INSTITUTO FEDERAL DO RIO GRANDE DO SUL

GABRIEL H. SCHAEFFER

RESUMO DO VÍDEO:

A HISTÓRIA NÃO CONTADA DO COLAPSO DA ENGENHARIA DE SOFTWARE

ERECHIM
2025

1.0 Introdução

O vídeo, publicado e apresentado por Renato Augusto no YouTube, propõe uma análise crítica sobre a atual crise na engenharia de software, destacando uma regressão no raciocínio e capacidade de implementar soluções sólidas por parte da nova geração de programadores, muito influenciada pelas inteligências artificiais, traçando um paralelo com as crises que ocorreram na área do desenvolvimento quando o mesmo ainda estava em seus primórdios, nas décadas de 60 e 70.

2.0 Marcos da Engenharia de Software

No vídeo, Renato apresenta um panorama histórico, iniciado na crise do software nos anos 60, passando pela consolidação da engenharia de software, até a ascensão das boas práticas e técnicas nos anos 80, 90 e adiante.

2.1 1960-70: A Origem do Caos

Em uma época onde computadores ainda ocupavam salas inteiras, eram programados em cartões perfurados manualmente e tinham acesso extremamente restrito, surge a crise do software, consequência direta da impotência dos métodos de desenvolvimento de software diante de um grande avanço do hardware.

Atrasos nos projetos, falhas sem justificativa, falta de orçamento e prejuízos milionários causados por bugs indetectáveis eram comuns na época, forjando um pensamento conservador entre os programadores, resumido na máxima “se funcionou, deixa assim”.

Nesse contexto de desenvolvimento, a manutenção de projetos era extremamente complexa e centrada em um nicho muito pequeno de programadores dentro da empresa. Em muitos dos casos, apenas um desenvolvedor era capaz de compreender e manter o sistema.

Nesse sentido, percebeu-se que era necessário dar uma atenção especial à parte de organização do projeto e parar de escrever códigos improvisados.

2.2 1970-80: Os Primeiros Esforços

Sob um olhar crítico acerca do caos da década passada, surgem os primeiros esforços rumo à formalização do desenvolvimento de software. Artigos pioneiros já esboçavam, à época, vislumbres de padrões de desenvolvimento que hoje em dia utilizamos extensivamente, como o encapsulamento. Essa década serviu como o marco inicial da revolução na Engenharia de software que viria a ocorrer na década seguinte.

2.3 1980-90: O novo paradigma

Paralelamente à popularização dos computadores pessoais, uma nova forma de abordar o desenvolvimento de software começou a florescer, enraizada nos princípios iniciais estabelecidos na década passada.

Nesse contexto, o que conhecemos hoje por “engenharia de software” começa a ganhar força, e a Programação Orientada a Objetos começa a se popularizar através de linguagens como o C++, que resgataram e refinaram os conceitos de POO de linguagens mais antigas.

2.4 1990-2010: A Era de Ouro

Em contraste com o caos das décadas passadas, os anos 90 começaram com um ar muito mais confiante acerca do desenvolvimento de software, com o conceito de POO se consolidando de vez.

Contudo, nesse contexto, a orientação a objetos começou a apresentar algumas falhas no que tange à padronização de código. A forma de programar era muito exclusiva de cada empresa, dificultando a transição de programadores.

Assim, surgiram tentativas de padronização, que se consolidaram através do livro “Design Patterns”, que catalogou 23 padrões de projetos. Diante disso, surgiu uma força que unificou a comunidade de desenvolvimento. Criou-se, finalmente, uma “linguagem universal” que todos podiam compreender.

Ademais, o desenvolvimento da linguagem Java, inspirada em C++, consolidou de vez a orientação a objetos, que virou o padrão oficial do mercado. Além do Java, outras linguagens mais modernas também davam os primeiros suspiros, como Python, PHP e C#.

Paralelamente, nascem outras alternativas em resposta à padronização corporativa, buscando algo mais flexível, como o “manifesto ágil”, diversificando o ecossistema da engenharia de software.

Posteriormente, surgem outros livros revolucionários. O “Clean Code”, que pontua princípios que ajudam a manter o código limpo, e o “Object Calisthenics”, que enumera 9 regras que impulsionam a qualidade de um código orientado a objetos.

Contudo, o vasto ecossistema criado acerca do desenvolvimento de projetos começou a se tornar esmagador em muitos casos. Projetos simples sofriam com o *Over-engineering,* visto que para abordar problemas simples eram criadas soluções excessivamente complicadas. Para piorar, a crise financeira da época ofuscou os esforços na área da engenharia de software, já que agora os holofotes estavam voltados à mera sobrevivência e permanência no mercado.

2.5 2010-20: O Tempo está Acabando

Diante da atenuação da crise financeira, as empresas começaram a se restabelecer e voltar ao normal. Contudo, outra revolução tecnológica se mostrava iminente: a ascensão dos smartphones.

O novo mercado de telefones celulares era a chance perfeita para recomeçar do zero e redimir os erros do passado, priorizando o simples e eliminando o over-engineering. Contudo, a oportunidade foi desperdiçada, e permanecemos reféns da complexidade técnica.

Por outro lado, nessa mesma época, buscando a inserção rápida no novo mercado do digital, surge outro problema, o *under-engineering,* consequência da priorização da entrega pela qualidade. Isso formou uma geração de programadores que não sabe projetar software de

fato, visto que as empresas estavam deixando essa parte de lado em busca de se consolidar no mercado mobile mais rápido do que a concorrência.

2.6 2020-Presente: Fim da Linha

Com a eclosão da pandemia do Covid-19, uma crise global se instaurou. Empresas fechando, demissões em massa e mudanças drásticas na forma de trabalhar, visto que diante deste cenário o isolamento social era obrigatório.

Nesse contexto, o software se tornou o alicerce do mercado de trabalho, visto que toda a forma de emprego e serviço agora era feita no modelo home office, dependendo de programas de computador de comunicação. Investimentos massivos em software impulsionaram as empresas e o programador agora era extremamente valioso.

Contudo, com a necessidade crescente por tecnologia, se intensifica a pressão sobre o programador, que precisava trabalhar para fazer entregas rápidas e manter sistemas de larga escala. Nesse prisma, o pensamento técnico foi mais uma vez deixado de lado, valorizando mais um produto final funcional do que um projeto de qualidade.

Paralelamente, modelos de linguagem generativa começam a surgir em grande escala, com o anúncio do GPT da OpenAI, seguido pelo Github Copilot, transformando a forma como se escreve código. Contudo, a qualidade da geração de código por parte das IAs é questionável, visto que muitos dos dados que as alimentam são de qualidade duvidosa. Mesmo assim a IA passou a ser usada extensivamente no desenvolvimento de software, às vezes terceirizando totalmente o raciocínio que deveria partir de humanos, formando uma geração que depende de IA para desenvolver projetos, e que deixa de lado a arquitetura e qualidade do software: os vibe-coders. Assim, o futuro se encontra incerto, e cabe a nós, programadores em formação mudar isso, aprendendo a escrever código escalável, seguro e de qualidade, sem depender de ferramentas generativas, mas sim usá-las como assistentes.

3.0 Tecnologias que um Engenheiro de Software deve conhecer

Um engenheiro de software precisa dominar diversas tecnologias e práticas de desenvolvimento. Isso inclui compreender paradigmas de programação, como a POO, princípios de design como SOLID. Também são essenciais as práticas de engenharia de software voltadas à qualidade, como Clean Code, TDD (Test-Driven Development), Domain-Driven Design (DDD) e arquiteturas modernas (microsserviços, Clean Architecture), que garantem que o software evolua de forma sustentável.

Além disso, é importante ter fluência em linguagens e ferramentas relevantes para o ecossistema atual, como C++, Java, Python e frameworks como Spring, bem como domínio de ambientes de produção modernos. Por fim, embora ferramentas de IA generativa estão se tornando cada vez mais presentes no desenvolvimento, o verdadeiro diferencial continua sendo o pensamento crítico por parte do programador.

4.0 Análise Crítica

O vídeo oferece uma visão histórica e crítica da engenharia de software, ressaltando que muitos dos desafios atuais são reflexos de crises identificadas desde os anos 60. Apesar de avanços como programação orientada a objetos e padrões de projeto terem promovido organização e sustentabilidade, práticas recentes têm contribuído para uma nova deterioração do desenvolvimento de software.

A distorção do Manifesto Ágil, juntamente com o over engineering e o Vibe Coding, revela uma cultura que prioriza a entrega rápida e superficialidade em vez de qualidade e planejamento técnico. O uso excessivo de inteligência artificial, ao substituir o pensamento crítico, agrava essa perda de fundamentos, e forma uma geração de programadores que não sabem desenvolver e analisar software de forma crítica e racional.