A Abstração no Pensamento Computacional

Leila Ribeiro

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

Abstract. *The idea that Computing Science provides a kind of ability that is useful for everyone has been advocated since decades. However, developing computacional thinking skills at schools has proven to be a difficult task. In our view, the power of computational thinking comes from the skill of creating abstractions in an organized and systematic way. In this paper we discuss how abstraction creation can be developed in different ways using CS foundations and argue that, to be effective, languages and tools used in schools should aid the process of creating abstractions.*

Resumo. *A ideia de que a Ciência da Computação desenvolve habilidades relacionadas à resolução de problemas tem sido defendida há décadas. Porém, desenvolver o pensamento computacional de forma efetiva na Educação Básica não é uma tarefa trivial. Na nossa visão, o verdadeiro poder do pensamento computacional é a habilidade de criar abstrações de uma maneira organizada e sistemática. Neste artigo, discutimos como a criação de abstrações pode ser desenvolvida de diferentes maneiras usando os fundamentos da Computação e argumentamos que, para serem eficazes, as linguagens e ferramentas computacionais usadas nas escolas devem focar no processo de criação de abstrações.*

1. Introduction

Desde o início da Computação, muitos têm destacado que essa área desenvolve *uma habilidade diferente* relacionada à abstração [Dijkstra 1970, Knuth 1974, Perlis 1982, Papert 1980, Wing 2006, Aho 2011] que parece ser complementar às desenvolvidas por outras ciências. De fato, é notável que a maioria dos programadores habilidosos tem uma maneira sistemática e organizada de abordar problemas, que pode ser usada não apenas para o desenvolvimento de software, mas que é aplicável em contextos mais gerais. Essa maneira de raciocinar é frequentemente chamada de *pensamento computacional*[1].

Desde os anos 80, muitas escolas introduziram programação em seus currículos com o intuito de desenvolver o pensamento computacional. Porém, notou-se que o aprendizado da programação não garante que se consiga aplicar essa nova forma de raciocinar em diferentes contextos. Uma das razões pode ser que a programação nas escolas foi tipicamente vista como um fim e não como um meio[Denning 2017]. Mas será que todos realmente precisam aprender a programar? As máquinas estão se tornando cada vez mais inteligentes, hoje podem realizar muitas tarefas diferentes por meio de comandos de voz, a inteligência artificial teve um impacto significativo em muitas atividades da vida diária nos últimos anos, em um futuro próximo, os computadores provavelmente serão capazes

[1]Há muitas definições diferences de pensamento computacional. Aqui usaremos este termo tanto para denotar uma habilidade quanto o processo mental que envolve diferentes habilidades.

de fazer muito mais [Elliott 2017]. Considerando tudo que os estudantes já aprendem nas escolas, o que torna o aprendizado de uma linguagem de programação diferente do aprendizado de uma língua estrangeira? O que torna construir um programa de computador diferente de construir modelos usando Matemática ou Física? Essas tarefas requerem habilidades de abstração, todas as áreas da ciência trabalham com abstrações. Línguas são abstrações, números são abstrações, equações da Física são abstrações de fenômenos naturais, símbolos representando elementos na tabela periódica são abstrações, enxergar nosso DNA como uma cadeia de letras é uma abstração, notas em uma partitura são abstrações. Quando estudamos Matemática, Física ou Biologia, aprendemos a resolver problemas nessas áreas usando as abstrações desses campos do conhecimento. A Computação não é a única área que desenvolve habilidades de resolução de problemas. A pergunta que é frequentemente feita pelos professores de escola é: *Por que eu devo ensinar programação para desenvolver habilidades de abstração e resolução de problemas se essas habilidades já são desenvolvidas ao ensinar Matemática ou Física?*

Neste artigo, propomos a perspectiva de que **Computação desenvolve a capacidade de criar abstrações de maneira sistemática.** As abstrações estão por toda parte porque essa é a forma humana de lidar com complexidade. A capacidade de criar abstrações é útil para fornecer uma consciência sobre o que são abstrações e como lidar com elas, ajudando a alcançar um nível mais alto de compreensão do mundo. Por exemplo, aprender diferentes idiomas requer habilidades relacionadas a *usar* conceitos da língua estudada (palavras, gramática). Mesmo se alguém domina diferentes idiomas, criar um novo idioma pode ser um grande desafio, pois a criação requer uma compreensão profunda sobre o que é uma língua. O processo de inventar uma nova língua fornece uma visão mais abrangente de todas as línguas que a pessoa conhece.

A Computação fornece técnicas para criar abstrações de forma sistemática [Kramer 2007, Liskov 2020]. Aprendemos a analisar situações, construir soluções gerais, decompor e relacionar problemas e a trabalhar em muitos níveis diferentes de abstrações, habilidades que são muito importantes para dominar complexidade em qualquer área do conhecimento. Para ser criativo e inovador, habilidades altamente valorizadas no século XXI, não é suficiente ter imaginação, é fundamental ter conhecimento e domínio técnico.

A capacidade de criar abstrações é a grande contribuição do pensamento computacional e, portanto, o desenvolvimento dessa habilidade deve ser feito intencionalmente. Mas o que é necessário para criar abstrações? De que tipo de abstrações estamos falando? O objetivo principal deste artigo é discutir essas perguntas. Propõe-se a perspectiva de que os conceitos e habilidades que precisamos realmente vêm dos fundamentos da Computação, a programação deve ser vista como um meio para trabalhar essas habilidades. É importante destacar que não estamos dizendo que a programação/codificação não deve ser ensinada nas escolas, a programação é uma excelente maneira de desenvolver o pensamento computacional. No entanto, os professores devem saber o foco é desenvolver habilidades de abstração e resolução de problemas via programação. Essa percepção muda o enfoque do processo de ensino/aprendizagem.

A Seção 2 revisa brevemente a noção de pensamento computacional, apresentando a definição que será usada neste artigo. Nas Seções 3 e 4 discutimos as principais abstrações da área da Computação. Considerações finais são apresentadas na Seção 5.

2. Pensamento Computacional

Há bastante tempo tem sido observado que a Computação desenvolve uma habilidade única que tem sido chamada de *pensamento algorítmico* ou *pensamento computacional*. Nos anos 70, E. Dijkstra [Dijkstra 1970] e D. Knuth [Knuth 1974] falaram sobre programação como a arte de dominar complexidade para resolver problemas; nos anos 80, A. Perlis escreveu que a programação ajuda a aprender (pois é basicamente uma atividade de ensino) [Perlis 1982], e S. Papert apresentou as *ideias poderosas* que poderiam revolucionar o processo de aprendizagem através do ensino de Computação para crianças (em Mindstorms [Papert 1980]). O tema voltou com força mais tarde com J. Wing [Wing 2006, Wing 2008], que argumentou sobre a relevância do desenvolvimento do pensamento computacional na Educação Básica. Mais recentemente, A. Aho [Aho 2011] e P. Denning [Denning 2017] argumentaram que construir modelos (computacionais) deve ser a essência do pensamento computacional. Desde então, houve uma miríade de definições diferentes de pensamento computacional (veja, por exemplo, [Cansu and Cansu 2019]). A maioria das definições propõe conceitos como decomposição, reconhecimento de padrões, abstração, resolução de problemas e algoritmos como relevantes.

Uma das razões pelas quais há muitas definições diferentes é que nossa compreensão do que é Computação mudou muito nos últimos 50 anos, começando da ideia de que a Computação é uma ciência para construir computadores até noções mais recentes a Computação envolve tanto construir quanto implementar e analisar algoritmos [Denning and Tedre 2019]. A própria definição do que é um algoritmo foi alvo de debates. A que usaremos neste artigo é a definição a seguir, que é bem geral, e enfatiza porque algoritmos podem ser também chamados de modelos computacionais.

DEFINIÇÃO 1: ALGORITMO, MODELO

*Um **algoritmo** é uma descrição precisa e finita de um processo. Um algoritmo é **um modelo** abstrato de um processo, ou seja, um **modelo computacional**.*

O pensamento computacional está, portanto, relacionado às habilidades necessárias para construir algoritmos ou modelos (computacionais), que podem ser programas, especificações, projetos, etc. Segundo B. Liskov, um conceito essencial na Computação é a abstração. No contexto de resolução de problemas, usamos a abstração de diferentes formas, tanto para descrever modelos abstratos da realidade (usando os construtores necessários para descrever processos) quanto para auxiliar no processo de construção do modelo (decompondo o problema não apenas em partes, mas também em níveis de abstração) [Liskov 2010]. A abstração ser a essência do pensamento computacional foi corroborada por muitos autores (veja, por exemplo, [[Papert 1980, Aho 2011, Wing 2006]). Embora a programação possa ser uma forma eficaz de desenvolver o pensamento computacional (porque permite animar/executar o modelo para analisar se ele de fato é a solução esperada), a ênfase deve ser nos conceitos necessários para construir o modelo, e esses são tipicamente relacionados aos fundamentos da Computação e à engenharia de software, e não ao aprendizado de linguagens de programação. O pensamento computacional fornece, entre outras, duas habilidades essenciais: a habilidade de criar abstrações (para descrever os algoritmos/modelos) e a habilidade de construir e argumentar sobre modelos de forma organizada e sistemática.

Está fora do escopo deste artigo revisar e discutir diferentes definições de pensamento computacional, mas é neceessário apresentar uma definição para esclarecer nossa perspectiva sobre o que é o pensamento computacional. A definição a seguir é baseada em [Lee et al. 2011] e foi escolhida porque destaca o fato de que construir abstrações é uma das habilidades principais providas pela Computação, e que técnicas como decomposição (em partes e em níveis) e generalização podem ser usadas para construir modelos computacionais. A automação também é relevante porque distingue a Computação de outras ciências: a possibilidade de animar/executar as descrições de processos que criamos torna mais fácil entender e validar esses modelos. Finalmente, é fundamental a análise crítica do modelo que está sendo construído, não apenas considerando questões técnicas, mas também de questões éticas e sociais.

DEFINIÇÃO 2: PENSAMENTO COMPUTACIONAL

Pensamento computacional *é o processo mental usado para descrever e analisar processos forma sistemática usando os fundamentos e técnicas da ciência da computação. Pensamento computacional envolve o domínio de* problemas *(analisar, comparar e descrever situações/problemas de forma precisa, objetiva e sistemática) e* algoritmos e modelos *(criar, descrever e avaliar processos usando algoritmos/modelos computacionais). O pensamento computacional desenvolve:*

Abstração *Identificar características essenciais de situações, e problemas e construir modelos usando diferentes representações para informações e processos, bem como aplicar técnicas baseadas nos fundamentos da Computação, individual e colaborativamente, considerando inúmeros aspectos e usando diferentes níveis de abstração;*

Automação *Selecionar e usar linguagens, paradigmas computacionais, plataformas, ferramentas e máquinas para automatizar soluções;*

Análise *Analisar criticamente problemas e algoritmos/modelos considerando tanto aspectos técnicos, como correção, eficiência e viabilidade, quanto aspectos socioemocionais, como éticos, usabilidade e sociais.*

Um algoritmo é uma descrição abstrata de um processo. Para descrever processos, deve-se estudar tanto (i) as abstrações que são usadas na descrição em si (da mesma forma que se usa números, operações sobre números para descrever de forma abstrata situações usando expressões e equações matemáticas), isso será abordado na Seção 3; e (ii) como construir essa descrição de processo (normalmente, descrevemos um processo complexo e pode não ser trivial chegar ao modelo que corresponde fielmente à realidade), essa questão será abordada na Seção 4. A Computação é a ciência que estuda *como construir, analisar e automatizar descrições de processos*.

3. Abstrações para Descrever Processos

Para entender o tipo de abstrações que que usamos na Computação, é necessário primeiro esclarecer o que se quer descrever com essas abstrações. Os modelos que criamos são modelos de computação, ou seja, eles descrevem comportamento. Modelos computacionais descrevem processos, por exemplo, o processo de fazer um bolo, de controlar um avião, de crescimento de uma planta, de busca de informações na web, etc.

As descrições de processos (computacionais) têm sido investigadas há décadas na Computação. Existem definições de álgebras de processos que definem as operações necessárias para descrever processos – veja, por exemplo, [Milner 1980, Hoare 1985]. Embora as definições possam diferir (como as muitas definições diferentes de aritmética), operações como a **composição**, **escolha** e alguma espécie de **repetição** aparecem em todas elas. Linguagens de programação implementam esses operadores de forma diferente dependendo do paradigma da linguagem (imperativo, funcional, declarativo, etc). Não importa qual linguagem seja escolhida para descrever processos (linguagem natural, linguagem de programação, linguagem de especificação, etc), essas operações fornecem uma base para descrever computações.

A definição de um processo envolve, em muitos casos, descrições dos recursos utilizados e/ou produtos gerados pelo processo. Em muitas situações, esses recursos/-produtos não são adequadamente representados apenas por números ou palavras (embora seja teoricamente possível codificar tudo como números, na prática isso torna modelos extremamente difíceis de construir e entender). Por exemplo, para explicar como encontrar uma rota em um mapa, precisamos explicar primeiro o que é um mapa (que é uma descrição abstrata do mundo real). A Computação fornece maneiras de organizar informação de forma adequada, tornando possível construir processos para manipular essa informação. **Estruturas de dados** são usadas para representar a informação de forma abstrata. As principais abstrações usadas para descrever informação são **registros**, **listas** e **grafos**. Ser capaz de representar a realidade/informação usando essas estruturas (ou combinação dessas estruturas) é uma habilidade extremamente importante. Isso permite criar modelos abstratos complexos de informação.

Resumindo: para ser capaz de descrever informação, é necessário saber quais são as **principais formas de estruturar dados (registros, listas, grafos)**; para ser capaz de descrever processos, é necessário dominar as **principais operações para descrever algoritmos (composição sequencial/paralela, operadores de escolha e repetição)**. Geralmente, construir uma solução computacional para um problema envolve não apenas escolher e usar algumas dessas abstrações, mas sim construir novas abstrações para representar informações do mundo real, bem como construir descrições de processos complexos. Essa atividade exige a criação de novas abstrações.

4. Construção de Abstrações para Descrever Processos

Como podemos construir uma solução computacional para resolver um problema?. Simplesmente conhecer as abstrações da Computação (revisadas na seção anterior) não é o suficiente, da mesma forma que não é suficiente ter um dicionário e entender as regras de gramática de um idioma para saber escrever um livro. É necessário dominar técnicas para construir soluções algorítmicas para problemas reais.

Vamos recapitular brevemente a história do computador. Nos anos 40 e 50, surgiram os primeiros computadores. Os programas que executavam nesses computadores eram escritos sem qualquer método específico. Logo os primeiros computadores comerciais apareceram e as empresas começaram a contratar muitos programadores, esperando que, ao trabalharem juntos, eles seriam capazes de construir sistemas complexos. No entanto, isso não aconteceu, levando à chamada *crise do software*. O motivo desta crise foi basicamente a falta de métodos para construir sistemas computacionais.

Na Computação, as áreas que provêm os fundamentos para a construção e análise de problemas e dos correspondentes sistemas computacionais são a engenharia de software (que surgiu como consequência da crise do software) a teoria da computação,. Quando abordamos um problema, a solução idealmente deve ser independente da linguagem de programação concreta. Solucionar problemas diretamente em termos de linguagens de programação tipicamente mistura questões de naturezas muito diferentes (por exemplo, o problema de ordenar uma lista não tem relação com como essa estrutura pode ser implementada em alguma linguagem). É importante entender a distinção entre a solução conceitual de um problema e sua implementação. A programação é útil porque fornece um método concreto e muito eficaz para praticar conceitos de construção de soluções, da mesma forma que o material dourado é um excelente auxílio para entender números e aritmética básica. Porém, é necessário enfatizar no ensino os métodos de engenharia de software para criar as soluções computacionais, caso contrário estaremos fornecendo aos estudantes o mesmo ambiente que tínhamos antes da crise do software [2].

Agora vamos ilustrar as técnicas para criar soluções usando um exemplo muito simples. Aqui vamos propositalmente usar um exemplo cuja solução não é um programa de computador, enfatizando que o pensamento computacional pode ser usado em diferentes contextos. Imagine que queremos construir casas como a mostrada na Figura 1(a) usando papel, cola e tesoura. Um passo útil é **decompor** o problema em problemas menores identificando subproblemas. Podemos decompor o problema como ilustrado na Figura 1(b). Esta é uma decomposição horizontal, semelhante ao que chamamos de **modularização**. Observe que a **decomposição** envolve também uma operação de **composição**: precisamos saber como juntar as partes de forma adequada para resolver o problema. Algumas vezes, o problema é tão complexo que não permite a identificação de todas as suas partes de uma só vez, precisamos trabalhar em níveis de abstração diferentes usando **refinamentos**. Por exemplo, poderíamos construir a casa como mostrado na Figura 1(c). A diferença fundamental entre (b) e (c) é que na abordagem (c) identificamos inicialmente 2 partes da solução: um número e uma casa, então basta colar o número na casa. Nesta solução, assume-se que a casa é entregue pronta. Depois, em um passo de abstração subsequente, pode-se definir como construir a casa. Na solução (b), o problema é dividido diretamente em três partes, ou seja, identificamos mais detalhes do problema de uma única vez. Quando se trabalha com problemas complexos, identificar todas as partes da solução de uma única vez é difícil, pois são muitos detalhes. Por isso trabalhar em níveis de abstração diferentes (decomposição vertical) é essencial.

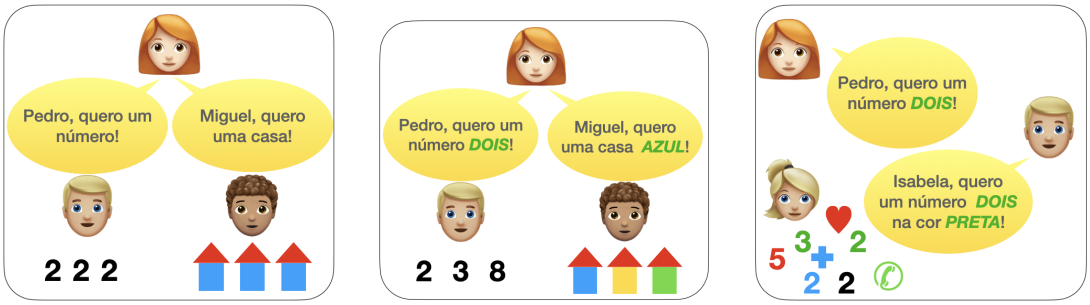
(a) (b) (c)

Figura 1. Exemplo de decomposição: Construção de casas

[2]Pode-se usar abordagens como a apresentada em [Felleisen et al. 2018], que enfatiza as técnicas para construir algoritmos e não a programação em si (os conceitos do livro são apresentados no contexto da linguagem Racket, mas são, em princípio, independentes de linguagem de programação).

Decomposição envolve enxergar um problema (e sua solução) não como um todo, mas em partes ou níveis. Por um lado, significa identificar as partes de um problema, bem como formas de juntar as soluções desses subproblemas para obter a solução (decomposição horizontal). Por outro lado, significa visualizar o problema em diferentes níveis de abstração, compreender as relações entre esses diferentes níveis (decomposição vertical).

A interação entre essas duas formas de decomposição é extremamente importante: permite quebrar um problema em suas partes constituintes e tratar cada uma separadamente, resolvendo-a em pequenos passos de abstração. A decomposição também é o que permite o trabalho colaborativo eficaz. Imagine agora que Luiza, Pedro e Miguel irão construir casas juntos. Pedro é responsável por cortar números 2 e Miguel irá construir casas azuis. Para construir as casas, Luiza pegará as partes fará a colagem. Esta situação é ilustrada na figura 2(a). Quando trabalhamos em grupo, é possível que os membros tenham habilidades mais gerais do que as que realmente precisamos. Por exemplo, Pedro pode ser capaz de cortar qualquer número e Miguel pode construir casas de qualquer cor. Luiza poderia usar essas habilidades para construir diferentes casas, mas ela teria que dizer explicitamente o número e a cor que ela deseja (Figura 2(b)). Isso é um exemplo de usar uma solução mais geral para resolver uma solução específica.



(a) (b) (c)

Figura 2. Exemplos de generalização (a) e (b) e Transformação (c)

Generalização envolve identificar padrões, o que muda e o que não muda, e definir o comportamento de processos de forma genérica. Procedimentos genéricos têm um número maior de entradas do que os específicos (porque a entrada é o que torna o procedimento adaptável a diferentes situações).

Quando um problema é decomposto para ser resolvido colaborativamente, é imperativo que as interfaces sejam claramente definidas e que cada membro da equipe saiba exatamente o que cada parte deve fazer (especificação). A Figura 3 mostra uma definição da tarefa de Pedro, as interfaces, a especificação e as ações concretas que ele realizará para cumprir essa tarefa, tanto no caso específico quanto no caso genérico.

Quebrar problemas em partes permite o reuso de soluções existentes, mas fazer isso corretamente requer capacidade de comparar problemas, o que é usualmente co-

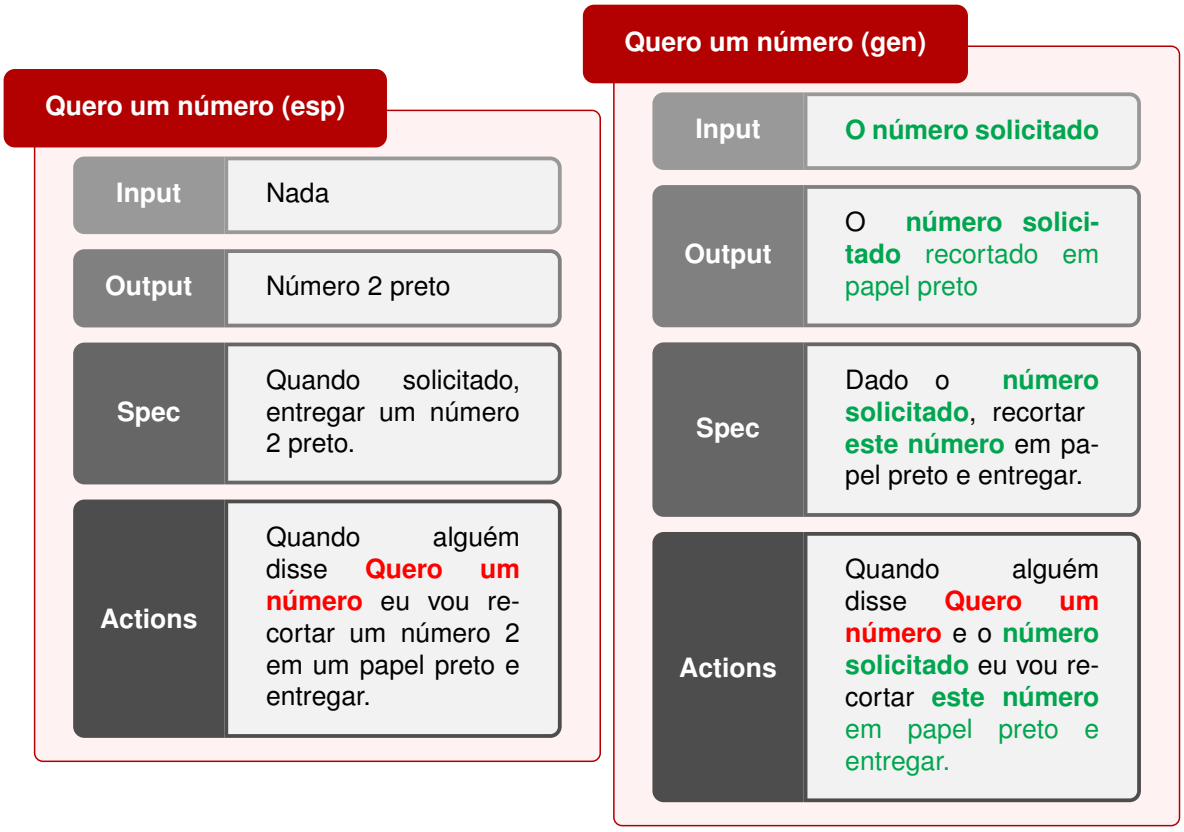


Figura 3. Tarefas do Pedro: (esp) versão específica (gen) versão genérica

nhecido como **transformação** ou **redução**. Imagine agora que Pedro se encontrou com Isabela, e ela é capaz de cortar qualquer símbolo em qualquer cor (inclusive números pretos, que é exatamente o que ele precisa). Então, ele decide mudar seu comportamento: ele agora pedirá a Isabela o número desejado (veja Figura 2 (c)). A entrada, saída e especificação da tarefa dele permanecem as mesmas, apenas a forma como ele resolve o problema mudou porque ele transformou sua tarefa em uma instância da tarefa de Isabela. Transformar um problema em outro requer grande capacidade de abstração porque é necessário entender ambos os problemas e a relação entre eles. Observe que essa relação não é apenas no nível sintático, é necessário comparar a semântica dos problemas para verificar se é realmente possível reutilizar uma solução. No exemplo, a compatibilidade semântica poderia significar verificar se o tamanho ou o tipo de letra do número feito por Isabela é exatamente o que Pedro necessita. Transformação, ou redução, é uma habilidade essencial que auxilia na resolução de problemas [Armoni et al. 2006].

Transformar problemas envolve identificar como instâncias de um problema podem ser vistas também como instâncias de outro problema. A transformação, ou redução, fornece um método poderoso para comparar e relacionar modelos. Além de permitir o reuso, é um conceito essencial para investigar os limites do Computação.

A decomposição, a generalização e a transformação são técnicas importantes para construir as descrição abstrata de um processo, ou seja o modelo ou algoritmo que é a

solução de um determinado problema. Dominar essas técnicas permite entender, resolver e analisar problemas de forma mais eficiente. Mas além de utilizar técnicas isoladamente, é necessário um método sistemático para abordar a resolução de problemas. A resolução de problemas deve começar com a definição do problema a ser resolvido e a definição dos requisitos. Em seguida, é çonstruído um esboço da solução (especificação), que tipicamente não é código, é uma descrição abstrata da solução. Depois pode-se ir gradualmente refinando esse modelo chegando em níveis mais concretos até se obter um programa que implemente a solução desejada (se for o caso). Quando a proposta de solução estiver pronta, é necessário realizar uma análise, pois o simples fato de um programa existir não garante que ele se comporta da forma desejada. A análise deve envolver questões como *Minha solução realmente resolve o problema? Poderia ser implementada de forma mais eficiente? Poderia ser aplicada em outros contextos (seriam necessárias modificações)? A solução é fácil de compreender?* Mas também é necessário levantar questões relacionadas a aspectos não técnicos, como *A solução tem um viés? Ela ajuda a sociedade de alguma forma? Pode ser usada para fins errados (e como isso pode ser evitado)?* A Computação fornece ferramentas muito poderosas, portanto, questões éticas e sociais devem ser parte da educação em Computação nas escolas. O pensamento crítico deve ser baseado em fatos científicos e argumentação lógica.

Não estamos defendendo que os estudantes da Educação Básica sejam especialistas em programação ou desenvolvimento de software. No entanto, os fundamentos da engenharia de software são muito úteis em contextos gerais para construir soluções, esses fundamentos fazem parte do desenvolvimento de um pensamento computacional.

5. Considerações Finais

Neste artigo, defendemos que a habilidade que representa a grande contribuição do pensamento computacional é a criação de abstrações. Isso é o que fazemos o tempo todo quando construimos soluções computacionais: definimos novos tipos de dados para representar informações, construímos algoritmos para descrever processos usando muitos níveis de abstração. E construímos essas soluções usando técnicas bem definidas e seguindo processos sistemáticos. Portanto, para realmente desenvolver habilidades de pensamento computacional, devemos focar nos fundamentos da computação e da engenharia de software.

A programação pode ser um meio importante para desenvolver o pensamento computacional, pois provê um arcabouço prático para trabalhar com conceitos abstratos. Portanto, o fato de hoje podermos gerar código automaticamente com inteligência artificial não significa que não devemos ensinar programação nas escolas, pois o objetivo da programação é ser outro: desenvolver habilidades de pensamento computacional. Mas nas escolas, devemos ter um outro olhar sobre as atividades típicas de programação. Por exemplo, para um programador, testar é uma forma de encontrar erros. Para um estudante da Educação Básica, isso também poderia ser visto como uma forma de entender o poder e as limitações do pensamento indutivo ao validar uma hipótese[3] Outra questão relevante seria discutir os impactos sociais de algoritmos de aprendizado de máquina (fortemente

[3]Poderia-se trabalhar os conceitos de raciocínio indutivo e dedutivo através de uma analogia entre realizar testes (indutivo) e fazer uma demonstração matemática (dedutivo) para garantir a correção de um determinado programa.

baseados em raciocínio indutivo). As diferenças entre homem e máquina também podem ser investigadas ao entender os limites da computação.

Outro ponto fundamental é que devemos nos concentrar em princípios e não em sintaxe. Por exemplo, apresentar recursão como *uma função que se chama a si mesma* é conceitualmente diferente de *uma forma de resolver um problema decompondo-o em um problema menor do mesmo tipo*. O último levanta questões como: *quando o processo de decomposição termina? O que é um problema menor? Como podemos comparar problemas?* Questões que os fundamentos da Computação respondem. Não devemos esconder esses fundamentos dos estudantes, eles não são mais difíceis do que os fundamentos da Matemática, Física ou Química, e são muito importantes pois a Computação fornece ferramentas poderosas para entender e transformar o mundo do século XXI.

É importante enfatizar novamente que não estamos defendendo que não se ensine programação. Pode-se desenvolver habilidades de pensamento computacional de forma eficaz via programação, desde que se compreenda o que de fato os alunos devem desenvolver. As habilidades de Computação que são realmente relevantes para todos estão relacionadas a construir modelos computacionais abstratos. Isso é corroborado também por iniciativas recentes como TeachAI, que defendem que o conhecimento sobre as fundamentos da Computação são essenciais na era da Inteligência Artificial [TeachAI and CSTA 2024].

Finalmente, essa discussão estabelece uma demanda de linguagens e ferramentas computacionais para serem usadas nas escolas: elas devem ser construídas para apoiar o objetivo de desenvolver uma abordagem sistemática para construir soluções computacionais criando e trabalhando com abstrações (tanto de dados quanto de processos, e enfatizando o uso de decomposição, especialmente níveis de abstração, generalização e transformação). Além disso, são necessários métodos de avaliação do processo de ensino-aprendizagem adequados para mensurar o desenvolvimento dessas habilidades.

Referências

Aho, A. V. (2011). Ubiquity symposium: Computation and computational thinking. *Ubiquity*, 2011(January):3–8.

Armoni, M., Gal-Ezer, J., and and, O. H. (2006). Reductive thinking in computer science. *Computer Science Education*, 16(4):281–301.

Cansu, S. and Cansu, F. (2019). An overview of computational thinking. *International Journal of Computer Science Education in Schools*, 3(1):11 pp.

Denning, P. and Tedre, M. (2019). *Computational Thinking*. MIT Press.

Denning, P. J. (2017). Remaining trouble spots with computational thinking. *Communications of the ACM*, 60(6):33–39.

Dijkstra, E. (1970). Notes on structured programming. Technical Report 70-WSK-03 (EDW249), Universidade Tï¿½cnica de Eindhoven.

Elliott, S. W. (2017). *Computers and the Future of Skill Demand*. OECD.

Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. (2018). *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall.

Knuth, D. E. (1974). Computer programming as an art. *Communications of the ACM*, 17(12):667–673.

Kramer, J. (2007). Is abstraction the key to computing? *Commun. ACM*, 50(4):36–42.

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J., and Werner, L. (2011). Computational thinking for youth in practice. *ACM Inroads*, 2(1):32–37.

Liskov, B. (2010). The power of abstraction. In *DISC 2010 Distributed Computing*, volume 6343 of *Lecture Notes in Computer Science*, pages 3–3. Springer.

Liskov, B. (2020). Reflections on programming methodology. Talk given at Heidelberg Laureate Forum.

Milner, R., editor (1980). *A Calculus of Communicating Systems*. Springer Berlin Heidelberg.

Papert, S. (1980). *Mindstorms: children, computers and powerful ideas*. Basic Books Inc. Publishers.

Perlis, A. J. (1982). Special feature: Epigrams on programming. *ACM SIGPLAN Notices*, 17(9):7–13.

TeachAI and CSTA (2024). Guidance on the future of computer science education in the age oa ai. https://www.teachai.org/cs. available at https://www.teachai.org/cs.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, 49(3):33–35.

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3717–3725.