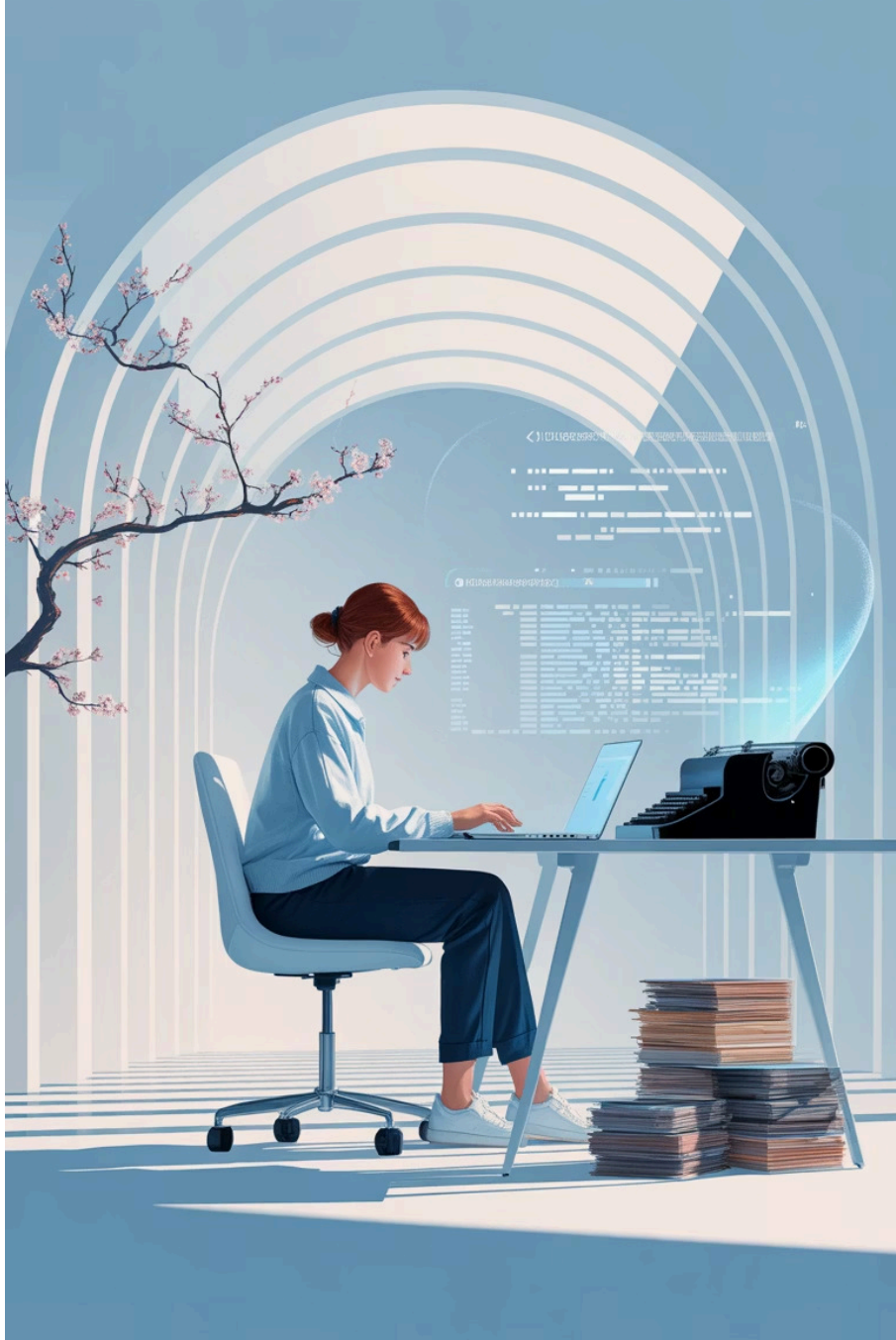
A Crise, Evolução e o Futuro da Engenharia de Software

Uma jornada histórica sobre os desafios perenes, avanços metodológicos e dilemas contemporâneos que moldam nossa profissão e seu futuro.



Agenda

Origens e Crises

A crise perene do software: dos anos 60 à era da IA

Evolução Histórica

Da programação caótica às metodologias estruturadas

Desafios Contemporâneos

Overengineering vs. Underengineering e a "Cracolândia Digital"

O Caminho à Frente

Como resgatar a engenharia de software através da responsabilidade e conhecimento

A Crise Perene do Software

Uma constante em nossa história

A Primeira Crise do Software (1968)

Em 1968, na Alemanha, especialistas concluíram que, apesar do avanço do hardware, o desenvolvimento de software estava **"totalmente fora de controle"**.

- Projetos atrasavam constantemente
- Orçamentos estouravam sem controle
- Sistemas falhavam sem explicação aparente
- Código era uma **"verdadeira bagunça, sem organização, sem padrão, sem separação de responsabilidade"**

O pensamento dominante: *"Se funcionou, deixa assim"*, devido à extrema dificuldade de manutenção.

Causas da Crise Original

Descompasso Tecnológico

"A principal causa é o descompasso entre as melhorias alcançadas em termos de poder computacional e a habilidade dos programadores de efetivamente se organizarem para fazer uso dessa capacidade." - *Alisson Vale*

Crise de Oferta

Demanda por software muito maior que a capacidade de desenvolvimento existente, criando pressão por entregas rápidas sem metodologia.

Crise de Manutenção

Projetos mal estruturados e recursos escassos tornavam a manutenção de sistemas existentes um pesadelo operacional contínuo.

De Crise a Realidade Perene

> "A crise do software nunca foi uma crise no sentido tradicional, mas sim uma **descrição de uma realidade que não conseguíamos enxergar**."

- Alisson Vale

Ao longo de décadas, percebemos que a "crise" é, na verdade, um conjunto de **problemas perenes** inerentes à complexidade do desenvolvimento de software, não uma situação temporária que seria resolvida com uma única solução definitiva.

Esta percepção mudou fundamentalmente como abordamos os desafios na engenharia de software: de buscar uma "cura" para aprender a "conviver e gerenciar" a complexidade inerente.

A Crise de Significado (Século XXI)

Hoje, enfrentamos uma **"crise de significado"** no desenvolvimento de software:

- Abundância de fatos ("quem, quando, o quê e onde")
- Escassez de compreensão e sabedoria
- Alto turnover nas equipes
- Confusão entre indivíduos e organizações sobre seus propósitos

> "Sabemos tantas coisas que não compreendemos! Toda a sabedoria de fatos é, a rigor, incompreensível, e só pode justificar-se estando a serviço de uma teoria".
>
> - Ortega Y Gasset

A Crise Atual (Era da IA)

"Uma geração inteira de programadores está perdendo a capacidade fundamental de raciocinar, projetar e implementar soluções sólidas."

Promessa vs. Realidade

A era da IA prometia aumentar a produtividade, mas trouxe uma "regressão disfarçada de produtividade".

Resultado Prático

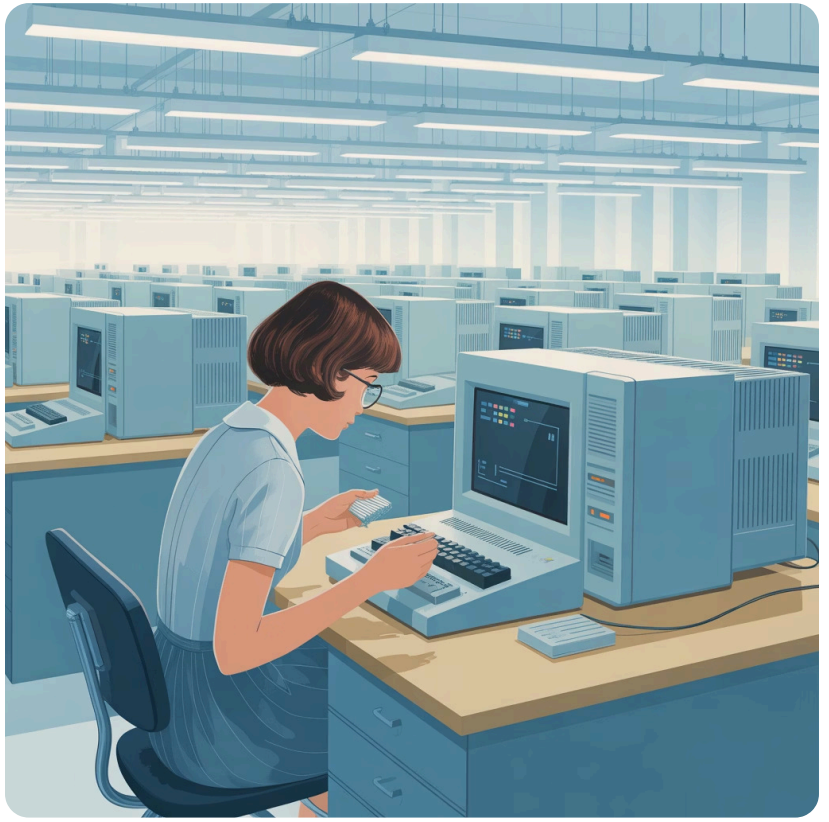
"Código cada vez mais porco" e sistemas mais frágeis e vulneráveis.

Estamos enfrentando uma "crise sem precedentes na engenharia de software" onde a terceirização do pensamento para a IA está comprometendo fundamentos essenciais da disciplina.

A Evolução da Engenharia de Software

Da desordem à busca pela ordem (e ao caos recorrente)

Década de 1960: A Origem do Caos



Não existia ainda uma disciplina formal de engenharia de software, apenas "um bando de programadores escrevendo código direto no metal".

- Programação manual em cartões perfurados
- Uso de linguagens de máquina ou assembly
- Ausência de metodologias estruturadas
- Falta de ferramentas de depuração avançadas
- Documentação praticamente inexistente

O foco era fazer funcionar, sem preocupação com manutenção futura ou reutilização de código.

Década de 1970: A Semente da Organização

Programação Estruturada

Edsger Dijkstra (1970) propõe estruturar o código com base em "sequência, condição (if-else) e repetição (loop)"

Encapsulamento de Decisões

David Parnas (1972) introduz conceito de "esconder complexidade através do design de código"

Evolução das Linguagens

Pascal, Algol, Modula e Ada incentivam clareza, modularização e legibilidade

A programação começa a ser vista como uma disciplina estruturada, exigindo metodologia e organização. A ideia de **"deixa assim que funciona"** começa a ser questionada por uma visão mais sistemática do desenvolvimento.

Década de 1980: A Ascensão da OO

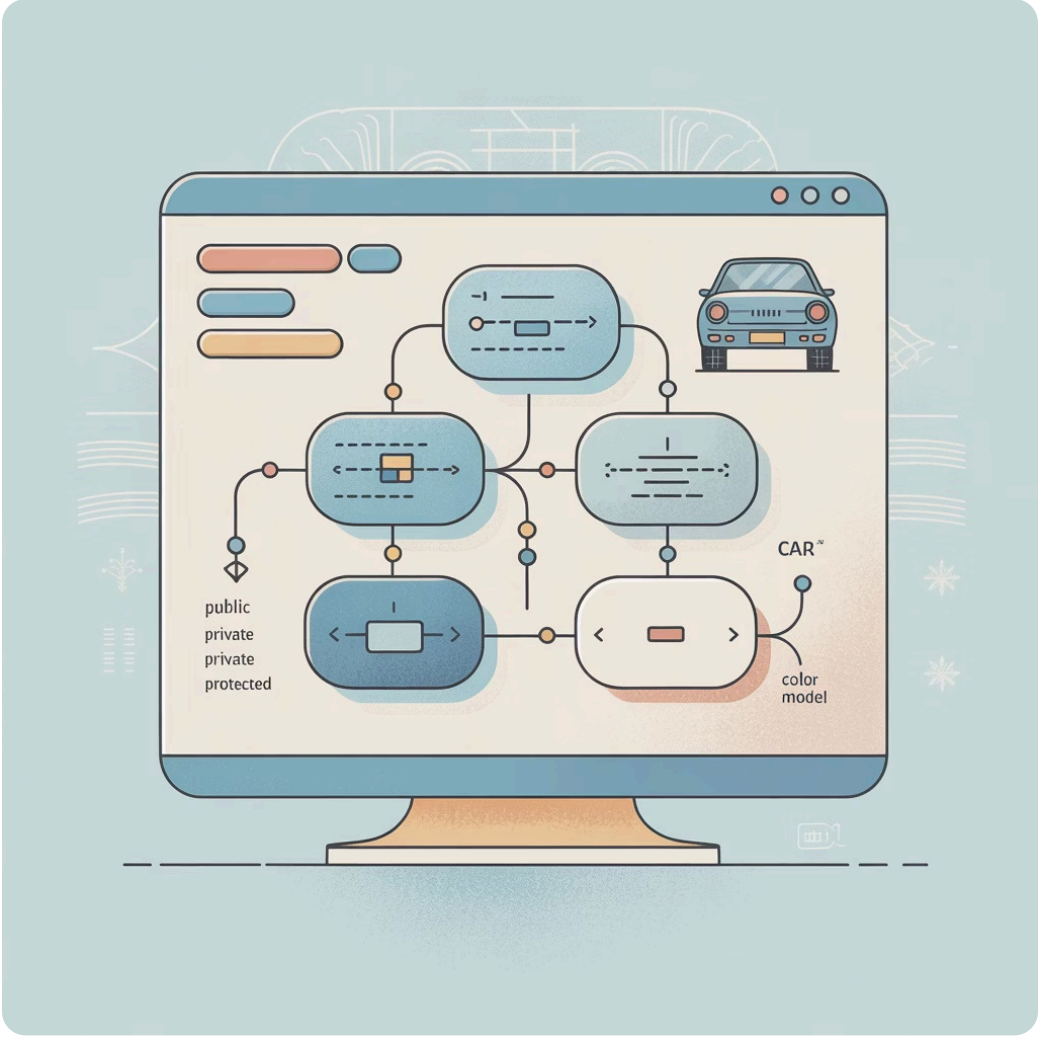
Engenharia de Software

Ganha força a ideia de tratar software como algo que **"precisa de arquitetura e engenharia"**, não apenas programação ad-hoc.

Programação Orientada a Objetos

Popularizada por C++ e Objective C, a OO propunha "quebrar o software em partes menores, reutilizáveis e isoladas, cada uma com uma única responsabilidade".

Isso representou uma verdadeira **"mudança de mentalidade"**, transformando o programador em um **"arquiteto de soluções"**.



Década de 1990: O Boom e a Padronização

Design Patterns (1994)

A "Gangue dos Quatro" cataloga 23 padrões de projeto, criando **"um vocabulário universal para descrever estruturas de software que resolviam problemas mais comuns"**.

Java (1995)

Popularizou definitivamente a Orientação a Objetos, com foco em portabilidade e legibilidade. A OO se tornou o **"padrão obrigatório do mercado"**.

Esta década consolidou metodologias estruturadas e estabeleceu uma base comum para comunicação entre desenvolvedores, elevando o nível de profissionalização da área.

Anos 2000: Maturidade e Novas Abordagens

Princípios SOLID (Robert Martin)

Conjunto de princípios para **"evitar código frágil, acoplado e bagunçado"**, criando **"uma geração de programadores mais conscientes do impacto do design de código"**.

Manifesto Ágil (2001)

Criado por 17 desenvolvedores, buscava uma alternativa mais **"flexível, colaborativa e focada na entrega de valor pro cliente"**, em oposição às metodologias rígidas da época.

Domain-Driven Design (Eric Evans, 2003)

Nova forma de pensar software, **"modelando olhando pro domínio do negócio"** e fazendo **"desenvolvedores e especialistas falarem a mesma língua"**.

O Dilema entre Estratégia e Arquitetura

Uma busca contínua por equilíbrio

Estratégia vs. Arquitetura: Definições

Estratégia

O "porquê" e o "o quê" do software

- Visão de longo prazo
- Objetivos de negócio
- Decisões tecnológicas
- Planejamento de recursos
- Metodologias de desenvolvimento

Arquitetura

O "como" do software

- Estrutura do sistema
- Componentes e suas relações
- Padrões e práticas
- Escalabilidade e manutenibilidade
- Decisões de implementação

O grande desafio não é determinar **"qual é mais importante"**, mas **"como equilibrá-las no contexto do seu projeto"**.

A Evolução da Relação

1 Era da Programação Estruturada

"Estratégia? Qual estratégia?" era a mentalidade dominante, com foco apenas em problemas imediatos.

Resultado: "sistemas legados que até hoje são pesadelos para manter".

2 Era da Orientação a Objetos

"A estratégia começou a ganhar importância: era preciso planejar antes de codificar."

Arquitetura começa a ser valorizada como disciplina fundamental.

3 Era dos Padrões de Projeto

Representam a **"fusão perfeita entre estratégia e arquitetura"**, oferecendo tanto benefícios estratégicos quanto arquiteturais.

4 Era das Arquiteturas Modernas

Microsserviços, Cloud-Native e Serverless buscam a **"fusão de estratégia e arquitetura"**, onde "a melhor arquitetura emerge de equipes auto-organizadas."

Os Novos Vilões

Overengineering e Underengineering

Overengineering: O Excesso de Engenharia

Após a "era de ouro" da engenharia de software (décadas de 1990 e 2000), a obsessão por padrões e boas práticas levou à **"complexidade desnecessária"**.

- Princípios se tornaram dogmas rígidos
- Padrões se tornaram obrigações inquestionáveis
- **"Cada novo problema simples ganhava uma arquitetura de guerra"**
- Simplicidade sufocada por abstrações excessivas

O overengineering foi temporariamente silenciado pela recessão global (2007-2008), mas ressurgiu posteriormente.

> "O certo era fazer complexo, mesmo quando o simples resolvia."

Underengineering: A Escassez de Engenharia

Explosão Digital

Com o boom dos smartphones e a corrida por presença digital, empresas priorizaram a **"entrega, entrega e entrega"** acima de tudo.

Ágil Distorcido

Metodologias ágeis foram **"distorcidas e deturpadas"**, transformando a daily em **"interrogatório"** e o programador em **"entregador de tickets"**.

Pressão da Pandemia

A crise da COVID-19 intensificou a pressão por velocidade, com o programador sob **"pressão insana"** para entregar, sacrificando o **"pensamento técnico"**.

Resultado: **"Sistemas começaram a ser feitos sem estrutura nenhuma"**, sem domínio, sem testes e com **"regras de negócio jogadas dentro de controller"**.

A Era da Inteligência Artificial

E a "Cracolândia Digital"

IA Generativa: Promessa e Realidade

A Promessa Sedutora

Ferramentas como GPT, GitHub Copilot, ChatGPT, Gemini, Cursor e Claude surgiram com a promessa de:

- Gerar código completo e funcional
- Explicar conceitos complexos
- Refatorar código legado
- Analisar vulnerabilidades

"Digite um comentário e ganhe um bloco de código funcional" - uma proposta irresistível para muitos.

A Realidade Problemática

Começaram a surgir problemas como:

- Código duplicado e vulnerável
- Soluções de procedência duvidosa
- Sistemas frágeis e difíceis de manter

A "Mediocridade Regurgitada"

> "A IA aprendeu com a nossa mediocridade e hoje ela tá apenas regurgitando o lixo que a gente jogou lá atrás e não voltou para catar."

O Problema do Treinamento

A IA foi treinada com **"tudo que tinha por aí"**:

- Repositórios públicos cheios de código lixo
- Respostas do Stack Overflow de qualidade variável
- Documentação incompleta
- Posts antigos de fóruns

O "Vibe Coding"

Surge uma **"nova era de programadores que idolatram a terceirização do raciocínio"**, onde:

- **"Pensar demais atrasa"**
- **"Estudar boas práticas é perda de tempo"**
- **"Arquitetura é frescura"**

Um **"nome bonito para justificar a preguiça"**.

⊗ Consequência: "A próxima grande crise do software não é mais uma possibilidade, ela é só uma questão de tempo."

O Caminho para Salvar a Engenharia de Software

Responsabilidade e Conhecimento

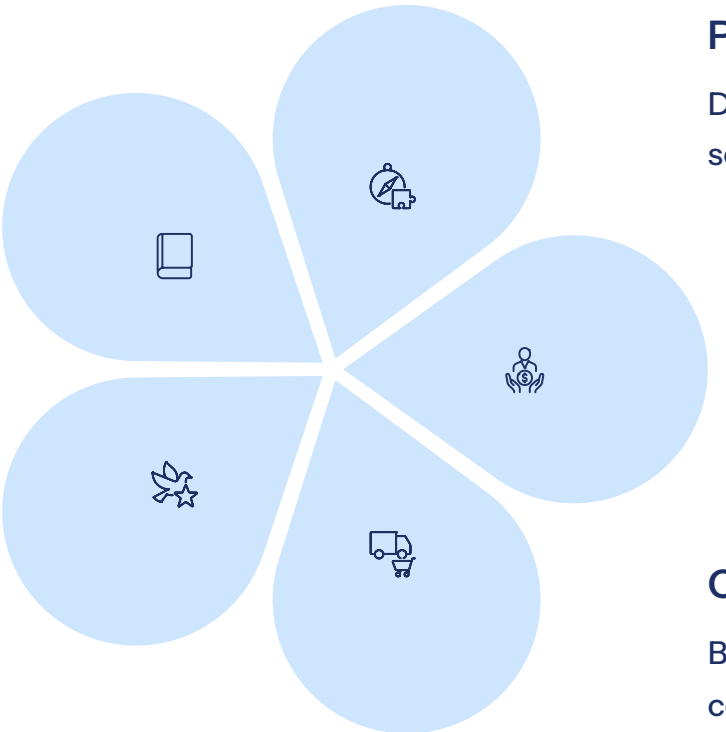
Como Resgatar a Engenharia de Software

Reaprender Fundamentos

"Voltar a estudar os fundamentos" da engenharia de software, algoritmos e estruturas de dados.

Consciência do Impacto

"Código porco também mata". Software roda em hospitais, carros autônomos, aviões e equipamentos que mantêm pessoas vivas.



Pensamento Crítico

Desenvolver **"critério"** e capacidade de avaliar soluções, não apenas copiar código.

Responsabilidade Técnica

Ter **"vergonha na cara quando for escrever um código"** e assumir responsabilidade pela qualidade.

Compreensão de Domínio

Buscar **"significado"** além dos fatos, conectando-se ao propósito do negócio.

> "A profissão de programador não vai ser destruída pela IA, ela vai ser destruída pela preguiça e pela burrice."

O futuro da engenharia de software depende de profissionais que entendam que a IA **"só reflete, ela não decide, ela não pensa, ela não projeta sistemas"**. Isso ainda é **"papel do engenheiro, do programador, do arquiteto de software que entende o impacto de cada linha escrita"**.