ALUNO: João Gabriel Santos Andrade Almeida

Em um célebre ensaio sobre engenharia de software, Frederick Brooks desmistifica a expectativa de soluções milagrosas para os desafios do desenvolvimento de software. Utilizando a metáfora de lobisomens folclóricos – criaturas familiares que se transformam em ameaças inesperadas, exigindo soluções específicas –, Brooks compara projetos de software a esses seres: aparentemente inofensivos no início, mas propensos a se tornarem pesadelos de prazos descumpridos, orçamentos excedidos e produtos falhos.

Brooks defende que não existe uma "bala de prata" – uma única tecnologia ou metodologia capaz de resolver todos os problemas de produtividade, confiabilidade e simplicidade no desenvolvimento de software. Ele explora detalhadamente as razões, apresentando uma análise que permanece pertinente mesmo décadas depois.

O autor identifica quatro características fundamentais do software que o tornam intrinsecamente complexo e impossível de simplificar por completo. A primeira é a complexidade inerente. Em relação ao seu tamanho, o software é mais complexo do que quase qualquer outra criação humana, pois não há repetição de partes – se duas partes fossem idênticas, seriam transformadas em uma sub-rotina. Isso difere de carros, edifícios ou computadores, onde elementos repetidos são comuns. Além disso, ao escalar um sistema de software, não se trata apenas de replicar os mesmos elementos em maior escala, mas sim de aumentar o número de elementos diferentes, que interagem de forma não linear, intensificando a complexidade.

A segunda característica é a conformidade. Ao contrário dos físicos, que confiam em princípios unificadores e leis naturais elegantes, os engenheiros de software lidam com complexidade arbitrária. Eles precisam interagir com sistemas humanos e institucionais projetados por diferentes pessoas, em diferentes épocas, sem qualquer lógica unificadora. Essa complexidade não pode ser eliminada apenas redesenhando o software – ela é externa.

A terceira é a mutabilidade constante. O software muda mais do que qualquer outra coisa que construímos, pois incorpora a função do sistema, e a função é justamente o que mais sofre pressão por mudanças. Além disso, como o software é "puro pensamento", é percebido como mais fácil de modificar do que objetos físicos. Todos solicitam alterações no software porque acreditam ser simples, ao contrário de mudanças em edifícios ou carros, onde os custos físicos são evidentes e desencorajam alterações desnecessárias.

A quarta característica notável é sua natureza invisível. O software carece de uma representação geométrica inerente. Enquanto a Terra possui mapas, os chips têm diagramas e os computadores, esquemas de conectividade, ao tentar diagramar o software, nos deparamos com múltiplos grafos direcionais sobrepostos: fluxo de controle, dados, dependências, sequência temporal e relações de namespace. Frequentemente, esses grafos não são planares, muito menos hierárquicos. Essa característica impede que nossa mente utilize algumas de suas ferramentas conceituais mais poderosas e dificulta significativamente a comunicação entre as equipes.

Brooks estabelece uma distinção crucial entre as dificuldades "essenciais" (intrínsecas à natureza do software) e as "acidentais" (relacionadas às ferramentas e processos utilizados). Ele demonstra que os grandes avanços do passado abordaram principalmente os problemas acidentais. As linguagens de alto nível foram, provavelmente, o avanço mais significativo, proporcionando ganhos de pelo menos cinco vezes em produtividade ao eliminar a complexidade acidental de lidar diretamente com bits, registradores e instruções de máquina. O time-sharing também trouxe melhorias consideráveis, preservando a continuidade do pensamento. No desenvolvimento batch, esquecíamos os detalhes (e, por vezes, até a direção geral) do que estávamos fazendo entre o envio do trabalho e a análise dos resultados. Ambientes de programação unificados como Unix e Interlisp atacaram as dificuldades acidentais de usar programas individuais em conjunto, oferecendo bibliotecas integradas, formatos de arquivo uniformes e ferramentas que se comunicavam facilmente.

Ao analisar as "balas de prata" propostas na época, Brooks se mostra consistentemente cético. Ada e outras linguagens avançadas da época são interessantes, mas são apenas mais linguagens de alto nível: o grande salto já havia ocorrido na primeira transição, ao superar a complexidade acidental das linguagens de máquina. Os ganhos adicionais seriam, necessariamente, menores. A programação orientada a objetos possui aspectos promissores, especialmente tipos abstratos de dados e hierarquias de tipos, mas apenas remove mais complexidade acidental. A complexidade essencial do design permanece inalterada.

A inteligência artificial também não oferece a solução mágica que muitos esperavam. Brooks cita David Parnas, que distingue entre AI-1 (usar computadores para solucionar problemas que antes só humanos resolviam) e AI-2 (técnicas específicas baseadas em regras e heurísticas). Para AI-1, assim que entendemos como o programa funciona, não consideramos mais como inteligência artificial — é um alvo em movimento. Para AI-2, as técnicas são muito específicas para cada domínio. O reconhecimento de fala tem pouca semelhança com o reconhecimento de imagem, e ambos são diferentes de sistemas especialistas. Mais importante: a

dificuldade no desenvolvimento de software não é expressar o que queremos dizer, mas sim decidir o que queremos dizer.

Os sistemas especialistas são notáveis, pois representavam o auge da IA em seu tempo. Brooks vislumbra valor em separar a lógica de raciocínio do saber específico da área, o que possibilitaria reaproveitar os motores de inferência e construir ferramentas para criar bases de conhecimento. Contudo, a maior dificuldade reside na obtenção de conhecimento: encontrar peritos que consigam expressar claramente o raciocínio por trás de suas ações e criar métodos eficazes para coletar e refinar esse conhecimento.

A "programação automática" é outra ilusão. Conforme aponta Parnas, "programação automática" sempre significou "programar usando uma linguagem mais avançada do que as existentes". Ela funciona bem em áreas muito específicas com atributos favoráveis – problemas definidos por poucos parâmetros, diversas soluções conhecidas e regras claras para escolher as técnicas. É difícil conceber como isso se aplicaria a sistemas de software comuns.