo software de código aberto não como um movimento dotado de valores e práticas políticas, mas como uma forma de desenvolver programas computacionais. O tipo de colaboração massiva através da Internet, inaugurado por Linus Torvalds e até hoje reencenado em projetos como o da linguagem de programação Rust, levanta a questão de como é possível coordenar um esforço de colaboração tão amplo em torno de tarefas inerentemente complexas. Esta questão, por sua vez, representa uma via concreta para a investigação das interações entre os fatores sociais e as dinâmicas cognitivas.

Também é provido de interesse o fato de tal colaboração ocorrer através da Internet, por meio de plataformas online, canais de comunicação e documentos virtuais. A sofisticação e importância deste meio tecnológico, sem o qual não seria possível a colaboração de programadores distribuídos através do globo, torna plausível a hipótese de que tais tecnologias produzem efeitos relevantes e identificáveis sobre as dinâmicas cognitivas envolvidas nesta colaboração, e sobretudo nos esforços de coordenação necessários para que a cooperação seja efetiva. Esta hipótese difere da hipótese anterior por atentar para o papel que as tecnologias desempenham na coordenação entre vários programadores, e não na atividade individual de cada programador.

Por fim, subsiste um interesse metodológico pela via de acesso que o desenvolvimento de software de código aberto apresenta para o estudo da prática de programação. Projetos de software que aderem a este modelo de colaboração massiva através da Internet tem como pré-condição a publicidade do processo de desenvolvimento, e tal publicidade cria uma via de acesso importante para

pesquisadores interessados em analisar processos de desenvolvimento de software. Este último ponto será discutido com mais vagar na próxima sessão.

1.2 MÉTODO E ESTRUTURA DO TEXTO

Os capítulos desta dissertação se organizam em torno das duas hipóteses colocadas na seção anterior. Métodos diferentes foram utilizados para tratar de cada uma destas hipóteses, e, portanto, optou-se por discutir as questões metodológicas juntamente com a estrutura do texto.

1.2.1 Modelo idealizado da prática de programação

Antes de tratar diretamente da análise do desenvolvimento do rustc, foi necessário explanar o que se trata a prática de desenvolvimento de programas computacionais, de forma a introduzir termos e conceitos relevantes para o entendimento do texto, e esclarecer de que modo ela pode ser compreendida como uma atividade cognitiva. O primeiro capítulo deste texto foi dedicado a estes pontos, caracterizando os problemas fundamentais enfrentados na elaboração de um software, para então descrever um modelo idealizado dos processos que um programador pode mobilizar para resolver tais problemas.

A partir deste modelo, foi possível discutir a caracterização da programação como uma atividade cognitiva, sob o prisma de diferentes perspectivas teóricas a respeito da cognição humana. Mais especificamente, o capítulo realiza uma crítica das pesquisas que descrevem os aspectos cognitivos da atividade dos programadores a partir da teoria de processamento de informação, e a partir daí propõe uma conceituação da programação inspirada na teoria cibernética da mente. Enxergar a programação através da perspectiva cibernética abriu um ângulo diferente para o tratamento da hipótese de que as ferramentas materiais desempenham um papel relevante nos processos cognitivos envolvidos na programação, e a seção final do primeiro capítulo se dedicou a explorar este ângulo enfocando o papel cognitivo desempenhado pelas linguagens de programação de alto nível.

Em termos metodológicos, é importante ressaltar que o modelo da atividade de programação que constituiu o fio condutor do primeiro capítulo não se baseou em

dados obtidos empiricamente. Desde o começo da elaboração desta pesquisa, era sabido que uma etnografia dos ambientes online de desenvolvimento de um programa computacional de código aberto não poderia prover todos os dados necessários para que se descreva um modelo completo da atividade de programação. Uma parte relevante dos processos realizados por cada programador que contribuiu para um projeto deste tipo nunca é publicizada, e a observação dos ambientes online permite visualizar apenas os resultados destes processos. Contudo, muitos destes processos envolvem justamente uma dimensão que esta dissertação almeja investigar, que diz respeito ao uso que o programador faz das ferramentas de sua prática. Para suprir esta lacuna, o projeto de pesquisa previa a realização de observações empíricas da atividade de um programador individual durante a elaboração de um programa computacional, ou de uma parcela significativa de um. Entretanto, o projeto subdimensionou tanto o tempo necessário para a realização desta etapa quanto o tempo exigido para a observação e análise das atividades de desenvolvimento do rustc. Assim, para que o último esforço pudesse ser posto a bom termo, foi feita a opção de abandonar o propósito de etnografar a prática de um programador.

A solução encontrada foi a elaboração de um modelo idealizado da programação, baseado em descrições desta prática encontradas em publicações de pesquisadores das ciências cognitivas e das ciências da computação, bem como em manuais utilizados no ensino da programação e da engenharia de software. Evidentemente a experiência do autor como programador influenciou a concepção do modelo, mas houve um esforço para que a caraterização da prática fosse balizada pelas fontes consultadas. O resultado obtido foi um modelo que, embora demasiadamente esquemático, reflete os processos essenciais envolvidos no desenvolvimento de um programa computacional.

1.2.2 Observação do desenvolvimento de rustc

Por sua vez, o segundo capítulo deste texto enxerga o desenvolvimento de um programa computacional de código aberto como uma atividade cognitiva realizada pela colaboração de vários programadores, e investiga a hipótese de que os elementos tecnológicos que mediam esta colaboração têm efeitos cognitivos

identificáveis. Esta seção do texto baseia-se no material coletado durante a observação etnográfica do desenvolvimento do rustc.

Anteriormente foi ressaltado que o desenvolvimento de um programa de código fonte aberto costuma acontecer em canais publicamente acessíveis através da Internet. Tal publicidade constitui uma via de acesso importante para uma pesquisa etnográfica, pois as tecnologias que suportam esta modalidade de desenvolvimento de software registram e publicizam cada linha de código alterada, cada mensagem trocada em discussões a respeito de questões técnicas ou organizacionais, o andamento de cada tarefa, bem como ações realizadas por sistemas automatizados e outros tantos dados relevantes. Através destes registros, o pesquisador pode reconstituir uma imagem bastante rica e detalhada das situações concretas do desenvolvimento de software de código aberto. No âmbito desta dissertação, considerou-se que tais registros constituem base suficiente para uma descrição da prática capaz de tratar das questões já elencadas.

A opção por trabalhar a partir de tais registros publicamente acessíveis põe o pesquisador numa situação similar à daquele programador que, ao se interessar por um projeto de desenvolvimento de código aberto, opta por contribuir para ele e busca as informações necessárias para fazê-lo. Caso deseje se assegurar que sua contribuição será aceita, o programador poderá buscar se informar sobre como opera o projeto, quais tarefas estão pendentes de execução, como os participantes do projeto esperam receber contribuições e quais os padrões técnicos e organizacionais vigentes. Para fazê-lo, as informações que ele terá à sua disposição são justamente aquelas presentes nos tipos de documentos e registros analisados nesta dissertação. Assim, o pesquisador que conduz a etnografia deste tipo de projeto de software pode recorrer a um artifício antigo dos etnógrafos, que consiste em observar as práticas que se deseja estudar a partir do ponto de vista de um neófito.

No que tange o material levantado, é possível distinguir três esforços distintos de observação e análise que, apesar de decorrerem paralelamente, foram guiados por objetivos diferentes. O primeiro deles consiste no levantamento de informações sobre o projeto da linguagem de programação Rust, do qual o desenvolvimento do compilador rustc é apenas uma parte (ainda que uma parte crucial). O projeto será descrito em detalhe no segundo capítulo, cabendo aqui destacar apenas que se trata de um esforço de larga escala realizado em múltiplas frentes. Neste momento, o foco

foi entender o contexto sobre o qual o desenvolvimento do rustc ocorre, e também obter um modelo de seus processos de desenvolvimento a partir daquilo que é descrito na documentação do projeto. Portanto, nesta etapa o enfoque foi primariamente documental, e abrangeu a leitura das páginas oficiais de promoção do projeto, sua documentação organizacional e técnica (com especial ênfase naqueles documentos que diziam respeito especificamente ao desenvolvimento do rustc), artigos e teses acadêmicas a respeito da linguagem de programação Rust, textos sobre os processos de desenvolvimento, ideais e aspectos técnicos de rustc veiculados em blogs e redes sociais pelos seus desenvolvedores. Também foram acompanhadas discussões realizadas nos fóruns online e canais de bate-papo oficiais do projeto Rust, bem como assistidos vídeos de palestras a respeito da linguagem rust em conferências de profissionais da indústria de software.

O segundo esforço foi pautado pela necessidade de entender a operação e uso dos principais artefatos tecnológicos mobilizados no desenvolvimento de rustc. Este esforço foi guiado pelo pressuposto de que uma compreensão detalhada dos artefatos seria necessária para identificar e descrever os possíveis papéis cognitivos que eles desempenham na atividade sob análise. Neste momento, foram analisadas documentações e manuais da plataforma online que sedia o desenvolvimento do rustc, chamada GitHub, bem como do Git, sistema de controle de versões sobre o qual o GitHub foi construído. Também foram investigados diversos sistemas de softwares mobilizados - ou mesmo criados especificamente - para auxiliar o desenvolvimento do rustc, incluindo aí programas de análise automatizada de código fonte, sistemas que facilitam ou automatizam tarefas organizacionais e comunicacionais, sistemas de testes de software, integração e entrega contínua, entre outros. Realizar esta tarefa exigiu leitura de documentações técnicas e em alguns casos exame do código fonte e dos arquivos de configurações destes softwares, além da observação de seus usos em situações concretas.

O esforço final envolveu a observação e análise detalhada de situações específicas do desenvolvimento de rustc. A análise prévia permitiu identificar os principais tipos de atividades envolvidas no desenvolvimento do software rustc. Foi identificado que o desenvolvimento do compilador rustc é levado adiante através de três tipos de atividades principais: a) idealização de novas funcionalidades e melhorias através da publicação e debate de propostas, b) relato e análise de erros no

funcionamento do software e c) modificações do código fonte do programa visando corrigir erros ou implementar novas funcionalidades. Estes três tipos de atividade de forma alguma esgotam tudo o que é feito no âmbito do desenvolvimento do compilador, pois uma miríade de outras atividades e processos são necessários na operacionalização deste projeto de software. Entretanto, as atividades selecionadas constituem o núcleo do esforço de programação envolvido no desenvolvimento do compilador. A partir daí algumas instâncias de cada uma destas tarefas foram selecionadas para uma análise mais minuciosa, sem que esta segunda etapa de seleção se pautasse por qualquer critério definido.

A análise conduzida varreu os principais canais de desenvolvimento do rustc, que serão detalhados no segundo capítulo, buscando identificar todos os registros de interações envolvidas na execução da instância de tarefa sob escrutínio. Tais registros incluem trocas de mensagens entre participantes do projeto, contribuições de código fonte e de documentação, entre outros. Os registros foram reunidos e ordenados temporalmente em fichas, onde foram inscritas também notas sobre os artefatos de softwares mobilizados na tarefa em execução e interpretações iniciais sobre os processos em análise. Num segundo momento, estes registros foram analisados e comparados com os materiais documentais produzidos pelos desenvolvedores de rustc para explicar os processos de desenvolvimento adotados neste projeto. Esta etapa permitiu um melhor entendimento destes processos, e também um entendimento do quão próxima a prática real encontra-se das descrições e prescrições contidas nos documentos de referência. O material recolhido neste ponto embasa as descrições de situações concretas do desenvolvimento do rustc apresentadas no segundo capítulo.

Por fim, o texto contém uma conclusão que sumariza os resultados obtidos e discute brevemente a relação entre os referenciais teóricos mobilizados e as perspectivas da antropologia social.

2 PROGRAMAÇÃO E COGNIÇÃO

A palavra cognição usualmente designa uma ampla gama de processos mentais realizados por humanos ou outras espécies animais. O presente capítulo busca esclarecer o sentido específico em que tal noção, a princípio tão difusa e abrangente, é empregada neste texto, a fim de construir um entendimento da programação de computadores como atividade cognitiva. Para cumprir este último objetivo será necessário apresentar um modelo da prática de programação, o que trará o benefício adicional de introduzir o leitor a diversos conceitos que serão pertinentes para a leitura nos capítulos posteriores.

2.1 PROGRAMAÇÃO E RESOLUÇÃO DE PROBLEMAS

A cognição usualmente é concebida como um processo interno, restrito aos limites do crânio e da pele. Assim, um estudo da programação como atividade cognitiva parece implicar a necessidade de penetrar a cabeça dos programadores enquanto os mesmos realizam a sua prática, ou ao menos tentar algum tipo de acesso indireto às funções cognitivas internas por meio de testes psicológicos em condições controladas. No entanto, o presente estudo vai na direção contrária, e busca analisar a programação em seu contexto de prática, através do exame das representações materiais externas produzidas durante o processo de programação. A opção pelo estudo dos processos cognitivos a partir das representações externas envolvidas nestes processos terá seus fundamentos teóricos delineados em seções posteriores deste capítulo.

2.2 ELABORAÇÃO DE UM PROGRAMA COMPUTACIONAL

Em termos gerais, a elaboração de um programa computacional consiste primeiro na ideação de uma tarefa a ser executada pelo programa, e depois na construção de um programa capaz de executar tal tarefa. A ideação da tarefa pode ou não ser realizada por alguém que saiba programar computadores, enquanto a construção do programa necessariamente o é. Este capítulo se restringe à descrição da atividade de um programador na implementação de um programa que realize uma

tarefa previamente definida, deixando de lado a etapa de concepção desta tarefa. Para fins de simplificação, a descrição assumirá uma situação onde um programador individual parte do zero para elaborar um programa completo. Na realidade, os programadores despendem boa parte do seu tempo na modificação de programas já existentes, e frequentemente operam de forma colaborativa. Caberá ao capítulo posterior tratar da atividade de programação em seu caráter coletivo e apresentar o contexto mais amplo desta prática.

Seguindo o modelo da atividade de programação expresso por BROOKS (1975), a descrição dividirá o desenvolvimento de um programa em três etapas principais: apreensão do problema, planejamento e codificação. Tal divisão traz ganhos analíticos, já que permite captar os processos distintos envolvidos na elaboração de um software. No entanto, como em qualquer modelo, corre-se o risco de passar uma imagem excessivamente ordenada da prática descrita, implicando que a mesma requer a execução de tarefas discretas em uma ordem sequencial irreversível. Em situações reais, é comum que as tarefas se interpenetrem, e a execução daquilo que neste texto é descrito como uma etapa posterior pode exigir um retorno a etapas anteriores e vice-versa.

2.2.1 Apreensão do problema

Todo programa computacional começa com uma ideia a respeito de como ele deve se comportar, geralmente na forma de uma descrição da tarefa a ser executada pelo software. Tal descrição pode ser elaborada pelo programador, ou comunicada a ele por outra pessoa, e usualmente é expressa na linguagem cotidiana do diálogo humano - referida como “linguagem natural” no âmbito da programação e da ciência da computação, a fim de distingui-la das linguagens formais utilizada para desenvolver programas computacionais. Tal descrição pode ser tão simples e sucinta como “O programa deve operar o envio de arquivos entre computadores conectados pela Internet”, ou tão complexa como a especificação de intrincados cálculos a serem realizados por um software de computação científica. Da mesma forma, a descrição pode variar no nível de detalhamento da especificação da tarefa a ser realizada. As seções posteriores apresentarão as etapas envolvidas na elaboração de um software tomando por exemplo a construção de um programa cuja descrição da tarefa é a

seguinte: “O programa deve permitir que dois usuários conectados pela internet joguem damas”.

A descrição da tarefa pode ser pensada como o enunciado do problema a ser resolvido pelo programador. Assim sendo, examinar e entender a descrição é a primeira etapa necessária para a resolução de tal problema. Uma característica importante dos problemas de elaboração de software é que eles exigem conhecimentos não apenas no *domínio de conhecimento da programação* e ciências da computação, mas também no *domínio de conhecimento da tarefa* que o programa deve executar: a criação de um software que manipule representações de formas geométricas provavelmente envolverá conhecimentos em geometria, enquanto um programa para correção gramatical de textos implicará conhecimentos linguísticos. Embora exista a possibilidade de que os conhecimentos do domínio da tarefa sejam repassados para o programador juntamente com a descrição da tarefa a ser executada pelo programa, em contextos práticos é sabido que tais descrições muitas vezes omitem certas informações, assumindo tacitamente que o programador já possui certos conhecimentos sobre o domínio da tarefa ao invés de especificá-lo em detalhe (PENNINGTON; GRABOWSKI, 1990, p. 50). Por exemplo, a descrição já citada do programa para jogo de damas assume tacitamente que o programador conhece as regras deste jogo. Como consequência disto, o programador poderá ter de buscar os conhecimentos adicionais necessários durante o processo de entendimento e resolução da tarefa. Daí decorre que a atividade de programação permite - e até mesmo exige - que o programador continuamente aprenda e mobilize novos e variados conhecimentos durante todo o processo de resolução do problema (BROOKS, 1975, p.5).

A etapa de apreensão do problema se encerra quando o programador considera que alcançou um grau suficiente de compreensão da tarefa a ser executada pelo software. Neste ponto, ele entenderá que está pronto para partir para a próxima etapa, que consiste no planejamento da solução a ser implementada. Evidentemente, as etapas posteriores do processo podem revelar insuficiências na compreensão do programador, exigindo um novo esforço de apreensão do problema.

2.2.2 Planejamento

A etapa de planejamento onde o programador cria o plano que guia a elaboração do programa. O processo de programação pode ser visto como um mapeamento[4] dos conhecimentos do domínio da tarefa a ser resolvida pelo programa, até o domínio da linguagem de programação, possivelmente passando por vários domínios intermediários (BROOKS, 1983, p. 54). Sob este ponto de vista, o fim da fase de planejamento é alcançado quando o programador consegue realizar o mapeamento do domínio da tarefa até um domínio intermediário que seja considerado adequado para ser mapeado diretamente para o domínio da linguagem de programação - ou seja, até o plano que guiará o esforço posterior de codificação.

Cabe ressaltar novamente que a descrição a seguir reflete um modelo idealizado da tarefa do programador, informada pela literatura de engenharia de software, por manuais e programação, e por estudos empíricos sobre o tema. Neste modelo, o planejamento é dividido em etapas delimitadas, e realizado através de uma série de técnicas mais ou menos formalizadas que produzem representações materiais do plano (tais como diagramas e escritos). Contudo, os processos de planejamento adotados por programadores em situações reais variam enormemente, sendo balizados tanto pelas preferências e conhecimentos de cada programador, quanto pelas circunstâncias do projeto de software em questão. Na prática é possível encontrar tanto procedimentos de planejamento extremamente minuciosos e formais quanto situações em que o programador não emprega nenhuma técnica estabelecida de planejamento, e não produz nenhum tipo de representação material de seu plano.

O planejamento de programas costuma envolver dois níveis de projeto. O primeiro deles, de caráter mais geral, consiste na elaboração de uma estrutura de sistema composta de vários módulos relacionados, cada um dos quais responsáveis por uma função necessária para a realização da tarefa que o programa deve resolver (PENNINGTON e GRABOWSKI, 1990, p.50). A literatura a respeito de metodologias de projeto de software apresenta vários métodos distintos para realizar essa forma de decomposição modular (JEFFRIES et al, 1981, p.256-257). Também existem várias possibilidades de formas representacionais para a expressão da estrutura modular do programa, sendo particularmente comum o uso de diagramas para este fim.

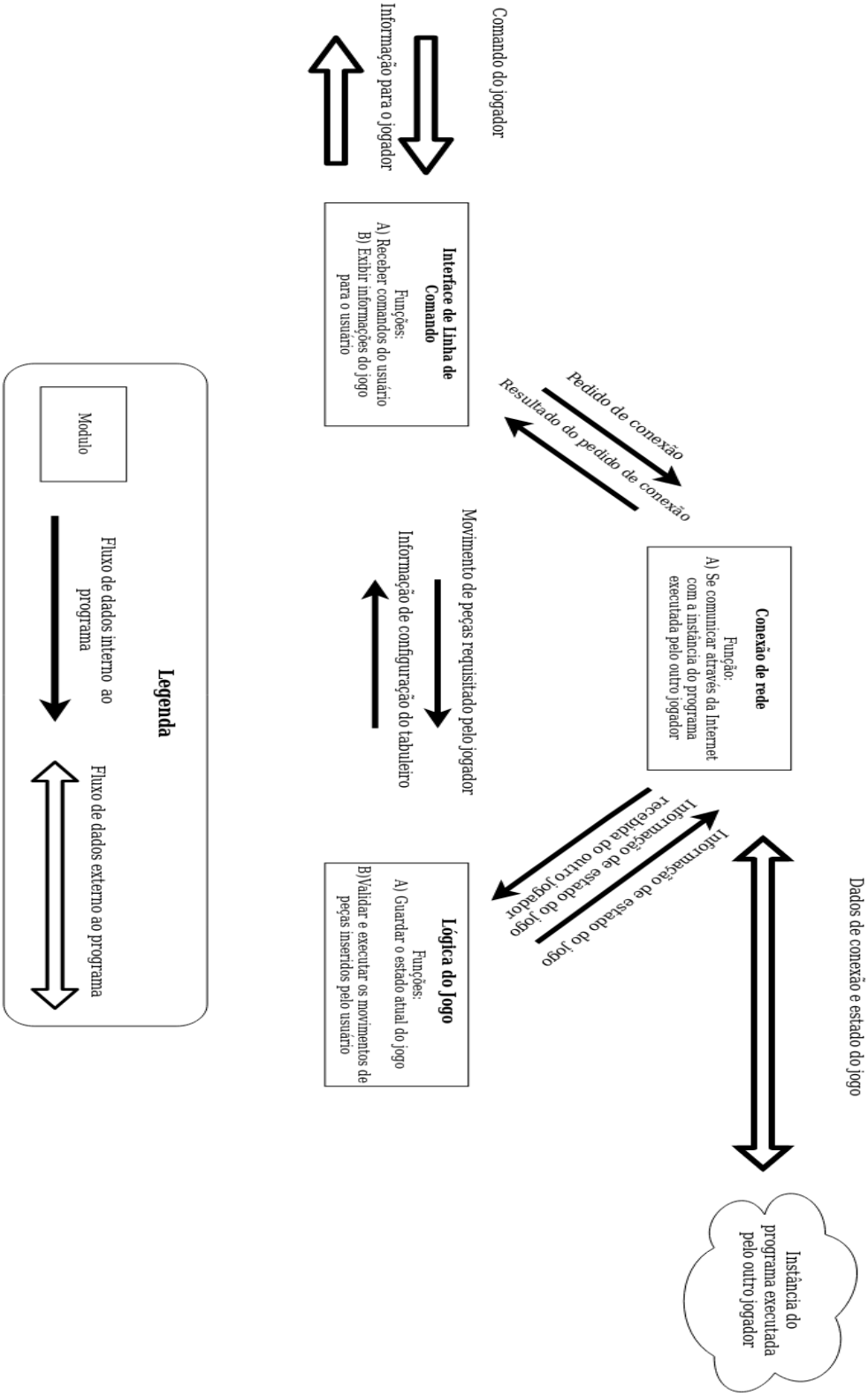
---

[4] O termo mapeamento pode ser entendido provisoriamente como estabelecimento de correspondências entre domínios de conhecimento - ou, mais especificamente, entre representações de objetos e relação oriundos destes domínios. Posteriormente esta noção será tratada em maior detalhe.

Um diagrama da estrutura modular do programa de jogo de damas através da Internet é apresentado na Figura 1 (p.34). A opção de representação adotada neste diagrama enfatiza as funções de cada módulo, os fluxos de dados pelos quais os módulos distintos se comunicam, e os fluxos de dados que compõe os inputs e outputs do programa. A ordenação temporal destes fluxos é omitida, já que a representação apresenta sincronicamente todos os fluxos de informação que ocorrem no sistema.

Diagramas utilizados em projetos de software possuem diferentes graus de complexidade e formalidade, a depender do propósito para os quais são destinados e das inclinações de seus autores. Alguns não passam de esboços criados para demonstrar algum ponto durante uma discussão, enquanto outros são elaborados por meio de linguagens formais de modelagem e servem como documentação de referência para o projeto. A figura encontra-se próximo ao polo mais informal deste espectro.

Figura 1 – Diagrama de estrutura modular.



Fonte: elaborada pelo autor

O segundo nível de projeto envolve uma especificação mais detalhada do funcionamento de cada módulo do sistema, o que implica a especificação de como as informações do domínio da tarefa que o programa resolve são representadas no módulo, e que tipo de operações computacionais são realizadas sobre essas representações. Nos termos usados nas ciências da computação, essa fase envolve a especificação das estruturas de dados e algoritmos implementados no módulo[5].

Um programa de computador usualmente opera sobre informações do domínio da tarefa que ele busca resolver, o que significa que o software deve ter a capacidade de criar alguma representação interna destas informações. Tal representação deve manter as propriedades e relações das informações que são relevantes para a execução da tarefa, e ao mesmo tempo codificar tais informações em símbolos que possam ser armazenados na memória do computador e processados pelo mesmo. Geralmente os símbolos nos quais as informações foram transformadas são agrupados em estruturas de dados que estabelecem algum tipo de relação entre os símbolos do grupo.

O programa de jogo de damas usado como exemplo precisa ser capaz de guardar na memória a posição das peças remanescentes no tabuleiro no momento atual do jogo e receber dos jogadores ordens para movimentar peças. A configuração do tabuleiro poderia ser representada por uma matriz numérica de oito colunas por oito linhas, na qual elementos da matriz que representam células vazias do tabuleiro recebem o valor 0, e elementos que representam células com peças do primeiro jogador recebem o valor 1 e aqueles com peças do segundo jogador recebem o valor 2. Sob essa representação, a diferença de cores entre as peças de cada jogador, como presente num jogo tradicional de damas, é codificada no programa como uma diferença entre números. Por outro lado, as ordens de movimento dos jogadores podem ser inseridas no programa por meio da indicação da posição atual da peça e da posição para qual ela deve ser movida. A figura 2 (p. 36) exemplifica esta representação.

[5] Programas suficientemente simples - ou seja, compostos por um, ou alguns poucos algoritmos - podem ser planejados apenas pela especificação detalhada dos algoritmos, sem a necessidade de dividir sua funcionalidade em diferentes módulos. A decomposição modular é uma técnica de controle de complexidade, empregada apenas quando há um patamar de complexidade que a justifique.

Figura 2 - Configuração inicial de um tabuleiro de damas representada por meio de uma matriz numérica.

1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	2	0	2	0	2	0	2
2	0	2	0	2	0	2	0
0	2	0	2	0	2	0	2

Fonte: elaborada pelo autor

A utilização de uma matriz numérica para representar a configuração de um tabuleiro de damas parece natural, mas certamente não é a única opção concebível. Seria igualmente possível representar a configuração através de uma lista de coordenadas, por exemplo. O ponto fundamental é que *um mesmo conjunto de informações pode ser representado de várias formas diferentes* no interior do programa, e *a escolha da modalidade de representação a ser empregada é dotada de consequências*, já que diferentes formas de representação podem implicar não apenas diferentes *demandas de espaço de armazenamento* do computador, mas também diferentes *custos de processamento* para realização de operações computacionais sobre os dados. Assim, a definição de como as informações relevantes serão representadas no interior do programa constitui um dos mapeamentos entre o conhecimento do domínio da tarefa e o conhecimento de programação que estão no cerne da fase de planejamento.

Nesta etapa do planejamento também são definidos os algoritmos utilizados para operar sobre as estruturas de dados escolhidas. Na definição de Donald Knuth, um algoritmo consiste num “conjunto finito de regras que define uma sequência de operações para resolver um tipo específico de problema” (KNUTH, 1997, p.4-5). Este autor determina ainda uma série de critérios para definição de um algoritmo: [i] Um algoritmo deve terminar após um número finito de passos, [ii] ser precisamente

definido de forma que cada passo possa ser levado adiante de forma não ambígua e rigorosa, [iii] pode ou não possuir inputs, que são quantidades passadas para o algoritmo antes que ele comece a ser executado, e [iv] deve possuir outputs que são quantidades que possuem uma relação especificada para com os inputs (caso estes existam)[6]. Por fim, Knut diz ainda que um algoritmo [v] deve ser construído de forma que todas as operações que ele executa sejam suficientemente básicas a ponto de poderem ser performadas com exatidão e num tempo finito por uma pessoa utilizando papel e lápis.

Assim, a escolha de um algoritmo para resolução da tarefa envolve outro mapeamento entre o domínio da tarefa a ser executada pelo programa e o domínio da programação, dado que o trabalho a ser feito envolve a elaboração de uma *sequência de passos* pelas quais a tarefa pode ser realizada, com a ressalva de que esses passos devem ser passíveis de serem posteriormente implementados como instruções computacionais em uma linguagem de programação.

A escolha de um algoritmo obedece às mesmas considerações que aquelas envolvidas nas estruturas de dados: uma mesma tarefa pode ser resolvida por meio de *diferentes algoritmos*, e a escolha de qual algoritmo será empregado é relevante na medida em que diferentes algoritmos podem implicar diferentes usos de memória do computador, ou diferentes tempos de processamento para resolução da mesma instância da tarefa. De fato, existe uma grande interconexão entre algoritmos e estruturas de dados, já que a escolha de como representar dados tem efeitos diretos sobre as operações que podem ser realizadas sobre os mesmos.

É comum que a formação - ou melhor, a instrução formal - de um programador inclua o aprendizado de várias estruturas de dados consideradas fundamentais, e de vários algoritmos para realizar operações básicas sobre tais estruturas. O objetivo deste ensino não é somente familiarizar o programador com a lógica subjacente à criação de algoritmos computacionais, mas também fornecer algoritmos básicos que podem ser combinados para formação de algoritmos mais complexos. Ou seja, tais conhecimentos permitem que o programador modele os problemas que seus

[6] Algoritmos que não recebem inputs não são apenas concebíveis teoricamente, mas utilizados em aplicações práticas. Um exemplo de uso seria retornar como output o resultado de uma função constante, ou seja, uma função na qual o output é sempre uma quantidade especificada que não varia de acordo com os inputs.

programas devem resolver em termos de estruturas de dados e algoritmos já conhecidos (SKIENA, 2008, p.19).

Mas como o programador especifica os algoritmos a serem utilizados para criar o programa? A elaboração de qualquer programa minimamente complexo exigirá o emprego de alguma forma de *decomposição do problema* em pedaços menores que podem ser tratados *separadamente*. Estudos de psicologia da programação costumam caracterizar o procedimento utilizado para este fim como uma técnica de *refinamento passo a passo* (PENNINGTON e GRABOWSKI, 1990, p.50). Algumas pesquisas empíricas apresentam evidências de que esta técnica é de fato utilizada no projeto de software (JEFFRIES et al, 1981; BROOKS; 1975). Além disso, o refinamento passo a passo é citado como a técnica padrão de planejamento em grande parte da literatura sobre metodologias de projeto de software (JEFFRIES et al, 1981, p.256-257).

A elaboração de algoritmos por meio do refinamento passo a passo inicia-se pela definição da série de etapas necessárias para realizar a tarefa que o algoritmo deve operar, e é levada adiante por meio de sucessivas revisões nas quais cada etapa é novamente decomposta em subetapas num nível de detalhamento cada vez maior, até que o programador considere que o algoritmo (ou uma parte dele) está suficientemente especificado para poder ser implementado diretamente em uma linguagem de programação.

Para exemplificar o processo de refinamento passo a passo, será realizada a decomposição da função B do módulo “Lógica do Jogo”. O ponto de partida da decomposição é a descrição desta função na Figura 1: *Validar e executar os movimentos de peças ordenados pelo usuário*.

O diagrama e a etapa de escolha das representações de informações internas ao software apresentam alguns conhecimentos importantes para a decomposição da função. Em primeiro lugar, sabe-se que o módulo Lógica do Jogo recebe o comando de movimento de peças do módulo Interface, e que esse comando é representado por um par de coordenadas indicando a posição atual da peça, e a posição para qual ela deve ser movida. Além disso, sabe-se que o módulo Lógica do Jogo envia informações de configuração do tabuleiro para o módulo Conexão de Rede, para que este módulo comunique o movimento realizado à instância do programa sendo executada pelo

outro jogador. Assim, é possível esboçar uma descrição inicial do algoritmo em questão.

Input: Coordenadas da peça a ser movida e da posição para qual se deve movê-la.

Etapas:

1. Validar o movimento ordenado pelo jogador
2. Executar movimento
3. Enviar informação do estado do jogo para o módulo Conexão de Rede.

Esse primeiro passo da decomposição envolve basicamente a especificação dos inputs e outputs, e a divisão em etapas das subfunções especificadas na descrição da função. A próxima etapa envolve a especificação de como realizar cada uma destas subfunções, o que implica a aplicação de conhecimentos sobre as regras do jogo de damas.

A primeira subfunção verifica se o movimento ordenado pelo jogador é válido diante das regras do jogo de damas e da situação atual do tabuleiro. Como a situação atual do tabuleiro é guardada pelo módulo Lógica do Jogo, assume-se que a função em questão não precisa realizar procedimentos adicionais para obter essa informação.

1. Validar o movimento ordenado pelo jogador - Recebe por input as coordenadas das casas de origem e destino da peça a ser movimentada:
    a. Checar se a primeira coordenada indicada contém uma peça, e se essa peça é do jogador que ordenou o movimento.
        i. Se as duas condições são verdadeiras, continuar para o passo 1.b.
        ii. Se uma das condições não é verdadeira, retornar uma informação indicando que o movimento é inválido e parar a execução desta função.
    b. Checar se o movimento tencionado é para uma casa de destino que é adjacente e em diagonal a casa onde está posicionada a peça a ser movida.

i. Se todas as condições são verdadeiras, continuar para o passo 1.c.
ii. Se alguma das condições é falsa, retornar uma informação indicando que o movimento é inválido e parar a execução desta função.

c. Checar se a casa de destino é posicionada à frente da casa na qual está posicionada a peça a ser movida. Checar se a casa de destino está vazia.
   i. Se todas as condições são verdadeiras, o movimento é válido. Continuar para a subfunção 2 para proceder a execução do movimento.
   ii. Se alguma das condições é falsa, continuar para o passo 1.d.
d. Checar se a casa de destino é ocupada por uma peça adversária. Checar se a próxima casa após a casa de destino, na mesma diagonal do movimento, está vazia.
   i. Se todas as condições são verdadeiras, o movimento é válido. Continuar para a subfunção 2 para proceder a execução do movimento.
   ii. Se alguma das condições é falsa, retornar uma informação indicando que o movimento é inválido e parar a execução desta função.

Os outros passos da tarefa a ser desempenhada pelo módulo podem ser especificados através do mesmo procedimento. A partir daí o planejamento pode tomar vários rumos. O programador pode desejar especificar o processo de recebimento de input e o passo 3, o que implica especificar as interfaces de comunicação entre os módulos. Da mesma forma, ele pode optar por especificar num nível ainda maior de detalhe os passos 1 e 2[7].

7 O leitor familiarizado com o jogo de damas poderá notar que o planejamento realizado até agora não comporta todas as regras tradicionais do jogo. Por exemplo, a transformação de peças simples em damas - um componente crucial das regras - não é prevista nem pelo algoritmo esboçado e nem pela descrição de como o software deve representar as peças e o tabuleiro de jogo. Uma especificação incompleta como esta poderia ser elaborada por um programador dotado de insuficiente conhecimento sobre o domínio da tarefa que seu programa deve realizar.

Cabe observar que algumas pesquisas sugerem que a caracterização do planejamento de programas como refinamento passo a passo apresenta uma visão excessivamente ordenada do trabalho dos programadores. Existem evidências de que o processo de planejamento não segue o curso unilateral que vai de um plano mais abstrato - isto é, com menos detalhes - até um plano mais concreto - ou seja, especificado e detalhado - dado que o trabalho nos níveis mais abstratos pode ter implicações ou para os níveis concretos, e vice e versa, exigindo que o planejador "salte" constantemente entre diferentes níveis de abstração (PENNINGTON e GRABOWSKI, 1990, p. 50-51). De fato, algumas metodologias de projeto de software optam por começar a decomposição da tarefa no caminho inverso realizado aqui, especificando primeiro os componentes num nível mais detalhado, de forma a identificar possíveis limitações implicadas por estes componentes para o projeto da estrutura geral do sistema (JEFFRIES et al, 1981, p.257)[8].

2.2.3 Escrita do código fonte

Uma vez que o programador considere que seu planejamento está concluído, ele pode começar a etapa de escrita do código fonte, na qual o nível mais detalhado da formulação do plano - se houver alguma formulação explícita - é implementado em uma linguagem de programação específica (PENNINGTON e GRABOWSKI, 1990, p.52).

Cada programador possui seu método de trabalho para codificação (e alguns não possuem método algum). De forma geral, o programador procederá a codificação escrevendo trechos de código para expressar o plano de seu algoritmo. Para verificar se o código escrito produz o efeito esperado, o programador continuamente executará a versão parcial do programa observando o comportamento dos trechos que ele escreveu. Caso as execuções revelem que um determinado trecho de código não funciona corretamente, o programador buscará entender o motivo da falha e corrigir o código de acordo, reiniciando o ciclo de codificação e execução para testes. Após

8 Outro ponto a ser destacado ainda é que a representação passo a passo constitui uma forma analítica de planejamento de algoritmo. Outras técnicas para resolução de problemas de programação podem apresentar diferentes ênfases, como o uso extensivo de construção de protótipos, soluções de tentativa e erro etc. (PENNINGTON e GRABOWSKI, 1990, p.51-5).

garantir que o seu programa possui o comportamento desejado, o programador pode optar por alterar ou reescrever determinados trechos de código para melhorar a performance ou manutenibilidade dos mesmos.

Os conhecimentos mobilizados durante a codificação são variados, e incluem tanto saberes gerais sobre como processos de manipulação de símbolos são executados por um computador, quanto conhecimentos específicos a respeito da linguagem de programação e demais tecnologias utilizadas para a elaboração do código fonte.

Durante todo este processo, é possível que o programador mobilize uma série de ferramentas para auxiliar seu trabalho. Editores de texto para programação possuem funcionalidades para facilitar a escrita de código fonte, facilitar a leitura e compreensão do código, e até mesmo analisar o comportamento do mesmo durante a execução. Da mesma forma, diversas ferramentas podem ser utilizadas para realização de testes que verificam se o programa se comporta conforme o esperado. Tais ferramentas ocupam o papel central no processo de codificação, e a proficiência no uso das mesmas é um componente essencial no conjunto de habilidades de um programador.

À primeira vista, o processo de codificação pode se assemelhar a um processo de tradução, já que a descrição da tarefa a ser executada pelo programa, ou a especificação do algoritmo, é transformada no código fonte escrito em linguagem de programação. No entanto, a metáfora da tradução talvez não seja a mais adequada, já que a atividade do programador não envolve apenas a *manutenção de um significado* através de diferentes linguagens. Preocupações como a performance do programa em tempo de execução e uso de memória, a capacidade do mesmo de lidar adequadamente com erros, e uma miríade de outras questões associadas às tecnologias utilizadas para construção do programa são levadas em conta durante a elaboração do código. Da mesma forma, a manutenibilidade do código é uma preocupação central no desenvolvimento de aplicações utilizadas em contextos reais: o código deve não apenas ser capaz de *executar corretamente e eficientemente a tarefa* para o qual foi criado, mas também ser escrito de forma a *facilitar a tarefa dos programadores* que futuramente serão incumbidos de entendê-lo e modificá-lo. Afinal, um programa que permanece em uso frequentemente precisará de correções e até

mesmo novas funcionalidades, e as decisões que o programador toma durante a elaboração do programa podem tanto facilitar quanto dificultar modificações futuras.

Além disso, cabe destacar que a codificação não se resume à implementação de um plano pré-estabelecido. Mesmo nos casos onde a etapa de planejamento é realizada com grande rigor e um plano detalhado é produzido, a etapa de codificação pode, e costuma, implicar *alterações e revisões do plano* conforme a sua implementação revela restrições, impedimentos, e mesmo furos no plano, que não foram previstos durante o planejamento. Já em contextos onde o planejamento prévio é realizado com menor rigor - ou de forma menos explícita - o trabalho de escrever o código se sobrepõe ainda mais ao esforço de delinear o algoritmo e a estrutura do sistema.

Quando a codificação é finalizada obtém-se o produto final do trabalho do programador: o código fonte de um programa que pode ser compilado e executado para realizar a tarefa para o qual foi concebido. Terminar a codificação é criar uma representação na qual o plano para realização da tarefa, conforme descrito na linguagem do domínio da mesma, foi convertido em código fonte na linguagem do domínio da programação, ou seja, em sequências de expressões que designam instruções de manipulação de símbolos passíveis de serem executadas por um computador.

2.3 PROGRAMAÇÃO E A TEORIA DE PROCESSAMENTO DE INFORMAÇÃO

Terminada a descrição do trabalho de criação de um programa, torna-se possível retomar o tema chave deste capítulo, que diz respeito à caracterização da programação como uma atividade cognitiva.

Alguns estudos sobre os aspectos cognitivos da programação buscaram seu quadro conceitual em uma teoria da cognição que, assim como a própria atividade de programação, emergiu a partir do advento dos computadores digitais[9]. Um dos escritos seminais deste arcabouço teórico é o livro *Human Problem Solving*, publicado em 1972 pelos pioneiros da inteligência artificial Allen Newell e Herbert A. Simon. A explicação da capacidade humana de resolução de problemas expressa nesta obra

[9] Exemplos da aplicação da teoria de processamento de informação à atividade de programação podem ser encontrados no de ORMEROD, 1990 e BROOKS, 1975.

parte do postulado de que o pensamento humano é uma modalidade de processamento de informação. Sob esta perspectiva, o pensamento é visto como um processo que recebe como inputs informações - oriundas do ambiente, do estado psicológico do indivíduo, ou da memória do mesmo - e aplica uma série de transformações a estes inputs, de forma a gerar sequências de “estados de informação”. No que tange a resolução de problemas, o processamento realizado envolveria gerar vários estados de informação até encontrar um que satisfaça o problema em questão (EASTMAN, 1969, p.669-670). Como é evidente, esse tratamento da cognição humana sugere que a mesma possui algum grau de similaridade com a operação de computadores digitais. De fato, Newell e Simon postulam que tal teoria consiste numa abstração da atividade efetuada pelos computadores, de forma a manter o caráter essencial do processamento de informação e descartar as especificidades de como esse processo é realizado pelas máquinas (NEWELL e SIMON, 1972, p.5).

Computadores são sistemas de processamento de símbolos - isto é, sua atividade consiste na aplicação de operações formais sobre expressões formadas por símbolos, de forma a transformá-las em outras expressões do mesmo tipo. Daí decorre que o modelo da cognição humana construído à imagem destes dispositivos se baseia na suposição de que entre os humanos a resolução de qualquer problema minimamente complexo é sempre operada por meio da manipulação de símbolos - presumivelmente realizada pelo cérebro, que ocuparia o lugar de processador central no aparato mental humano (NEWELL e SIMON, 1972, p.72). Neste contexto, o conjunto de problemas tratados em *Human Problem Solving* - que inclui a prova de teoremas lógicos, o xadrez e uma modalidade aritmética na qual letras substituem os dígitos numéricos - foram selecionados justamente porque sua resolução necessariamente envolve a manipulação de símbolos.

De forma geral, os modelos cognitivos da programação construídos com base na teoria de processamento da informação objetivam identificar como os programadores representam mentalmente os problemas de programação - e também os conhecimentos necessários para resolver esses problemas - e quais operações eles utilizam para transformar a representação do estado inicial do problema na representação da solução desejada. Alguns dos estudos identificados possuem um caráter primariamente teórico (ORMEROD, 1990), enquanto outros se baseiam na

observação empírica do trabalho de programadores em contextos laboratoriais controlados (BROOKS, 1975). O objetivo mais ambicioso deste tipo de estudo seria o agenciamento dos modelos cognitivos elaborados em sistemas computacionais capazes de gerar programas de forma automatizada (BROOKS, 1975, 141-143). Este último ponto exemplifica a ênfase simulacional que perpassa a agenda da teoria de processamento de informação, afinal, originários do mesmo campo científico que as disciplinas de inteligência artificial, os adeptos desta teoria sempre tiveram a criação de simulações computacionais de comportamentos cognitivos não apenas como uma ferramenta de pesquisa, mas também como um critério de validação das inferências teóricas. Segundo esta perspectiva teórica, um bom modelo de um comportamento cognitivo deve descrever as manipulações de símbolos através das quais o comportamento é realizado em um nível de detalhe tal que um computador digital (ou outro processador similar) seja capaz de executá-las (NEWELL e SIMON, 1972, p.11).

Tais estudos certamente possuem méritos importantes. As pesquisas deste tipo que se baseiam em observações empíricas produzem descrições altamente detalhadas e minuciosas das atividades de programadores - rigor descritivo que provavelmente deriva do objetivo de elaboração de um modelo cognitivo suficientemente bem especificado para ser convertido em um programa computacional. Além disso, o caráter esquemático dos modelos produzidos não é desprovido de valor analítico, já que auxilia na identificação dos diferentes processos cognitivos envolvidos em diferentes etapas do processo de elaboração e manutenção de um programa computacional.

No entanto, seus limites também são evidentes. Em primeiro lugar, se levarmos a sério a ideia de elaboração de uma simulação computacional da atividade do programador como critério de validação, podemos acusar estes estudos de terem falhado em seus próprios termos. Afinal, até a presente data este esforço não logrou produzir um sistema capaz de operar automaticamente a conversão de tarefas arbitrárias informalmente descritas em programas computacionais capazes de executar tais tarefas.

Em segundo lugar, é possível argumentar que a atividade de programação é precondição do processamento simbólico, e, portanto, não pode ser reduzida a ele. Para desenvolver este argumento é necessário introduzir a ideia de sistema formal.

A noção de *manipulação de símbolos* implica distinção entre um *mundo de fenômenos* e um *sistema formal* que se baseia na *codificação* de elementos do mundo dos fenômenos em símbolos. Uma vez que algo é codificado como um símbolo, torna-se possível aplicar operações formais sobre este símbolo, ou seja, manipulá-lo apenas com referência a sua forma, sem que seja necessário interpretar seu significado. Em um sistema formal a manipulação das expressões simbólicas sempre resulta em outras expressões simbólicas, e essas novas expressões podem ser interpretadas como se significassem algo sobre os fenômenos originalmente codificados em símbolos (HUTCHINS, 1995, p.359-360). É através deste expediente que regularidades do mundo físico podem ser codificadas em equações formais que podem ser operacionalizadas para gerar previsões sobre acontecimentos no mundo.

Um computador digital é um processador de símbolos definido por um sistema formal que estabelece quais operações o computador é capaz de executar. Da mesma forma, uma linguagem de programação é um sistema formal que pode ser utilizado para dar instruções de execução de operações para computadores. Assim, fica evidente que atividade do programador é em grande parte definida pela lida com sistemas formais. No entanto, também fica claro que tal atividade é logicamente anterior à manipulação de símbolos. Afinal, o trabalho do programador é *codificar elementos do mundo dos fenômenos em símbolos* da linguagem de programação, e definir quais operações disponibilizadas pela linguagem serão aplicadas sobre estes símbolos. O programador opera precisamente na dobra entre o sistema formal e o mundo dos fenômenos, transformando em expressões simbólicas coisas que por si só não possuem um caráter formal pré-estabelecido. Só então torna-se possível que o computador execute o processamento simbólico definido pelo programador.

Seja como for, mesmo que nem estes pontos e nem as inúmeras diferenças entre a mente humana e computador - que aliás têm sido reconhecidas pelos próprios cognitivistas, à exemplo de Howard Gardner (1996) - fossem levadas em conta, ainda assim a teoria de processamento da informação dificilmente serviria aos propósitos deste trabalho.

Ao definir a cognição como processamento simbólico realizado por um processador central, os teóricos de processamento de informação abraçaram uma agenda de pesquisa voltada primariamente para a elucidação das representações através das quais o conhecimento é armazenado e transformado por este

processador, e das operações que o mesmo é capaz de realizar (Gardner, 1996). Além disso, como o sistema cognitivo composto pelo processador central foi localizado no interior do corpo, ou mais especificamente do crânio, a unidade de análise da cognição adotada passou a ser a dos eventos simbólicos internos a um indivíduo (HUTCHINS, 2010).

Mas escolher o que enfocar é também escolher o que não enfocar, e no caso dos cognitivistas da teoria de processamento de informação, a opção de enfocar os processos tidos como centrais na cognição levou a uma negligência deliberada do estudo dos processos considerados periféricos - tais como a percepção e os aspectos do sistema motor - e também dos impactos cognitivos da emoção e dos contextos ambiental, sociocultural e histórico (GARDNER, 1996, p.56-57). Em particular, a escolha de desenfatizar os aspectos contextuais da cognição foi em grande parte guiada pela unidade de análise adotada, já que a investigação da cognição como um processo interno e individual deixa pouca margem para este tipo de consideração.

Os objetivos desta dissertação não incluem a investigação da natureza dos processos internos da mente de um indivíduo. Pelo contrário, a questão de fundo que anima esta pesquisa diz respeito ao papel das *relações sociais* e dos *artefatos materiais* nos *processos cognitivos*. Para este fim, faz-se necessária uma perspectiva teórica que coloque no centro de seu modelo justamente aquilo que os teóricos de processamento de informação optaram por negligenciar - isto é, os contextos nos quais, e através dos quais, ocorrem os processos cognitivos.

2.4 ECOLOGIA COGNITIVA

A teoria da cognição a ser mobilizada nesta dissertação tem suas raízes no trabalho de Gregory Bateson, autor cujas pesquisas sobre cognição tomaram um caminho bem distinto daquele trilhado pela ortodoxia das Ciências Cognitivas, que na época de atividade deste autor era composta principalmente pela teoria de processamento de informação. De fato, as noções de Bateson sobre a mente alcançam uma formulação coerente no final da década de 60 (BATESON, 1972, p.13), e *Steps to an Ecology of Mind*, livro que introduz estas ideais conforme expressas ao longo da vasta obra do autor, foi publicado em 1972, mesmo ano de publicação do já

citado *Human Problem Solving*, de Newell e Simon. Embora contemporâneos, ambos os livros expressam visões muito distintas sobre os fenômenos mentais.

A abordagem cibernética da mente, da qual Bateson é um dos principais formuladores, também emprega a noção de informação - e mesmo o termo processamento de informação - para tratar da cognição. No entanto, a diferença fundamental entre a abordagem cibernética e a abordagem de processamento de informação é que enquanto a última concebe a mente como formada por eventos simbólicos internos (ao corpo, e de forma mais estrita ao crânio), a primeira entende que a mente é composta por circuitos de informação que se estendem através do corpo e ao longo do ambiente (HUTCHINS, 2010).

O contraste entre as teorias pode ser apreciado também pelo entendimento do conceito de informação e processamento empregado por Bateson. Para este autor, informação - em sua unidade mais básica - é diferença que faz diferença. A ideia aqui é que diferença é sempre uma relação abstrata entre uma coisa e outra, e o mundo é pleno destas relações. Há um número potencialmente infinito de diferenças entre qualquer par de coisas, por mais simples que elas sejam, e um número igualmente amplo de diferenças entre os elementos que compõe uma mesma coisa - sejam estes elementos partes que podemos enxergar ou manusear, sejam eles as moléculas que compõe as substâncias de que as coisas são feitas (BATESON, 1972, p.321). Mas apenas quando uma diferença é incorporada em algum tipo de circuito e produz outras diferenças no interior deste circuito é que ela se torna informação (BATESON, 1979, p.110).

Se o assunto parece por demais abstrato, cabe tratar de um exemplo concreto. Consideremos o caso de um mapa e do território por ele representado. Cada ponto do território possui inúmeras diferenças em relação a qualquer outro ponto - diferenças de relevo, altitude, tipo de solo, vegetação etc. No entanto, nenhum mapa contém todas essas diferenças, dado que o criador do mapa é obrigado a selecionar apenas algumas destas diferenças para inscrever na representação. Ou seja, apenas algumas diferenças presentes no território são informações, tornando-se diferenças no mapa (BATESON, 1972, p. 320-321).

Esse processo, por meio do qual uma diferença é transformada em outra diferença, é central para o pensamento de Bateson, já que estas transformações consistem justamente em processamento de informação. No exemplo apresentado,

essas transformações acontecem ao longo dos complexos circuitos por meio dos quais as diferenças do território são codificadas no mapa. Tais circuitos - assim como em qualquer outro caso minimamente complexo de processamento de informação - não são compostos apenas por um único processo de transformação de diferenças, mas sim por várias longas cadeias de codificação.

De fato, as diferenças do território passam por vários lugares antes de irem parar no mapa. Elas primeiro são observadas pelo olhar do cartógrafo, e este ato de percepção por si só implica um complexo circuito de processamento de informação. Os olhos do cartógrafo são estimulados pelos raios de luz refletidos pelas superfícies do ambiente, raios estes recebidos pelas células fotorreceptoras da retina, que por sua vez se encarrega de transformar as diferenças de luz em diferentes reações físicas e químicas que atravessam os circuitos neurais, e (presumivelmente) acabam por compor uma imagem no pensamento do cartógrafo. Da mesma forma, os diversos instrumentos de mensuração utilizados pelo cartógrafo também captam diferenças no ambiente, e as transformam em símbolos numéricos, tais como mensurações de altitude. Tanto os circuitos de processamento de informação que têm lugar através da percepção e do pensamento do cartógrafo, quanto aqueles que perpassam os instrumentos de mensuração, fazem parte dos circuitos por meio dos quais as diferenças no território são codificadas em diferenças no mapa.

O exemplo permite ver que a noção de processamento de informação em Bateson é bastante abrangente. De fato, o autor se propõe a empregá-la para descrever fenômenos tão aparentemente díspares como a operação da retina, os processos genéticos que definem o padrão de desenvolvimento de um organismo, a transmissão de informações em processos evolutivos, o funcionamento de máquinas dotadas de capacidade de autorregulação, entre outros. Sob essa perspectiva, a manipulação de símbolos em sistemas formais é apenas um caso específico de processamento de informação, dentre muitos outros existentes.

2.4.1 Programação sobre a luz da teoria cibernética da mente

Cabe questionar então como o quadro teórico de Bateson se aplica à atividade do programador. Anteriormente, foi observado que a programação pode ser descrita como um mapeamento da descrição de uma tarefa, expressa na linguagem do

domínio da tarefa, para um código escrito em linguagem de programação. Sob uma perspectiva batesoniana, esse mapeamento é um processo no qual informações na descrição da tarefa, combinadas a informações advindas de outras fontes, são transformadas em informações expressas no código fonte.

A descrição anterior da atividade de elaboração de programas buscou justamente delinear alguns dos processos de transformação de diferenças que compõem esta atividade: [a] a decomposição modular dos aspectos principais do processo de resolução da tarefa que o programa deve realizar, [b] a elaboração de um diagrama de sistema que expressa estes módulos e suas interações, [c] a especificação dos algoritmos relativos a cada um destes módulos através de refinamento passo a passo (ou outra técnica equivalente), [d] a representação de elementos do domínio da tarefa como símbolos e estruturas de símbolos, [e] a codificação dos algoritmos e estruturas de símbolos obtidas em uma linguagem de programação. Estes processos consecutivos revelam uma *cadeia de transformações representacionais*, onde a cada etapa as *diferenças que compõem uma representação*, acrescidas de informações provindas de outras fontes (como conhecimentos de computação e do domínio da tarefa), são *transformadas em diferenças em uma outra representação*. Na metáfora cartográfica, em cada etapa uma representação é tomada como território, e mapeada em uma nova representação, que por sua vez se tornará território na etapa seguinte. De fato, esta metáfora se revela ainda mais adequada diante da caracterização da programação como um processo de *mapeamento* entre dois domínios distintos.

Neste ponto, tornam-se evidentes algumas das vantagens que podem ser obtidas ao aplicar o modelo cognitivo de Bateson para tratar do caso em questão. Em primeiro lugar, a concepção de cognição batesoniana parece suficientemente ampla para dar conta de diversos processos envolvidos na programação - sejam estes processos caracterizáveis como manipulações simbólicas ou não. Afinal, para dar conta de uma atividade cognitiva que opera na passagem do mundo dos fenômenos para um sistema formal, faz-se necessária um conceito de cognição mais abrangente do que o modelo de processamento simbólico.

Em segundo lugar, o arcabouço teórico batesoniano é capaz de dar conta do papel que os elementos materiais externos ao corpo desempenham na cognição - algo essencial na investigação de uma atividade como a programação, que é definida

pelo uso de artefatos tecnológicos. Essa capacidade do modelo de mente proposto por Bateson decorre de duas de suas características centrais: a) a noção de mente como um agregado de componentes em interação e b) a ideia de que a delimitação daquilo que faz ou não parte deste agregado deve sempre levar em conta o contexto do comportamento cognitivo em questão.

A noção da mente como agregado de componentes em interação parte do princípio que a *mente* não é algo que deriva de *propriedades intrínsecas* dos elementos que a compõem, mas da forma como estes elementos se *relacionam*. Este princípio objetiva prover uma explicação de como os fenômenos mentais podem emergir a partir da matéria, sem que seja necessário recorrer a algum tipo de idealismo - ou, no limite, a explicações sobrenaturais. Neste ponto, o quadro teórico de Bateson apresenta uma grande afinidade com as noções expostas por Marvin Minsky (1986), cognitivista da área de inteligência artificial que concebe a mente como uma espécie de sociedade - ou seja, um fenômeno que emerge a partir das relações entre elementos que por si só não apresentam as características deste fenômeno. Bateson (1979, p.92-93) assinala que é possível que uma parte dos elementos que compõem uma mente sejam eles mesmos dotados de todas as características mentais, mas sempre será possível decompor estes elementos até encontrar um nível em que as partes componentes por si só não possuem caráter mental.

Considerada separadamente, a ideia da mente como agregado de partes em interação não produz um quadro teórico capaz de dar conta do papel que os elementos materiais externos ao corpo desempenham na cognição. De fato, seria possível utilizá-la para defender um modelo estritamente internalista da mente. No entanto, essa noção adquire um novo significado quando somada à ideia de que a investigação dos fenômenos mentais deve traçar sua *unidade de análise* de forma *contextual*, levando em conta todos os *circuitos de processamento de informação relevantes* para o fenômeno a ser explicado. Bateson (1972, p.325) ilustra este princípio com o proverbial exemplo do cego e sua bengala: a bengala poderia ser considerada parte da mente do cego quando o fenômeno a ser explicado é o comportamento de locomoção do mesmo, dado que através da bengala circulam alguns dos fluxos de informação essenciais para que o cego se locomova. Por outro lado, se o comportamento a ser estudado é o de um cego parado, sem fazer uso da bengala, então a bengala não faz parte de nenhum circuito relevante e não deve ser

considerada como parte da mente em questão. Operando a junção dos princípios citados, podemos perceber que a bengala, embora constitua um artefato material externo ao corpo, pode ser caracterizada em algumas circunstâncias como um dos componentes de uma mente - tudo depende da forma como ela interage com os outros componentes.

Os parágrafos anteriores nos levam ao seguinte princípio: o *critério* para definir se um elemento pode ser considerado como *componente de um processo cognitivo* não deriva da *localização* deste elemento (ou seja, se ele encontra-se ou não dentro do corpo ou do crânio), mas sim da *função* que o mesmo desempenha no processo. Este critério funcional está na base dos recentes conceitos externalistas da cognição, que se opõem à ideia da mente como algo encerrado no aparato neural. Exemplos deste tipo de teoria incluem não apenas externalismos originados na filosofia cognitivista, como aquele exposto por Andy Clark e Chalmers (1998), mas também a teoria da cognição distribuída elaborada pelo antropólogo cognitivo Edwin Hutchins (1995, 2010). Esta última teoria apresenta evidente influência batesoniana.

Coube a Hutchins o trabalho de tirar as consequências dos princípios citados e integrá-los num quadro teórico e metodológico que propicia o estudo etnográfico da *cognição como um processo socialmente distribuído*. Uma das ideias basilares da abordagem deste autor é a seguinte: uma vez que se concebe que as atividades cognitivas são operadas por sistemas cognitivos funcionais (ou, na linguagem de Bateson, mentes) compostos por pessoas e artefatos materiais, é possível observar diretamente - ou seja, etnograficamente - o funcionamento destes sistemas cognitivos através da análise das representações artefatuais que o sistema mobiliza e das operações que ele realiza para transformar representações em outras representações. Ou seja, essa perspectiva possibilita observar aqueles processos cognitivos que ocorrem fora dos limites do corpo do indivíduo, mas que ainda assim compõem o sistema cognitivo no qual o mesmo está situado. O argumento central é que, na medida em que estes processos diretamente observáveis podem revelar muito sobre a organização e operação do sistema cognitivo a ser analisado, é possível apresentar uma explicação esclarecedora deste sistema e, sobretudo, do caráter cognitivo de seus aspectos materiais e sociais, sem que seja necessário investigar ou supor algo sobre os processos cognitivos internos aos indivíduos que participam do mesmo (HUTCHINS, 1995, p.128-129).

A descrição da atividade de programação realizada na primeira parte deste capítulo baseou-se na proposta metodológica de Hutchins. O *sistema cognitivo* sendo abordado na descrição *não se resume ao programador,* uma vez que é composto também pelos diferentes *artefatos* e *modalidades representacionais* que ele utiliza para realizar o mapeamento entre o domínio da tarefa e o domínio da programação. Ao longo da descrição, buscou-se supor o mínimo possível sobre os processos que ocorrem no interior da mente do programador, e enfocar em detalhe as diferentes representações materiais mobilizadas, e a maneira como é elaborada uma cadeia de mapeamentos através da transformação de representações.

A descrição anterior enfocou sobretudo os artefatos e modalidades representacionais utilizadas no desenvolvimento de um programa. Uma hipótese central desta dissertação é que tais elementos materiais têm efeitos concretos sobre as dinâmicas cognitivas da atividade de programação. Para averiguar esta hipótese, na seção seguinte será examinado o papel desempenhado pelas linguagens de programação, ferramentas centrais para os programadores. Tal exame permitirá também demonstrar que a cadeia de mapeamentos que constitui a atividade de programação não termina quando o programador acaba de elaborar o código fonte, mas se estende até os processos operados nas entranhas da máquina.

2.4.2 A função cognitiva das linguagens de programação

A descrição anterior da atividade do programador parou no momento em que o programador dava por encerrada a codificação. Um ponto fundamental omitido é que em geral, a codificação é uma etapa marcada por ciclos de escrita do código, testes do programa para verificar se o código escrito produz o comportamento desejado, e modificação do código de acordo com o resultado dos testes.

Para testar um programa é necessário executá-lo. Durante a codificação tudo que o programador tem são arquivos de texto contendo o código fonte do programa. Contudo, tais arquivos não podem ser diretamente executados por um computador, já que computadores são capazes de executar apenas programas definidos em *linguagem de máquina***,** que consiste numa linguagem na qual as instruções a serem executadas pelo computador são expressas por meio de códigos numéricos (usualmente representados de forma binária ou hexadecimal).

De fato, os primeiros computadores eletrônicos eram programados exclusivamente através de linguagem de máquina. No entanto, gradualmente foram sendo desenvolvidas *linguagens de alto nível*, e hoje virtualmente todos os programas são feitos utilizando este tipo de linguagem. As linguagens de alto nível permitem designar operações a serem executadas pelo computador através de expressões mais abstratas do que os códigos de instruções computacionais da linguagem de máquina, de forma a facilitar o trabalho dos programadores.

O termo *abstração* aqui é usado em um sentido bastante geral. Nesta acepção, dizer que A é uma abstração de B implica que A é algum tipo de modelo ou representação de B que suprime alguns detalhes do mesmo (RAPAPORT, 2020, p.552-553). Assim, algumas das vantagens proporcionadas pelo uso de linguagens de alto nível ao invés de linguagens de máquinas derivam do fato de que as primeiras permitem livrar o programador do encargo proporcionado por certos detalhes e especificidades característicos das últimas.

Um exemplo de vantagem deste tipo deriva do fato de que as linguagens de alto nível abstraem as especificidades das diversas linguagens de máquina existentes. Dois computadores construídos a partir de diferentes arquiteturas computacionais terão diferentes linguagens de máquina, fazendo com que um programa expresso na linguagem de máquina de um deles não possa ser executado pelo outro. No entanto, o código de um programa elaborado numa linguagem de alto nível pode ser traduzido em vários códigos binários específicos para diferentes linguagens de máquina. Isto permite ao programador escrever um único código que pode ser executado por muitos computadores diferentes, algo que não é possível em linguagem de máquina.

Outra abstração com vantagens similares é a seguinte: linguagens de alto nível costumam agregar várias instruções de linguagem de máquina em uma “instrução composta”, que designa uma operação computacional mais complexa. Portanto, o programador que usa uma linguagem de alto nível constrói o seu programa a partir de blocos básicos mais potentes do que as “pequenas peças” disponibilizadas pelas linguagens de baixo nível.

Contudo, a função cognitiva das linguagens de alto nível não repousa apenas sobre sua capacidade de livrar o programador do intrincado nível de detalhamento do código exigido pelas linguagens de máquina. Uma das principais vantagens é que o

uso de linguagens de alto nível demanda *habilidades cognitivas diferentes* daquelas demandadas pelas linguagens de máquina.

Enquanto as linguagens de máquina são de natureza estritamente numérica, com códigos compostos exclusivamente por sequências de números binários ou hexadecimais, as linguagens de alto nível possuem um caráter textual. Nessas linguagens, as operações computacionais que compõem os blocos básicos disponibilizados para o programador são representadas por expressões compostas por palavras, abreviaturas, e outros tipos de símbolos derivados das notações formais da lógica e matemática. Além disso, linguagens de alto nível possibilitam que o programador crie identificadores - compostos por letras, números, e outros símbolos - para expressões de seus programas. Estes identificadores funcionam como rótulos para partes do programa, e usualmente são utilizados para rotular valores ou sequências de operações com palavras ou frases que esclarecem sua função no programa.

A importância do caráter textual das linguagens de programação dificilmente pode ser subestimada e, em alguma medida, é evidenciada pelas atitudes dos próprios programadores. Um notável livro de boas práticas de programação (MARTIN, 2009) contém uma coleção de opiniões de alguns programadores reconhecidos sobre quais são as características que distinguem a qualidade do código fonte de um programa. Um tema recorrente entre as diferentes opiniões diz respeito à legibilidade, expressividade e até mesmo qualidade estética do código. Assim, alguns programadores destacam que o código deve ser elegante e dotado de estilo, enquanto outros dizem que ele deve expressar corretamente as ideias de desenho do sistema e ser de fácil leitura. Fica claro que várias das características desejáveis citadas pelos programadores poderiam muito bem ser direcionadas a uma peça de literatura. De fato, um dos programadores consultados sugeriu que um bom código fonte (ou, nos termos do livro, um código *limpo*) “[...] é legível como uma prosa bem escrita”[10] (MARTIN, 2009, p.8). A importância fundamental deste tipo de característica, relativa à legibilidade do código, é que um código bem escrito - isto é, dotado das características citadas - pode ser mais facilmente entendido não apenas pelo escritor original do código, como por outros programadores que venham a lidar com ele.

[10] No original: “Clean code reads like well-written prose.”

Talvez a representação mais extrema deste tipo de raciocínio seja o paradigma de programação “letrada” proposta pelo cientista da computação Donald Knuth. Para este autor, a tarefa principal de um programador não é apenas construir um programa que diga ao computador o que ele deve fazer, mas também - e ao mesmo tempo - escrever código que explique a outros seres humanos o que se quer que o programa faça. Segundo Knuth, alcançar este objetivo exigiria conceber os programas de computador como trabalhos de literatura (KNUTH, 1983).

As atitudes e opiniões destes programadores mostram que eles não apenas colocam o problema da inteligibilidade dos programas como central, mas consideram que a solução desse problema passa pela natureza textual dos mesmos. Afinal, um programa escrito numa linguagem de alto nível é um texto, e como tal pode ser escrito através do emprego de técnicas de composição textual que o tornam mais facilmente inteligível. De fato, o ambiente sociocultural no seio da qual foram gestadas a computação e a programação é fortemente centrado no texto, e tem a leitura e a escrita como habilidades básicas necessárias para as pessoas que dele participam. Por outro lado, o mesmo não poderia ser dito da elaboração de códigos baseados em longas sequências de zeros e uns. Nesse caso, não há uma elaborada tradição que nos indique as melhores formas de escrever códigos binários que sejam de fácil interpretação por outros seres humanos, e a maioria das pessoas consideraria extremamente desafiadora a tarefa de elaborar ou extrair qualquer informação de um código deste tipo.

O argumento sendo desenvolvido aqui é que uma das funções cognitivas básicas das linguagens de alto nível é que elas mudam o conjunto de habilidades cognitivas necessárias para elaborar e entender o programa de computador, substituindo a habilidade de expressar e interpretar significado através sequências numéricas pela habilidade de realizar as mesmas ações por meio de representações textuais. A vantagem dessa substituição repousa no fato de que a primeira habilidade é pouco desenvolvida na maioria das pessoas (e talvez nem mesmo possa ser desenvolvida em alto grau pelo aparato cognitivo humano), enquanto a segunda costuma ser bem desenvolvida nos ambientes socioculturais onde é praticada a programação. Esse argumento é consistente com a teoria da cognição distribuída de Edwin Hutchins, que afirma que a transformação do tipo de capacidade necessária

para realizar uma tarefa é uma das principais funções desempenhadas pelos artefatos mobilizados em sistemas cognitivos (HUTCHINS, 1995, p.170-171).

Uma linguagem de programação constitui uma espécie de interface entre o computador e o humano que pretende programá-lo. No caso das linguagens de baixo nível, o princípio que guia o projeto da interface é a correspondência exata entre as instruções apresentadas pela linguagem e as operações que o computador é capaz de executar. Ou seja, as considerações envolvidas nas linguagens de baixo nível dizem respeito sobretudo às características da máquina. Por outro lado, as linguagens de alto nível são interfaces elaboradas para facilitar a tarefa do humano, e, portanto, são construídas levando em consideração- implícita ou explicitamente, e em maior ou menor grau de elaboração - as habilidades cognitivas do mesmo.

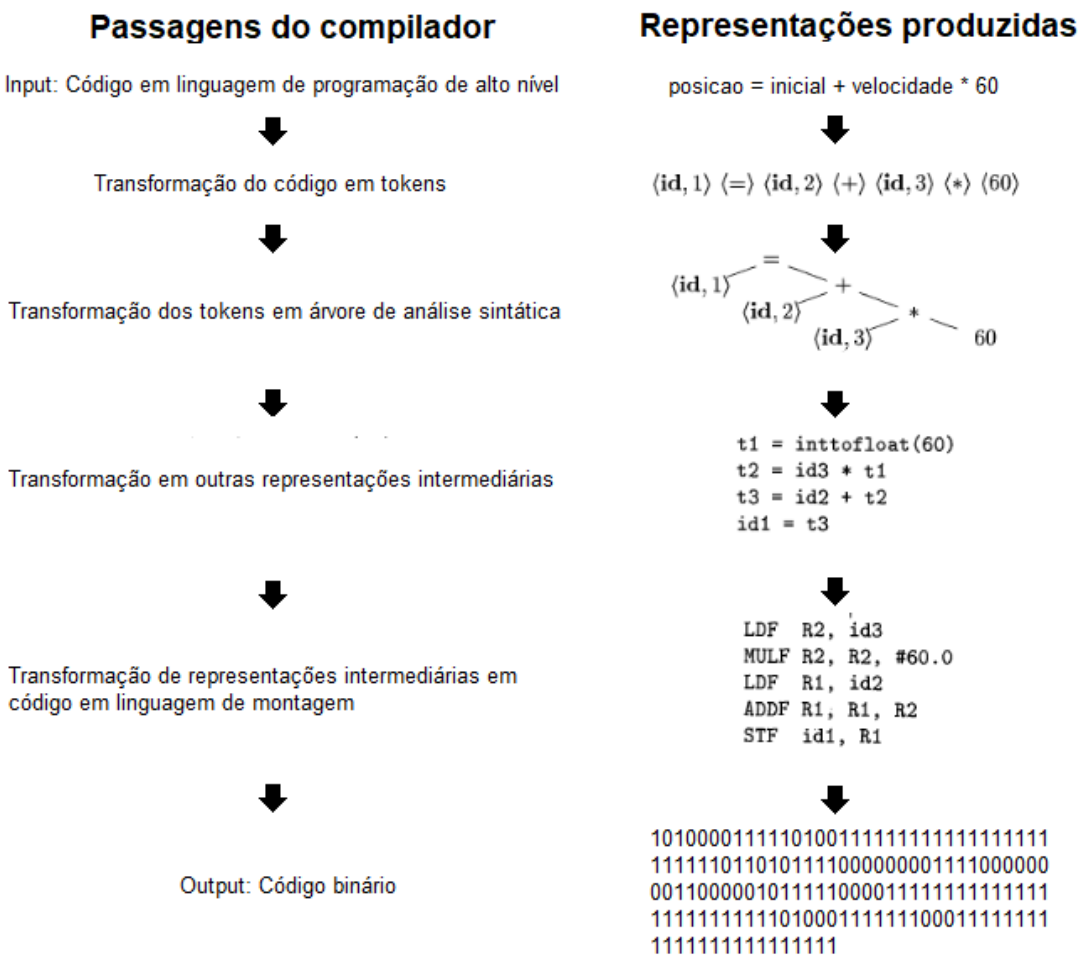
A atual seção discorreu sobre a função cognitiva das linguagens de programação. No entanto, uma questão técnica premente foi deixada sem resposta: Se o computador entende apenas linguagem de máquina, como ele seria capaz de executar um código escrito em linguagem de alto nível? A resposta desta questão demonstrará como o tipo de análise proposta pode ser levada até o interior da máquina.

2.4.3 Cognição nas entranhas da máquina

A resposta da pergunta levantada na seção anterior é a seguinte: para que um programa escrito em linguagem de alto nível possa ser executado pelo computador, ele deve ser **compilado**.

A compilação é o processo por meio do qual o código fonte de um programa computacional escrito em uma linguagem de programação é traduzido para outra linguagem. O caso mais comum deste processo é a tradução de código em uma linguagem de alto nível para um código binário expresso em uma dada linguagem de máquina. O programa computacional que realiza o processo de compilação é chamado **compilador**. Tipicamente, um compilador opera através de uma sequência de “passagens”, cada uma das quais transforma uma representação do código recebido como input em outra representação intermediária, até que se alcance a representação final em linguagem de máquina. A figura abaixo exemplifica as transformações de representação operadas no processo de compilação.

Figura 3 – Transformações representacionais produzidas pelas passagens de um compilador



Fonte: Adaptado de Aho; Lam; Sethi e Ullman (2009).

Um notável paralelo pode ser traçado entre o processo de compilação e o processo de elaboração do programa pelo programador - sobretudo quando essa elaboração é feita por refinamento passo a passo. Nos dois casos, trata-se de um processo que parte de uma representação de caráter mais abstrato e opera sucessivas rerepresentações com níveis cada vez maiores de detalhamento. Em última instância, é como se os processos operados pelo programador e aqueles operados pelo compilador fossem fundamentalmente da mesma natureza, o que permite encará-los como partes de uma mesma cadeia de transformações representacionais. Sob esse ponto de vista, a construção de um programa pode ser pensada como uma *cadeia de propagação e transformação de representações* que

parte de uma descrição de tarefa *altamente "abstrata"* - isto é, produzida na linguagem natural do diálogo humano, e com *baixo nível de detalhamento e formalidade* - e tem como ponto de chegada uma sequência de *instruções computacionai*s especificadas numa notação numérica que, por corresponder à maneira *altamente detalhada e formal* com que *computadores representam as operações* que são capazes de realizar, pode ser diretamente executada por uma máquina a fim de realizar a tarefa em questão.

A cada passo deste processo, o que se observa é o mapeamento por meio do qual algumas diferenças presentes em uma representação são codificadas em diferenças em outro tipo de representação. Contudo, esse processo não é realizado nem apenas no interior do cérebro humano, nem apenas através das representações artefatuais que o humano manipula diretamente, já que uma parte significativa das transformações representacionais é operada de forma automatizada pelo compilador.

Evidentemente, várias diferenças podem ser verificadas entre os processos cognitivos desempenhados pelo programador e aqueles desempenhados pelo computador. A mais evidente delas é que todos os processos realizados pelo programador são de natureza simbólica, enquanto o ser humano faz muito mais do que realizar operações puramente formais. Contudo, todos esses processos podem ser abarcados sob a mesma noção de cognição, e analisados a partir do mesmo procedimento metodológico, sem que para isso seja necessário supor que o computador é um cérebro eletrônico, ou que o cérebro humano é um computador implementado em tecido orgânico.

2.4.4 Fechamento

O presente capítulo apresentou um modelo da atividade de um programador individual, discorreu sobre a natureza cognitiva esta atividade a partir do aparato teórico da teoria cibernética da mente, e discorreu sobre algumas das vantagens que tal aparato apresenta para a análise em questão, destacando sobretudo a sua capacidade de tratar do papel dos artefatos materiais na cognição.

Ao capítulo posterior caberá introduzir o projeto de desenvolvimento de software que essa dissertação pretende examinar, e demonstrar como a abordagem teórica escolhida pode ser mobilizada para tratar ao mesmo tempo do caráter social

da cognição e da relação deste com os artefatos materiais mobilizados no sistema cognitivo em questão.

3 AUTO-ORGANIZAÇÃO E COGNIÇÃO

O capítulo anterior enfocou o papel que os artefatos materiais desempenham nas atividades cognitivas enfrentadas por um programador individual. No presente capítulo o foco passa a ser o papel que os artefatos desempenham na realização de atividades cognitivas colaborativas. Esta questão será tratada a partir da análise do processo de desenvolvimento do rustc, compilador da linguagem de programação Rust.

A linguagem de programação Rust é um projeto de software de código aberto de larga escala, levado adiante através da colaboração de um grande número de programadores em um meio tecnológico complexo. O objetivo deste capítulo é tratar das inflexões que este meio produz sobre a coordenação das atividades de desenvolvimento de Rust.

3.1 A LINGUAGEM RUST

O desenvolvimento do que viria a ser tornar a linguagem Rust começou em 2006, como um projeto pessoal de Graydon Hoare, engenheiro de software especializado em linguagens de programação. Em 2009, o projeto começou a ser patrocinado pela empresa Mozilla, que na época era empregadora de Hoare. A partir daí o projeto passa ser desenvolvido por uma equipe interna da organização e eventualmente adere ao modelo de desenvolvimento de código aberto. A primeira versão oficial da linguagem foi lançada em 2015. Atualmente, o projeto Rust continua em pleno desenvolvimento, apoiado por uma gama mais ampla de patrocinadores, programadores dedicados em tempo integral e centenas de colaboradores voluntários.

Rust tem atraído crescente atenção de desenvolvedores e organizações, figurando como uma das linguagens de programação mais populares dentre aquelas que emergiram na última década. Essa atenção decorre, em grande parte, dos diferenciais que Rust apresenta em relação a outras linguagens de programação.

3.1.1 Uma nova linguagem de programação de sistemas

Rust é uma linguagem voltada para a programação de sistemas. Isso implica que ela é uma linguagem destinada à construção de softwares complexos, que muitas vezes funcionam como plataforma para outros softwares. O principal exemplo de software deste tipo seria um sistema operacional, cuja função primária é intermediar o uso do hardware do computador por outros programas. Softwares de sistema costumam ter altas demandas de desempenho, dado que lentidão num sistema desencadeará lentidão nos softwares que o usam como plataforma. Além disso, a programação de softwares de sistema implica que o programador deve ser capaz de exercer um controle mais próximo e granular de certos detalhes de funcionamento do programa e, possivelmente, da interação do mesmo com o hardware. Essa modalidade de programação se contrapõe à programação de aplicações, que desenvolve softwares que operam tomando por base os sistemas e são diretamente utilizados pelos usuários. A programação de aplicações costuma ter demandas menos estritas de desempenho, e também não necessariamente requer que o programador exerça controle direto sobre a interface do programa com a máquina.

As diferenças entre esses dois tipos de software se refletem nas diferenças entre as linguagens de programação utilizadas para desenvolvê-los. Linguagens de programação de aplicações costumam abstrair mais detalhes da operação do programa, evitando que o programador tenha de se preocupar com os mesmos. Tais abstrações facilitam enormemente o trabalho do programador e tornam o programa mais seguro, pois diminuem a margem de erro humano na codificação. Isto é possível porque tais linguagens utilizam abstrações que implicam em menor eficiência de uso de recursos computacionais - ou seja, perda de desempenho - dos programas nela escritos e também abstrações que impedem que o programador controle certos aspectos da operação do programa.

Por outro lado, a programação de sistemas exige o uso de linguagens que não apresentam abstrações tão poderosas, dado que esse tipo de programação não pode incorrer nas penalidades trazidas pelo uso de tais abstrações. Quando utiliza uma linguagem de programação de sistemas, o programador precisa manejar vários detalhes que seriam abstraídos por uma linguagem de programação de aplicações.

Isso cria o risco de que o programador cometa erros ao manejar essas operações. Muitos desses erros não são possíveis nas linguagens de programação de aplicação, nas quais os detalhes em questão encontram-se abstraídos e fora do controle do programador[11].

A principal inovação a qual Rust se propõe é disponibilizar abstrações que facilitam o trabalho do programador e previnam que ele cometa certos tipos de erros tipicamente facultados por linguagens de programação de sistemas, sem com isso violar os requisitos de desempenho e controle que balizam esses tipos de linguagens.

3.2 O PROJETO RUST

O projeto da linguagem Rust, como qualquer outro projeto de linguagem de programação, tem no seu núcleo a elaboração de documentos que definem a sintaxe e a semântica da linguagem[12], bem como a criação de um compilador que implementa essa definição e permite que a linguagem seja operacionalizada. É justamente o desenvolvimento do compilador de rust, chamado de **rustc**, que constitui o foco de interesse deste capítulo. Contudo, o escopo total do projeto vai bem além desse núcleo, desdobrando-se em uma série de iniciativas que servem de suporte para o desenvolvimento e utilização da linguagem.

Uma destas iniciativas consiste na elaboração de documentos de referência e manuais didáticos para a linguagem, que podem ser utilizados por usuários que desejem aprender a programar em Rust ou tirar dúvidas específicas durante a sua utilização. Tal iniciativa é bastante importante para o uso e a adoção da linguagem, já

[11] Um ponto central dessa distinção entre linguagens de programação de sistemas e de aplicações diz respeito ao problema de alocação de memória. Para manipular um dado qualquer um programa precisa alocar esse dado num espaço da memória de trabalho do computador. Após terminar de utilizar o dado, o programa precisa realizar uma desalocação, de forma que o espaço de memória em questão seja liberado para outros usos. Linguagens de programação de aplicações costumam possuir meios de automatizar a administração de memória, fazendo com que o programador não precise se ocupar desse detalhe durante a escrita do código. Contudo, os mecanismos de administração automatizada de memória fazem com que os programas por vezes sofram penalidades de desempenho, e também retiram certo grau de controle da mão do programador. Por este motivo, esses mecanismos usualmente não estão presentes nas linguagens de programação de sistemas. Por isso, os programadores que utilizam essas linguagens precisam explicitar os processos de alocação e desalocação de memória no código fonte, o que abre margem para erro humano. Uma das principais vantagens de Rust é que ela apresenta uma solução que evita este tipo de erro sem incorrer nas penalidades dos sistemas tradicionais de administração automatizada de memória.

[12] De forma resumida, a sintaxe de uma linguagem de programação diz respeito às regras que definem quais expressões são válidas na linguagem, enquanto a semântica diz respeito ao significado destas expressões em termos das instruções computacionais que elas engendram.

que a falta de documentação de qualidade é um obstáculo significativo para o uso de qualquer tecnologia complexa.

Outra iniciativa crucial diz respeito ao desenvolvimento, fomento e suporte do ecossistema de software no qual a linguagem Rust se insere. A noção de *ecossistema de software* é de uso corrente no universo da programação e, segundo Lungu (2009, p. 27), refere-se a coleções de software que coevoluem a partir de um ambiente comum. Quando se referem à ecologia de uma linguagem de programação, os programadores estão tratando daqueles softwares que gravitam em torno da linguagem e que existem para facilitar ou potencializar o seu uso. Uma amostra não exaustiva dos membros deste ecossistema inclui programas de edição de texto com funcionalidades específicas para escrita de código na linguagem, ferramentas de análise automatizada de código para detecção de possíveis erros e pontos de melhoria, ferramentas para teste de software, pacotes de software (chamados "bibliotecas") que constituem componentes pré-fabricados que podem ser combinados para facilitar o desenvolvimento de outros softwares, entre outros. Todos esses elementos são fundamentais para as práticas modernas de desenvolvimento de programas computacionais, e a falta de qualquer um deles implicaria dificuldades para o uso efetivo da linguagem, constituindo uma barreira para adoção da mesma. O projeto Rust trata desta questão através de iniciativas que objetivam não apenas desenvolver determinados itens de ecossistema, mas também estabelecer as condições para que o mesmo evolua organicamente.

Outro eixo de iniciativas é o fomento de uma comunidade de usuários e entusiastas em torno da linguagem Rust. O esforço realizado sob este eixo envolve a criação de eventos da comunidade, produção de conteúdos sobre o projeto para circulação em redes, vigilância da observância dos códigos de conduta da comunidade nos espaços oficiais de interação online, realização de programas de inclusão de grupos sociais sub-representados, entre outros.

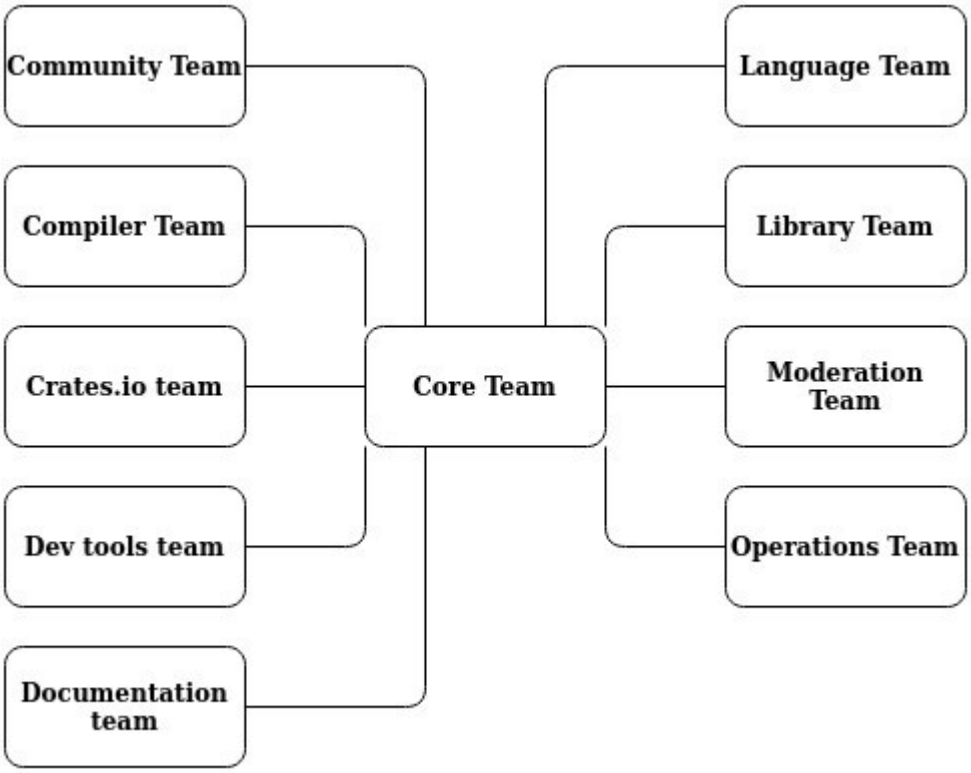
Por fim, existem algumas iniciativas que servem para organizar e suportar as iniciativas citadas anteriormente. Isto envolve tanto a manutenção da infraestrutura computacional e dos processos necessários para o desenvolvimento da linguagem, quanto a operação e melhoria da estrutura de governança do projeto Rust.

3.2.1 Organização

Para levar adiante estas iniciativas diversas Rust conta com uma estrutura organizacional que provê uma divisão de responsabilidades a respeito das variadas iniciativas do projeto.

A principal estrutura organizacional de Rust são os *Times*. Um time consiste num conjunto de pessoas responsáveis por alguma área do projeto Rust. Os membros de time possuem autoridade sobre a área de atuação, cabendo a eles sancionar contribuições de voluntários, ouvir as partes interessadas sobre os rumos da área de atuação do time e tomar as decisões finais a este respeito, bem como realizar contribuições variadas (técnicas, organizacionais etc.) para a efetivação das iniciativas de sua esfera de responsabilidade. Exemplos de times incluem: o *Compiler Team*, responsável pelo desenvolvimento dos aspectos internos do compilador de Rust; o *Language Team*, responsável pelo desenho e implementação de novas funcionalidades para a linguagem Rust; o *Community Team*, responsável por várias iniciativas de fomento e suporte à comunidade de entusiastas de Rust; e o *Dev tools Team*, responsável por contribuir para o ecossistema de ferramentas de software utilizadas para facilitar o desenvolvimento de programas em Rust. Ao todo, existem nove times responsáveis por áreas específicas do projeto. Além destes nove, há ainda o *Core Team*, constituído pelas lideranças dos outros times e responsável por guiar os valores gerais e a visão que guia o projeto, definir prioridades para os outros times, tratar de temas que atravessam as áreas de responsabilidade de múltiplos times e ter a palavra final quanto à inclusão de uma nova funcionalidade na linguagem. A figura 4 apresenta um organograma dos times de projeto.

Figura 4 - Organograma dos times de Rust.



Fonte: elaborada pelo autor.

Os times costumam ter em torno de oito membros em média, com o menor deles tendo quatro e o maior quinze. É possível que a mesma pessoa seja membro de mais de um time: o Compiler Team e o Language Team, por exemplo, possuem membros em comum. É comum que um ou dois dos membros sejam designados como líderes do time, e são essas lideranças que costumam representar o time no Core Team. Para além dos times, existem também estruturas organizacionais responsáveis por iniciativas mais específicas. Estas estruturas são chamadas de *grupos de trabalho*, e costumam ser compostas por um número de membros menor do que os times e também podem possuir líderes. Um grupo de trabalho pode estar associado a um time quando a iniciativa pela qual é responsável encontra-se contida na área de atuação do time. Esse é o caso do *Rustc Dev Guide working group*, que se encontra associado ao Compiler Team e é responsável pela manutenção de um documento que serve de guia para desenvolvedores interessados nos aspectos internos do compilador de Rust. No entanto, existem grupos de trabalho sem associação com times, como o *Game development working group*, cujos focos são iniciativas para

promover o uso da linguagem Rust no desenvolvimento de jogos eletrônicos. Embora membros de times frequentemente participem de grupos de trabalho, também é comum que estes grupos possuam participantes não ligados a nenhum time. Contudo, ao menos nos grupos associados ao Compiler Team, deve haver pelo menos um líder do grupo de trabalho que seja membro do time.

Cabe destacar que esta estrutura organizacional está em constante mudança e sofreu grandes alterações desde o início do projeto. De fato, a adequação e melhoria, dos processos e estruturas organizacionais é um dos grandes focos dos envolvidos no projeto. A evolução da organização em times evidencia este ponto. O Core Team é o mais antigo dos times do projeto. Em 2015, ele continha oito membros, que compunham o principal corpo de governança. A partir de 2015, os envolvidos no projeto realizaram um processo decisório e optaram por acrescentar cinco *subtimes* ao Core Team. Ao longo do tempo, esses subtimes passaram a ser referidos como times (e alguns passaram a conter subtimes), e gradualmente foram sendo acrescidos de outros times - na maior parte surgidos de desdobramentos daqueles já existentes - até que se atingisse o número atual.

O esforço de aprimoramento organizacional ocorre tanto a nível do projeto como no interior de cada time. O Compiler Team, por exemplo, possui um grupo de trabalho voltado especificamente para questões organizacionais, e os membros do time dedicam algumas de suas reuniões regulares para discussão de questões não técnicas, que usualmente são de natureza organizacional.

3.2.2 Participantes

O projeto Rust, assim como outros projetos de código aberto de grande envergadura, é levado adiante tanto por voluntários quanto por pessoas pagas por diversas organizações.

Como o desenvolvimento de Rust ocorre através de canais de público acesso na Internet, qualquer pessoa interessada pode se engajar voluntariamente no projeto. As opções de contribuições disponíveis para um voluntário são variadas e incluem: contribuições de código fonte ou documentação para algum dos softwares desenvolvidos no âmbito do projeto, relatórios de *bugs* experienciados no uso destes softwares e até mesmo realização de atividades organizacionais (como a organização

de reuniões de grupos de trabalho). A realização de uma contribuição usualmente não requer nenhum tipo de engajamento prévio por parte do voluntário, tampouco produz necessariamente algum vínculo posterior. Um voluntário pode realizar uma contribuição de código fonte apenas uma vez e nunca voltar a contribuir, pode realizar contribuições infrequentes ou pode passar a contribuir constantemente para o projeto. Evidentemente, determinados tipos de contribuições (como atividades organizacionais) podem indicar um compromisso de maior prazo, mas não há vínculo formal que impeça o voluntário de encerrar sua participação se assim desejar.

Existem algumas pessoas que são pagas por empresas para participar do desenvolvimento de Rust como seu trabalho integral ou parcial, em caráter permanente ou temporário. Historicamente a maior empregadora de participantes do projeto Rust é a Mozilla, mas outras empresas também empregam pessoas para este fim. Quando uma empresa patrocina um contribuidor de Rust, ela pode direcionar as contribuições do mesmo para iniciativas específicas do projeto que beneficiam a empresa. O projeto Rust tem buscado atrair mais patrocínios de empresas para diversas finalidades, incluindo contribuições financeiras ou contratos para participantes voluntários que tem um bom histórico de contribuições para o projeto.

Tanto voluntários quanto desenvolvedores recorrentes podem participar dos times responsáveis pela administração e direcionamento do projeto. Os critérios de adição de novos membros são decididos por cada time. No que tange o Compiler Team, existe um documento que indica a trajetória para que uma pessoa seja incluída como membro. Segundo esse documento, uma pessoa usualmente começa a participar do projeto do compilador quando se responsabiliza por uma das tarefas publicadas nas páginas da Internet que sediam o desenvolvimento do compilador, ou quando se engaja num dos grupos de trabalho do time. Pessoas que contribuem constantemente, são capazes de trabalhar com certo grau de independência em suas tarefas e agem de acordo com o código de conduta do projeto, podendo ser formalmente reconhecidas como contribuidoras do Compiler Team. Esse reconhecimento não torna a pessoa um membro do time, mas lhe garante alguns privilégios, como a permissão para revisar e aprovar contribuições de código para o compilador, e certas obrigações, como a exigência de que o contribuidor mantenha observância ao código de conduta. Um contribuidor experiente e reconhecido por exercer um papel ativo na formulação dos direcionamentos do compilador ou do time

pode ser convidado para se tornar membro do último. Ser um membro do time implica participar das decisões sobre o reconhecimento dos contribuidores de Rust e a inclusão de novos membros do time, ter a capacidade de liderar grupos de trabalho como representante do time, e também ter um papel central nos processos de tomada de decisão a respeito dos rumos do compilador.

O projeto Rust é levado adiante pelos esforços de uma grande quantidade de pessoas. Embora não seja possível verificar o número global de contribuidores para todas as iniciativas do projeto, o projeto Rust mantém registros a respeito das contribuições para o projeto do compilador. A partir desse registro, pode-se verificar que, até a noite em que foram escritas estas páginas, 5.237 pessoas (das quais pelo menos uma é um robô) já se engajaram, em algum momento, em contribuições de código fonte ou documentação para o desenvolvimento do compilador. Um olhar rápido sobre esses dados permite ver que a distribuição de contribuições por pessoa segue um padrão comum a diversos projetos de código aberto, nos quais uma pequena porcentagem dos participantes realiza a grande maioria das contribuições (MOCKUS; FIELDING; HERBSLEB, 2002). Quando se observa a lista 50 pessoas que mais realizaram contribuições, podemos perceber que cada uma dessas pessoas realizou várias centenas ou mesmo milhares de contribuições (numa margem de contribuições por pessoa que vai de 500 contribuições a aproximadamente mais de 14 mil). Também é possível verificar que várias dessas pessoas são, ou foram, membros de times e grupos de trabalho de Rust. Por outro lado, as estatísticas mostram que mais de 80% dos 5.237 que contribuíram para o compilador de Rust possuem menos de 10 contribuições computadas, e aproximadamente 40% possuem apenas uma contribuição[13]. Os parágrafos anteriores colocam uma questão premente: como é possível que um projeto de tamanha complexidade seja levado adiante através desse modelo no qual a participação é massiva, heterogênea e calcada em vínculos extremamente flexíveis? Como é possível que esses muitos participantes, que se encontram distribuídos ao redor do mundo, consigam agir com a coordenação necessária para levar adiante os objetivos do projeto? Esta questão tem sido colocada

[13]Tais dados são mantidos pelos próprios membros do projeto Rust, e a metodologia utilizada considera como contribuição os envios de código fonte e os comentários de revisão de código fonte realizados no repositório online onde está contido o código fonte do compilador de Rust, e também boa parte da documentação da linguagem. A noção de repositório e revisão de código, bem como o processo de envio de contribuições de código fonte, serão explicadas posteriormente.

de várias formas desde a emergência do modelo de desenvolvimento de código aberto. Especialmente recorrente é a comparação com os modelos de coordenação e organização tradicionais da indústria do software. Eric Steven Raymond (2000), reconhecido praticante e apologista do desenvolvimento de código fonte aberto, delineia o contraste entre esses modelos através de imagens peculiares: enquanto os projetos tradicionais de software seriam construídos em catedrais nas quais alguns poucos especialistas trabalham em isolamento, a criação de um software de código aberto se daria num vibrante bazar ao ar livre, populado por programadores praticantes das mais diversas abordagens.

O campo de pesquisas acadêmicas sobre desenvolvimento de código aberto também demonstrou significativo interesse pela questão, e produziu uma descoberta relevante: os próprios artefatos utilizados no desenvolvimento de código aberto - tais como canais de comunicação, documentos online e até mesmo o próprio código fonte - são em boa parte responsáveis pela coordenação e organização das atividades dos desenvolvedores (SCACCHI, 2007, p.29-34). De acordo com uma das modulações mais incisivas deste argumento, os artefatos *governam* as comunidades de desenvolvimento de código aberto, visto que organizam o trabalho dos programadores mediante a incorporação e operacionalização de regras que agem sobre o comportamento dos mesmos (PAOLI & D'ANDREA, 2008).

O problema da coordenação na elaboração do rustc é um tema central deste capítulo e, como sugere a pista deixada pelas pesquisas citadas, dificilmente poderia ser examinada sem que se enfocassem os artefatos tecnológicos envolvidos no desenvolvimento deste software. Assim sendo, as próximas seções objetivam uma descrição de alguns dos artefatos centrais que compõem o complexo meio tecnológico no qual, e através do qual, o projeto Rust opera.

3.3 O MEIO

O desenvolvimento de Rust ocorre ao longo de múltiplas páginas e aplicações Web. Existem múltiplos canais de bate-papo dedicados a diversos propósitos, como realização de reuniões dos times, discussões sobre iniciativas específicas do projeto, recepção e suporte a novos contribuidores voluntários etc. Além disso, existem fóruns da Internet voltados a discussões de questões sobre o desenvolvimento e os rumos

do projeto Rust, páginas detalhando processos de desenvolvimento adotados por cada time, bem como um blog onde são veiculadas notícias e atualizações sobre o desenvolvimento das iniciativas do projeto.

O Compiler Team mantém um website com documentos e informações detalhadas sobre seus processos. Neste site, é possível encontrar o código de conduta adotado, os links para os canais de comunicação utilizados, o detalhamento dos processos organizacionais do time, a ata das reuniões públicas realizadas pelo mesmo, uma explicação do caminho necessário para que um contribuidor do compilador passe a ser membro do Compiler Team, bem como uma listagem de todos os grupos de trabalho associados a este time.

Da mesma forma, um dos grupos de trabalho do Compiler Team tem por foco manter e aprimorar a documentação do compilador em um site, com uma espécie de guia para as pessoas que desejam entender o funcionamento interno deste software a fim de participar de seu desenvolvimento. A documentação é bastante compreensiva, contendo desde a arquitetura através da qual os diferentes elementos do compilador se relacionam até o detalhamento da série de transformações representacionais que o software opera. Consta também uma exposição do processo através do qual uma nova funcionalidade pode ser ideada e implementada no rustc.

A importância desse tipo de documento para um projeto de código aberto não pode ser subestimada. Para contribuir efetivamente no desenvolvimento do código fonte de um software, é necessário obter algum grau de compreensão sobre o funcionamento do mesmo e dos processos através dos quais ele é desenvolvido. Por este motivo, a produção de documentação a respeito dos detalhes internos do software e, em alguma medida, dos processos de desenvolvimento costuma ser considerada uma boa prática na engenharia de software. Num projeto como Rust estes documentos podem ser vistos como bases de conhecimento compartilhados do projeto, que podem ser utilizados como referência por contribuidores já engajados no desenvolvimento do software em questão, e também como material introdutório para novos colaboradores.

O Compiler Team também utiliza canais de comunicação públicos para discussões relacionadas ao desenvolvimento do rustc. O primeiro desses canais é o Fórum Rust Internals, no qual qualquer interessado pode criar um usuário e participar de discussões. As discussões são estruturadas em tópicos, que são basicamente

postagens feitas pelos usuários e que receberam outras postagens como respostas. O fórum pode ser utilizado para assuntos relacionados a todos os times de Rust e as postagens recebem um rótulo para indicar a qual área do desenvolvimento se referem. O fórum contém tópicos sobre anúncios importantes por parte dos times ou grupos de trabalho, dúvidas sobre assuntos específicos (por exemplo, detalhes do funcionamento interno do rustc) e discussões sobre propostas de mudanças na linguagem, no compilador ou em outras iniciativas do projeto Rust.

O Compiler Team também faz uso do Zulip, um serviço de bate-papo textual. No Zulip são definidas streams, que representam fluxos de mensagens ordenadas temporalmente. Mensagens dentro das streams podem ser divididas em tópicos, que são agrupadores de mensagens sobre um mesmo assunto. Todas as mensagens das streams do Compiler Team são arquivadas em página web públicas para consultas futuras. Neste babo-papo novos participantes do projeto se apresentam para a comunidade, enquanto outros participantes colocam e respondem questões sobre o funcionamento do rustc, processos de desenvolvimento, discutem questões de planejamento e implementação de alterações no compilador etc. Da mesma forma, é através do Zulip que o Compiler Team realiza suas reuniões regulares. No Zulip existem streams dedicados tanto a assuntos gerais do compilador quanto a grupos de trabalho, e os membros do time possuem uma presença bastante ativa nestas streams.

Esses meios de comunicação são bastante importantes para o desenvolvimento do compilador, já que muitas das ideias e questionamentos que mais tarde informarão a realização de alterações e melhorias no código do mesmo são inicialmente aventadas e debatidas no Zulip e no Fórum Rust Internals. Da mesma forma, esses canais parecem ser veículos importantes para a formação de laços entre a comunidade de desenvolvedores de Rust.

Contudo, o elemento mais importante do ecossistema artefatual no qual opera o projeto Rust é a plataforma GitHub, que constitui o nexo para onde converge todo o esforço de desenvolvimento.

3.3.1 GitHub

O modelo de desenvolvimento de código aberto se baseia no acesso público ao código fonte e aos meios de participação no esforço de elaboração do software. A partir da emergência e consolidação desse modelo de desenvolvimento, surgiram diversas plataformas online que funcionam não apenas como meio de hospedagem e publicização de código fonte, mas também como ambientes estruturados para suporte a práticas de desenvolvimento colaborativo de software.

A mais popular destas plataformas é o GitHub. Lançada em 2008 por uma pequena companhia estadunidense, em 2018 a plataforma foi adquirida pela Microsoft. Atualmente o GitHub possui mais de 176 milhões de repositórios de código fonte, sendo 46 milhões de acesso público[14]. A linguagem Rust, assim como muitos outros projetos de software de código aberto, é desenvolvida através do GitHub.

O GitHub permite navegar pela sua extensa base de repositórios públicos, ler ou fazer o download do código fonte destes repositórios. Além disso, é possível criar um usuário para enviar contribuições para repositórios já existentes ou mesmo criar novos repositórios. Também é possível localizar outros usuários, verificar seus repositórios e contribuições realizadas para repositórios de outros usuários e também contatá-los por meio de mensagens. Por este motivo, o GitHub é frequentemente utilizado como um local na qual programadores desenvolvem um portfólio público de projetos próprios ou contribuições para projetos *open-source*, de forma a ganhar reconhecimento perante outros desenvolvedores ou mesmo visibilidade perante organizações que contratam profissionais de software.

Cabe destacar que o GitHub é usado também como plataforma de desenvolvimento por organizações de diversas naturezas, incluindo empresas privadas. A plataforma permite que organizações detenham repositórios, públicos ou privados, e oferece serviços e funcionalidades específicas. Atualmente muitas das principais empresas da indústria global de software possuem repositórios no GitHub e frequentemente utilizam a plataforma para hospedar seus projetos de código fonte aberto.

3.3.1.1 Repositórios

[14] O GitHub não é utilizado apenas para repositórios de código públicos. Também é possível criar repositórios privados, cujo acesso é restrito a uma ou mais pessoas. Contudo, este texto trata apenas de repositórios públicos, voltados ao desenvolvimento de código fonte aberto.

Um repositório GitHub é um conjunto de páginas web que disponibilizam o acesso a um conjunto de arquivos textuais, usualmente de código fonte de um programa computacional, e oferecem mecanismos para que esses arquivos possam ser elaborados e modificados colaborativamente. O projeto Rust conta com diversos repositórios GitHub. Grande parte deles são destinados ao armazenamento e desenvolvimento dos softwares produzidos pelas diversas iniciativas do projeto. Contudo, existem muitos repositórios que não abrigam código fonte, mas sim outros tipos de arquivos de texto. De fato, o projeto Rust detém vários repositórios dedicados à publicação e elaboração colaborativa de documentos cujo conteúdo trata tanto de questões organizacionais quanto assuntos técnicos, bem como repositórios para armazenamento dos conteúdos exibidos nas páginas web do projeto, entre outros. O próprio Compiler Team possui um repositório onde são guardados os conteúdos da página web na qual são publicizados os documentos do time.

O foco deste capítulo é a atividade de desenvolvimento levada adiante no repositório que abriga o código do compilador rustc. Ao abrir este repositório, o usuário do GitHub depara-se com uma árvore de diretórios e arquivos similar àquela encontrada ao abrir qualquer diretório de um computador pessoal. Os diretórios desta árvore contêm os arquivos de código fonte do rustc, atualmente composto por mais de 14 mil arquivos de código fonte que totalizam centenas de milhares de linhas de código, bem como diversos arquivos de documentação sobre o funcionamento do compilador e da linguagem. Contudo, os primeiros arquivos com que o usuário se depara, antes de adentrar em qualquer diretório, não fazem parte do código do rustc. Essa página inicial contém uma grande quantidade de arquivos de configuração do software e do repositório, e também documentos destinados aos contribuidores do repositório. Dentre estes, o que mais chama a atenção são aqueles arquivos que contém documentos como as licenças de propriedade intelectual do rustc, o código de conduta do projeto, diretrizes para aqueles que desejam contribuir, e uma apresentação textual - exibida em destaque na página - que introduz o repositório, indica documentos a respeito do projeto e do compilador, e explica como fazer download e manejar o código fonte do mesmo.

A interface de um repositório GitHub contém um conjunto muito amplo de funcionalidades, e um inventário exaustivo das mesmas traria pouco benefício. As próximas páginas enfocam duas funcionalidades particularmente significativas para o

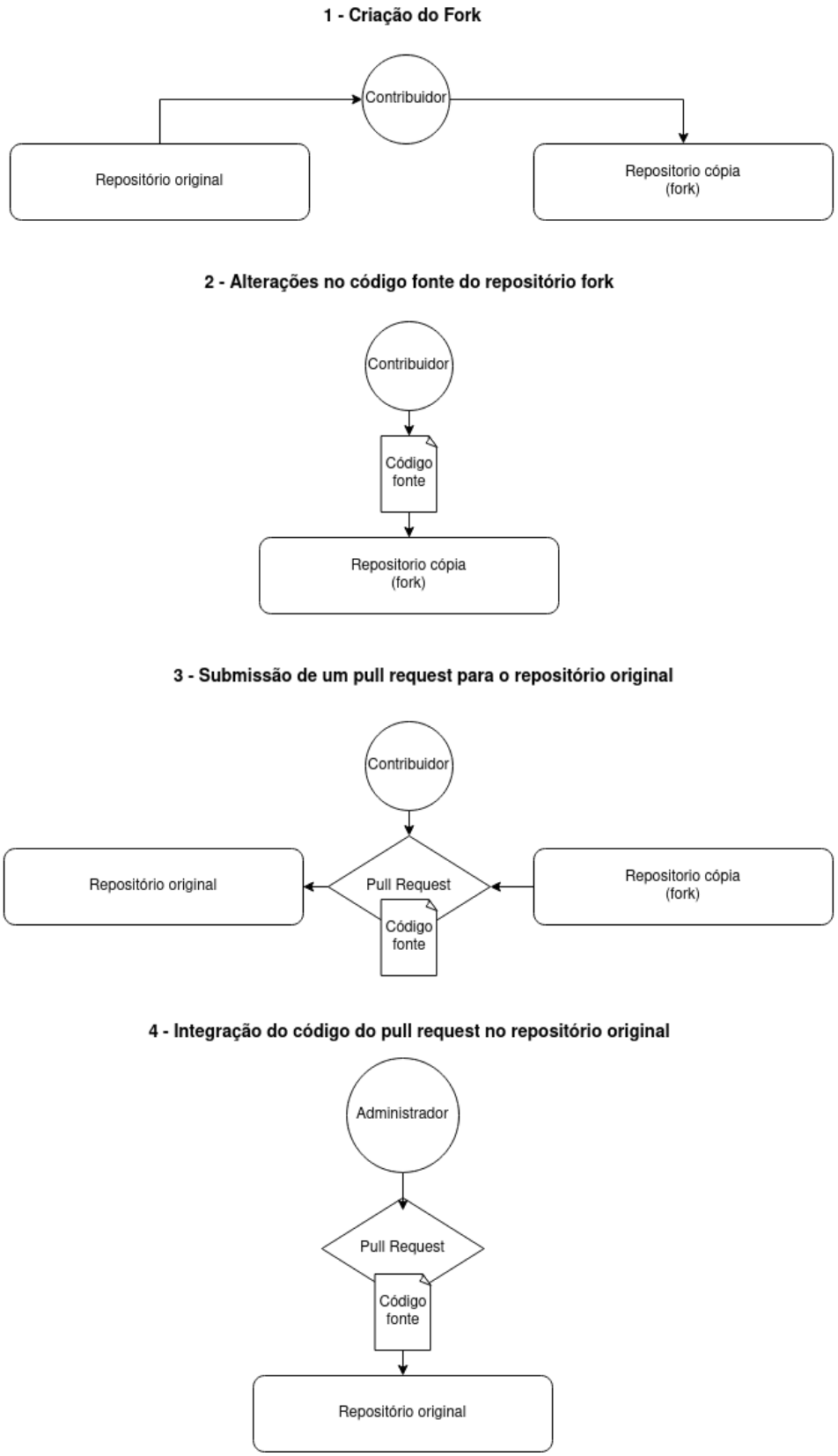
fomento da colaboração. Posteriormente, será demonstrado como essas funcionalidades estão implicadas nos processos de desenvolvimento do compilador de Rust.

3.3.1.2 Pull requests

A principal funcionalidade facultada por um repositório GitHub é a elaboração colaborativa dos arquivos textuais do repositório. Daí decorre uma questão central: quem tem autorização para alterar ou permitir alterações nestes arquivos? O GitHub possui um mecanismo de permissões que faculta ao dono de um repositório (usualmente quem o criou) selecionar aqueles que podem realizar diretamente alterações nos arquivos do mesmo. Em projetos de código aberto de larga escala é comum que essa permissão seja concedida a um grupo seleto de usuários que se tornam administradores do repositório, de forma a garantir segurança contra alterações indevidas.

Aqueles que não têm permissão para realizar modificações diretas nos arquivos do repositório ainda podem enviar contribuições através de um processo conhecido como **fork and pull**. Nesse processo, o usuário do GitHub que deseja modificar ou adicionar código ao repositório começa por fazer um **fork** do mesmo, isto é, começa por **clonar** o repositório no GitHub, de forma a obter um repositório cujos arquivos são exatamente idênticos àqueles do repositório para o qual deseja contribuir. A diferença é que o repositório fork está sob controle do contribuidor, e portanto pode ser alterado diretamente por ele. A partir daí, o contribuidor poderá fazer as modificações que desejar nos arquivos do repositório e, ao concluí-las, poderá utilizar uma funcionalidade do GitHub chamada **pull request** para solicitar que as modificações no fork sejam incorporadas ao repositório que deu origem ao mesmo. Cabe então a um administrador do repositório aceitar ou rejeitar as modificações de código contidas na **pull request**. Caso aceitem-nas, elas serão automaticamente aplicadas sobre os arquivos do repositório principal. A figura 5 ilustra este processo.

Figura 5 - Diagrama do processo de Fork And Pull



Fonte: elaborada pelo autor

Contudo, a avaliação da pull request raramente constitui uma simples aceitação ou rejeição, e sim um processo de revisão de código, através do qual o autor das modificações, os administradores do repositório, e quaisquer outras pessoas que desejem participar, podem colaborar no aperfeiçoamento do código submetido.

Vale destacar que as pull requests de um repositório continuam acessíveis (publicamente, caso o repositório seja público) mesmo após terem sido encerrados por uma aceitação ou rejeição. Ao abrir a interface de listagem de pull requests no repositório do compilador, é possível ter acesso a todos os mais de 40 mil pull requests já submetidos desde a criação deste repositório. A interface permite até mesmo filtrar as pull requests a partir de vários critérios. Esse mecanismo de busca pode ser utilizado para vários propósitos, como encontrar pull requests aberto que precisam de revisão ou encontrar pull requests que realizam modificações em um mesmo segmento do compilador.

3.3.1.3 Issues

Outra funcionalidade importante do GitHub são as **issues**, que constituem um meio para registrar e acompanhar o andamento de tarefas a serem executadas e problemas a serem resolvidos em um repositório. Registrar uma issue implica em prover um título e uma descrição para a questão a qual a issue se refere, e num repositório público como o do compilador de Rust qualquer usuário do GitHub pode fazê-lo. Assim como na pull request, o registro de uma issue cria uma página web específica para a mesma, e insere um link para esta página na listagem de issues do repositório, que por sua vez permite buscar issues a partir de critérios específicos. Qualquer outro usuário GitHub pode acessar uma issue registrada e deixar um comentário na mesma para participar da discussão a respeito da tarefa ou problema em questão. Quando a questão colocada pela issue se refere a uma modificação de código, como a correção de um bug ou adição de uma nova funcionalidade, é comum que a resolução da issue se dê por meio de uma pull request contendo as modificações de código solicitadas pela issue. O objetivo de uma issue é criar uma demanda, e não necessariamente atende-la.

A interface das issues apresenta um conjunto de funcionalidades relevantes para a colaboração e organização do trabalho. Algumas delas são exclusivas para

administradores de repositório, como a possibilidade de marcar a issue como resolvida, ou fechá-la para indicar que a questão descrita na issue por alguma razão não deve ser tratada. Pessoas com permissões adequadas também podem designar um usuário do GitHub - usualmente um administrador ou contribuidor envolvido no repositório - como responsável pela issue, ou aplicar rótulos customizáveis para classificar a issue ou sinalizar algo a respeito dela. Também é possível *mencionar* outro usuário do GitHub num comentário realizado numa issue, de forma a fazer com que esse usuário receba uma notificação que chama sua atenção para a menção. A funcionalidade de comentário também permite fazer referência a outras issues ou pull requests do repositório, e essas referências carregam um link para a página associada aos mesmos.

As issues são amplamente utilizadas no repositório do compilador - desde a criação do repositório, já foram registradas mais de 35 mil issues, e atualmente cinco mil delas estão abertas. A finalidade das issues é bastante variada. Elas podem ser utilizadas tanto pelos contribuidores e membros do time para acompanhar e documentar algum esforço de desenvolvimento (como a implementação de uma nova funcionalidade na linguagem), quanto por usuários do compilador para solicitar pequenas melhorias.

A seguir será realizada uma descrição de um tipo de atividade de desenvolvimento do compilador de Rust que é operada por meio dos mecanismos de issue e pull request.

3.4 O DESENVOLVIMENTO DE COMPILADOR DE RUST

Mudanças no código do compilador de Rust podem ser suscitadas tanto pela necessidade de corrigir erros quanto para acrescentar novas funcionalidades ao software. A adição de novas funcionalidades é um processo complexo e longo, que envolve várias discussões em torno da ideação da funcionalidade, e um período estendido de implementação e testes. Por outro lado, a correção de erros - ou, conforme o jargão dos programadores, bugs - costuma ser bem mais simples.

O objetivo desta seção é descrever a prática do desenvolvimento do compilador, de forma a inspecionar o papel que os artefatos possuem na coordenação deste processo. Para cumprir este propósito, optou-se por enfocar o processo mais

simples, de correção de erros. A maior simplicidade do processo permitirá descrever e analisar com maior minúcia os pontos relevantes para a investigação.

3.4.1 Relatórios de bugs

Um dos principais argumentos dos defensores do modelo de desenvolvimento de código aberto é que a liberação do acesso ao código permite que os próprios usuários de um software se engajem no desenvolvimento. Segundo esta perspectiva, o usuário do software que se depara com uma falha no mesmo poderia escrutinar o código fonte para encontrar as possíveis causas da falha e enviar um relatório de erro com essas informações para os desenvolvedores, ou mesmo tentar codificar uma solução para o problema e remetê-la como contribuição de código para ser integrada no software (RAYMOND, 2000).

Evidentemente, o engajamento dos usuários no desenvolvimento de código aberto requer que estes sejam dotados das competências necessárias para tal. Estas competências incluem não apenas conhecimento de programação - e em específico da linguagem de programação no qual é escrito o software em questão, mas também conhecimento dos procedimentos e ferramentas utilizados para enviar uma contribuição para os desenvolvedores do programa. Na grande maioria dos projetos de software, é de se esperar que apenas uma fatia diminuta dos usuários detenha os conhecimentos e habilidades requeridos para participar do desenvolvimento. No caso do compilador de Rust ocorre exatamente o contrário: qualquer usuário deste software já possui desde o início várias das capacidades requeridas. O código do rustc é escrito na própria linguagem Rust, e os usuários deste programa necessariamente já detém um mínimo conhecimento na linguagem, visto que eles o utilizam justamente para compilar o código Rust que escrevem. Da mesma forma, é provável que grande parte dos usuários do rustc tenham familiaridade com o uso do GitHub, já que atualmente esta é uma habilidade básica no mundo da programação.

Esses fatores certamente têm contribuído para que grande quantidade de relatórios de bug que vêm sendo submetidos para os desenvolvedores do compilador. Existem mais de cinco mil e quatrocentas issues (abertas ou já encerradas) rotuladas como relatórios de bugs no repositório do rustc, e toda semana algumas dezenas de novos relatórios são enviados. Por este motivo, os desenvolvedores do compilador

têm um grande suprimento de informações sobre erros que precisam ser corrigidos para aperfeiçoá-lo e, consequentemente, uma grande demanda de exame de relatórios e correção de erros.

Através dos relatórios de bug, é possível até mesmo corrigir uma versão defeituosa do rustc antes que ela seja lançada para uso. Para entender este ponto é necessário compreender o ciclo de lançamento de versões do compilador de Rust. A cada seis semanas é liberada uma nova versão estável do rustc, e os usuários da linguagem Rust podem baixar esta versão para obter novas funcionalidades e correções. A versão estável é aquela recomendada para uso por ter sido sujeitada a um processo de controle de qualidade mais estrito. Antes de seu lançamento, uma versão estável fica disponível por um período de seis semanas como versão beta. Isso permite não apenas que os usuários interessados possam obter as últimas funcionalidades sem ter que aguardar uma versão estável, mas também garante que essa versão beta poderá ser usada e testada por um período significativo. Durante esse período, os usuários da versão beta podem reportar os bugs encontrados, permitindo que os desenvolvedores do rustc corrijam os problemas antes da promoção da versão beta à estável. Quando as seis semanas se passam, e a última versão beta é lançada como versão estável, é criada uma nova versão beta contendo o produto das últimas semanas de desenvolvimento. Além disto, ao final de cada dia é lançada uma versão “noturna”, contendo o produto do desenvolvimento das últimas 24 horas. Os usuários da versão noturna não apenas têm acesso ao que há de mais novo no compilador, mas também podem encontrar bugs antes que eles adentrem numa versão beta.

3.4.2 Um exemplo etnográfico

Tomemos por exemplo o bug relatado pelo usuário do GitHub **matthew-mcallister**[15] em 21 de maio de 2020. Até a data de criação da issue deste relatório de bug, o usuário em questão não tinha participado de nenhuma forma do desenvolvimento do compilador de Rust - fato que é atestado tanto pelo histórico público do usuário no GitHub quanto por um comentário realizado pelo mesmo durante

[15] As descrições a seguir se referem aos participantes do projeto rust a partir de seus nomes de usuário GitHub, que é o principal meio de identificação utilizado nos processos de desenvolvimento de rustc.

o processo de resolução do bug. Já em sua primeira contribuição, matthew-mcallister foi capaz de se engajar efetivamente no desenvolvimento do rustc, apresentando um relatório de bug e enviando uma solução para o mesmo. A seguir, examinaremos essa contribuição a fim de destacar alguns fatores que permitiram que ela fosse levada a bom termo.

Em primeiro lugar, cabe destacar que matthew-mcallister claramente já possuía muitas das capacidades e conhecimentos necessários para contribuir para o repositório do compilador. Fica evidente que o usuário era familiar com o uso do GitHub e com o desenvolvimento de código fonte aberto, já que o histórico de contribuições dele na plataforma revela uso constante desde o final de 2014, com participações em diversos repositórios públicos. Da mesma forma, o histórico demonstra que matthew-mcallister frequentava repositórios de softwares escritos em Rust desde pelo menos 2017 - ou seja, há um indício de familiaridade com o uso da linguagem. Por fim, também verificou-se que o usuário já havia realizado contribuições para outros repositórios do projeto Rust, e que nestas oportunidades ele interagiu até mesmo com membros de times envolvidos nos repositórios.

O relatório de bug enviado por matthew-mcallister se adequa aos padrões esperados no repositório, conforme expresso no documento que apresenta os guias gerais para contribuições para o compilador de Rust. Isto é perceptível já no título da issue, que é o seguinte: "*Existence of trait alias breaks name resolution of associated consts*". Os documentos de referência indicam que as issues de relatório de bug devem ter títulos descritivos, que indiquem a funcionalidade da linguagem relacionada ao bug, as condições em que ele ocorre e parte da mensagem de erro exibida pelo compilador (caso haja uma). Um título apropriado ajuda outros usuários a identificar rapidamente o assunto tratado pela issue, e até mesmo buscar por ela entre as muitas issues do repositório. O título citado do relatório de bug claramente segue os padrões, já que indica que o problema ocorre quando duas funcionalidades específicas da linguagem chamadas *trait alias* e *associated consts* são usadas em conjunto num trecho de código, e também que o problema afeta uma etapa da compilação chamada "*name resolution*".

O relatório de bug contido no texto de abertura da issue também segue as práticas utilizadas do repositório. O texto começa identificando a versão do compilador na qual foi verificado o erro - neste caso uma versão noturna - e prossegue

apresentando pequenos exemplos de código Rust que causaram o erro ao serem compilados. O primeiro exemplo, com 3 linhas de código, apresenta um código que compila corretamente, e o segundo apresenta o mesmo código acrescido de duas linhas que causam o erro de compilação. Estas duas informações, versão do rustc e exemplos de código que causam o erro, são importantes porque permitem que outras pessoas reproduzam o bug - isto é, utilizem o compilador de modo a causar o bug propositalmente - para investigar suas causas e encontrar a solução. Após os exemplos de código, existe um bloco de texto que apresenta a mensagem lançada pelo compilador quando ele é utilizado para compilar o exemplo de código que causa o erro. No caso em questão, o problema é que os exemplos de código apresentados são válidos de acordo com as regras de sintaxe da linguagem, e, portanto, deveriam ser compilados, mas o compilador não reconhece um deles como tal e lança uma mensagem de erro. A mensagem de erro apresentada pelo compilador é crucial para o tratamento do bug, pois pode apresentar indícios importantes para investigação da causa do erro.

Logo após a abertura da issue matthew-mcallister adicionou um rótulo para sinalizar que a issue tratava de um bug. Seis horas depois, o usuário **jonas-schievink** adicionou quatro outros rótulos na issue: um para sinalizar que o tratamento desta issue era responsabilidade do Compiler Team, outro para indicar que o bug só poderia ser reproduzido numa versão “noturna” do compilador, e dois outros para marcar as funcionalidades da linguagem envolvidas no bug relatado pela issue. A criação e associação de rótulos customizados com issues e pull requests é uma funcionalidade do GitHub amplamente utilizada no repositório do rustc, e existe um procedimento definido de triagem de bugs que prescreve como e quando os rótulos devem ser utilizados. Os rótulos sinalizam informações relevantes para o tratamento de uma issue, e também facilitam que outros usuários a localizem em meio as outras issues do repositório e se engajem em sua resolução. Existe até mesmo uma ferramenta de software que gera automaticamente a pauta da reunião semanal do Compiler Team através de buscas pelas issues e pull requests do repositório, e esta ferramenta identifica as issues relevantes para a reunião através de seus rótulos.

No dia seguinte matthew-mcallister realizou um novo comentário, no qual ele referencia uma outra issue do repositório, que contém um relatório de bug ainda submetido um ano antes pelo usuário **athre0z** e até então não solucionado, para

assinalar que os dois bugs poderiam ser relacionados. Constatar essa possível relação é útil porque as informações levantadas em um relatório podem ser úteis para o tratamento de outro. O relatório referenciado contém rótulos deixados por **jonas-schievink** e também comentários de **Centril**, membro do Team Language, e **petrochenkov**, membro do Compiler Team, que discutiram uma hipótese acerca da causa do bug, e talvez esta informação tenha sido útil para a investigação de matthew-mcallister. No caso em questão, foi descoberto que os dois bugs possuíam a mesma causa raiz, o que implica que a mesma correção de código poderia resolver ambos os problemas. É provável que matthew-mcallister tenha encontrado o relatório de bug de athre0z utilizando a busca de issues do GitHub para identificar issues associadas à *trait alias*, funcionalidade da linguagem Rust associada ao bug. De fato, seria possível fazê-lo tanto a partir do título da issue quanto a partir de seus rótulos, já que tanto um como mencionam a funcionalidade.

Neste ponto, após o relatório de bug ser submetido e rotulado, é comum que outros usuários participem do exame e resolução do bug por meio de comentários. Esses comentários podem incluir informações que auxiliem no tratamento do bug, tais como novos exemplos de código que causam o mesmo bug no compilador e hipóteses sobre a causa do erro. Também é comum que sejam deixados comentários ou novos rótulos para assinalar a gravidade do bug, ou para indicar que ele foi considerado grave o suficiente para ser discutido na reunião semanal do Compiler Team. O objetivo é sempre reunir as informações e realizar os encaminhamentos necessários para que o bug seja examinado e solucionado. Embora a solução em si não seja operada na issue, não é incomum que alguns usuários envolvidos realizem comentários na mesma com atualizações sobre o processo de resolução do problema. Qualquer usuário pode se encarregar de prover uma contribuição de código que solucione o bug através de uma pull request, e não raramente o usuário pode manifestar essa intenção através de um comentário na issue do relatório. Em geral uma issue de relatório de bug é dada por encerrada quando a contribuição de código que resolve o bug é aceita e integrada ao repositório.

Como é possível perceber, relatórios de bug costumam ser tratados colaborativamente por frequentadores do repositório. Contudo, as coisas correram de forma diferente no bug em questão. Após a inserção de rótulos e o comentário do autor do relatório, nenhum outro usuário inseriu qualquer comentário na issue. Isso

ocorreu porque o próprio autor do relatório proveu rapidamente uma solução para o bug. Três dias após ter aberto a issue de relatório de bug, matthew-mcallister abriu uma pull request contendo uma contribuição de código destinada a corrigir o bug.

No comentário de abertura da pull request matthew-mcallister assinala que sua contribuição deve consertar tanto o bug expresso em seu relatório quanto aquele descrito no outro relatório de bug que ele indicou ter relação com o seu. Para fazer isso, ele referenciou diretamente os links para as issues dos relatórios. Também assinalou que a correção do bug foi simples, e que havia sido sua primeira vez identificando e testando bugs no compilador de Rust. A relativa simplicidade da correção é atestada pelo código contido na pull request, que revela que a resolução do bug envolveu a alteração de apenas duas linhas de código do compilador e, como é norma no repositório de Rust, alguns testes automatizados para o bug em questão.

Os testes automatizados consistem em programas que podem ser executados para verificar se o código submetido na contribuição realiza o objetivo para o qual foi escrito. No caso em questão, os testes automatizados consistem em compilar os exemplos de código com problemas de compilação citados nos dois relatórios de bugs a serem resolvidos pela contribuição. Caso o compilador consiga fazê-lo corretamente fica evidenciado que a contribuição de código da pull request corrigiu o bug. Os testes submetidos por matthew-mcallister passaram, demonstrando que sua contribuição de código resolveu o problema em foco.

O primeiro comentário da pull request aconteceu segundos após a sua abertura. O autor deste comentário não é um usuário humano do GitHub, mas sim rust-highfive, um sistema automatizado que é disparado sempre que uma nova pull request é aberta no repositório do compilador. Quando isso acontece rust-highfive deixa um comentário que agradece ao autor da pull request, apresenta algumas informações sobre como uma contribuição de código deve ser feita e - o que é mais importante - indica um outro usuário do GitHub para realizar a revisão de código.

A revisão de código é um processo de controle de qualidade. A ideia é que antes de ser integrada no repositório, uma contribuição de código deve ser revisada por outro programador, a quem cabe a responsabilidade de rejeitar ou aceitar o código, ou mesmo condicionar a aceitação à realização de alterações específicas no código fonte submetido por parte do autor da contribuição. Ao revisor cabe avaliar não apenas a corretude do código - isto é, se ele produz o efeito que se propõe e não

ocasiona outros problemas - mas sobretudo sua legibilidade e manutenibilidade, a fim de verificar se ele poderá ser facilmente lido e alterado por outros programadores. Esta prática é amplamente difundida tanto no desenvolvimento de software de código fonte aberto quanto no desenvolvimento de software proprietário. De fato, grande parte das funcionalidades da interface de Pull Request do Github são desenhadas para suportar revisões de código.

No repositório do compilador todas as contribuições de código precisam obrigatoriamente passar por revisão, e a aceitação do código por parte do revisor permite que o código submetido na contribuição seja integrado no repositório. Para escolher um usuário para realizar a revisão rust-highfive consulta uma lista de usuários aptos a revisar contribuições de código, que contém uma indicação de quais partes do código compilador o usuário possui maior afinidade (isto é, interesse e/ou experiência)[16]. A partir daí rust-highfive verifica quais partes do código do compilador foram modificadas pela pull request em questão, identifica quais revisores potenciais tem afinidade por estas partes do código, e seleciona aleatoriamente um revisor dentre estes. A seleção automatizada de rust-highfive não precisa ser acatada, e não é incomum que os envolvidos escolham outro revisor, seja porque o revisor “sorteado” recusou o encargo, seja porque outro usuário é considerado mais apto para examinar aquela contribuição.

No caso em questão, o revisor selecionado foi estebank, membro do Compiler Team. Dezenove dias após a realização abertura da pull request e seleção do revisor, estebank realizou um comentário no qual aprovou a contribuição sem exigir qualquer modificação por parte do autor. Isso iniciou o processo automatizado de integração do código da pull request ao repositório, que teve por consequência o encerramento das duas issues de relatórios de bugs que foram resolvidos pela contribuição de código.

[16] Os usuários contidos na lista incluem não só membros de times do projeto Rust, mas também outros contribuidores que possuem a prerrogativa de aprovar ou rejeitar contribuições no repositório do compilador.

3.5 COORDENAÇÃO

A descrição anterior segue as ações de matthew-mcallister no relato de um bug no repositório do compilador e na submissão de uma contribuição de código para corrigi-lo. No entanto, ela também permite perceber que várias outras pessoas participaram de alguma forma neste processo. A atuação de estebank certamente foi importante, já que a revisão e aprovação da pull request é um passo necessário na aceitação de qualquer contribuição de código. Mesmo ações realizadas antes de matthew-mcallister ter submetido o relatório de bug tiveram um papel na contribuição realizada pelo mesmo: o exemplo de código no relatório de bug de athre0z, submetido um ano antes, foi refletido diretamente na implementação dos testes automatizados que compuseram a pull Request de matthew-mcallister. E, embora não seja possível afirmar com certeza, é bem possível que as ações de jonas-schievink, Centril e petrochenkov tenham fornecido informações relevantes para a elaboração do código que corrigiu o problema.

Se assumirmos que as ações de todos os envolvidos tinham por objetivo a resolução do bug, podemos afirmar que as ações foram coordenadas para este fim. Isto é, elas foram organizadas de forma a funcionarem efetivamente em conjunto para cumprir o objetivo para o qual estão destinadas. A questão que emerge a partir daí é a seguinte: quais mecanismos produziram esta coordenação?

Segundo a teoria da coordenação proposta por Malone e Crowston (1994, p.90), coordenação consiste na administração de dependências entre atividades. A natureza destas dependências define qual o tipo de problema de coordenação é colocado por um determinado conjunto de atividades, bem como suas possíveis soluções. Algumas atividades apresentam dependências sequenciais, na qual a conclusão de uma atividade é pré-requisito para a próxima. Nestes casos, faz-se necessário coordenar a sequência na qual as atividades são realizadas (HEYLIGHEN, 2016, p.14-15). Um exemplo deste tipo de situação é a relação entre a submissão da contribuição de código e sua revisão: para que a revisão seja executada, é preciso que antes a contribuição tenha sido submetida. Quando atividades sequencialmente relacionadas são atribuídas a agentes diferentes, o agente que executa uma atividade precisa saber que atividades que são pré-requisitos para a sua foram executadas para

que ele possa começar a agir. No exemplo, o revisor de código precisa saber que há uma contribuição aguardando revisão.

Atividades que contribuem para a conclusão do mesmo objetivo, mas não apresentam uma relação sequencial, são atividades que podem ser executadas em paralelo. Ainda assim, a execução paralela destas atividades por diferentes agentes coloca o problema de como dividir eficientemente as atividades entre os atores, e este também é um problema de coordenação (HEYLIGHEN, 2016, p.14-15). Se dois agentes decidirem executar a mesma atividade independentemente eles estarão duplicando esforços sem que isso contribua para a conclusão do objetivo principal, e no limite podem até mesmo interferir no trabalho um do outro. A qualquer momento o repositório de Rust tem um grande número de relatórios de bugs que ainda não foram corrigidos, e a submissão de modificações de código para corrigi-los pode ser realizada paralelamente. Quando um usuário decide trabalhar na resolução de um bug, é comum que ele deixe um comentário sinalizando isto na issue do relatório, de forma que outros usuários possam ver que o bug já está sendo corrigido e não desperdicem esforços codificando uma correção.

Diversos mecanismos podem ser utilizados para resolver estes problemas de coordenação. Um destes mecanismos é um plano de trabalho pré-definido, que especifique o que cada executor deve fazer, quando ele deve começar a fazê-lo e quanto tempo tem para concluir sua atividade. Caso todos os agentes guiem sua atuação pelo plano e cumpram seu papel à risca, a coordenação será efetiva. Outra opção é a existência de um coordenador central encarregado de dividir o trabalho entre os atores, definir a ordem sequencial adequada para as atividades, indicar para os agentes o momento em que eles podem iniciar suas atividades, e realizar os ajustes e remanejamentos necessários durante o processo. Por outro lado, é possível que os agentes prescindam tanto de um plano completo quanto de um coordenador, desde que consigam se comunicar efetivamente durante a realização das atividades. Neste cenário, basta que os agentes saibam o que precisa ser feito, pactuem algum tipo de divisão de atividades e comuniquem a atividade que estão realizando e o momento em que ela foi concluída (HEYLIGHEN, 2016, p.16-17).

Todas as abordagens acima apresentam potenciais e limitações específicas, e podem até mesmo ser combinadas de diversas formas para coordenar atividades. Contudo, nenhum destes mecanismos aparenta ser capaz de explicar a ação

coordenada envolvida no tratamento e correção do bug. Claramente não há nenhum plano pré-definido, já que, embora todos os envolvidos provavelmente sabiam os passos necessários para que um bug seja examinado e corrigido, nenhum deles tinha como saber quem se encarregaria de cada atividade, e quando estas atividades seriam concluídas. Da mesma forma, ninguém assumiu um papel de coordenação dizendo o que cada um dos envolvidos deveria fazer. Por fim, não foi encontrado nenhum registro de troca de mensagens para fins de coordenação entre os envolvidos, nem através de comentários no GitHub e nem através do canal oficial de bate papo do Compiler Team[17]. Na verdade, isso nem mesmo seria possível, já que os envolvidos não sabiam de antemão quem executaria cada atividade. Por exemplo, antes de enviar o relatório de bug matthew-mcallister não tinha como ter certeza de que jonas-schievink é que alocaria os rótulos de classificação na issue, e, portanto, não tinha como alertá-lo que a submissão estava concluída e pronta para ser classificada.

3.5.1 Estigmergia

Como é possível explicar a ação coordenada de agentes que não se valem de nenhum dos mecanismos de coordenação citados? A resposta para essa questão foi encontrada num campo de pesquisa distante das ciências sociais e das ciências cognitivas: o estudo do comportamento dos insetos eusociais. Consideremos, por exemplo, o comportamento dos cupins na construção de um cupinzeiro. Em um primeiro momento um observador verá cada cupim vagando para coletar barro e depositá-lo em um lugar qualquer. No entanto, uma vez que um cupim deposita barro em algum lugar, outros cupins são levados a depositar mais barro em cima, de modo a contribuir para o incremento de uma pilha de barro já existente em vez de iniciar uma nova. Desta forma os cupins aumentam estas pilhas até que elas se tornem as grandes estruturas que caracterizam os cupinzeiros. Embora trata-se de um

[17] Não há como descartar completamente a possibilidade de que os envolvidos tenham trocado mensagens de alguma forma, pois não há como escrutinar todos os canais de comunicação através dos quais eles poderiam fazê-lo. O que é possível afirmar com certeza é que mensagens sobre questões de coordenação do trabalho não foram trocadas nos canais previstos normalmente utilizados para este fim, isto é, o GitHub (nas issues e na pull request já citados) e o canal oficial de bate papo do Compiler Team.

comportamento evidentemente coordenado, ele também não apresenta marcas de nenhum dos mecanismos de coordenação citados anteriormente: não há razão para crer que os cupins elaborem planos, sigam ordens de algum executivo central, ou se comuniquem diretamente uns com os outros. A misteriosa "inteligência" dos cupins foi por muito tempo um enigma para os pesquisadores, que só veio a ser resolvido quando o biólogo francês Pierre-Paul Grassé propôs que a coordenação destes insetos ocorre indiretamente, através dos vestígios que suas ações deixam no ambiente em que eles agem. Para conceituar esse mecanismo de coordenação Grassé cunhou o termo **estigmergia** (GRASSÉ, 1959 apud HEYLIGHEN, 2016, p.1)[18]**.**

O conceito de estigmergia já recebeu múltiplas definições na literatura, mas nesta dissertação optou-se por utilizar a definição proposta pelo ciberneticista Francis Heylighen, (2016, p. 7, tradução nossa), que postula que a estigmergia "[...] é um mecanismo indireto e mediado de coordenação entre ações, no qual o vestígio que uma ação deixa em um meio estimula a realização de uma ação subsequente"[19]. Tal definição, que enfatiza a estigmergia como um mecanismo que alcança a coordenação através de um feedback positivo, baseia-se em três elementos centrais.

O primeiro deles é a ideia de *ação*, que seria um "[...] processo causal que produz uma mudança no estado do mundo" (HEYLIGHEN, 2016, p. 7, tradução nossa)[20]. É a mudança no estado do mundo efetuada pela ação que produz um vestígio que constituirá o estímulo para execução de outra ação. Heylighen enfoca a ação, e não o agente que a executa, pois objetiva estender o conceito de estigmergia também a ações sem agente - tais como reações químicas, por exemplo (HEYLIGHEN, 2016, p. 7-8). No âmbito desta dissertação, entretanto, as ações examinadas têm agentes identificáveis.

O segundo, particularmente importante para o enfoque deste capítulo, é a noção de *meio*, que é aquilo cujo estado é alterado pelas ações. Heylighen utiliza o termo meio, ao invés do mais comumente adotado *ambiente*, para denotar não todo o mundo exterior aos agentes, mas apenas aquela parte do mundo que eles podem

[18] Não foi possível ler o trabalho referenciado de Pierre-Paul Grassé, pois ele foi escrito em francês, língua que não domino, e não localizei nenhuma tradução.
[19] No original: [...] stigmergy is an indirect, mediated mechanism of coordination between actions, in which the trace of an action left on a medium stimulates the performance of a subsequent action.
[20] Tradução nossa. No original "[...] a causal process that produces a change in the state of the world"

alterar e perceber. Afinal, na estigmergia o meio é aquilo que fornece interação e comunicação entre os agentes - ou seja, o meio funciona como *mediador* (HEYLIGHEN, 2016, p. 12-13). Como veremos a seguir, a investigação sobre o papel que os artefatos desempenham na coordenação dos desenvolvedores de rustc é em grande medida uma investigação sobre as propriedades e características do meio sob a qual opera um processo estigmergético.

O último elemento desta definição é a noção de *vestígio*[21], ou *marca*, que constitui a mudança perceptível que uma ação causa no meio. O termo vestígio, preferido por Heylighen, denota que a mudança feita no meio pode ser uma consequência não intencional da ação (HEYLIGHEN, 2016, p. 13). No caso desta dissertação o termo marca se revela mais apropriado, justamente por carregar a ideia de intencionalidade.

No processo sob análise podemos perceber que os usuários do GitHub envolvidos foram de alguma forma estimulados a agir por marcas previamente deixadas por outros usuários em um meio compartilhado. Neste caso, o meio compartilhado é o repositório GitHub do compilador de Rust, e as marcas são quaisquer mudanças perceptíveis no repositório, como adição de rótulos, criação issues, comentários, e até mesmo submissões de alterações de código. Quando matthew-mcallister abriu uma issue com um relatório de bug, ele imprimiu uma marca no repositório que foi percebida por jonas-schievinkr, que por sua vez rotulou a issue, deixando outras marcas que poderiam ajudar usuários a localizar o relatório de bug. A submissão do relatório de bug por athre0z também deixou uma marca no repositório na forma de uma issue que foi localizada vários meses depois por matthew-mcallister, e levou o último a referenciar a issue em seu próprio relatório de bug e a usar o exemplo de código contido nela para formular os testes automatizados para sua contribuição de código.

Até mesmo ação de bot[22] rust-highfive pode ser compreendida da mesma forma, afinal ela foi disparada pela abertura do pull request - marca que a contribuição

[21] No original *trace*.

[22] Bot é uma abreviação de robot, termo correntemente usado para softwares que simulam ações humanas de maneira repetida. Embora o termo seja frequentemente utilizado para programas que se passam por seres humanos em redes sociais, a simulação realizada pelo bot não necessariamente tem por objetivo ludibriar os seres humanos que interagem com ele. No caso de rust-highfive o próprio sistema se identifica como bot quando realiza comentários em pull requests do GitHub.

de código de matthew-mcallister imprimiram no meio. Evidentemente há uma diferença entre dizer que um ser humano foi estimulado a agir por uma marca no meio e dizer o mesmo de um sistema de software como rust-highfive. No primeiro caso, o termo estímulo indica que a pessoa em questão percebeu a marca e identificou nele uma possibilidade de ação. Já no segundo caso a ação é uma mera resposta automática à marca. Seja como for, sob o ponto de vista da análise da coordenação entre as ações o efeito é o mesmo: tanto o software quanto o ser humano são levados a agir por marcas previamente dispostos em um meio compartilhado, e suas ações imprimem novas marcas, levando à emergência de um padrão de auto-organização por estigmergia.

Ao fim e ao cabo, num projeto de software de código aberto a própria escrita do código fonte frequentemente é guiada por interações estigmergéticas (HEYLIGHEN, 2007). A funcionalidade da linguagem Rust que apresentou o bug corrigido por matthew-mcallister, chamada *trait alias*, foi proposta pela primeira vez em 2016, implementada inicialmente ao longo de 2017 e 2018, e até o momento em que estas páginas foram escritas o desenvolvimento da funcionalidade ainda não foi dado por concluído. A implementação inicial evidentemente consistiu em alterações no código do compilador que, como sói ocorrer, introduziram bugs que foram corrigidos por meio de novas alterações de código. A própria pull request de matthew-mcallister é apenas uma de uma série de contribuições de código por meio das quais a funcionalidade *trait alias* tem sido corrigida e melhorada. Em certo sentido, cada contribuição dessa série gerou uma marca no repositório que estimulou as contribuições que vieram depois delas. Afinal, foi a implementação de *trait alias* que ocasionou os bugs que levaram a sua modificação, e foi sobre esse código modificado que matthew-mcallister agiu para operar ainda outra correção. Esta cadeia de modificações de código exemplifica o mecanismo de feedback positivo característico da estigmergia.

A ideia de que um mecanismo de coordenação observado inicialmente em insetos pode ser aplicado a uma atividade tão complexa quanto o desenvolvimento de um software de código aberto pode soar implausível. No entanto, cabe destacar que a estigmergia é tida como um padrão de auto-organização verificada em ampla variedade de sistemas complexos, noção validada pela sua aplicação em campos tão

diversos quanto a elaboração de algoritmos de inteligência artificial e a análise da colaboração entre seres humanos (HEYLIGHEN, 2016, p.2-3).

Atividades colaborativas através da Web têm se mostrado um solo particularmente fértil para a emergência desta forma de coordenação (HEYLIGHEN, 2007). De fato, os projetos de desenvolvimento de software de código aberto (BOLICI; HOWISON; CROWSTON, 2009; ROBLES; GUERVÓS; GONZALEZ-BARAHONA, 2005), juntamente com outras iniciativas de caráter similar como a Wikipédia (SMART; CLOWES; HEERSMINK, 2017, p.137-138), têm sido frequentemente citados por pesquisadores como exemplos de colaboração baseada em estigmergia. A prevalência da estigmergia neste tipo de atividade pode ser explicada pelas vantagens trazidas por esse modelo de coordenação, e pela forma como ele soluciona questões inerentes ao modelo de colaboração através da Web.

A vantagem mais evidente da estigmergia é a simplicidade da solução que ela coloca para os problemas de coordenação. A coordenação sequencial das atividades é resolvida naturalmente, visto que uma nova atividade só é iniciada quando um agente percebe no meio as marcas que revelam que estão presentes as condições necessárias para a realização da atividade. Da mesma forma, o problema de divisão de atividades paralelizáveis entre diferentes agentes pode ser resolvido se os agentes tiverem algum meio de verificar através do meio quais atividades já estão sendo executadas, de forma a não duplicarem esforços na execução de atividades que já estão em andamento. Com efeito, se assumirmos que cada agente só irá escolher executar as atividades para as quais têm a competência necessária, é possível postular que a estigmergia é um mecanismo suficiente para distribuir eficientemente as atividades entre aqueles com a capacidade para realizá-las (HEYLIGHEN, 2016, p.21).

Entretanto, do ponto de vista do desenvolvimento de código aberto, o mais importante não é apenas que esta solução é simples, mas que ela coloca poucas demandas sobre os agentes (HEYLIGHEN, 2016, p.19-21). A correção de bug realizada por matthew-mcallister ilustra este ponto com clareza, dado que a coordenação por estigmergia foi um fator crucial para que este usuário que nunca havia colaborado para o desenvolvimento do compilador de rust conseguisse realizar a correção do bug sem precisar se comunicar diretamente com qualquer contribuidor do repositório.

A estigmergia não requer que os agentes se engajem em qualquer tipo de planejamento, dado que cada agente só precisa saber do estado atual do trabalho, sem se preocupar com os próximos passos ou objetivo final. Isto é ilustrado pelo fato de que matthew-mcallister foi capaz prover uma contribuição para o desenvolvimento da funcionalidade trait alias (através de uma correção de bug) sem precisar ter qualquer informação sobre quais seriam os próximos passos previstos para o desenvolvimento desta funcionalidade ou se havia alguém trabalhando para concretizar estes passos. Isto também evidencia o fato de que na coordenação por estigmergia, os agentes podem realizar atividades que contribuem para um objetivo de forma completamente independente (e até mesmo sem estarem conscientes) dos outros agentes que trabalham para o mesmo objetivo

O fato de que matthew-mcallister não precisou se comunicar diretamente com os outros participantes também é uma decorrência da estigmergia, pois a comunicação indireta através dos efeitos produzidos no meio é suficiente para garantir a coordenação sequencial na execução das atividades e a divisão do trabalho. De forma similar, a estigmergia não exige nenhum tipo de memória dos agentes, pois a informação necessária sobre o estado do trabalho está representada no meio. Mesmo sem ter participado previamente no desenvolvimento do compilador ou consultado outro agente que houvesse, matthew-mcallister foi capaz de localizar o relatório de bug de athre0z e verificar que o bug relatado não havia sido resolvido, pois todas estas informações necessárias estavam acessíveis no repositório.

Especialmente importante para o desenvolvimento de código aberto é o fato de que a coordenação por estigmergia não exige que os agentes envolvidos estejam presentes ao mesmo tempo e no mesmo lugar, desde que eles tenham acesso a um meio comum tal como a plataforma GitHub. De fato, esta é uma condição de possibilidade para que pessoas dispersas ao longo de vários locais do mundo (e em diferentes fusos horários) possam colaborar efetivamente. Esta é uma das razões pelas quais a estigmergia é tão presente em atividades colaborativas na Web.

Até agora foi ressaltada a importância da coordenação por estigmergia para a realização de uma atividade específica dentro do repositório. Contudo, esse modelo de coordenação tem implicações mais gerais para o desenvolvimento do compilador de Rust - e, de forma mais geral, para o desenvolvimento de código aberto.

A primeira implicação diz respeito à inclusão de novos contribuidores. O exemplo de **matthew-mcallister** indica que a coordenação por estigmergia torna mais fácil que que novos contribuidores consigam contribuir com sucesso para o desenvolvimento do compilador de Rust. Se tal hipótese for verdadeira este pode ser um fator relevante para explicar a grande quantidade de voluntários que têm se engajado no repositório.

Outra implicação diz respeito ao papel da estigmergia na coordenação dos esforços dos vários usuários que trabalham simultaneamente no desenvolvimento do compilador de Rust. As estatísticas do repositório do compilador demonstram que a todo tempo existem vários usuários distintos trabalhando paralelamente em várias atividades. Na semana que precedeu a escrita destas páginas 67 novas issues foram abertas no repositório por 53 usuários diferentes, e 72 issues foram marcadas como encerradas. Da mesma forma, 98 pull requests com contribuições de código foram submetidos por 58 usuários diferentes, e 111 pull requests foram aceitos com suas contribuições de código incorporadas no repositório[23]. Evidentemente esses esforços também precisam ser coordenados de alguma forma.

A coordenação dos esforços de vários agentes trabalhando paralelamente costuma ser um problema complicado que, como qualquer outro problema de coordenação, se torna ainda mais difícil quando a quantidade de agentes aumenta. Planos de coordenação tendem a tornarem-se mais complexos e propensos a falhas quando envolvem maior quantidade de agentes. Da mesma forma, as demandas de comunicação podem aumentar conforme cresce o número de agentes, e um coordenador central terá maior dificuldade para coordenar muitos atores. A estigmergia, por sua vez, não é dependente de nenhum dos fatores que dificultam a escalabilidade de outros métodos de coordenação, e pode escalar eficientemente para atividades de qualquer tamanho e com qualquer quantidade de agentes (HEYLIGHEN, 2016, p.19-20). Sem dúvida esse é um fator que permite que projetos como Rust funcionem eficientemente com uma grande quantidade de contribuidores.

[23] Dados relativos a seman de 5 de outubro de 2020 a 12 de outubro de 2020. Cabe ressaltar que o termo pessoas diferentes indica apenas os usuários únicos que abriram as issues ou as pull requests: é possível que a mesma pessoa tenha aberto uma issue e submetido uma pull request, e neste caso ela será computada nas duas contagens. Da mesma forma, é possível que algumas das issues fechadas sejam aquelas que estão na contagem das issues abertas ou não. Este último ponto se aplica também a pull requets.

3.5.2 Outros mecanismos de coordenação

Embora a noção de estigmergia permita explicar muitos aspectos da coordenação envolvida no desenvolvimento do compilador de Rust, seria errôneo indicar que ela é o único mecanismo de coordenação utilizado neste âmbito.

Este ponto pode ser atestado pela atuação do bot rust-highfive na pull request aberta por matthew-mcallister. Anteriormente salientamos que a ação do bot rust-highfive nesta pull request caracterizou estigmergia na medida em que esta ação é desencadeada pela verificação de uma condição específica no meio. Embora mantenhamos este argumento, também fica claro que o bot foi desenhado para cumprir de forma automatizada uma função de coordenação, que consiste na divisão das atividades de revisão de código, e o faz aos moldes de um coordenador central: atribuindo uma atividade para um agente dentro de um conjunto de agentes disponíveis de acordo com seus próprios critérios. Desta forma, a ação do revisor do código não foi desencadeada por algum tipo de marca previamente deixada no meio, e sim pela mensagem direta através da qual coordenador rust-highfive que lhe delegou uma atividade.

A noção de que uma ferramenta de software como rust-highfive pode ser caracterizada como coordenador de seres humanos pode causar alguma estranheza. Todavia, quando discutimos mecanismos de coordenação o que importa não é a natureza dos agentes, e sim a função que eles desempenham. Se rust-highfive cumpre a função que caberia a um coordenador humano de forma análoga ao que faria este humano, então a natureza maquínica do bot não deve ser empecilho para que o consideremos um coordenador. Também não deveria ser razão de objeção o fato de que o bot realiza a divisão das atividades através da execução de um algoritmo pré-definido, dado que um coordenador humano delegasse as atividades através da execução rígida de um dado conjunto de regras não deixaria de ser considerado coordenador. Por fim, o fato de que a adesão à atribuição de rust-highfive é opcional (já que os contribuidores podem não aceitá-la e indicarem outro revisor) também não é um empecilho, visto que entre grupos exclusivamente humanos não é incomum verificar arranjos similares.

Um outro exemplo de correção de bug permitirá ver de forma ainda mais evidente a presença de outros mecanismos de coordenação no desenvolvimento do

compilador. O relatório de bug intitulado “ICE with incorrect turbofish”, criado pelo usuário **cramertj** em vinte de maio de 2019, foi marcado inicialmente pela estigmergia característica do desenvolvimento de código aberto. Os quatro primeiros usuários a participarem da issue, que incluíram membros do Language Team e Compiler Team, se basearam nas informações contidas no relatório de cramertj ou nos comentários destes próprios usuários para enriquecer a issue com novas informações pertinentes para o relatório de bug, na forma de comentários e rótulos, sem que para isso tivessem que se comunicar diretamente uns com os outros[24]. Essa atividade inicial ocorreu no próprio dia de submissão do relatório de erro, bem como no dia seguinte, 21 de maio de 2019.

No entanto, a atividade subsequente da issue, iniciada no dia 23 de maio, exibe um padrão de coordenação bastante diferente. Neste dia ocorrem dois eventos periódicos relevantes no cotidiano de desenvolvimento de Rust: a reunião semanal de triagem do Compiler Team e o lançamento de uma nova versão estável do compilador, que ocorre a cada seis semanas. A reunião semanal de triagem é o fórum no qual os membros do Compiler Team discutem desenvolvimentos recentes entendidos como importantes o suficiente para merecerem a consideração do time. Dentre as principais pautas dessas reuniões estão os relatórios de bugs que, por serem considerados críticos, são selecionados para avaliação do time. A definição desta pauta costuma ocorrer num processo de pré-triagem realizado antes da reunião. Durante o processo de pré-triagem que antecedeu por algumas horas a reunião do dia 23 de maio de 2019, tornou-se claro o bug relatado por cramertj, que até agora estava presente apenas na versão beta do compilador, seria incluído na nova versão estável a ser lançada no mesmo dia. Por este motivo, este bug foi selecionado para discussão na reunião do Compiler Team.

Quando a reunião teve início, o bug foi o primeiro assunto em pauta. O objetivo inicial da discussão era buscar uma solução para o bug antes que a versão fosse lançada. O usuário **pietroalbini**, membro do Operations Team responsável pelo

[24] Aqui cabe novamente salientar que a assertiva de que não houve comunicação entre os usuários sobre este relatório de bug em seu período inicial deriva do fato de que essa comunicação não foi verificada nos canais públicos em que ela normalmente ocorreria, ou seja, o repositório GitHub e o canal de bate-papo oficial do Compiler Team. Comprovar que não houve comunicação alguma entre esses usuários por quaisquer canais evidentemente não é possível através dos meios empregados nesta pesquisa.

processo de lançamento da versão, informou que seria necessário prover uma correção para o bug num período de uma hora e meia para que esta pudesse ser incorporada na nova versão antes do lançamento. Um membro do Compiler Team se prontificou a tratar da solução, e imediatamente foi iniciado um esforço conjunto incluindo outros membros de time que se dispuseram a acompanhar o desenvolvimento e prover a ajuda necessária. Deste ponto em diante, várias pessoas se engajaram na busca pela solução, e os envolvidos se mantiveram em comunicação constante através de canais de bate-papo, dos comentários da issue na qual foi relatado o bug, e também nos comentários da pull request no qual foi trabalhada a modificação de código que deveria corrigir o problema. As mensagens trocadas consistiram em ideias de possíveis soluções para o bug, atualizações sobre o andamento da correção, pedidos para que alguns envolvidos realizassem com urgência atividades necessárias para acelerar o andamento da correção, e até mesmo discussões sobre meios de prevenir problemas similares no futuro. Ao fim e o cabo a correção foi codificada a tempo, mas problemas técnicos impediram que a correção fosse incluída na versão lançada no mesmo dia.

Esse relato evidencia de forma bastante clara a presença de coordenação por comunicação direta. Isto é visível tanto nas discussões que houveram para coordenar sequencialmente as atividades de correção do bug e lançamento da versão quanto nas próprias mensagens trocadas pelos envolvidos na reunião e no processo de resolução do bug para dividir atividades e atualizar uns aos outros sobre o trabalho já realizado.

Por que motivo esse esforço teve que recorrer à coordenação por comunicação direta, ao invés de basear-se apenas em estigmergia? A resposta é que neste caso a urgência para que o bug fosse corrigido antes do lançamento de versão gerou uma necessidade de ação coordenada rápida que dificilmente poderia ser suprida pela estigmergia. Caso não houvesse comunicação direta, é provável que algum tempo tivesse se passado até que algum usuário percebesse a marca deixada pelo relatório de bug e decidisse enviar uma contribuição de código para saná-la, e mais tempo ainda até que alguém realizasse a revisão do código da contribuição. Este ponto é atestado pelo relatório de bug de athre0z, que ficou estagnado por meses até que matthew-mcallister lhe solucionasse. A estigmergia é bastante útil quando as atividades não precisam ser resolvidas dentro de um prazo específico, e esse é

frequentemente o caso em projetos de desenvolvimento de software de código fonte aberto. Por outro lado, quando uma atividade possui limites temporais outras formas de coordenação podem ser necessárias.

O exemplo em questão é só um de muitos possíveis que demonstram que o desenvolvimento do compilador de Rust não é coordenado apenas por estigmergia. De fato, o desenvolvimento do compilador possui até mesmo práticas institucionalizadas de coordenação, como as reuniões periódicas do Compiler Team, que são baseadas em mecanismos não estigmergéticos. Além disso, o exemplo também permite ver que esses outros mecanismos de coordenação podem operar lado a lado com a estigmergia.

3.6 ESTIGMERGIA E COGNIÇÃO DISTRIBUÍDA

Reconhecer que a estigmergia não é o único meio de coordenação vigente no desenvolvimento de Rust não implica diminuir a importância atribuída a ela neste âmbito. A exposição anterior buscou esclarecer o caráter basilar desse modo de auto-organização não apenas para o projeto Rust, mas para todo o modelo de desenvolvimento de código aberto via Web. Todavia, colocar a estigmergia em foco atende também a um outro propósito, que é central para esta dissertação: atentar para o papel que o meio desempenha na organização da cognição distribuída.

O primeiro capítulo introduziu a teoria cibernética da mente e o conceito de cognição distribuída que dela deriva. O foco então foi examinar a elaboração de um programa como uma cadeia de transformações representacionais que não ocorre apenas no interior da mente do programador, mas também através dos artefatos materiais que ele mobiliza para transformar representações em outras representações. Através desse exemplo foi introduzida a ideia de que a unidade de análise da cognição não é o indivíduo, mas um sistema funcional composto pelo indivíduo e os artefatos que ele mobiliza no curso de uma atividade.

Uma noção até então não explorada é que a cognição pode ser distribuída não apenas entre uma pessoa e os artefatos que ela mobiliza, mas também entre muitas pessoas que cooperam numa atividade cognitiva. Para isso, é importante voltar a ideia, brevemente introduzida no capítulo anterior, de que a mente cibernética é uma estrutura hierárquica e recursiva. Isto é, de acordo com Bateson (1979, p.92-93) a

mente é composta por vários elementos inter-relacionados, e cada um destes elementos também pode ser uma mente[25]. Segundo a terminologia da teoria da cognição distribuída, um sistema cognitivo pode ser composto por vários subsistemas cognitivos. Desta forma, embora o programador e os artefatos que ele mobiliza possam ser analisados como um sistema cognitivo, num esforço colaborativo de desenvolvimento de software esse sistema pode ser apenas um dos muitos subsistemas que compõem um sistema cognitivo maior, que envolve os diversas pessoas e artefatos envolvidos na realização de um dado conjunto de atividades.

Ao examinar as descrições de correções de bug realizadas neste capítulo a partir do aparato conceitual da teoria da cognição distribuída, podemos considerar que cada uma das atividades descritas foi executada por um sistema cognitivo distinto, composto por subsistemas funcionais caracterizados pelos usuários individuais envolvidos na análise e correção do bug e pelos artefatos que cada um deles mobilizou para cumprir sua parte na atividade. Por sua vez, estes sistemas de correção de bugs podem ser tomados como componentes de um sistema cognitivo ainda maior, composto por todas aquelas pessoas e artefatos entre os quais é distribuído o trabalho cognitivo de desenvolvimento do compilador de Rust.

O foco da descrição deste capítulo foi o problema da coordenação, questão que também é crucial para a análise da cognição distribuída entre diferentes pessoas. Conforme afirma Edwin Hutchins, a colaboração de várias pessoas na realização de qualquer atividade, seja ela de natureza cognitiva ou não, exigirá alguma forma de cognição distribuída para que as pessoas possam coordenar seus esforços. Ou seja, resolver um problema de coordenação é por si só uma atividade cognitiva. Quando a própria atividade sendo coordenada é de natureza cognitiva, então a cognição distribuída pode ser presenciada tanto no nível da cognição necessária para executar a atividade, quanto no nível da coordenação dos elementos do sistema que irá executar esta atividade (HUTCHINS, 1995, p.176).

Hutchins destaca que entre os seres humanos os problemas de coordenação costumam ser resolvidos através da organização social do trabalho. As ciências

[25] Evidentemente, essa definição recursiva da mente deve possuir um limite, no qual em algum nível da hierarquia de mentes aninhadas os elementos componentes deixarão de ter características mentais. Na ausência deste limite, a definição correria o risco de produzir uma regressão infinita de mentes compostas por outras mentes, e perderia a capacidade de explicar a emergência dos fenômenos cognitivos a partir da matéria (BATESON, 1979, p.92-93).

sociais há muito têm ressaltado que a organização social do trabalho muitas vezes produz propriedades a nível do grupo que são muito diferentes das propriedades dos indivíduos que o compõem. Em *Cognition in the Wild*, Hutchins demonstra etnograficamente que este ponto também se aplica a grupos que operam atividades cognitivas de forma distribuída, evidenciando que estes grupos podem possuir propriedades cognitivas diferentes daquelas verificadas a nível dos indivíduos. A organização social utilizada para distribuir a cognição entre os membros do grupo é listada como um fator causador desta diferença, bem como o são os artefatos técnicos que garantem novas capacidades cognitivas ao sistema na medida em que alteraram a forma como as atividades são realizadas pelos indivíduos (HUTCHINS, 195, p.170-173, 228).

Fica claro que no esquema de Hutchins é enfatizado o papel da organização social na coordenação do grupo e o papel dos artefatos na transformação das atividades cognitivas enfrentadas pelos indivíduos. Ao demonstrar o primeiro ponto, evidenciando que a organização social do grupo incide sobre as propriedades cognitivas do mesmo, Hutchins se contrapõe à ortodoxia cognitiva da época, que buscava examinar a cognição individual isolada do contexto sociocultural no qual ela ocorre (GARDNER, 1996, p.56-57). Ao demonstrar o último, ressaltando que as transformações realizadas pelos artefatos têm consequências para o esforço cognitivo dos indivíduos que os empregam, Hutchins se contrapõe à visão de que a cognição se encerra nos limites da pele e do crânio.

Um ponto explorado com menos ênfase pelo autor é o papel que o próprio meio material desempenha na coordenação do sistema cognitivo. Em uma passagem que se aproxima do conceito de estigmergia sem citá-lo, Hutchins (1995, p.200) afirma que o time de navegação que constitui seu objeto de estudo poderia ser modelado como um conjunto de agentes que agem quando percebem condições específicas no ambiente. Ao descrever a coordenação do time sob este modelo, Hutchins enfatiza que cada membro do time não apenas coordena sua ação com os outros membros, mas também com os dispositivos que mobilizam. O exemplo do autor é o de uma equipe incumbida de realizar repetidamente uma dada atividade com uma periodicidade de três minutos. Nesta situação, um membro da equipe verificará seu relógio para identificar o momento de iniciar novamente a atividade, e os outros membros do time observarão a atuação do primeiro para saber quando devem iniciar

suas respectivas ações. Portanto, o primeiro membro coordena sua ação com o relógio, e os demais membros coordenam-se com o primeiro. Este exemplo elucida que:

> Quando a natureza do problema é vista como coordenação entre pessoas e dispositivos, muito da organização do comportamento é removida do executante e cedida à estrutura do objeto ou sistema com o qual ele está se coordenando. Isso é o que significa coordenar: se colocar de tal forma que as limitações no seu próprio comportamento são dadas por algum outro sistema (HUTCHINS, 1995, p.200, tradução nossa).[26]

A estigmergia constitui um exemplo radical do papel do meio material na coordenação, pois nesta modalidade de auto-organização os agentes delegam a organização de seu comportamento ao meio. Por delegar, nos referimos ao fato de que cada agente deixa que as condições percebidas no meio guiem suas ações, e a estrutura do meio, na medida em que vai sendo modificada pelas ações, provê a coordenação necessária para os atores. É justamente pelo fato de que a coordenação emerge do meio que a estigmergia livra os agentes dos encargos de comunicação, planejamento, memória e etc que costumam ser necessários para a coordenação. Se, como afirma Hutchins, o próprio trabalho organizacional necessário para coordenar um sistema cognitivo é uma forma de cognição distribuída, então podemos afirmar que na estigmergia este trabalho cognitivo é distribuído não apenas entre os atores, mas entre estes e o meio material no qual o sistema opera.

Em seções anteriores, nós delineamos alguns dos efeitos que a coordenação por estigmergia produz no desenvolvimento do rustc. Muitas dessas características - como por exemplo o paralelismo massivo - poderiam ser analisadas como propriedades relevantes do sistema cognitivo em questão. Se a asserção anterior sobre o papel que o meio desempenha na estigmergia estiver correta, seria de se esperar que a origem de algumas destas propriedades poderia ser encontrada nas características do meio tecnológico no qual é desenvolvido o rustc. Este ponto pode ser evidenciado pelo papel mnemônico que as funcionalidades do GitHub desempenham no sistema cognitivo em questão.

---

[26] No original: When the nature of the problem is seen as coordination among persons and devices, much of the organization of behavior is removed from the performer and is given over to the structure of the object or system with which one is coordinating. This is what it means to coordinate: to set oneself up in such a way that constraints on one's behavior are given by some other system.

3.6.1 Memória através do meio

Sempre que um usuário realiza uma ação no repositório do compilador, tal como a submissão de uma pull request, a abertura de uma issue, ou mesmo a inserção de comentários ou rótulos em issues e pull requests, esta ação fica registrada na plataforma. A partir daí estes registros ficarão visíveis para qualquer um que acesse o repositório, pelo tempo que este existir. Para facilitar o manuseio deste histórico, o GitHub oferece diversos mecanismos de busca e consulta dos registros. Quando concebemos o desenvolvimento de Rust como um sistema de cognição distribuída, torna-se fácil perceber que os registros de ações realizadas no repositório funcionam como a memória compartilhada do sistema.

Do ponto de vista da coordenação por estigmergia, o ponto relevante é que através destes registros os agentes podem verificar o estado de execução do trabalho para coordenar suas ações. Todavia, a função realizada por esta memória compartilhada não se restringe a coordenação, visto que os agentes podem a qualquer momento consultá-la para obter outras informações relevantes para a execução de suas atividades. Um uso simples é para compreensão do código fonte do software: quando uma pessoa encontra algo que lhe parece incompreensível em um trecho do código do compilador, ela pode consultar a plataforma para identificar em qual pull request o trecho de código foi inserido, ler os comentários desta pull request em busca de algo que explique o código incompreensível, e até mesmo encontrar a issue que motivou a modificação de código.

A memória inscrita em artefatos é sempre uma pedra de toque para qualquer perspectiva externalista da cognição, pois trata-se de um exemplo ostensivo do uso do mundo material para execução de uma função cognitiva que de outra forma seria exercida através dos mecanismos internos ao corpo humano[27]. Por outro lado, ela também pode servir de ilustração da distribuição da cognição entre pessoas, pois já houve quem postulasse que grupos culturais funcionam como uma espécie de

[27] Um exemplo importante do papel que a memória exerce nas discussões da sobre a hipótese da mente extracorpórea é o célebre artigo no qual Clark e Chalmers (1998) apresentam a noção de *cognição extendida*. Neste escrito, os autores comparam o uso da memória biológica humana com a prática de consultar um caderno de notas para rememorar informações. O argumento traçado é que tanto o caderno de notas quanto a memória biológica operam uma função análoga no processo mnemônico desempenhado pelas pessoas. Isso implicaria que, sob certas condições, um artefato como um caderno pode ser considerado uma parte integrante do aparato cognitivo de uma pessoa.

memória distribuída (ROBERTS, 1964, apud HUTCHINS, 1995, p.177). A memória provida pela plataforma GitHub se reflete de alguma forma nestes dois pontos, na medida em que evidencia um grupo de pessoas que compartilha a memória de suas ações através de artefatos técnicos. O que é crucial é que ao fazê-lo, o grupo adquire propriedades mnemônicas que não estariam presentes caso a memória fosse implementada por outros meios, como por exemplo a capacidade de qualquer membro do grupo acessar a memória sem precisar se comunicar com nenhum outro membro. Este ponto, por sua vez, tem implicações importantes para um processo de desenvolvimento de código aberto, pois reduz as demandas de comunicação necessárias entre os agentes e evita o risco de que determinadas informações fiquem restritas a alguns participantes do projeto.

3.7 RECAPITULAÇÃO

O presente capítulo continuou a investigação do papel que os artefatos materiais desempenham na cognição distribuída envolvida no desenvolvimento de um programa computacional. A passagem do nível do programador individual para a descrição de um projeto que envolve múltiplos programadores permitiu ressaltar que os artefatos não operam apenas transformações nos processos cognitivos que cada programador se defronta ao realizar uma atividade, mas também podem produzir efeitos na distribuição da cognição *entre* os programadores. Na medida em que estão envolvidos na coordenação das atividades cognitivas, os artefatos operam na distribuição *social* da cognição.

4 CONCLUSÃO

A presente dissertação se propôs a analisar o desenvolvimento de um software como uma prática cognitiva levada a cabo coletivamente, com o objetivo de investigar as formas pelas quais esta prática é influenciada pelo contexto material e social no qual ela ocorre.

O primeiro capítulo se propôs a definir os termos nos quais a programação pode ser pensada como uma prática cognitiva. Após rejeitar os modelos cognitivos que reduzem a prática do programador a uma forma de processamento de símbolos,

a atividade de programação foi definida como um processo de mapeamento no qual a descrição informal de uma tarefa é mapeada para um código simbólico que pode ser executado por uma máquina para operar a tarefa. O mapeamento entre um ponto e outro seria realizado através de uma cadeia de mapeamentos intermediários, alguns deles executados diretamente pelo programador (através de seu aparato cognitivo biológico e do uso de artefatos materiais) e outros operados no interior da máquina. O crucial é que todos estes processos de mapeamento, sejam eles executados no interior do crânio do programador, no interior da máquina, ou através da manipulação de outros artefatos, são considerados como partes de um mesmo processo cognitivo.

O capítulo inicial também explora o papel que os artefatos materiais desempenham na cognição. A análise enfoca as linguagens de programação de alto nível, que constituem as principais ferramentas de trabalho mobilizadas pelos programadores. O argumento apresentado é que o uso de tais linguagens, em contraste com a programação em linguagem de máquina, modifica o conjunto de capacidades cognitivas necessárias para a criação de um programa computacional, uma vez que permite escrever os programas não como códigos numéricos, mas como textos escritos em linguagem formal. Como a escrita e leitura de textos tendem a ser mais familiares para a maioria das pessoas do que a composição e interpretação de longas sequências numéricas, há uma redução no esforço cognitivo necessário por parte dos programadores.

O segundo capítulo expande a unidade de análise para além do programador individual, analisando a atividade colaborativa de desenvolvimento do rustc, o compilador da linguagem de programação rust. Esta parte da dissertação se volta para o exame dos efeitos cognitivos causados pela interação entre o contexto material e o modo de organização adotada pelos desenvolvedores. A análise do material etnográfico revelou que grande parte da colaboração é coordenada através de estigmergia, modo de organização no qual a interação entre os agentes envolvidos é mediada pelo ambiente. O argumento desenvolvido, na linha da teoria da cognição distribuída, enxerga o desenvolvimento de rustc como um sistema cognitivo, e pondera que algumas das propriedades deste sistema devem ser produzidas pelo modelo de organização social adotado pelos indivíduos que o constituem. Uma vez que o meio ocupa um papel central na organização por estigmergia, se deduz que algumas das propriedades verificadas neste sistema se originam no meio através do

qual ele opera. A partir daí, busca-se demonstrar que a plataforma GitHub, nexo central do complexo meio tecnológico através do qual é desenvolvido o rustc, produz efeitos identificáveis sobre o sistema cognitivo em questão. Mais especificamente, observa-se que os registros do processo de desenvolvimento realizado na plataforma são tomados como uma espécie de memória compartilhada do sistema cognitivo.

4.1 A MENTE EM CONTEXTO

Esta dissertação tratou de assuntos que, à primeira vista, podem ser considerados distantes das questões constituintes da antropologia social. Entretanto, a escrita destas páginas foi inspirada por um modo de enxergar a cognição que é, no entendimento do autor deste texto, muito próximo da perspectiva que a antropologia tem empregado para estudar os mais diversos fenômenos.

Existe uma tradição que se propõe a estudar a cognição como uma coisa que ocorre no interior de alguns organismos (e também, no limite, de determinadas máquinas). Sob este ponto de vista, o que interessa é descrever o funcionamento e as capacidades desta coisa, e também delinear as características dos aparatos biológicos ou maquínicos que lhe servem de suporte. Levada ao limite, essa operação de reificação acaba por estabelecer um objeto de estudo que aparta a mente do contexto na qual ela opera. Este foi de alguma forma o caminho trilhado no passado pelas principais linhas das Ciências Cognitivas, representadas nesta dissertação pelos seus expoentes da teoria de processamento de informação.

Há, por outro lado, uma perspectiva alternativa, que vê a cognição não como uma propriedade do organismo, mas como um fenômeno que emerge das interações entre o organismo e o ambiente. Visto sobre este prisma inerentemente ecológico, o objetivo do estudo da cognição não é mais esmiuçar o funcionamento de uma mente apartada do mundo, mas sim descrever e entender os comportamentos cognitivos em relação ao contexto nos quais eles ocorrem, sejam estes contextos de natureza material, sociocultural, ou de outra ordem. Este é o caminho trilhado por vários autores e correntes intelectuais que têm se contraposto aos pressupostos do cognitivismo ortodoxo (HUTCHINS, 2010).

Não seria enganoso dizer que a antropologia social também adota abordagens ecológicas, pois sua perspectiva privilegia as relações sobre os termos, e se dedica a estudar os fenômenos socioculturais em contexto, por meio do trabalho de campo e de descrições etnográficas minuciosas. Esta dissertação está construída sobre a premissa de que a ciência capaz de descrever os emaranhados de relações que constituem a vida social, sem com isso escamotear as complexidades dos cenários nos quais residem tais emaranhados, poderia também descrever os circuitos de informação através dos quais a mente emerge no mundo.

REFERÊNCIAS

AHO, Alfred; ULLMAN, Jeffrey; LAM, Monica S.; SETHI, Ravi. **Compilers**: principles, techniques & tools. 2. ed. Boston: Pearson Addison Wesley, 2009. 1009 p.

APGAUA, Renata. O Linux e a perspectiva da dádiva. **Horizontes Antropológicos**, [s.l.], v. 10, n. 21, p.221-240, jun. 2004. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0104-71832004000100010&lng=en&nrm=iso>. Acesso em: 14 set. 2020.

BATESON, G. **Steps to an ecology of mind:** collected essays in anthropology, psychiatry, evolution, and epistemology. San Francisco: Chandler Publishing Co, 1972.

BATESON, Gregory. **Mind And Nature**: a necessary unit. New York: E . P. Dutton, 1979.

BERGQUIST, Magnus; LJUNGBERG, Jan. The power of gifts: organizing social relationships in open source communities. **Information Systems Journal**, [s.l.], v. 11, n. 4, p.305-320, out. 2001. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1046/j.1365-2575.2001.00111.x>. Acesso em: 05 jan. 2021.

BOLICI, F., HOWISON, J., CROWSTON, K.: Coordination without discussion? Socio-technical congruence and Stigmergy in Free and Open Source Software projects. In: 2nd STC, ICSE 2009.

BROOKS, Ruven. **A Model of Human Cognitive Behavior in Writing Code for Computer Programs.** Springfield: National Techical Information Service, 1975. 149p. Disponível em: <https://archive.org/details/DTIC_ADA013582>. Acesso em: 9 out. 2020.

BROOKS, Ruven. Towards a theory of the comprehension of computer programs. **International Journal Of Man-machine Studies**, [s.l.], v. 18, n. 6, p.543-554, jun. 1983. Elsevier BV. http://dx.doi.org/10.1016/s0020-7373(83)80031-5.

CLARK, A. **Supersizing the mind**: Embodiment, action, and cognitive extension. Oxford: Oxford University Press, 2008.

CLARK, A.; CHALMERS, D. The Extended Mind. **Analysis**, [s.l.], v. 58, n. 1, p. 7-19, 1 Jan. 1998. Oxford University Press (OUP). http://dx.doi.org/10.1093/analys/58.1.7.

COLEMAN, Gabriella. CODE IS SPEECH: Legal Tinkering, Expertise, and Protest among Free and Open Source Software Developers. **Cultural Anthropology**, [s.l.], v. 24, n. 3, p.420-454, ago. 2009. Disponível em:

<https://anthrosource.onlinelibrary.wiley.com/doi/abs/10.1111/j.1548-1360.2009.01036.x>. Acesso em: 15 fev. 2020.

CROWSTON, Kevin. A Coordination Theory Approach to Organizational Process Design. **Organization Science**, [S.L.], v. 8, n. 2, p. 157-175, abr. 1997. Institute for Operations Research and the Management Sciences (INFORMS). http://dx.doi.org/10.1287/orsc.8.2.157.

D'ANDRADE, Roy. **The development of cognitive anthropology**. Cambridge: Cambridge University Press, 1995.

EASTMAN, Charles M. Cognitive Processes and Ill-Defined Problems: A Case Study from Design. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 1, 1969, Washington. **Proceedings.** San Francisco: Morgan Kaufmann Publishers Inc., 1969. p. 669 - 690. Disponível em:<https://www.researchgate.net/publication/220814441_Cognitive_Processes_and_I11-Defined_Problems_A_Case_Study_from_Design>. Acesso em: 9 set. 2020.

ECK, David J.. **Introduction to Programming Using Java:** Version 8.1, July 2019. 2019. Disponível em: <http://math.hws.edu/eck/cs124/downloads/javanotes8-linked.pdf>. Acesso em: 09 out. 2020, 747 p.

EVANGELISTA, Rafael de Almeida. **Traidores do movimento:** política, cultura, ideologia e trabalho no software livre. 2010. 240 f. Tese (Doutorado) - Curso de Antropologia Social, Programa de Pós-graduação em Antropologia Social, Universidade Estadual de Campinas, Campinas, 2010. Disponível em: <http://repositorio.unicamp.br/jspui/handle/REPOSIP/280201>. Acesso em: 07 nov. 2019.

EVANGELISTA, Rafael. O movimento software livre do Brasil: política, trabalho e hacking. **Horizontes Antropológicos**, [s.l.], v. 41, p.173-200, 28 dez. 2013. Disponível em: <https://journals.openedition.org/horizontes/578#quotation>. Acesso em: 07 nov. 2019.

GARDNER, Howard. A nova ciência da mente: uma história da revolução cognitiva. São Paulo: EDUSP, 1996.

GOODENOUGH, Ward. 1957. Cultural anthropology and linguistics. In **Report of the Seventh Annual Round Table Meeting in Linguistics and Language Study** (Language and Linguistics monograph 9), Georgetown University.

GRASSÉ, P. P . La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes natalensis et Cubitermes sp. La théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs. Insectes Sociaux, 6(1), 41–80, 1959.

HEYLIGHEN, Francis. Why is Open Access Development so Successful? Stigmergic organization and the economics of information. In: B. Lutterbeck, M. Baerwolff and R. A. Gehring (Ed.), **Open Source Jahrbuch,** 2007 (pp. 165–180). Lehmanns Media.

HEYLIGHEN, Francis. Stigmergy as a universal coordination mechanism I: definition and components. **Cognitive Systems Research**, [S.L.], v. 38, p. 4-13, jun. 2016. Elsevier BV. http://dx.doi.org/10.1016/j.cogsys.2015.12.002.

HUTCHINS, Edwin. **Cognition in the Wild**. Cambridge: MIT Press, 1995.

HUTCHINS, Edwin. Cognitive Ecology. **Topics In Cognitive Science**, [s.l.], v. 2, n. 4, p.705-715, 1 abr. 2010. Disponível em: <https://onlinelibrary.wiley.com/doi/full/10.1111/j.1756-8765.2010.01089.x>. Acesso em: 13 out. 2020.

INGOLD, Timothy. Da transmissão de representações à educação da atenção. **Educação**, Porto Alegre, v. 1, n. 33, p.6-25, 2010. Disponível em: <http://revistaseletronicas.pucrs.br/ojs/index.php/faced/article/view/6777/4943>. Acesso em: 10 junho de 2021.

JEFFRIES, R., TURNER, A. A., POISON, P. G. and ATWOOD, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), **Cognitive Skills and their Acquisition**. Erlbaum: Hillsdale, 1981.

KARANOVIC, J. Free Software and the Politics of Sharing. In: HORST, Heather A.; MILLER, Daniel (Ed.). **Digital anthropology.** London: Berg, 2012. p. 185-204.

KNUTH, D. E.. **The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms**. Addison Wesley Longman Publishing Co: Redwood City, 1997.

KNUTH, D.E.. Literate Programming. **Technical Report STAN-CS-83-981**. Stanford University, Department of Computer Science, 1983.

LATOUR, B; WOOLGAR, S. **A Vida de Laboratório**: a produção dos fatos científicos. Rio de Janeiro, Relume Dumará. 1997.

LUNGU, Mircea F.. **Reverse engineering software ecosystems**. 2009. 186 f. Tese (Doutorado) - Curso de Filosofia, Faculty Of Informatics, University Of Lugano, Lugano, 2009. Disponível em: https://www.researchgate.net/publication/39731581_Reverse_engineering_software_ecosystems. Acesso em: 16 out. 2020.

MALONE, Thomas W.; CROWSTON, Kevin. The interdisciplinary study of coordination. **Acm Computing Surveys**, [S.L.], v. 26, n. 1, p. 87-119, mar. 1994. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/174666.174668.

MARTIN, Robert C. **Clean Code**: a handbook of agile software craftmanship. Boston: Pearson Prentice Hall, 2009. (Robert C. Martin Series).

MINSKY, Marvin. **The Society of Mind**. New York: Simon and Schuster, 1986.

MOCKUS, Audris; FIELDING, Roy T.; HERBSLEB, James D.. Two case studies of open source software development. **Acm Transactions On Software Engineering And Methodology**, [S.L.], v. 11, n. 3, p. 309-346, jul. 2002. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/567793.567795.

MURILLO, Luis Felipe Rosado. **Tecnologia, política e cultura na comunidade brasileira de software livre e de código aberto.** 2009. 172 f. Dissertação (Mestrado) - Curso de Programa de Pós-graduação em Antropologia Social., Instituto de Filosofia e Ciências Humanas, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2009. Disponível em: <https://www.lume.ufrgs.br/handle/10183/16879#>. Acesso em: 07 nov. 2019.

NEWELL, Allen; SIMON, Herbert A.. **Human Problem Solving.** Englewood Cliffs: Prentice-hall, 1972. 920 p.

ORMEROD, Tom. Human Cognition and Programming. In: HOC, J.-m. et al. **Psychology of Programming.** Londres: Academic Press Ltd, 1990. Cap. 1.4. p. 63-82. (Computers and People Series).

PAOLI, Stefano de, D'ANDREA, Vincenzo. 2008. “How artefacts rule web-based communities: practices of free software development”. *International Journal Of Web Based Communities*, 4(2): 199-219, Disponível em: <http://www.inderscience.com/offer.php?id=17673>. Acesso em: 07 nov. 2020.

PENNINGTON, Nancy; GRABOWSKI, Beatrice. The Tasks of Programming. In: HOC, J.-m. et al. **Psychology of Programming.** Londres: Academic Press Ltd, 1990. Cap. 1.4. p. 45-62. (Computers and People Series).

RAPAPORT, William. **Philosophy of Computer Science**. Buffalo, 2020. 938 p. Em construção (draft).

RAYMOND, Eric Steven. **The Cathedral and the Bazaar.** 2000. Disponível em: <http://www.catb.org/~esr/writings/cathedralbazaar/cathedral-bazaar/index.html>. Acesso em: 30 out. 2019.

ROBERTS, J.. The self-management of cultures. In W. Good enough (ed.), Explorations in Cultural Anthropology: Essays in Honor of George Peter Murdock. McGraw-Hil, 1964.

ROBLES, Gregorio; GUERVÓS, Juan Julián Merelo; GONZALEZ-BARAHONA, Jesus M.. Self-Organized Development in Libre Software: a model based on the stigmergy concept. In: 6TH INTERNATIONAL WORKSHOP ON SOFTWARE

PROCESS SIMULATION AND MODELING, 6. 2005, St. Louis, Missouri, Usa. **Proceedings [...]**. p. 1-11.

SCACCHI, Walt. Free/Open Source Software Development: recent research results and methods. **Advances In Computers**, [S.L.], p. 243-295, 2007. Elsevier. http://dx.doi.org/10.1016/s0065-2458 (06)69005-0.

SKIENA, Steven S.. **The Algorithm Design Manual.** 2. ed. Londres: Springer, 2008. 730 p.

TUOMI, Ilkka. Internet, Innovation, and Open Source: Actors in the Network. **First Monday**, Chicago, v. 1, n. 6, 08 jan. 2001. Disponível em: <https://www.firstmonday.org/ojs/index.php/fm/article/view/824/733>. Acesso em: 30 out. 2020.

GLOSSÁRIO

Estigmergia: Mecanismo mediado e indireto de coordenação de ações, caracterizado por ciclos de feedback positivo nos quais o vestígio que uma ação deixa no meio estimula a realização de outras ações.

GitHub: Plataforma online de hospedagem de código fonte e desenvolvimento colaborativo de software

Issue: Funcionalidade da plataforma GitHub que permite registrar e acompanhar o andamento de tarefas pendentes de execução no contexto de um repositório. Um de seus usos mais comuns é o registro e tratamento de Relatórios de Bug.

Pull Request: Funcionalidade da plataforma GitHub utilizada para submissão de contribuições de código fonte para um repositório. Possui mecanismos voltados para discussão e revisão de código.

Relatório de Bug: Registro de uma ocorrência de defeito ou comportamento inesperado enviada para os responsáveis pela manutenção de um programa computacional.

Repositório: No contexto do GitHub, um repositório é um conjunto de páginas web que configuram um local virtual onde ficam armazenados arquivos de código fonte. Tais páginas oferecem mecanismos para que esses arquivos possam ser elaborados e modificados colaborativamente.

Rustc: Software que executa o processo de compilação de código fonte escrito na linguagem de programação Rust.