



Khalil Stemmler

S O L I D

S O L I D

Introduction to Software Architecture and Design Principles with Node.js & TypeScript

| Learn to write testable, flexible &
maintainable code.

SOLID: O Projeto de Software e manual de arquitetura

1. Sobre este livro on-line 1.

Este livro é atualizado continuamente 2.

Baixando uma cópia 3. . .

Linha do

tempo 4.

Atualizações 5. Acessando
o livro 6.

Comentários 2.

Introdução 1.

Minha história 1. Minhas
primeiras pesquisas 2. Por

que escrevi isso livro 1. Habilidades práticas de
design não estão sendo ensinadas 2.

Design ruim é extremamente caro 3. Há
muito a aprender com o passado 3. Como escrevi este livro

1. Dois tipos de sabedoria: Sophia e Phronesis 2.

Sophia 3.

Phronesis 4.

A abordagem deste livro à sabedoria

4. Como dominar o design de software

1. Etapa 1: Identificar o propósito do código

2. Etapa 2: Identificar as qualidades do código que o tornam
realizar bem o seu propósito 3.

Passo 3: Dominar as técnicas daqueles que o estão fazendo bem 4.

Passo 4: Desenvolver nossos próprios princípios praticando e
construindo experiência

5. Como este livro o beneficiará 6.

Como este livro está organizado

7. TypeScript

8. Recursos 1.

Artigos

3. Parte I: Up to Speed

1. 1. Complexidade e o mundo do design de software 1.

Objetivos do ____

capítulo 2. O objetivo do design de
software 3. ____

Complexidade 4. O que é complexidade em software?

1. A complexidade é um fenômeno incremental 5.

Tipos de complexidade 1. ____

Complexidade essencial ____

2. Complexidade acidental 6.

Causas da complexidade acidental ____

1. Obtendo os requisitos errados 2.

Dependências (acoplamento forte) ____

3. Obscuridade (baixa coesão) ____

4. Mais desenvolvedores

5. Recursos de linguagem e API 6.

Otimização prematura 7.

Outras causas ____

7. Como detectar a complexidade

1. Ripple

2. Carga cognitiva

3. Fraca capacidade de ____

descoberta 4. Pobre

compreensibilidade 8. ____

Mitigando a complexidade 1. Programação

tática versus estratégica 2. Programação extrema: a metodologia

original de ____

desenvolvimento ágil 3.

Práticas técnicas 9. Valores (da Programação Extrema)

1. Feedback

2. Simplicidade

3. Comunicação 10. ____

Resumo 11. ____

Exercícios 12. ____

Referências 1. ____

Artigos 2. ____

Livros ____

3. Artigos

2. 2. Artesanato de software 1.

Objetivos do capítulo

2. Profissionalismo 3.

Uma breve história do desenvolvimento de software

1. Programação da velocidade de aceleração (50s)

2. A crise do software dos anos 60-80 3. Bolha

pontocom, OOP e Extreme Programming (1995 – 2001)

4. Agile (2001 - hoje)

5. O Manifesto Ágil 6. A Era

(Equivocada) do Ágil 7. Por que o Ágil não funcionou?

8. Software Craftsmanship (2006 - hoje)

9. O Manifesto do Software Craftsmanship 10. De volta ao básico (XP)

4. Artesanato: Profissionalismo no desenvolvimento de software 1. Definição

2. Você é um

artesão de software?

5. Entendendo o manifesto 1. Não

apenas um software funcional, mas também um software bem elaborado
Programas

2. Não apenas respondendo à mudança, mas também agregando valor

constantemente 3. Não apenas indivíduos e interações, mas também um
comunidade de profissionais

4. Não apenas colaboração com o cliente, mas também parcerias
produtivas 6.

Princípios de artesanato 1. Para

escrever um software bem elaborado...

2. Para agregar valor constantemente...

3. Envolve-se na comunidade...

4. Considere-se um parceiro..

7. Como criar hábitos a partir dos princípios do artesanato 8. Resumo

9. Exercícios

10. Referências

1. Livros 3.
3. Uma visão de 5.000 pés de design de software 1.
Objetivos do capítulo
2. Níveis de design 3. A
pilha e roteiro de design e arquitetura de software 4. Recurso: A pilha 5. Recurso: O mapa 6. Etapa 1: Limpar
Código 7. Etapa 2: Paradigmas
de programação 8. Etapa 3:
Programação orientada a objetos e modelagem
de domínio 9. Etapa 4: Princípios de design 10. Etapa 5: Padrões de design 1.
Críticas aos padrões de design 11.
Etapa 6: Princípios de arquitetura
12. Etapa 7: Estilos de arquitetura
1. Estrutural 2. Baseado em mensagens 3.
Distribuído 13. Etapa 8: Padrões de arquitetura 14.
Etapa 9: Padrões corporativos 15.
Resumo 16. Exercícios 17. Referências 1.
Artigos 2. Livros

-
-
-
-
-
-
-
-
-
4. Parte II: Humanos e Código 1. 4.
Desmistificando o Código Límpio 1.
Objetivos do capítulo
2. Afinal, o que é código limpo?
1. De acordo com a comunidade 2. De acordo com os especialistas 3.
Padrões de codificação limpa 1.
O que é um padrão de codificação?
2. Por que precisamos de padrões de codificação?
4. Design Simples 5.
Surgimento

6. Estrutura x Experiência do Desenvolvedor

1. Estrutura 2.

Experiência do desenvolvedor

3. Experiência do desenvolvedor para empresas de ferramentas para

desenvolvedores 4. APIs não

são apenas URLs 5. A experiência do desenvolvedor é importante para
todas as empresas 7. Equilíbrio entre estrutura e experiência

do desenvolvedor 1. Estrutura e experiência do desenvolvedor na prática: Angular
e reagir

2. Estrutura x Experiência do Desenvolvedor na Prática: Objeto

Programação Orientada vs. Programação

Funcional 8.

Conclusão 9.

Resumo 10.

Exercícios 11.

Referências

1. Artigos

2. Livros 2. 5. Design Centrado no Homem para Desenvolvedores

1. Objetivos do

capítulo 2. Design centrado no ser
humano 1. O que é?

2. Como isso é útil para nós?

3. Casos de uso do

desenvolvedor 3. Como descobrimos as coisas — A psicologia do ser humano
Ação

1. Descoberta e compreensão 2. Os 7

estágios da ação 3. Princípios
fundamentais do design

4. Conhecimento na Cabeça versus Mundo 1.

Conhecimento na Cabeça 2.

Conhecimento no Mundo 3. Como
isso é útil para nós?

5. Affordances 1.

Exemplos da vida real 2.

Por que isso é útil?

3. Recursos em linguagens de programação 4.

Recursos em padrões de projeto

5. Como fazer bem as possibilidades

6. Significantes

1. Exemplos da vida real

2. Por que isso é útil?

3. Significantes intencionais no desenvolvimento de software 4. Significantes acidentais no desenvolvimento de software 5. Como usar bem os significantes

7. Restrições

1. Exemplos da vida real (restrições físicas)

2. Exemplos da vida real (restrições culturais)

3. Exemplos da vida real (restrições semânticas)

4. Exemplos da vida real (restrições lógicas)

5. Exemplos de restrições no desenvolvimento de software 6. Como usar bem as restrições

8. Mapeamento

1. Exemplos da vida real

2. Por que isso é útil?

3. Como fazer mapeamentos

bem 4. Agrupamento

5. Proximidade 6. Agrupamento no desenvolvimento de software 7. Proximidade
no

desenvolvimento de

software 9. Feedback 1.

Tipos de erros 2. Exemplos da vida real 3. Por que isso é útil?

4. Feedback no desenvolvimento de software

5. Como fazer feedback bem 10.

Modelos conceituais 1.

Exemplos da vida real 2.

Por que isso é útil?

3. Modelos conceituais no desenvolvimento de software

4. Como fazer bem modelos conceituais

11. Testando seu código para limpeza

1. Pergunte: O que meu código faz?

2. Pergunte: Encontre o código que precisa ser

alterado 3. Pergunte: Altere este código sem introduzir bugs

12. Resumo 1.

Uma filosofia para código amigável ao ser humano

13. Exercícios 1.

Considere os fundamentos da interação ao projetar software 2.

Aprenda mais sobre

Human-Centered Design 14. Recursos 1. Artigos 2. Livros

3. 6. Organizando

as coisas 1

Objetivos

do capítulo 2. Sintomas de

estrutura de projeto

ruim

1. Esquecer o que o sistema faz 2. Recursos

difíceis de localizar 3. Energia

gasta alternando arquivos 3. Abordagens de estrutura de projeto

comuns 1. Apenas evolua com o tempo 2. Pacote

por infraestrutura/tecnologia/tipo

3. Pacote por domínio 4. Recursos (casos de uso) . . .

1. Os recursos são fatias verticais 2.

Os recursos são o ponto de entrada 3.

Arquitetura gritante e pasta orientada a recursos

estrutura

5. Estrutura do projeto front-end orientado a recursos 1.

Páginas

2. Estrutura do projeto

6. Estrutura do projeto back-end orientado a recursos 1.

Casos de uso

7. Conteúdo

compartilhado 8. Benefícios de uma organização de projeto

orientada a recursos 9. Regras de organização do projeto

1. Use convenções de alto nível 2.

Pertence a um recurso ou é compartilhado 3.

Agrupe arquivos relacionados a um recurso específico

4. Combata o framework 10.

Resumo

11. Referências

1. Artigos 2.

Livros 12.

Exercícios

4. 7. Documentação e repositórios 1.

Objetivos do _____

capítulo 2. Objetivos do usuário e perguntas a serem abordadas,

em um repositório 3. Empurre a

complexidade para baixo 4.

Testes como .

documentação

5. Resumo 6.

Exercícios

7. Recursos 1. Artigos

5. 8. Nomeando coisas 1. Os sete

princípios de nomeação 2. Princípio de nomeação #1.

Consistência e singularidade

1. Consistência na nomeação 2.

Consistência

com tudo 3. Singularidade 4. Regra: Evite usar palavras semelhantes para expressar

conceito (nomes do dicionário de sinônimos)

5. Regra: Siga a linguagem de programação e o projeto _____
(nomear) convenções de

codificação 6. Regra: Evite nomes de variáveis muito _____

semelhantes por erros ortográficos (ou usando grafias alternativas corretas)

7. Regra: não use o mesmo nome para expressar diferentes
conceitos dentro do mesmo namespace _____

8. Regra: Não recicle nomes de variáveis 9.

Regra: Nomes devem ser únicos, independentemente do caso 3.

Princípio de nomenclatura nº 2: Compreensibilidade

1. Conhecimento no mundo 2.

Representação de conceitos do mundo real

3. Nomes específicos de domínio são um investimento de longo
prazo 4. A camada de domínio descreve as principais regras

de negócios 5. Nomeando as coisas em camadas

mais técnicas 6. Usando arquitetura e estruturas para ditar como

nomear as coisas

7. Existe uma construção para tudo 8.

Regra: Não coloque sílabas maiúsculas aleatoriamente dentro das palavras

9. Regra: Evitar nomes com dígitos 10.

Regra: Use nomes pronunciáveis 11.

Regra: Use o princípio CQS (Command Query Separation) para nomear métodos

12. Regra: Documente efeitos colaterais em métodos com vários efeitos colaterais

notáveis 13. Regra: Use conceitos de domínio para se referir a coisas do

negócio 14. Regra: Use conceitos técnicos para expressar técnicas
coisas

15. Regra: Use inglês simples (gramaticalmente correto) (sem erros ortográficos)

16. Regra: Não omita vogais ou abrevie desnecessariamente
palavras

17. Regra: Evite nomes enganosos 18.

Regra: Evite usar negativos em métodos que retornam booleano

19. Regra: Evite nomes irrelevantes

20. Faça distinções significativas

4. Princípio de nomenclatura nº 3:

Especificidade 1.

Especificação excessiva

2. Múltiplas variantes 3. Falta de contexto necessário 4.

Especificação insuficiente 5.

Anotações de tipo

opcionais 6. API de ganchos de reação

7. Regra: Nomeação singular/plural 8. Regra: Evite referindo-se a coisas como variáveis,

classes ou métodos no nome 9. Regra:

Nomeie as pessoas por suas funções 10.

Regra: Especificidade contra uniões

11. Regra: Evite parâmetros blob 12. Regra: Evite

parâmetros de série numérica 13. Regra: Utilize namespaces

14. Regra: Use abreviações com moderação 5 . . .

Princípio de nomeação nº 4: Brevidade 1 .

Compressão 2

Contexto 3. A

lei, reiterada 4. Regra: Use

inglês conciso 5. Regra: Abster-se de

letras não convencionais

variáveis 6.

Regra: Agrupamento indica contexto 7. Regra: A

operação e o nome do recurso devem estar em
contexto

8. Regra: Não use prefixos de membros desnecessários 9. Regra:

Refatore prefixos de membros a partir de objetos 6. Princípio de
nomenclatura nº 5: capacidade de pesquisa .

1. Regra: Evite usar constantes numéricas 2. Regra:

Mantenha a documentação atualizada

7. Princípio de nomenclatura nº 6: Pronunciabilidade 1 .

Regra: abreviações muito padrão não precisam ser pronunciáveis 2

Regra: Use

maiúsculas e minúsculas para sinalizar quebras de palavras em variáveis 3

Regra: Booleanos devem fazer uma pergunta ou fazer uma
afirmação

4. Regra: Não omita vogais 8.

Princípio de Nomeação #7: Austeridade 1 .

Nem todo mundo tem o mesmo senso de humor que você 2. Regra:

Não use conceitos temporariamente relevantes 3. Regra: Evite ser

fofo, engraçado, inteligente 4. Regra: Não inclua

referências à cultura popular 9. Resumo 10. Exercícios 11. Recursos 1. Artigos

2. Livros 6. 9 .

Comentários

1. O código explica o quê e como, os comentários explicam o porquê 2. Os .
comentários desorganizam o código

1. [Transformar comentários em código claro, explicativo e declarativo](#)
2. [Comentários ruins](#)
3. [Quando escrever comentários](#)
4. [Demonstração](#)
 1. [Exemplo: Adicionando contexto adicional](#)
 5. [Resumo](#)
 6. [Exercícios](#)
 7. [Recursos](#)
 1. [Livros](#)
10. [Formatação e estilo](#)
 1. [Objetivos do capítulo](#)
 2. [Verdades de legibilidade objetiva](#)
 3. [Espaço em branco](#)
 1. [Use regras de espaçamento óbvias](#)
 2. [Mantenha a densidade de código baixa](#)
 3. [Quebre horizontalmente quando necessário](#)
 4. [Prefira arquivos menores](#)
 4. [Consistência](#)
 1. [Capitalização](#)
 2. [Espaço em branco consistente](#)
 5. [Narrativa](#)
 1. [Código do jornal e o princípio de redução](#)
 2. [Manutenção de um nível consistente de abstração](#)
 3. [O código deve descer em abstração em direção ao nível inferior](#)
 4. [detalhes](#)
 4. [Manter métodos relacionados próximos uns dos outros](#)
 6. [Impondo regras de formatação com ferramentas](#)
 1. [ESLint](#)
 2. [Prettier](#)
 3. [Husky](#)
 8. [Resumo](#)
 9. [Exercícios](#)
 10. [Recursos](#)
 1. [Artigos](#)
 11. [Tipos](#)

2. Entendendo os tipos 1.
Estático x dinâmico 2.
Forte (ou estrito) x fraco 3. Por
que linguagens com tipagem estática?
 1. Capturar erros bobos
 2. Técnicas de abstração 3.
Reforçar a política
 4. Tornar o implícito, explícito 5.
Comunicar a intenção do projeto mais
facilmente 6. Linguagens não tipificadas não escalam muito bem
4. TypeScript: nossa linguagem tipada estaticamente
 1. A anotação “type” 2. Tipos
são opcionais 5. Tipos
em TypeScript 1. Tipos
implícitos convenientes 2. Tipos
explícitos 3. Tipos
estruturais 4. Tipagem
nominal 5. Tipagem
pato 6. Ambiente
tipos 6. Recursos
básicos da linguagem TypeScript 1. Tipos
primitivos 7. .
 - Programação orientada a objetos 1.
Classes 2.
Herança de classe 3.
Propriedades estáticas
 4. Variáveis de instância
 5. Modificadores de
acesso 6. Modificador
somente
leitura 7.
 - Interfaces 8. .
Genéricos 8. Tipos
especiais 1. Asserções de
tipo 2. A palavra-
chave “type” 3.
Aliases de tipo 4. Tipo de união 5. Tipo de interseção

6. [Enum](#)

7. [___. Qualquer](#)

8. [Void](#) 9. [Tipos embutidos](#)

e literais 10. [Tipos](#)

de [proteção](#).

9. [Resumo](#) 10.

[Exercícios](#) 11.

[Recursos](#)

1. [Artigos](#) 9. 12. [Erros e exceções](#)

1. [Objetivos do](#)

capítulo 2. [Erros e loucuras de tratamento](#)

de exceções 1.

[Retorno nulo](#) 2. [Log](#)

e lançamento 3. [Problemas com essas abordagens](#)

3. [Compreendendo erros e exceções](#) 1.

[Erros](#) 2.

[Exceções](#) 4.

[Uma filosofia para tratamento de erros](#)

1. [Recursos e casos de](#)

uso 2. [Um caminho feliz, vários caminhos](#)

tristes 3. [Aregar erros com uniões](#)

5. [Construir uma infraestrutura de tratamento](#)

de erros 1. [O que queremos em nossa API de tratamento de erros?](#)

2. [Apresentando o tipo Qualquer](#)

3. [Modelando o tipo de resposta](#) 4.

[Conectando-o](#) 5.

[Encapsulando a construção da mensagem de erro em classes](#)

6. [Uma filosofia para tratamento de exceções](#)

1. [Lidando com o código de outras](#)

pessoas 2. [Embrulhe o código de E/S em](#)

um try/ bloco catch 3. [Transformar exceções em](#)

[erros significativos](#) 4. [Decidir quando](#)

[lançar exceções](#) 5. [Lançar exceções quando um consumidor deve](#)

[ser forçado a](#)

[corrigir o problema](#) 6. [Lidar](#)

[com o nada](#) 7. [Resumo](#)

8. Exercícios

9. Recursos 1.

Artigos 2.

Livros 5.

Parte III: Phronesis 1. 13.

Recursos (casos de uso) são a chave 1.

Objetivos do

capítulo 2. Abordagens de código-primeiro para

desenvolvimento de software

1. Programação tática 2.

Programação imperativa 3. Dados,

comportamento e namespaces 4. Desvantagens

de uma abordagem API-first 3. Uma filosofia orientada a recurso/caso de uso para design de

software 1. Recurso = caso de

uso 4. A

anatomia de

um recurso 1. Dados

2. Comportamento 3. Espaçamento

de nomes 5. O

ciclo de vida

de um recurso

1. Descoberta 2.

Planejamento 3.

Estimativa 4. Arquitetura 5. Testes

de aceitação 6. Design e implementação 6. Como

resolver nossos objetivos no design de software 7. Construir fontes de feedback (à prova de er

8. Perguntas

1. Isso não é apenas XP?

2. Tenho que fazer tudo aqui?

9. Resumo 10.

Referências 1.

Livros 2.

Artigos 2.

14. Planejamento

1. Objetivos do

capítulo 2. A razão mais comum para o fracasso dos projetos

3. Declaração de direitos
 1. Para clientes 2.
Para desenvolvedores
 3. Permaneça no script 4. Por que planejar?
 1. Priorize - Faça as histórias mais importantes primeiro 2. Coordene - Mantenha todos sincronizados e no saber
 3. Recalibrar — Voltar aos trilhos quando sairmos dos trilhos 5. Como o planejamento funciona 1. 1 —
Escopo de um projeto 2. 2 —
Negociar o primeiro plano de liberação 3. 3 — Executar o primeiro plano 4. 4 — Medir a velocidade
 6. Por que essa abordagem funciona
 7. Resumo 8. .
Exercícios 9.
 - Referências 1.
 - Artigos 2.
 - Livros 3.
 - Documentos
3. 15. Clientes 1.
 - Objetivos do capítulo
 2. Condução versus direção
 3. Quem é o cliente?
 1. A pessoa que toma decisões de negócios 2. Um especialista no domínio
 3. Deve estar disponível para perguntas 4.
Responsável pelo sucesso ou fracasso do projeto 5. Geralmente uma equipe inteira 4. Localizando um cliente 5. E se você puder? t encontrar um cliente?
 6. Resumo 7. .
Exercícios 8.
 - Referências 1.
 - Artigos 2.
 - Artigos

3. Livros 4.

16. Aprendendo o Domínio 1.

Objetivos do capítulo

2. Onde estamos até agora?

3. Domain-Driven Design 1. Um
modelo compartilhado

2. O domínio 3.

Aprendendo o domínio sem um modelo compartilhado 4.

Benefícios de um modelo

compartilhado 5. Padrões, princípios,

práticas 4. Construindo um modelo compartilhado com Event

storming 1. Branco Rótulo: Um domínio de
exemplo 5. 1 — Plete os eventos do

domínio 1. Classifique os eventos 2.
cronologicamente 2. Empurre os
eventos para as bordas 3. Benefícios da
modelagem com eventos 6. 2 —

Identifique comandos 7. 3 — Identifique agregados (opcional)

8. 4 — Decompondo em subdomínios 1.

Subdomínios 2.

Mapas de contexto — documentando as relações entre os
subdomínios 3. Domínios

essenciais, genéricos e de suporte 9. 5 —

Utilizar a linguagem onipresente em todos os lugares 10. E as
consultas?

11. Modelagem de eventos

12. Resumo 1.

Conceitos de Domain-Driven Design 2.

Conclusão 13.

Referências 1.

Artigos 2.

Livros 5.

17. Histórias 1.

Objetivos do capítulo

2. Histórias de usuários

1. Responsabilidades do

desenvolvedor 2. Responsabilidades do cliente

3. INVESTIR: Princípios de escrita de histórias

1. I — As histórias devem ser independentes
2. N — As histórias devem ser negociáveis
3. V — As histórias devem fornecer valor 4.
- E — As histórias devem ser estimáveis 5.
- S — Prefira histórias curtas 6. T
- As histórias devem ser testáveis 4.

História -formatos de gravação

1. Formato de comando/consulta
2. Como [função], quero [recurso], para que [valor]
5. Escrever histórias para requisitos não funcionais 6. Resumo
7. Exercícios .
8. Referências

1. Artigos 2.

Livros 6. 18.

Estimativas

e pontos da história 1. Objetivos do

capítulo 2. Sobre
estimativas 3. Tempo

ideal (esforço) ,

4. Story points 5.

Como fazer estimativas usando os story points recomendados
tabela

6. Refatoração de histórias

1. Fusão — quando as histórias podem ser consolidadas 2.

Divisão — quando uma história é muito grande

3. Divisão — quando uma história contém requisitos funcionais e não
funcionais 4. Spike — quando

há incerteza 7. Técnicas de estimativa de

equipe 1 Dedos voadores 2. Planning

poker 3. Maneiras

de lidar com estimativas

não unâimes 8. Como melhorar as estimativas 9.

Quando atualizar as estimativas 10.

Resumo 11. Referências

— .

—

1. Artigos 2.

Livros 3.

Vídeos 7.

19. Planejamento de lançamento

1. Objetivos do

capítulo 2. O que é planejamento de lançamento?

1. Para o cliente 2. Para

desenvolvedores 3.

Solicitando histórias 4.

Tamanho da iteração

5. Exemplo de plano de
lançamento 6.

Estabilidade do plano 1. Eventos que

afetam o plano 2.

Reconstruções regulares 7.

Lidando com bugs 8. Infraestrutura

de planejamento 1. Caminhando

pelo esqueleto 2. Iteração zero: iteração de funcionalidade

zero 3. Iteração de festa da pizza (ou programação mob)

9. Resumo 10.

Exercícios 11.

Referências 1.

Livros 8.

20. Planejamento da iteração 1.

Objetivos do capítulo

2. Reunião de planejamento da iteração

3. Compreensão de uma história

4. Listar as tarefas 1.

Tarefas técnicas 5.

Inscrir-se para tarefas 6.

Mantendo acompanhamento do plano de iteração 7.

Exemplo de plano de iteração 8.

Resumo 9.

Exercícios 10.

Recursos 1. Artigos

2. Livros

9. 21. Compreendendo uma história 1.

Objetivos do capítulo

2. Entrevistando um especialista no domínio

1. Regras para uma boa entrevista 3.

Documentação com pseudocódigo 1. Resumo

de recursos 2. Primitivas de

domínio 3. Fluxo de trabalho

de recursos

4. Passo 1 — Obtenha uma compreensão de alto nível de todo o

fluxo de

trabalho 1. Dados de entrada e

saída 2. Estados de sucesso, estados de falha e dependências 3.

Investigar casos extremos com cenários hipotéticos 5. Etapa 2 —

Documentar as primitivas de domínio (tipos de dados)

1. Restrições 2.

Variantes e máquinas de estado

6. Etapa 3 — Documente as etapas 1.

Algoritmo de caso de uso: o nível mais alto de abstração 2. Subetapas 7.

Resumo 8.

Exercícios 9.

Referências 1.

Artigos 2. Livros

10. 22. Testes

de aceitação

1. Objetivos do capítulo 2 O que é
teste de aceitação?

1. Uma forma de contratar e exercitar os cenários de sucesso e fracasso

3. Compreender (e apreciar) os testes de aceitação

1. O caminho para os testes de aceitação (final dos anos 90 até agora)

2. Uma divisão de responsabilidades 3.

Ferramentas de teste de aceitação inicial

4. Falha em ganhar onipresença 5.

A necessidade de melhores práticas de TDD 6.

BDD (desenvolvimento orientado a comportamento)

7. Dado-quando-então (exemplo)

4. Redação de testes de aceitação

1. Ferramentas
2. Formato nº 1: Testes de linha única
3. Formato nº 2: Testes de Jest no estilo "Dado-Quando-Então"
4. Formato nº 3: Testes de recursos "Dado-Quando-Então" (preferencial)

5. Perguntas frequentes 1. Quem faz o

- quê? _____
2. Cadência _____

6. Resumo 7 .

Exercícios 8.

Referências 1.

Artigos 11.

23. Paradigmas de Programação _____

1. Objetivos do _____

capítulo 2. Programação estruturada

1. Provas matemáticas 2. _____

Parando com os GOTOS 3. _____

Decomposição funcional 4. O _____

nascimento do teste como o conhecemos hoje 5. Qual é.

o modelo conceitual correto? _____

6. Problema nº 1: Estado mutável compartilhado (acoplamento)

7. Problema nº 2: Complexidade ciclomática 8. _____

Problema nº 3: Polimorfismo inseguro 3. Orientado
a objetos 1. A ideia _____

principal: uma teia de objetos 2. Sequência _____

seleção, iteração e indireção 3. Como OO resolve os problemas
estruturados programação enfrentada? _____

4. Herança - como errar fazendo OO 5. Como fazer OO

corretamente _____

4. Funcional _____

1. Um modelo conceitual completamente diferente 2. _____

Noções básicas de _____

função 3. Não é a mesma sequência, seleção, iteração e indireção

4. Composição _____

5. Usando a _____

composição para construir sistemas do mundo real _____

- 6. Por que programação funcional?
- 5. Os princípios de design são independentes de paradigma
- 6. Escolhendo um paradigma de programação 7.
- Resumo 8.
- Exercícios 9.
- Recursos 1.
 - Artigos 2.
 - Livros 3.
 - Vídeos 12.
- 24. Uma arquitetura orientada a objetos 1. Objetivos do capítulo 2. Decidindo sobre uma arquitetura 1. 1. Use os requisitos 2. 2. Consulte os princípios de design 3. 3. Consulte os princípios de arquitetura 4. 4. Consulte os padrões de arquitetura 5. 5. Desenhe em um quadro branco
- 3. Aprimorando suas habilidades em arquitetura
- 4. Uma arquitetura orientada a objetos para aplicativos de negócios 1. MVC: a trivial arquitetura em camadas 2. Qual é o problema com o MVC?
- 3. Você não pode testar a unidade de infraestrutura 4. Código principal versus código de infraestrutura 5. Como as solicitações funcionam em uma arquitetura OO: scripts de transação versus modelos de domínio
- 6. Inversão de dependência e a regra de dependência 7. Por que desacoplar o núcleo do código de infraestrutura novamente?
- 8. Implantação como um monólito modular 5.
- Resumo 6.
- Referências 1.
 - Artigos 13.25.
- Estratégias de teste 1. Objetivos do capítulo 2. Tipos de teste 1.
 - Unidade
 - 2. Integração 3.
 - Ponta a ponta

4. Aceitação 3.

Preocupações: O que precisamos testar?

1. Recursos (consultas)

2. Recursos (comandos)

3. Código central

4. Infraestrutura 5.

Banco de dados (infra)

6. API GraphQL ou RESTful (infra)

7. Componentes de infraestrutura compartilhados (infra)

8. APIs externas e integrações (infra)

4. Estratégia de teste para um aplicativo desacoplado 1.

Teste de unidade em todo o código da

camada de domínio 2. Testes de caso de uso: teste de aceitação dos recursos
da camada do

aplicativo como testes de unidade 3.

Adaptadores de entrada do teste de integração

4. Adaptadores de saída do teste de integração 5. Teste de ponta a ponta

para consultas

e confiança

adicional

5. Resumo 6.

Referências 1. Livros 2. Artigos 14. 26.

O esqueleto

ambulante 1. Objetivos do capítulo 2. O que é um esqueleto ambulante?

1. Por que ponta a ponta?

2. Expor problemas de implantação e conexão com antecedência 3.

O que você quer dizer com ambiente de produção?

4. Por que compilações automatizadas?

5. O que fazemos com os componentes não cobertos pelo teste de ponta
a ponta?

3. Demonstração de alto nível (exemplo)

1. Pequenos

passos 2. Escolha o(s) cenário(s) mais simples

3. Um teste de ponta a ponta com

falha 4. Use objetos de página para escrever testes E2E

declarativos 5. Buscando pontos inexistentes

6. Back-end: o mínimo necessário

4. Resumo 5.

Referências 1.

Artigos 2.

Livros 3.

Vídeos 15.

27. Pair Programming 1. Objetivos

do capítulo 2. O que
é?

3. Papéis

4. Técnicas de troca 1. Usando
um relógio 2. Ping
pong/pipoca 3. . . .

Emparelhamento de estilo

forte 5. Qual é o resultado de uma sessão de programação em par?

6. Com que frequência você deve programar em par?

7. Considerações 1.

Como é um bom par 8. Resumo 9.

Recursos 1.

Artigos 2. Livros

16. 28. Fluxo
de Trabalho

do Desenvolvimento Orientado a Testes 1. Objetivos do

capítulo 2. Regras e

considerações do Desenvolvimento Orientado a Testes

1. Red-Green-Refactor 2. Não

se precipite 3. Tenha seus testes

rodando ao seu lado enquanto você codifica 4. Confirmando depois

de passar no teste 3. TDD Pré-requisitos 1.

Estratégia de teste razoável

pensada 2. Crie uma compilação -test-deploy arquitetura

de teste 3. Scripts separados para aceitação/unidade,

integração e

Testes E2E

4. Pode alternar perfeitamente entre ambientes 4. Por que o TDD

do mundo real é tão difícil 1. Diferentes

desafios de TDD no desenvolvimento front-end vs. back-end

2. Como uma indústria, lutamos para concordar com a terminologia .
- de teste 3. Muitos falham em reconhecer que o TDD começa com a arquitetura 4. Entender o que testar (e como testá-lo) é difícil .
5. Double Loop TDD e as duas escolas de pensamento TDD
 1. Clássico (Inside-Out/Chicago) e Mockist (Outside In/London)
 - TDD 6. Preenchendo lacunas: Adicionando E2E e Testes de Integração 1. Testes de E2E
7. Refatoração é onde o design acontece 8. Resumo 9. .
Conclusão de Phronesis
10. Exercícios
11. Referências
 1. Artigos 6.

Parte IV: Fundamentos do Desenvolvimento Orientado

a Testes 7. Parte V: Design Orientado a Objetos (com Testes)

1. 5. Programação Orientada a Objetos e Modelagem de Domínio 2. Calistenia de Objetos 3. Cheiros de Código
 1. Odores de código dependem do idioma, contexto e desenvolvedor
4. Antipadrões
 1. Antipadrões não são determinados 1. Modelos de domínio anêmicos vs. Sistema de componentes de entidade
 2. DRY vs Overengineering 2. Overengineering 3. Complexidade ciclônica 5. .
- Refatoração 8.

Parte VI: Princípios de design 1. 6.

- Princípios de design 1. SOLID 1
- Princípio da Responsabilidade Única 2. Princípio Aberto -Fechado (OCP)
 3. Princípio da Substituição de Liskov (LSP)
 4. Princípio de Segregação de Interface
 5. Princípio de Inversão de Dependência (DIP)

1. Terminologia 2. .
Inversão de Dependência 3.
Usando um objeto fictício 4.
As principais vitórias da Inversão de Dependência 5. Inversão de Contêineres de Controle e IoC

2. Os Princípios do Menor 1.
Princípio do Menor Esforço 2.
Princípio do Menor Surpresa (Surpresa)

 3. Lei de Deméter (Princípio do Menor Conhecimento)

 3. Projeto por contrato (DBC)

 4. Separação de preocupações 1.
Postagens de blog .
relacionadas 5. CQS (separação de consulta de comando)

 6. YAGNI 7.
Design Simples 8.
KISS (Keep It Simple, Silly) .

 9. DRY, WET, Regra de Três 10. Os Quatro Princípios Primários de Design Orientado a Objetos 11. Composição sobre herança 1. Busque hierarquias de classes rasas 12. Encapsule o que varia 13.

 - Programe para interfaces, não para implementações
1. Relação com a arquitetura de portas e adaptadores 2. Relação com o Princípio de Inversão de Dependência 14. O Princípio de Hollywood 15. Todo software é composição 16. Padrões de projeto são complexos 1. Conheça-os, mas saiba quando precisar deles 17. Separação de responsabilidades 1 Exemplo : controlador sobrecarregado 1.

 - Separação de preocupações 18. Esforce-se para um acoplamento fraco entre objetos que interagem 19. Princípio da menor resistência 20. Diga, não pergunte 21. Sem e, ou ou mas 2. Projeto simples 1. Capítulo metas

2. Os quatro elementos do design simples
 1. Executa todos os
testes 2. Sem
duplicação 3. Maximiza
a clareza 4. Minimiza o número de elementos

3. Outras definições de projeto simples

1. Intenção
2. Aceitação orientada para
testes 3. Acoplamento e
coesão 4.
Mínimo 4.
- Conclusão 5.

Recursos 9. Parte VII: Padrões

- de projeto 1. 7. Padrões
de projeto 1. Padrão
de fábrica 10. Parte VIII : Arquitetura

- Essencial 1. 8. Princípios de
arquitetura 1. Acoplamento
e coesão 2.
Contratos 3. Princípios de
componentes 1. Princípio de equivalência de
reutilização-liberação 2. Princípio de fechamento comum (CCP)
3. O Princípio de Reutilização Comum (CRP) .
4. Componentes estáveis
5. Componentes voláteis

4. Lei de Conway 5.
A regra da dependência 6.
Fronteiras 7.

- Preocupação transversal 8.
Os princípios da economia 1. O
princípio do custo de oportunidade 2. O
princípio do último momento responsável 9.

Separação Preocupações 2.

9. Estilos de arquitetura 1.
Estrutural 1.
Arquiteturas baseadas em componentes
2. Arquiteturas em camadas

3. Arquiteturas monolíticas 2.

Baseadas em

mensagens 1. Arquiteturas

orientadas a eventos 2. Arquiteturas

Publish-Subscribe 3. Distribuídas

1. Arquiteturas cliente-servidor 2.

Arquiteturas peer-to-peer

3. 10. Padrões de arquitetura 1.

Arquitetura em camadas (n camadas)

1. Camadas

1. Camada de _____

domínio 2. Camada de _____

aplicativo 3. Camada de _____

infraestrutura 4. _____

Camada de adaptador 2.

Arquiteturas semelhantes

1. Portas e adaptadores 2.

Arquitetura de fatia

vertical 2 Fonte de

eventos 3.

Microkernels 4.

Microsserviços 5. Baseados no espaço 11. Parte IX: Construindo Aplicações Web com Design Orientado a

Arquitetura Hexagonal e CQRS

1. 11. Construindo um aplicativo DDD do mundo

real 1. Sobre este capítulo

2. Objetivos do

capítulo 3. Design orientado a

domínio 1. Linguagem ubíqua

2. Implementando DDD e garantindo a pureza do modelo de domínio

3. DDD aborda as deficiências do MVC 1. Slim Modelos

(sem lógica) 2. Escolha seu

veneno de modelagem de objeto 3. _____

Preocupações com a camada não especificada no

MVC 4. Efeitos colaterais indesejáveis com a falta de um domínio

modelo

5. Comportamento e forma do

modelo 4. Benefícios técnicos

5. Desvantagens técnicas 6.

Alternativas ao DDD 7.

Blocos de construção do

DDD 1.

Entidades 2.

Objetos de valor

3. Agregados 4.

Serviços de

domínio 5.

Repositórios 6.

Fábricas 7. Eventos de

domínio 8.

Conceitos de arquitetura

1. Subdomínios 2. Contextos delimitados 4 O Projeto : DDDForum — um fórum inspirado no app

1. Sobre o projeto 2. O código 5.

Como planejar um novo projeto 1.

Design imperativo 1.

Abordagens de design imperativo são para aplicativos CRUD pequenos e simples 2.

Dimensões que influenciam a abordagem de design que devemos

adotar 3. Caso de uso design orientado 1. Casos de

uso e atores 2. Aplicativos são agrupamentos de casos de uso 3. Um caso de uso é um comando ou uma

consulta 4. Artefatos de caso de uso 5. Requisitos funcionais documentam a lógica de negócios 6.

Paralelos com o design API-first 7. Etapas para implementar o design de caso de uso 4.

Planejando com diagramas de

caso de uso UML 1. 1 — Identificando

os atores 2. 2 — Identificando os objetivos do ator 3. 3 —

Identificando os sistemas que precisamos criar 4. 4 — Identificando os casos de

5. Papéis, limites e Lei de Conway no caso de uso
Design
6. Resumo dos diagramas de casos de uso
5. Event Storming 1.
Por que precisamos de event storming 2.
Como conduzir uma sessão de event storming 6. Modelagem
de eventos 6. Construindo
o DDDForum 1. Arquitetura do
projeto 1. Decisão 1: Vamos
usar o Domain-Driven
Padrões de projeto
2. Decisão 2: vamos usar uma arquitetura em camadas
3. Decisão 3:
vamos implantar um monólito modular 4. Decisão 4: vamos
usar CQRS

(Segregação de resposta de consulta de comando)
5. Decisão 5: Não vamos usar Event Sourcing 2.
Começando
com os modelos de domínio 1. Modelando um
Agregado de Usuário 2. Emitindo
Eventos de Domínio a partir de um Agregado de Usuário 3. Escrevendo Eventos de Domínio
4. Construindo Eventos de Domínio Assunto 1.
Marcando um Agregado que acabou de criar um Domínio
Eventos
2. Como sinalizar que a transação foi concluída 3. Como
registrar um manipulador em um evento de domínio?
4. Quem determina quando uma transação é concluída?
5. Recurso 1: Criando um membro 1.
Emitindo uma solicitação de API
2. Serviços de aplicativos/casos de uso 3.
Dentro da transação do caso de uso CreateUser 6. Recurso
2: Vote a favor de uma postagem 1.
Compreendendo a lógica do domínio de votação
2. Manipulando a solicitação de postagem de
votação positiva 3. Dentro do caso de uso Upvote Post

4. Princípios de design agregados 5. _____
Usando um serviço de domínio 6.
Persistindo na operação de postagem de upvote
7. Recurso 3: Obter postagens populares
 1. Modelos de leitura
 2. Manipulação de uma solicitação de API para obter postagens populares 3. Usando um repositório para buscar os modelos de leitura 4. Implementando paginacão
7. Para onde ir a partir daqui?
8. Recursos 9.
Referências
2. Mapeamento de Contexto de um Monólito Modular
 1. Padrão de Kernel Compartilhado para infraestrutura compartilhada 2. Padrão de Parceria para comunicação de subdomínio
 3. Padrão de Serviço Open-Host 4. Camada Anticorrupção para comunicação com modelos
3. Padrões 1.
 1. Caso de uso 2. Repositório 3. Modelo de domínio 4. Script de transação 5. Evento de domínio 6. Caixa de saída transacional 7. Unidade de trabalho
 8. Agregados 9. Comandos 10. Consultas 11. Objetos de valor 12. Entidades 13. Manipuladores de eventos 14. Relógio (tempo)
12. Parte X: Desenvolvimento Avançado Orientado a Testes
 1. Duplas de teste: Mocks & Stubs 2. Teste contra comportamento, não implementação 3. Teste de caso de uso 4. Princípios de teste de ponta a ponta

1. [Notas](#)
2. [Recursos](#) 1.
[Artigos](#) 5.

[Construindo uma plataforma de teste de ponta a ponta](#) 6. [Testes de ponta a ponta estáveis com o padrão de objeto de página](#) 7. [Testando dependências gerenciadas versus não gerenciadas](#) 8. [Adaptadores de entrada de teste de integração: GraphQL , REST,](#)
CLI 9. [Princípios de teste de unidade](#) 10.
[Princípios de teste de aceitação](#) 13.
Parte XI:
[Acima e além](#) 1. [DevOps](#) 14. [Conclusão](#)

Sobre este livro on-line

Obrigado por ler este livro! Atualmente está sendo trabalhado por mim e transformado em algo incrível (com o feedback dos compradores e revisores da pré-venda do livro).

Este livro é atualizado continuamente

Acredito que foi Julian Shapiro quem disse que os livros são um “meio ruim para educação e discussão”. Eles ficam desatualizados rapidamente, contêm preenchimento para atingir a contagem de palavras e é difícil vincular conteúdo suplementar que pode ser valioso para o aprendizado.

Essa é uma das principais razões pelas quais eu gostaria que a fonte primária deste livro fosse este artefato online, vivo e que respira.

Se você comprou este livro, recebeu por e-mail a versão atual com acesso vitalício e a capacidade de baixá-lo e lê-lo em vários formatos: aqui (web), PDF e EPUB.

Baixando uma cópia

Para baixar a versão mais recente do livro, acesse a página de downloads em wiki.solidbook.io/downloads e escolha entre PDF ou EPUB.

Linha do tempo

Cronograma de lançamento — Você pode acompanhar o cronograma do livro na [página de atualizações](https://wiki.solidbook.io/actualizacoes). Pretendo terminar de escrever solidbook.io até agosto de 2021.

Atualizações

Recebendo atualizações — Enviarei um e-mail para você sempre que houver uma atualização. Você também pode encontrar atualizações na **página de atualizações** do site. Você também pode me seguir no Twitter para atualizações.

Acessando o livro

Lendo este livro online — Para ler este livro online, use o *link mágico* que foi enviado para o seu e-mail. Precisa reenviar o link para si mesmo? Sem problemas, basta acessar wiki.solidbook.io e insira o e-mail com o qual você comprou o livro. Você deve receber um e-mail contendo seu link em alguns momentos.

Opinião

Enviando comentários — Existem várias maneiras de enviar comentários. Use o botão, envie-me um e-mail ou entre em contato comigo no Twitter.

A melhor maneira de me enviar feedback é escrever um e-mail para khalil@khalilstemmler.com ou DM no Twitter.

Introdução

Minha história

Tudo começou quando meu entrevistador me perguntou: “Como você projetaria sua camada de lógica de negócios?”

Estou sentado lá no meio de uma sala mal iluminada, na frente de três desenvolvedores de software seniores (nenhum deles parecendo muito agradável, devo acrescentar), esperando que nenhum deles tenha um vislumbre da gota de suor Eu apenas senti rolar para baixo do lado da minha cabeça.

Camada de lógica de negócios... Isso é praticamente outra linguagem para mim. Eu não tinha a menor ideia de como responder.

Após alguns segundos de deliberação, comecei a dizer palavras. Eu provavelmente disse algo sobre MVC (model-view-controller), como organizo minhas pastas e talvez algo sobre como estruturei meus controladores em um projeto Node.js e Express.js.

Cada palavra que saía da minha boca era um pouco mais de sujeira jogada no buraco que eu estava cavando para mim. Conforme eu falava, eu percebia cada vez mais que nada do que eu dizia tinha algo a ver com “lógica de negócios”.

Por fim, tive a sensação de parar de falar.

Fui então recebido com alguns segundos de silêncio *muito* desconfortáveis. Depois de um aceno de cabeça de um dos companheiros e alguns olhares um para o outro, um entrevistador disse “OK, acho que é isso. Obrigado, você tem alguma pergunta para nós?”

Aaaaaaab-claro que não, vou me ver fora — disse meu cérebro. Tentando apagar aquele momento da minha memória o mais rápido possível, provavelmente perguntei “Então, como é a cultura aqui?”

É seguro dizer que não consegui o emprego. Meu recrutador até me dispensou. Pensando no que fiz para me preparar para a entrevista, lembro-me de ter estudado:

- O que quer que eu tenha encontrado em “como se preparar para uma entrevista de desenvolvedor da web full-stack” Pesquisa
- no Google Os algoritmos em *Cracking the Coding Interview*
- Javascript e todas as suas peculiaridades de linguagem bobas como fechamentos, IIFEs e passagem por referência versus valor.

Não apenas isso, mas eu estava trabalhando na [construção de minha própria empresa iniciante](#) . durante o tempo em que estava na escola e pensei que todo aquele código que escrevi teria se traduzido em alguma experiência séria.

Posso ter estragado completamente aquela entrevista, mas olhando para trás hoje - foi uma das melhores coisas que poderia ter acontecido comigo. Foi nesse momento que percebi que havia todo um mundo de design e arquitetura de software que **eu precisava** aprender *sozinho*. Fiquei incrivelmente interessado neste tópico. eu ia aprender. E eu não iria falhar na próxima entrevista.

Minha pesquisa inicial

Meados de 2018, no meio das minhas pesquisas, eu precisava muito pagar as contas, então acabei conseguindo um emprego como Consultor Frontend. Eu estava procurando principalmente por um trabalho de baixo estresse que pudesse realizar para hackear minha carreira comprando e estudando o máximo possível de livros sobre design de software, padrões e arquitetura e aplicando tudo o que aprendi nesses livros para melhorar meu peso ~ 300k-line Código de inicialização do Node.js.

Todas as noites depois do trabalho por 8 meses, eu lia livros e escrevia códigos. Revisei tudo o que me ensinaram na escola, mas desta vez, aprendendo com os livros dos especialistas.

Comecei com Programação Orientada a Objetos. Quais foram os 4 princípios de OOP novamente? O que *uma classe abstrata* faz afinal? Por que você iria querer usar um método estático? Ouvi dizer que herança era uma coisa ruim, certo?
Por que devo evitar isso?

Enquanto aprendia, anotei todos os conceitos dos quais **ouvi falar, mas nunca me preocupei em tentar entender como** POJOs, injeção de dependência, inversão de dependência, inversão de controle, classes concretas, padrões de design e princípios, etc.

Revisitei ideias que conhecia, *mas* nunca entendi totalmente em profundidade, como acoplamento, coesão, gerenciamento de dependências e separação de preocupações. Logo, eu estava aprendendo muitas coisas novas, como a arquitetura hexagonal, a lei de Conway, o desenvolvimento orientado a casos de uso, TDD e os princípios SOLID.

O ponto em que tudo realmente valeu a pena foi quando descobri o Domain-Driven Design. Minha abordagem de aprendizado era *aprender fazendo*, além de ensinar os outros. Saí do meu emprego, refatorei a base de código da Univjobs usando práticas de Domain-Driven Design e comecei a compartilhar regularmente o que estava aprendendo sobre design e arquitetura de software com meus colegas on-line em meu blog pessoal.

Hoje, milhares de leitores todos os meses estão aprendendo a escrever um código **testável, flexível e sustentável** em khalilstemmler.com.

Por que escrevi este livro

Dois anos depois, após cerca de setenta postagens em blogs, dez mil assinantes de boletins informativos e a dissolução da Univjobs, comecei a coletar e-mails para um curso de Domain-Driven Design. Como o tópico da série mais popular do blog, meu plano era ensinar aos desenvolvedores como usar essa técnica para criar software de alta qualidade.

Quando comecei a aprender mais sobre quem estava interessado neste curso, percebi que a lacuna de habilidades em design de software era *profunda*.

Habilidades práticas de design não estão sendo ensinadas

A verdade é que não mudou muito sobre os fundamentos do design de software nos últimos 60 anos, mas há **uma enorme falta de treinamento nisso**.

Em geral, os desenvolvedores juniores e intermediários ainda ficam confusos e com falta de orientação sobre como desenvolver software de alta qualidade com confiança e consistência.

Desde coisas simples, como estruturar seu projeto ou nomear bem as coisas para que as coisas possam ser compreendidas, encontradas e alteradas, até como decidir sobre um estilo arquitetônico e organizar sua *lógica de negócios*, estamos deixando os desenvolvedores de lado.

Ninguém está ensinando aos desenvolvedores fundamentos essenciais de design e arquitetura de software

Tendo a concordar com Eric [Elliot](#), que diz que “99% dos desenvolvedores profissionais carecem de treinamento sólido em design de software e fundamentos de arquitetura. 3/4 dos desenvolvedores são autodidatas e 1/4 dos desenvolvedores são mal treinados por um currículo de CS disfuncional.

E quase nenhuma empresa compensa essas deficiências com treinamento e orientação internos. Em outras palavras, se você simplesmente aceitar o status quo e se recusar a oferecer treinamento interno, sua equipe será um cego guiando outro cego”.

Design ruim é extremamente caro

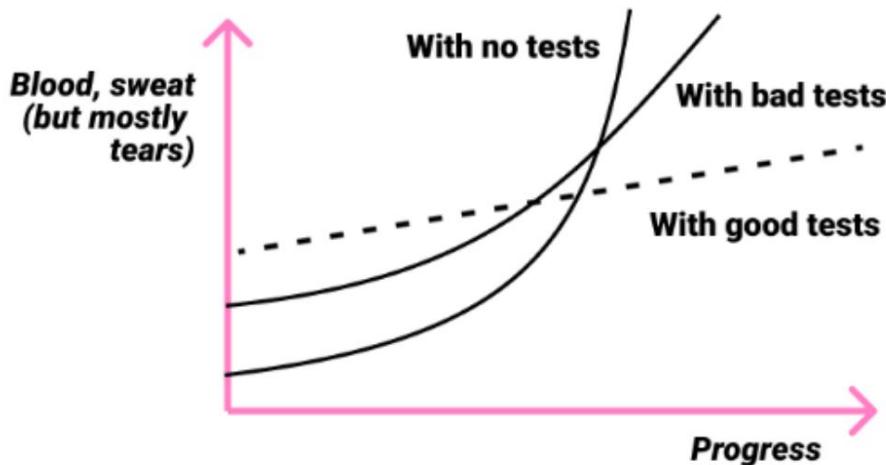
O mau design tem algumas consequências seriamente dispendiosas. De acordo com a CNBC, em setembro de 2018:

Aproximadamente US\$ 85 bilhões são gastos anualmente lidando com códigos ruins.

Isso me surpreendeu. eu escrevi um [artigo](#) sobre esse fenômeno notável e por que acho que isso está acontecendo, mas o resumo executivo de onde acho que isso vem é:

- Os desenvolvedores não estão **aprendendo habilidades essenciais de design de software**
- A maioria das empresas *diz que pratica Agile*

- Na verdade, praticar Agile significa ciclos de feedback estreitos e a capacidade de alterar e refatorar o código
- Para refatorar o código, precisamos de testes
- Para escrever testes, precisamos saber como escrever código testável
- **A maioria dos desenvolvedores não pode escrever código testável**
- Portanto, a produtividade cai com o tempo



Como projetos com bons testes, testes ruins e nenhum teste progredem ao longo do tempo.
Sem testes (ou com testes ruins), é difícil continuar progredindo com segurança porque novas mudanças introduzem regressões. A refatoração sem testes é uma ótima maneira de introduzir regressões.

Há muito a aprender com o passado

A cada poucos anos, uma tecnologia popular é lançada sobre ideias anteriores do passado. Redux é o padrão observador. [GraphQL é a inversão de dependência arquitetônica.](#)

Qualquer desenvolvedor de JavaScript da década de 2010 sabe como pode ser frustrante precisar se manter atualizado com as bibliotecas mais recentes e a melhor maneira atual de lidar com o estado, etc — apenas para que novas práticas e ferramentas surjam alguns meses depois.

Como indústria, rapidamente esquecemos nossas raízes e reinventamos a roda.

Eventualmente, chegou a um ponto em que decidi que não iria mais perseguir as ferramentas e tecnologias mais recentes. Decidi que aprenderia as coisas pelas quais as novas ferramentas e tecnologias são influenciadas e construídas.

Afinal, é a maldição de Santayana:

“e quem não conhece a história também está condenado a repetir seus erros”.

Como escrevi este livro

Dois tipos de sabedoria: Sophia e Phronesis

Aristóteles foi um dos professores mais influentes da história. Ele escreveu sobre física, lógica, metafísica, ética, música, política, linguística e muito mais.

Aristóteles exerceu uma influência única em quase todas as formas de conhecimento no Ocidente.

Em seu livro, *The Nichomachean Ethics*, ele distingue entre dois tipos diferentes de sabedoria: *sophia*, que significa “sabedoria factual” **e *phronesis*, que significa ***“sabedoria prática”.

Sofia

Já ouviu falar de um tópico chamado *filosofia*? A palavra grega originária, *philo**-*sophia**, significa “amor à sabedoria”.

Sophia é o tipo de conhecimento relacionado com o conhecimento estruturalmente ensinável, componível, lógico e factual que muitas vezes equiparamos à ciência.

Por exemplo, um livro intitulado “1000 fatos sobre tubarões” provavelmente será um livro com o lado *sophia*.

Os livros de programação (e documentação) do lado *Sophia* geralmente são bons para aprender novas linguagens, plataformas e estruturas. Eles ensinam novos fatos e **mostram o que você pode fazer**.

Phronesis

Phronesis, que é a sabedoria prática, é a sabedoria adquirida através da experiência.

Esse tipo de sabedoria está mais preocupado com a ação prática e com a melhor forma de agir em cenários que exigem isso.

Tradicionalmente, esse tipo de conhecimento é mais difícil de ensinar. Normalmente, é ensinado por meio de mentorias e aprendizado prático com os *próprios especialistas*, mas, muitas vezes, os especialistas escrevem livros e criam recursos que os alunos podem pegar e conduzir a si mesmos mais rapidamente no caminho da aquisição de sabedoria prática.

Os livros de programação *Phronesis* são os livros clássicos influentes e atemporais que resistem às tendências de software. Estes são os livros que **mostram o que você deve fazer**.

A abordagem deste livro à sabedoria

Uma versão inicial deste livro estava mais de acordo com a sabedoria do lado sophia e seguia uma abordagem *de primeiros princípios* para dominar o software

projeto. A abordagem baseou-se na ideia de que, se pudéssemos entender como funcionam os tópicos de baixo nível do design, poderíamos compor esses fatos em partes maiores que fizessem sentido. No [capítulo 3](#), examinamos isso com mais detalhes, observando como os padrões de design no nível de classe também podem ser aplicados no nível de arquitetura mais amplo.

Embora essa seja uma maneira simples de aprender *fatos* sobre design de software, ela não nos ajudará a dominar o ato prático de escrever um código de qualidade.

Em vez disso, **estamos usando um método desenvolvido por Aristóteles há mais de 2.000 anos para ser bem-sucedido em qualquer coisa (e em tudo) na vida.**

Veja como funciona.

Passo 1: Identificar o **propósito** da coisa que queremos usar. Conhecer o propósito de algo nos permite avaliar as maneiras pelas quais podemos usá-lo.

- por exemplo, facas são para cortar

Passo 2: Identifique as **qualidades** que o fazem cumprir bem o seu propósito.

- por exemplo, se as facas forem para cortar, uma faca afiada é uma **boa** faca, porque corta **bem**.

Etapa 3: imite **outras pessoas** que cumprem bem o objetivo

- por exemplo, **encontramos pessoas que são boas em cortar e as observamos.**

Passo 4: Desenvolver nossos próprios princípios através da **prática**

Nós descobrimos os princípios que guiam seu sucesso.

- por exemplo, **aprendemos o que os torna bons no corte.**

Este pode ser o ponto intermediário entre dois extremos.

- por exemplo, para cortar algo, a faca **não deve ser muito** longa **nem muito** curta. (*ponto médio*)

Também pode ser um mínimo ou um máximo.

- por exemplo, para cortar algo, a faca **deve ser** suficientemente afiada . (*mínimo*) por exemplo,
- para cortar algo, a faca **não deve ser muito** **pesada. (*máximo*)

Depois de bastante experiência, construímos hábitos e dominamos o que estamos praticando.

- por exemplo, **praticamos** o corte **até que** cortar **bem** seja **fácil e prazeroso**.

Essa é a abordagem.

Essencialmente, fazemos o que os profissionais estão fazendo e acumulamos nosso próprio conjunto de princípios por meio da experiência. Isso nos dá a capacidade de determinar os princípios corretos e sua quantidade correta para qualquer situação.

Como dominar o design de software

Vamos aplicar o método de Aristóteles ao ato **de escrever código** para aprender a dominar o design de software.

Etapa 1: identificar o objetivo do código

Qual é o propósito do código? Eu ouço todos os tipos de argumentos para o código ser uma arte, ciência, matemática e assim por diante.

Certamente depende de quem perguntamos, não é? As pessoas escrevem código por motivos diferentes. Os designers de interação fazem arte generativa, os cientistas de dados praticam a codificação exploratória e tenho certeza de que os matemáticos também têm seus próprios casos de uso. Até **os músicos codificam!**

Vamos considerar o propósito do código como uma arte por um momento.

- por exemplo, o código é para criar arte

Se for esse o caso, quais qualidades seriam necessárias para que o código seja considerado *uma boa arte*? Talvez o código precise ser inteligente, interessante, envolvente, único, inspirador, polarizador ou evocar uma emoção (tristeza, nostalgia, amor, angústia). Todas essas são qualidades potenciais para uma boa arte (bem, pelo menos as coisas estranhas que eu gosto).

É claro que poderíamos seguir esse caminho (e isso definitivamente seria divertido), mas não está de acordo com o que a maioria de nós está usando código.

Para a maioria de nós que trabalhamos como comerciantes qualificados para contratar, o **objetivo do código é criar produtos para os clientes**.

- por exemplo, o código é para criar produtos para clientes

Passo 2: Identifique as qualidades do código que o fazem cumprir bem seu propósito

Se estivermos usando código para criar produtos para clientes, quais são algumas das qualidades de um *bom* código escrito para atender a esse propósito?

O código tem dois consumidores: **os clientes e os desenvolvedores que o mantêm**.

Portanto, algumas das qualidades essenciais para cumprir bem o seu propósito são:

- **Atende às necessidades do cliente:** O produto tem requisitos — funcionais e não funcionais. Estamos nos encontrando com eles?
- **Flexibilidade:** Queremos ser capazes de adicionar novos recursos apenas adicionando um novo código e alterando o mínimo possível de código existente.
- **Testabilidade:** quando os requisitos mudam e temos que adicionar novos recursos, podemos continuar fazendo isso sem interromper os recursos existentes?
Para nos mantermos produtivos, precisamos de testes. Os testes têm muitos benefícios, mas os mais importantes para listar aqui são que eles provam a correção do negócio, evitam regressões e nos dão confiança para refatorar com segurança.

- **Manutenibilidade:** Na verdade, há muito mais qualidades de código que poderíamos listar, como consistência, número de elementos, clareza, simplicidade — mas, em última análise, é nossa capacidade de entender o código que mais afeta sua manutenção. Por que isso é tão importante? Bem, imagine que você é um cliente com um orçamento. Agora imagine que você está pagando por tempo e materiais (ou seja, você está pagando pelos gastos dos desenvolvedores trabalhando no projeto). Quanto tempo leva para os desenvolvedores produzirem uma nova funcionalidade que agregue valor ao projeto? Uma semana? Duas semanas? Um mês? Dois meses? Isso é muito tempo. E tempo é dinheiro. O código precisa ser projetado de forma que possa ser alterado de maneira econômica.

Portanto,

O objetivo do código: Código é para criar produtos para clientes. O código testável, flexível e sustentável que atende às necessidades dos usuários é **bom** porque pode ser alterado de maneira econômica **pelos desenvolvedores**.

Passo 3: Domine as técnicas de quem está fazendo bem

Se *phronesis* é “sabedoria prática”, então os *phronimos* são os sábios desenvolvedores; aqueles com experiência prática construindo software com as qualidades que queremos.

Entre minha *lista pessoal de phronimos* estão os seguintes desenvolvedores:

Martin Fowler, Robert C. Martin, Kent Beck, John Ousterhout, Scott Wlaschin, Vaughn Vernon, Steve Freeman, Matthias Noback, Greg Young, Eric Evans, Dan North, Gregor Hohpe, Udi Dahan, Gerard Meszaros, Kevin Henney, Michael Feathers, Pedro Moreira Santos, Vladimir Khorikov, Marco Consolaro, Alessandro Di Gioia, Thomas Pierrain, Adam Dymitruk e Kent C. Dodds.

O próximo passo é ler seus livros e postagens de blog, assistir suas palestras, fazer seus cursos, treinamentos práticos e conversar com eles no Twitter para aprender suas técnicas e filosofias de design de software.

Passo 4: Desenvolver nossos próprios princípios praticando e construindo experiência

Por último, *pratique*. Siga as técnicas que os profissionais usam e domine-as. Tire suas próprias conclusões e desenvolva princípios sobre quando usá-los com base em sua própria experiência.

Decida quebrar as regras depois de dominá-las.

Como este livro irá beneficiar você

Este livro é para qualquer desenvolvedor que sente que seu código piora em vez de melhorar com o tempo. O que você está lendo é um manual que ensina **aos desenvolvedores de software profissionais as melhores práticas essenciais de design e arquitetura de software** que eles não ensinaram na escola.

Este livro é para:

- Graduados em Bootcamp, desenvolvedores juniores, intermediários e até seniores - basicamente, qualquer desenvolvedor que queira aprender a dominar o design de software.
- Desenvolvedores frustrados com a escrita de código com bugs e com as coisas quebrando com o tempo.
- Desenvolvedores interessados nas técnicas testadas em batalha para enviar código de forma confiável, repetida e confiável sem introduzir regressões.
- Desenvolvedores que desejam aprender a projetar aplicativos de grande escala.
- Desenvolvedores que desejam aprender a escrever **software limpo, flexível, testável e de fácil manutenção**.
- Desenvolvedores que se preocupam com o envio de código de qualidade e desejam aprender como escrever um código que possa realmente ser testado.

Este livro tem dois objetivos. A primeira é mostrar as *incógnitas* comuns do design e arquitetura de software, desde o código limpo até os microsserviços e desde o design de objetos até o Domain-Driven Design. A segunda parte é apresentar a você as ideias mais influentes e a **sabedoria prática e açãoável** dos phronimos — nossos professores — sobre como escrever código testável, flexível e sustentável.

Dependendo da sua experiência, algumas dessas ideias podem ser novas para você, mas, na realidade, muito poucos desses tópicos e técnicas são *novos*. A grande maioria do que você vai ler são técnicas que existem há muito tempo. É, no entanto, provável que você não tenha sido exposto a alguns deles ou simplesmente não tenha sido mostrado como fazê-los completamente.

O que estou oferecendo é uma síntese de técnicas de vários desenvolvedores muito mais sábios do que eu. Se você encontrar algo particularmente inteligente ou útil, tenho

para creditar a experiência de meus predecessores. Se você encontrar um erro, pode presumir que a culpa é minha.

Como este livro está organizado

Dividi este livro em várias partes.

As Partes I a III colocam você a par das incógnitas desconhecidas do design de software, ensinam como tomar decisões de design mais *amigáveis ao ser humano* e resumem todos os hábitos e técnicas que os especialistas estão usando para criar um projeto de software de dentro do cliente. cabeçotes a serem implantados na produção.

As Partes IV a VII enfocam a mecânica de programação orientada a objetos, princípios de design, padrões e como enviar código simples e testável de forma incremental usando o Test-Driven Development.

No VIII e IX, mudamos nosso foco para a arquitetura, aprendendo diferentes arquiteturas para ajudar a informar melhor o esqueleto ambulante que decidimos. É aqui que aprendemos como construir uma arquitetura de aplicativo da web à prova de balas usando Domain-Driven Design, Hexagonal Architecture e CQRS.

Nos capítulos finais do livro, discutimos tópicos avançados de teste, como simulação, BDD e técnicas para obter o melhor retorno sobre o investimento de sua arquitetura de teste em aplicativos de back-end e front-end. Terminamos discutindo brevemente como o software é dimensionado no mundo real, tanto do ponto de vista do tráfego quanto das pessoas.

Alguns capítulos contêm exercícios para testar sua compreensão e adquirir experiência com as técnicas que abordamos. Usaremos uma mistura de questões de reflexão, katas de codificação e projetos para mantê-lo progredindo.

No final de cada capítulo, também incluo links para referências para que, se você quiser se aprofundar, leia os livros, blogs, artigos e assista aos vídeos que informaram o capítulo.

TypeScript

A linguagem que usamos para exemplos neste livro é TypeScript. Se você não está muito familiarizado com isso, não se preocupe, vou ensinar o que você precisa saber à medida que avançamos.

Introdução ao TypeScript: Para baixar o TypeScript e configurar um projeto, visite “[Como configurar um projeto TypeScript + Node.js](#)”.

Recursos

Artigos

- [https://howtobeastoic.wordpress.com/2016/09/20/sophia-vs-phronesis two-conceptions-of-wisdom/](https://howtobeastoic.wordpress.com/2016/09/20/sophia-vs-phronesis-two-conceptions-of-wisdom/) <https://en.wikipedia.org/>
- [wiki/Phronesis](https://medium.com/@ericelliott/wiki/Phronesis) <https://medium.com/@ericelliott/>
- [se o treinamento n%C3%A3o %C3%A9 realista voc%C3%A9 est%C3%A1 na ind%C3%BAstria errada-32e488b864ad](https://medium.com/@ericelliott/se-o-treinamento-n%C3%A3o-%C3%A9-realista-voc%C3%A9-est%C3%A1-na-ind%C3%BAstria-errada-32e488b864ad)

Parte I: Até a Velocidade

Programar é a arte de dizer a outro ser humano o que queremos que o computador faça. Mais especificamente, como comerciantes pagos , nossa missão é escrever código testável, flexível e sustentável. Como fazemos isso bem? Os últimos 40 anos introduziram uma série de movimentos influentes (XP, Agile, Craftsmanship), técnicas de programação (TDD, Pair Programming, CI/CD) e contaram com a descoberta de padrões e estilos de arquitetura (como DDD, arquiteturas em camadas, etc.) para desenvolver software testável, flexível e sustentável. Vamos nos atualizar no mundo do design de software.

1. Complexidade e o mundo do design de software

A complexidade é o principal desafio do design de software; isso prejudica as bases de código e as torna difíceis de manter. Estamos em busca de maneiras simples, testadas em batalha e repetíveis para conquistar a complexidade.

Objetivos do capítulo

- Entenda a complexidade e como ela encontra seu caminho no software
- Aprenda a reconhecer a complexidade no código
- Aprenda técnicas práticas para mitigar e superar a complexidade em projetos de software

O objetivo do design de software

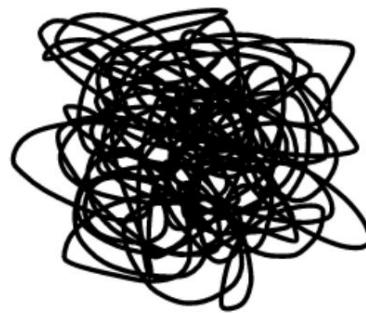
Na [Introdução](#), aprendemos que, como desenvolvedores de software pagos, o objetivo do design de software é:

Construir produtos que **atendam às necessidades do cliente** e que possam ser **alterados pelos desenvolvedores de maneira econômica**.

O inimigo que está em nosso caminho para realizar isso é a *complexidade*.

Complexidade

O estado padrão do mundo é a complexidade. Vemos isso em todos os lugares. Pense em quanto pouco esforço é necessário para o seu quarto ficar bagunçado, para os pratos se acumularem ou para o chão ficar sujo com o tempo. Parece que quando você aplica a menor quantidade de energia, o mundo tende à complexidade – ou como os físicos gostam de chamá-lo, à *entropia*: aleatoriedade, incerteza ou um estado de desordem.



A entropia foi usada pela primeira vez no estudo da termodinâmica, mas para nós - desenvolvedores de software - a entropia está no centro de nossa maior limitação como desenvolvedores de software. Para entender.

Simplificando: se um sistema é muito complexo para entendermos, não podemos mantê-lo.

O que é complexidade em software?

Complexidade é tudo o que torna o sistema difícil de **entender e modificar**.

Embora possa ter sido fácil fazer alterações na base de código no início do projeto, com o tempo, a complexidade estrangula nossa base de código - prejudicando nossa capacidade de entendê-la e, de repente, agora é muito mais difícil fazer alterações. Novos recursos levam mais tempo para serem construídos; eles exigem mais esforço, energia e *dinheiro*.

Complexidade é um fenômeno incremental

Se não fizermos nada além de adicionar novos recursos, mesmo apesar de nossos melhores esforços, a complexidade potencial de um sistema aumentará gradualmente.

Sem as estruturas de manutenção adequadas, mais código significa que há mais coisas para entender, mais maneiras de quebrar e mais coisas a serem consideradas ao adicionar ou alterar recursos.

Tipos de complexidade

Existem dois tipos de complexidade com os quais precisamos nos preocupar: complexidade essencial (complexidade que não podemos *controlar*) e complexidade incidental (complexidade que podemos *controlar*).

Complexidade essencial

Complexidade essencial é o tipo de complexidade sobre a qual não podemos fazer nada. Esse é o tipo de complexidade que faz parte da própria natureza do sistema.

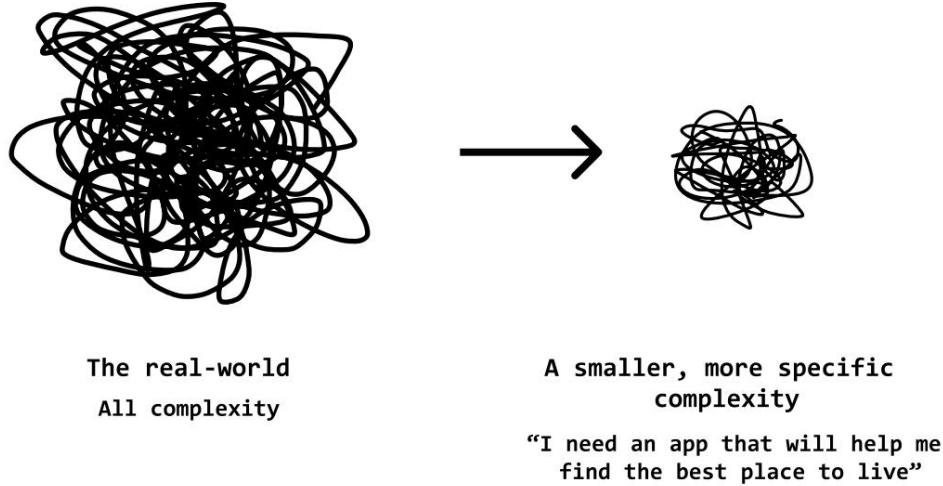
Para entender melhor a complexidade essencial, podemos examinar o *mundo real*.

O *mundo real* é uma colcha de retalhos de sistemas. Para solicitar um empréstimo, melhorar suas amizades, ganhar mais dinheiro ou casar com seu parceiro, existem sistemas que seguimos.

Alguns desses sistemas são muito conhecidos e padronizados, como o check-in para sua consulta no dentista. Outros sistemas são menos conhecidos e provavelmente mais subjetivos, como fazer amigos em uma nova cidade.

Quando temos o desafio de **desenvolver um produto para um cliente**, o que estamos fazendo é **melhorar um sistema do mundo real, desenvolvendo uma maneira automatizada ou melhor de obter o resultado final desejado**.

Por exemplo, se você foi contratado para criar um aplicativo que “ajuda as pessoas a encontrar o melhor lugar para morar”, você está potencialmente melhorando o sistema existente que envolve navegar na Internet por horas procurando aluguel de condomínios, apartamentos e casas em várias cidades.



Uma complexidade menor (mas ainda complexa) do mundo.

Para nosso aplicativo de aluguel de exemplo, a *complexidade essencial* que estamos adotando é, no total:

- os recursos (casos de uso), por
 - exemplo. inscreva-se, crie perfil, pesquise, inscreva-se para alugar
- os dados (estado) necessários para preencher as histórias do
 - usuário, por exemplo. "Ao exibir um aluguel, os usuários precisam ver a metragem quadrada, o número de quartos e as fotos dos quartos." o comportamento
- (lógica de negócios e regras de aplicação) que governa os dados, por exemplo. "Se eu definir a faixa ideal de aluguel em meu perfil para 50KM, não quero receber recomendações de lugares fora dessa faixa." e ignorando a aparência da interface do usuário (você)
- sabe, para deixar as coisas bonitas), semanticamente, o que precisamos apresentar ao usuário, por exemplo. "Depois de fazer login, devo ver meu painel com as
 - recomendações de aluguel mais recentes."

Se esses recursos são o que o cliente está pedindo e temos certeza de que cada um desses recursos é necessário para o usuário atingir seu objetivo, então temos a *complexidade essencial* do aplicativo.

Esta é a complexidade mínima. É o chão. É declarativo. Isso é o que precisamos construir, mas não *como* é construído.

Complexidade acidental

A complexidade acidental é introduzida assim que começamos a construir o sistema. Você pode pensar na complexidade acidental como **virtualmente qualquer outra complexidade que não seja a complexidade essencial**.

Este é o espaço de *implementação*. É o imperativo.

Toda vez que tomamos uma decisão de design, seja para criar uma classe aqui, usar uma função ali, fazer um loop aqui — estamos introduzindo algum tipo de complexidade.

Na realidade, existe *alguma* complexidade acidental que **pode ser evitada** — como duplicação, código morto e comentários desatualizados. Mas também existem outras formas sutis de complexidade que são tão fundamentais para a natureza da programação orientada a objetos moderna, como *estado* e *controle*, que, a menos que adotemos uma abordagem puramente funcional para o design de software, acharemos muito difícil removê-las completamente.

Abordagens puramente funcionais existem e as discutiremos em [23. Paradigmas de programação](#), mas sempre haverá compensações. É o que Fred Brooks disse em 1987, “não há balas de prata”.

Causas de complexidade acidental

Errar os requisitos

Uma das primeiras maneiras possíveis de introduzir a *complexidade acidental* é apenas obter os requisitos errados. Há o que *achamos* que precisamos construir e há o que o cliente realmente deseja. Deixar de transformar adequadamente as conversas com o cliente em requisitos *funcionais* e *não funcionais* é uma receita para o desastre.

Dependências (acoplamento forte)

Uma das primeiras coisas que aprendemos quando começamos a programar é dividir o problema em partes menores. Isso é **decomposição**.

A decomposição é ótima porque, em vez de focar em todo o problema, podemos concentrar nossos esforços em partes menores (componentes).

É uma ótima técnica, mas como o software não faz muito até conectarmos as peças, somos forçados a conectar as coisas. Dependendo das estratégias que usamos quando fazemos isso, podemos introduzir **dependências concretas**.

Uma *dependência* é um componente necessário para que outro componente funcione.

Pegue o seguinte código TypeScript, por exemplo:

```
class UserService
  { constructor
    ( banco de dados privado:
      MySqlConnection, logger privado: Logger
    ) {}

    criarUsuário () { ... }
  }
```

Para criar um UserService, precisamos passar o db e o logger como dependências (dependências concretas, para ser mais específico).

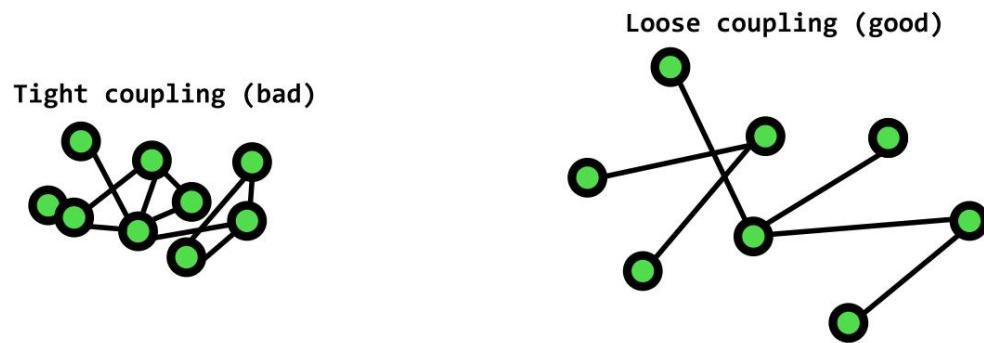
Concreções: Um objeto concreto é um objeto literal criado usando a palavra-chave `new`. A alternativa às concreções são as abstrações (interfaces, classes abstratas, tipos).

Isso pode não parecer uma coisa tão ruim, mas à medida que nossa base de código cresce em tamanho, realmente precisamos confiar em testes para garantir que as coisas ainda funcionem da maneira que queremos.

No exemplo acima, não há como testar um `UserService` isoladamente. Para testar um `UserService`, precisamos criar e passar um `MySqlConnection` e um `Logger`. Se o seu objeto `MySqlConnection` se conectar a um banco de dados real e ativo, isso tornará seus testes lentos. E se o seu objeto `Logger` imprimir muito ruído no console, seus testes serão especialmente confusos e difíceis de ler.

Como não podemos testar nosso `UserService` isoladamente, podemos dizer que ele está fortemente *acoplado*.

Acoplamento: Dependências são uma forma de *acoplamento*. O acoplamento é uma medida de quão entrelaçados dois componentes estão. Um componente pode se referir a uma rotina, método, função, classe, módulo ou até mesmo um componente arquitetônico inteiro, como o banco de dados ou o aplicativo da web. Precisamos entender como os componentes colaboram uns com os outros e nos esforçar para um acoplamento *flexível* entre eles.

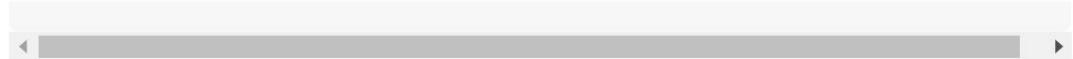


Existem outras formas de acoplamento também. Para citar alguns, há:

- **Acoplamento de subclasse** — Quando uma classe filha é conectada à classe pai.
 - ex.: SomeClass estende BaseClass
- **Acoplamento temporal** — Quando duas ações são agrupadas em um módulo porque acontecem ao mesmo tempo.
 - Ex.: Como depois de executar CreateUser, precisamos executar a operação SendAccountVerificationEmail , vamos apenas colocar os dois na mesma função CreateUserAndSendEmail e chamar essa função.
- **Acoplamento dinâmico** — Quando a ordem em que as coisas são executadas é importante.

○ por exemplo.:

```
email = Email()
email.setRecipient("foo@example.com")
email.setSender("me@mydomain.com")
email.send() // Isso seria errado fazer neste momento email.setSubject("Olá mundo")
```

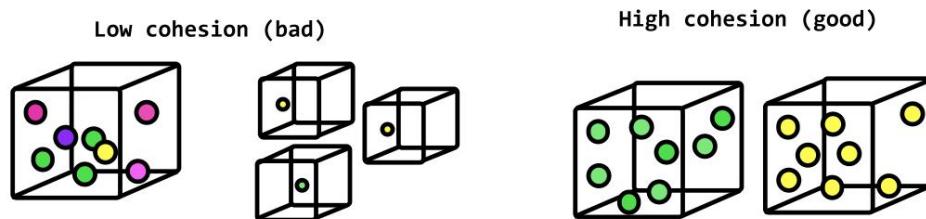


Não podemos evitar totalmente o acoplamento, mas **podemos** nos esforçar para um acoplamento *flexível* como forma de minimizar a complexidade.

Obscuridade (baixa coesão)

Se estivermos olhando para algum código e descobrirmos que é muito difícil entender *o que o código faz ou o que precisamos saber sobre o código em questão*, dizemos que ele é **obscuro** e tem **baixa coesão**.

Coesão: A coesão é uma medida de como os componentes relacionados estão dentro de um módulo específico. É uma medida de quanto eles pertencem um ao outro. Você tem coisas aqui principalmente focadas no tópico em questão? Ou existem construções aqui que provavelmente pertencem a outro lugar?



Exemplos de obscuridade e baixa coesão são:

- **Informações inúteis** — comentários que não fornecem nada de valor para o leitor

- **Variáveis, métodos, classes, etc. mal nomeados** — talvez eles digam uma coisa, mas façam outra, não sejam escritos usando termos do domínio, sejam excessivamente concisos ou ambíguos
- **Empacotamento ruim** — quando coisas relacionadas não são empacotadas
- juntas **Incorreto (ou confuso) abstrações** — **** pode ser que a abstração que você está criando não seja prontamente compreensível para a próxima pessoa.
- **Abstrações desnecessárias (precoces)** — ****também pode ser o caso de você criar uma abstração muito cedo quando ela ainda não é necessária — isso tem o efeito de tornar seu código mais complexo do que ele precisa ser Abstrações
- perdas — e há também o caso em que você provavelmente *deve* abstrair muito do trabalho para outra camada de código, porque isso pode tornar o código muito mais fácil de ler.

Idealmente, queremos buscar alta coesão porque torna o código mais compreensível.

Mais desenvolvedores

Como aprendemos em The Mythical Man-Month, adicionar mais desenvolvedores a um projeto tem o efeito adverso de introduzir mais complexidade.

Com mais cérebros, opiniões e criadores de abstrações, um estilo de codificação consistente e práticas técnicas disciplinadas são fundamentais para evitar que a complexidade exploda exponencialmente.

Recursos de linguagem e API

Quanto mais poder e opções sua linguagem de programação, biblioteca, estrutura ou API tiver, maior será a área de superfície para criar complexidade acidental.

Ferramentas com menos opções têm menos probabilidade de serem mal utilizadas de forma desastrosa. Se você já se sentiu sobre carregado ou desencorajado ao usar a programação orientada a objetos, não posso dizer que estou surpreso. OOP vem com grande poder. E com grande poder vem a possibilidade de você dar um tiro no próprio pé. Embora a programação funcional possa parecer mais atraente porque a área de superfície do que é possível é muito menor, a programação puramente funcional no mundo real apresenta um conjunto muito diferente de desafios.

Otimização prematura

O fortemente citado Donald Knuth diz que:

Otimização prematura é a raiz de todo o mal

90% do tempo, otimização significa introduzir mais estado. Adicionamos um cache, alguns truques de banco de dados, bagunçamos o Kubernetes, etc. No influente artigo de Ben Moseley , *Out of the Tar Pit* , ele argumenta que o estado não essencial é uma das principais causas da complexidade do software. Uma das melhores coisas que podemos fazer é evitá-lo até que seja absolutamente necessário.

Outras causas

Existem muitas outras formas de complexidade, mas, de alguma forma, acredito que todas elas impactam negativamente o acoplamento e (ou) a coesão: as duas melhores medidas que temos de quão bem estamos domando a complexidade.

- Duplicação — acoplamento (precisamos alterar o código em vários lugares)
- Código morto — coesão (obscurece nossa compreensão do que devemos fazer)
- Modularidade pobre — coesão (há uma mistura de abstração de alto e baixo nível que dificulta a compreensão do propósito do código)
- Documentação ruim — coesão (novamente, é difícil saber o que fazer)

Estes são todos os cheiros de código. Aprendemos sobre outros comuns e suas refatorações na [Parte V: Projeto Orientado a Objetos \(Com Testes\)](#).

Como detectar a complexidade

Agora que temos uma ideia decente de como é a complexidade, vamos discutir os sintomas.

Ondulação

Quando o que parece ser uma simples mudança de código na verdade significa que você tem que fazer modificações em várias outras partes do seu código, você pode estar experimentando *ripple*.

O Ripple sinaliza que aspectos do seu código podem estar fortemente acoplados, que há falta de encapsulamento ou que as dependências sabem muito umas sobre as outras.

Carga cognitiva

Quanto mais elementos você precisar manter em seu cérebro para concluir uma determinada tarefa, maior será a carga cognitiva. Indo para o *outro* lado da ondulação, a carga cognitiva pode sinalizar que temos *muita* abstração e encapsulamento. Isso acontece quando criamos muitos elementos, camadas, classes com muitos métodos, usamos muitas variáveis globais, falhamos em impor decisões de design consistentes e organizamos os arquivos em nosso código por infraestrutura (ou tipo) em vez de recurso — forçando para alternar entre os arquivos.

Fraca capacidade de descoberta

Digamos que você esteja trabalhando em um projeto e precise adicionar alguma lógica de validação adicional a um recurso EditUser . Quanto tempo leva para você encontrar o código onde precisa fazer a alteração?

A descoberta é sobre identificar onde as coisas estão, o que são e o que podemos fazer com elas - os recursos da base de código, um objeto, uma função ou método.

Fraca compreensibilidade

Agora vamos imaginar que encontramos a área de código envolvida na implementação do recurso EditUser . Estamos quase lá. Queremos adicionar a lógica de validação, mas, olhando para o código, **não temos ideia de como fazer isso**.

Compreender é **saber como executar a ação, realizá-la** e depois **confirmar que obtivemos sucesso** ao perceber que obtivemos o resultado pretendido.

Por exemplo, elevadores bem projetados simplificam o entendimento apresentando apenas duas opções: subir e descer. Quando pressionamos um dos botões, os elevadores acendem e fazem barulho, dando feedback imediato e nos avisando que realizamos a sequência de ação correta.

Ferramentas com baixa compreensão dificultam:

- Entender *como usar* algo.
- Entenda se estamos usando algo da *maneira correta*. Quando as ferramentas fornecem muitas opções para fazer a mesma coisa, ficamos questionando se tomamos a decisão correta ou não.
- Entenda se o que fizemos *funcionou*.

Abstrações com vazamento: você já se deparou com um obstáculo com React, Angular ou talvez um ORM que você goste? Quer projetemos nossas próprias abstrações ou usemos bibliotecas ou estruturas, “todas as abstrações não triviais eventualmente vazam”. Uma [abstração com vazamento](#) ocorre quando uma ferramenta deveria abstrair todas as coisas difíceis, mas, em vez disso, acabamos em uma situação em que somos forçados a aprender alguns dos detalhes internos da ferramenta para fazê-la funcionar da maneira que pretendíamos.

Mitigando a complexidade

Programação tática vs. estratégica

Existem duas abordagens gerais para sentar e escrever código: a abordagem *tática* e a abordagem *estratégica*.

Na **abordagem tática**, usamos a força bruta em uma solução até que ela esteja completa. Não estamos muito preocupados com as implicações de nossas decisões de design - estamos apenas codificando e usando o que funcionar melhor até que funcione da maneira que queremos.

A abordagem tática tem seus casos de uso prático. Quando você está construindo uma prova de conceito, você está essencialmente executando um experimento. Ao fazer programação exploratória, você está mergulhando no código, vendo o que pode fazer e voltando para respirar. Ambos os casos de uso são ótimos quando você deseja ver o que pode fazer com o código. A abordagem tática só faz sentido quando você está construindo coisas que tem certeza de que não vão durar muito.

O lado tático é 0% focado no aspecto de manutenção da programação. Portanto, se aplicarmos essa abordagem a um projeto da vida real, sim — podemos fazê-lo da primeira vez, mas é provável que não seja fácil para você ou qualquer outra pessoa de sua equipe fazer alterações futuras.

Com uma **abordagem estratégica**, há uma divisão 50/50 entre escrever código para o **cliente** e escrever código para **nossos mantenedores**. É como quando construímos um mercado; fazemos atividades que alimentam os dois lados do mercado porque são igualmente importantes, e não podemos ter sucesso a menos que tenhamos ambos.

A Sun Microsystems, a empresa por trás do Java — tem o seguinte a dizer sobre a manutenção de um projeto de software:

- 40% a 80% do custo vitalício de um software vai **para a manutenção e quase nenhum**
- **software é mantido durante toda a sua vida pelo autor original.**

Portanto, devemos *investir* na base de código seguindo técnicas que nos permitirão (e futuros mantenedores no futuro) mantê-la e manter a complexidade sob controle.

Extreme Programming: A metodologia original de desenvolvimento Agile

Extreme Programming é a metodologia de desenvolvimento OG Agile. Criado por Kent Beck durante a bolha pontocom, é uma estrutura de técnicas de melhores práticas conhecidas por criar software de alta qualidade que resiste ao teste do tempo.

Quando as empresas perceberam que poderiam gerar fluxos de receita enviando sites ao mercado mais rapidamente, a corrida começou.

Kent, que estava trabalhando em um projeto para a Chrysler, tornou-se o líder de um projeto substancial em 1996. Como líder do projeto, ele tomou nota das metodologias de desenvolvimento que estava usando em sua equipe, substituindo coisas que não funcionavam por outras. isso funcionou muito bem e, eventualmente, ele formalizou o que ficou conhecido como *Programação Extrema*: um conjunto *extremo* de práticas recomendadas para a entrega consistente de código de qualidade no prazo.

A influência da Programação Extrema (também conhecida como XP) em nossa indústria tem sido monumental. Ele definiu o padrão de como os desenvolvedores profissionais de software transformam as necessidades dos clientes em produtos nos quais os usuários confiam.

Práticas técnicas

Este é o conjunto original de práticas técnicas da Extreme Programming.

Opinião

- Programação em par — quando vários programadores
- Jogo de planejamento — Planejamento de lançamento (com o cliente) e planejamento de iteração (com sua equipe)
- Desenvolvimento orientado a testes (TDD) — novas funcionalidades são adicionadas primeiro escrevendo um teste com falha, fazendo-o passar e, em seguida, contemplando o design
- Equipe inteira — o *cliente* faz parte da equipe; eles precisam estar disponíveis para perguntas a qualquer momento

Processo contínuo

- Integração contínua — Evite conflitos de código significativos com outros desenvolvedores fazendo upload constante de seu código a cada poucas horas
- Melhoria de design/Refatoração — Quando começamos a ver sintomas de complexidade, devemos considerar a refatoração para tornar o sistema mais simples e genérico
- Pequenos lançamentos — lançamentos pequenos e frequentes que agregam valor

Compreensão compartilhada

- Padrão de codificação — decida sobre um conjunto *consistente* de convenções de codificação que você e sua equipe usarão ao longo do projeto
- Propriedade coletiva do código — Embora as equipes possam ser formadas, todos são responsáveis por todo o código em geral.
- Design simples — O código mais simples possível tem testes, não contém duplicação, maximiza a clareza e minimiza o número de elementos

- Metáfora do sistema (Design orientado ao domínio) — Use a linguagem do domínio para nomear recursos, métodos, variáveis, classes e assim por diante.

bem-estar do programador

- Ritmo sustentável — Os desenvolvedores não devem trabalhar mais de 40 horas por semana

Podemos aprender muito com essas práticas. Neste livro, discutiremos a maioria deles, mas focaremos especificamente no seguinte:

- Programação em Par
- TDD
- Refatoração e
- Design Orientado a Domínio

Valores (de Programação Extrema)

Os valores ditam a ação prática do mundo real. Chamamos esse comportamento *de prática*.

Gastamos muito tempo com as *práticas mencionadas acima*, mas sabemos que elas se originam fortemente dos valores originais de Kent Beck na metodologia Extreme Programming.

Opinião

O design ruim é realmente uma complexidade criada por nós mesmos.

Quando escrevemos código e criamos abstrações que são mais difíceis de entender e trabalhar do que o necessário, devemos tomar cuidado para corrigi-los.

Às vezes, porém, perdemos abstrações ruins, vamos muito longe e é muito desafiador e complicado de consertar, então simplesmente decidimos viver com nossos erros.

E se pudéssemos detectar designs ruins *mais cedo*?

Um valor importante do XP é **introduzir loops de feedback em nossos processos de escrita de código.**

Os loops de feedback nos forçam a parar por um momento e observar os resultados do que acabamos de fazer. Isso nos dá a chance de ver nosso progresso. Se estivermos propensos a fazer pausas, é mais provável que encontremos designs ruins antes que eles cheguem ao ponto em que não podemos mais mantê-los.

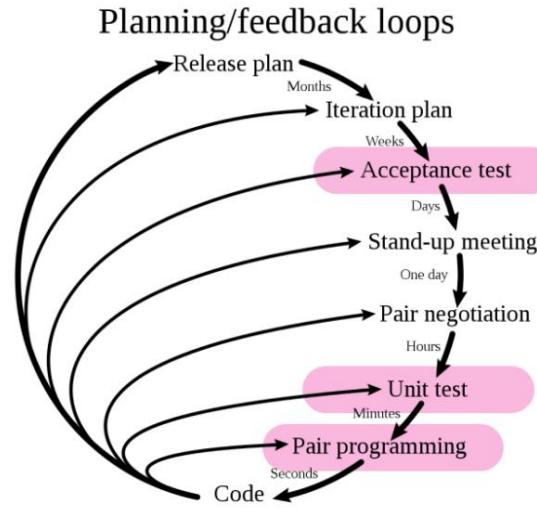
Uma forma de feedback que a maioria dos desenvolvedores profissionais faz em seus trabalhos diários são **as revisões de código**. Para aqueles que não realizam revisões de código, são discussões que você tem com sua equipe antes que seu código seja mesclado na produção. As revisões de código dão a seus colegas de equipe a chance de criticar seu código quanto à qualidade.

Se você está fazendo isso, provavelmente está trabalhando em uma *equipe* que valoriza o *feedback*.

No entanto, esta é a **última linha de defesa para feedback**.

Você já fez anotações em sua revisão de código em que as alterações que percebeu que precisava fazer eram *estruturais*?

E se eu dissesse que existem outras técnicas de feedback que você pode usar para obter feedback? Técnicas para obter feedback *muito mais cedo* no processo. Mesmo *enquanto* você está codificando.



No diagrama acima, podemos ver vários loops de feedback que existem na metodologia Extreme Programming. No contexto deste livro, os que vamos focar são:

- Programação em par
- TDD
 - Testes de aceitação
 - e testes de unidade

Simplicidade

Nosso objetivo deve ser construir a coisa mais simples possível que funcione e seja compreensível para os desenvolvedores.

Se houvesse uma maneira de pegar a *especificação formal* do que estamos sendo solicitados a construir - todas as histórias de usuários e casos de uso (os recursos - isto é) e, em seguida, entregá-la a um construtor sem código e desenvolvê-la de maneira limpa em uma solução para nós, provavelmente seria a coisa mais simples possível que poderia funcionar.

Infelizmente (ou felizmente — para nós), essas ferramentas não existem (ou existem, mas ainda não são boas no que fazem).

Portanto, temos que construí-lo nós mesmos.

Vamos imaginar dois desenvolvedores sentados separadamente, ambos encarregados de criar o mesmo recurso.

Para construí-lo, eles criam abstrações.

Eles escrevem classes, funções, objetos e modelam os recursos no código.

Considere que cada desenvolvedor tem sua própria abordagem para implementar os recursos. Se nenhuma das soluções dos desenvolvedores faz sentido uma para a outra, quem está errado?

Nenhum deles. Isso é difícil. Leva *anos* para os desenvolvedores entenderem como projetar boas abstrações e reconhecer as ruins.

Elaborar boas abstrações diretamente de seu cérebro na primeira tentativa é muito improvável de acontecer.

Obviamente, não temos anos para dominar isso. Precisamos ser produtivos *hoje*. Precisamos de uma maneira repetível e prática de desenvolver a coisa mais simples possível.

Na [Parte III: Phronesis](#), aprendemos sobre todas as técnicas *baseadas em recursos* que os especialistas usam em projetos ágeis do mundo real. TDD (Test-Driven Development) é uma dessas técnicas. Talvez o melhor para desenvolver a coisa mais simples possível que funcione. Juntamente com técnicas como Prioridade de Transformação, Padrões de Projeto e Calistenia de Objetos, definimos maneiras de inventar gradualmente abstrações simples e necessárias com TDD.

Aprendemos os fundamentos do TDD na [Parte IV: Princípios do Desenvolvimento Orientado a Testes](#), domine as coisas avançadas na [Parte X: Advanced Test-Driven Development](#), e aprenda como melhorar gradualmente nossos projetos orientados a objetos na [Parte V: Projeto Orientado a Objetos \(com testes\)](#), [Parte VI: Princípios de Projeto](#) e [Parte VII: Padrões de Projeto](#).

Comunicação

Vale a pena repetir que se 50% do que estamos fazendo é construir produtos que atendam às necessidades do usuário, os outros 50% estão tentando escrever códigos facilmente compreendidos por outros desenvolvedores.

Programar é a arte de dizer a outro ser humano o que ele quer que o computador faça —
Donald Knuth

Se valorizamos a comunicação, queremos usar a maneira mais eficaz de comunicar a intenção do programa. Isso significa escrever documentação? Poderia , mas para mim - a documentação mais eficaz é o próprio código.

Se escrevermos código em um estilo declarativo, sempre haverá uma única camada mais alta de abstração que representa o código o mais próximo possível da *complexidade essencial* . Em um contexto orientado a objetos, este será nosso código *de teste de aceitação* .

Nosso código de teste atua como um ponto de montagem para que outros desenvolvedores aprendam sobre o domínio, o que nosso código faz e permite que eles desçam uma camada mais profunda em um nível de abstração um pouco mais complexo. A complexidade é empurrada para baixo.

Na [Parte IX: Construindo Aplicações Web com Design Orientado a Domínio, Arquitetura Hexagonal e CQRS](#), aprendemos como usar um pequeno conjunto de padrões arquiteturais para encapsular a complexidade e representar o domínio/complexidade essencial da maneira mais declarativa possível em uma linguagem orientada a objetos contexto.

Um *padrão de codificação* consistente é uma questão igualmente importante se quisermos escrever um código que comunique bem sua intenção. Antes de escrever nosso primeiro teste, na [Parte II: Humans & Code](#), discutiremos como desenvolver (e aplicar) um padrão de codificação ** que resulte em um código que os humanos gostem de ler e trabalhar.

Resumo

- O objetivo do design de software é construir produtos que **atendam às necessidades do cliente** e que possam ser **alterados pelos desenvolvedores de maneira econômica**.
- Qualquer um pode aprender a escrever código e construir um produto, mas o que separa o amador do profissional é a capacidade de vencer a complexidade ou sucumbir a ela.
- XP é provavelmente a melhor metodologia de desenvolvimento para reduzir a complexidade em projetos de software orientados a objetos, e os valores XP ditam nossas técnicas e princípios.
- Práticas técnicas como TDD (consulte [a Parte IV: Fundamentos do desenvolvimento orientado a testes](#)) e programação em pares ([consulte a Parte III: Phronesis](#)). Eles nos ajudarão a escrever um código melhor e detectar designs ruins antes que tenham a chance de se solidificar em nosso código.
- Começaremos tratando-as como *regras* porque ainda não temos experiência com elas. Na [Parte III: Phronesis, aprendemos os princípios](#) orientadores por trás do uso especializado dessas técnicas, como “Construir o esqueleto ambulante no Sprint 0” e “Fazer com que o cliente escreva os testes de aceitação”
- Com o tempo, tratadas como regras — pratique as técnicas e você desenvolverá seu próprio conjunto de princípios. Então, você decide por si mesmo se vai usá-los ou não.

exercícios

Em breve

Referências

Artigos

- https://en.wikipedia.org/wiki/No_Silver_Bullet
- <https://khalilstemmler.com/wiki/leaky-abstraction/>
- https://en.wikipedia.org/wiki/Extreme_programming

- https://en.wikipedia.org/wiki/Extreme_programming_practices
- https://en.wikipedia.org/wiki/Chrysler_Comprehensive_Compensation_System
- https://en.wikipedia.org/wiki/Coding_conventions

livros

- Uma Filosofia de Design de Software por John Ousterhout
- Programação Extrema Explicada: Abrace a Mudança por Kent Beck O Mítico
- Homem-Mês por Fred Brooks Robert L. Glass:
- Fatos e Faláncias da Engenharia de Software; Addison Wesley, 2003.

Papéis

- Fora do poço de alcatrão, de Ben Moseley e Peter Marks

2. Artesanato de Software

O movimento Software Craftsmanship é sobre profissionalismo no desenvolvimento de software. Como James Clear escreve em seu popular best-seller *Atomic Habits*, se quisermos obter melhores resultados e manter nossos hábitos, temos que começar com nossa identidade.

Objetivos do capítulo

- Aprenda um breve histórico do desenvolvimento de software e os grandes desafios até aqui

- Para saber por que a habilidade de software é a solução para uma era equivocada de Ágil
- Aprenda as principais ideias por trás do artesanato de software
- Aprenda uma técnica para adotar hábitos de artesanato de software ao longo da vida

Profissionalismo

Antes de ser programador, fui paisagista.

Quando eu era adolescente, com a ajuda do meu padrasto, abri meu próprio negócio de jardinagem para arrecadar fundos para meus primeiros anos de universidade. Chamei-o de Steps Lawn Care. Era um nome adequado para a empresa, pois meu padrasto me ensinava o básico e me conduzia a um nível em que me sentia confortável o suficiente para fazer trabalhos de forma independente.

Comecei sem saber nada sobre paisagismo.

Nas primeiras semanas, saímos e fazíamos trabalhos juntos. Eu observava como ele trabalhava, ajudava no que podia e, com o tempo, eu fazia tudo sozinho.

Eventualmente, por meio de orientação e correção, aprendi como encontrar clientes, definir o escopo de um trabalho, fornecer uma estimativa para o trabalho e satisfazer o cliente fazendo o trabalho bem e no prazo.

Meu padrasto **me ensinou como ser um profissional** nessa indústria.

Percebi que a orientação, a maneira como aprendi o ofício de paisagismo, é **praticada há gerações** e remonta à Europa medieval com mestres e jornaleiros.



Le charpentier.

Der Zimmermann.

The carpenter.

“O carpinteiro” - por artista anônimo ([http://www.digibib.tu-bs.de/?
docid=00000286](http://www.digibib.tu-bs.de/?docid=00000286), Domínio público,

<https://commons.wikimedia.org/w/index.php?curid=981584>)

Naquela época, o mestre demonstrava a forma correta de realizar uma tarefa; então, os *jornaleiros tentam* imitar as habilidades do mestre e são corrigidos por quaisquer erros.

É interessante notar que este modelo, **o modelo de aprendizagem**, ainda é amplamente utilizado em ofícios tradicionais como siderurgia, panificação, encanamento e muitos mais.

Algumas profissões tradicionais valorizam tanto **o profissionalismo** que, devido à regulamentação do comércio, pode ser *illegal* sair por conta própria até que você tenha concluído *um programa de aprendizado*: um certo número de horas de prática com um membro mais experiente.

A programação, por outro lado, é um ofício incrivelmente jovem (e turbulento).

Esperamos que novos desenvolvedores aprendam o que é preciso para ser um profissional de forma independente e, muitas vezes, há distanciamento da geração mais antiga de programadores para os mais novos.

Não apenas isso, mas ainda estamos no processo de concordar com um procedimento padrão para produzir software de qualidade de maneira confiável e pontual (acredito que a resposta aqui seja Agile, mas com as práticas técnicas adequadas, como as defendidas no XP - Programação extrema).

Imagine se os encanadores ainda estivessem tentando descobrir como consertar canos com vazamentos.

Não é uma surpresa que tantos desenvolvedores fiquem confusos, despreparados e sem orientação para *trabalhar profissionalmente*, porque **nós, como indústria, não convergimos sobre o que significa trabalhar profissionalmente**.

Agora, vamos inverter por um segundo. Imagine que você é um desenvolvedor profissional e sabe as coisas certas a fazer, como TDD, Pair Programming, refatoração e afins. Isso é ótimo, mas sua vida *ainda* é difícil. Os gerentes de projeto não técnicos geralmente têm a tendência de recuar nessas coisas. Eles não veem o valor. Essas práticas parecem *recursos que não funcionam*.

Ah, desafios.

Podemos nunca regulamentar o setor de programação profissional, as empresas ainda serão rigorosas na contratação de instrutores profissionais e é improvável que um gerente não técnico insista para que você implemente as melhores práticas técnicas. Ainda mais, talvez nunca introduzamos um conceito como

aprendizados formais obrigatórios em nossa indústria (alguns tentaram - ver 8º [Luz](#)). 

— —

Bem, há algo que possamos fazer hoje?

Sim.

Hoje, o que *podemos fazer* é **adotar uma mentalidade artesanal**.

Software craftsmanship é profissionalismo no desenvolvimento de software.

A *habilidade de software* é uma mentalidade emergente compartilhada por uma crescente comunidade de desenvolvedores dedicados **a elevar o nível**.

Embora este guia seja sobre criação de software, outro título poderia ser “como ser um desenvolvedor de software profissional”.

Ser um profissional (ou artesão – considero que significam a mesma coisa) significa **assumir a responsabilidade por** nossas carreiras, clientes e comunidade.

Significa também **ter orgulho de** nossas soluções, trabalhar **com pragmatismo** e buscar melhorar nossa reputação com o trabalho de excelência que realizamos continuamente.

O profissionalismo é fundamental para escrever um código limpo.

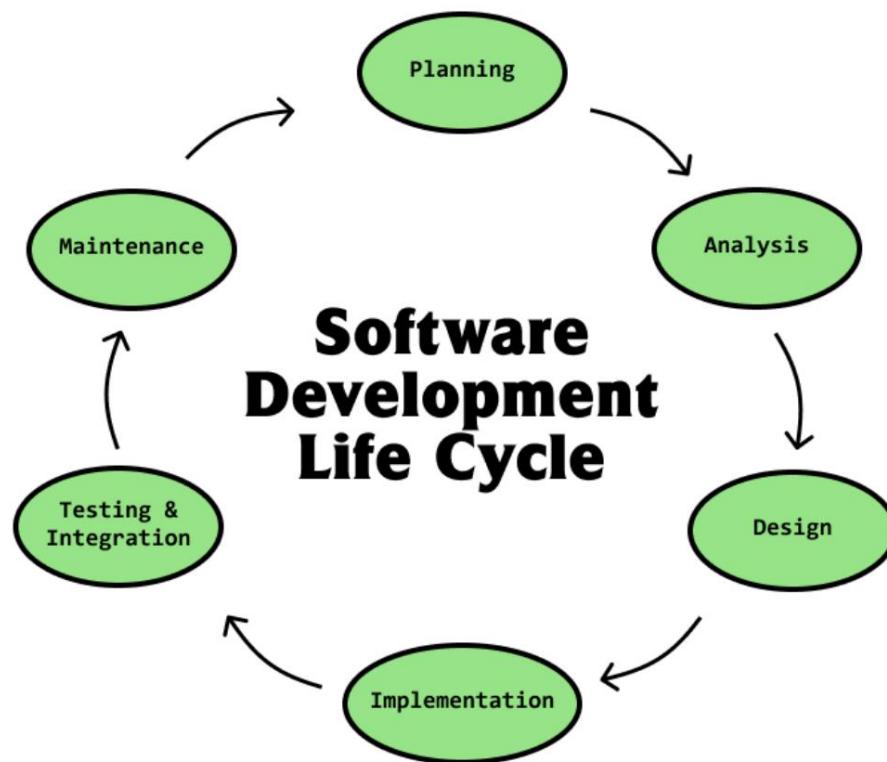
Uma breve história do desenvolvimento de software

Primeiro, para entender completamente a origem do artesanato e respeitar a urgência de por que você deve se esforçar para se tornar um hoje, precisamos discutir as mudanças significativas em nosso setor que nos deixaram onde estamos hoje.

Velocidade de aceleração de programação (50s)

Nos anos 50, a ciência da computação pegou. Nessa época, começamos a entender o ciclo de vida do software como o software é *planejado, projetado, construído e mantido*.

Chamamos isso de **SDLC (ciclo de vida de desenvolvimento de software)**.



O Ciclo de Vida de Desenvolvimento de Software demonstrou o ciclo geral em que os projetos de software operam.

Até então, considerávamos o desenvolvimento de software apenas o ato de escrever código e corrigir bugs. Não havia nenhum processo formal sobre como

isso funcionou. Mas em 1956, criamos [o modelo Waterfall: uma abordagem iterativa](#) para a construção de software.

Nos anos posteriores (por volta dos anos 80 e 90), os profissionais da indústria criariam outras maneiras de trabalhar, novamente opinativas, como XP (Extreme Programming), FDD (Feature-Driven Development), Scrum e assim por diante. Chamamos *isso de metodologias de desenvolvimento de software* e elas agem como **uma estrutura para passar por cada fase do SDLC**, projetadas especificamente para contornar práticas e armadilhas de codificação incorretas. Eles lhe dão regras a seguir.

Por exemplo, uma regra no XP é que o *cliente escreve testes de aceitação*, dessa forma, os desenvolvedores sabem o que devem construir e podem definir a conclusão com mais precisão.

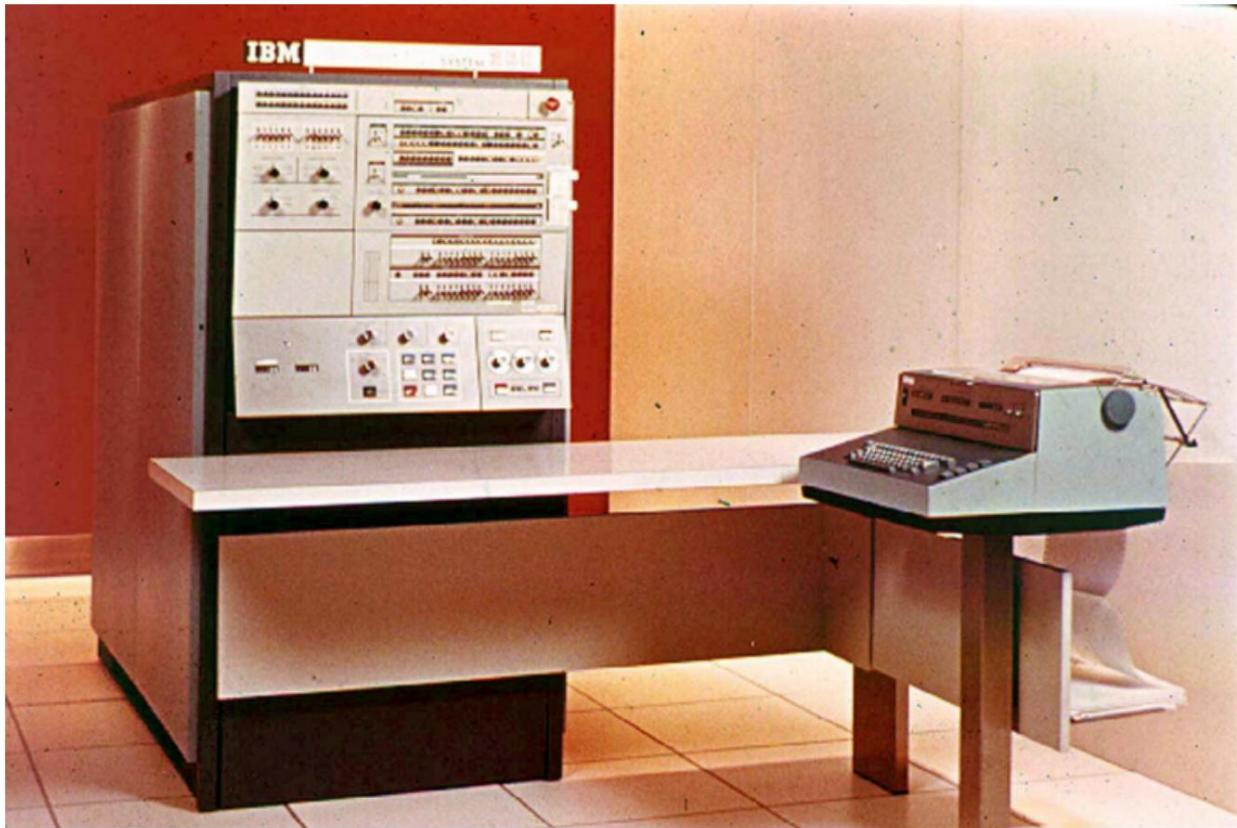
Metodologia de Desenvolvimento de Software: Uma abordagem para desenvolvimento de software, mais específica do que o Ciclo de Vida de Desenvolvimento de Software (SDLC) que formaliza regras, estratégias e práticas para a entrega de projetos de software. Exemplos: XP, FDD, Scrum, Cascata.

Mas nos anos 50 e 60, os processos eram bastante simplificados. E assim passamos a aprender algumas duras lições.

A crise do software dos anos 60-80

Dos anos 60 aos anos 80, descobrimos a maioria dos **principais problemas na engenharia de software** que continuariam a nos atormentar hoje; eles são baseados predominantemente *em produtividade e qualidade*.

Vou te dar um exemplo de cada.



O IBM System/360

Em 1964, a IBM anunciou o IBM *System/360*: o primeiro sistema de computador de uso geral com compatibilidade cruzada entre os modelos. A compatibilidade cruzada era enorme naquela época; a capacidade *de finalmente executar os mesmos programas em todos os sistemas, em vez de precisar se ater a modelos mais antigos e lentos para executar os programas necessários para o uso diário*, foi revolucionária.

Embora espetacular quando finalmente foi lançado, **a falha de produtividade era lendária**. A IBM levou uma década **inteira** e 1.000 **desenvolvedores para** concluir o sistema porque eles não desenvolveram uma arquitetura sólida e compreensível. O influente livro “*The Mythical Man-Month*”, escrito por [Fred Books](#), baseia-se em suas observações gerenciando o desenvolvimento do sistema IBM. Em 1975, depois de concluir o projeto e publicar o livro, nós, como indústria, aprendemos que “adicionar mão de obra a um projeto de software tardio torna-o mais tarde”. Esta é uma das muitas coisas que tivemos que aprender da maneira mais difícil, pois havia poucos projetos tão grandes quanto este antes.



O Therac-25: uma máquina que matou quatro pessoas e deixou duas com ferimentos permanentes

Uma das **falhas de qualidade mais famosas** é a história de 1986-87 do Therac 25: uma máquina de terapia de radiação com defeito que funcionou mal e emitiu dez vezes a quantidade de radiação aceitável, matando quatro pessoas e deixando dois pacientes com lesões permanentes.

O que aconteceu aqui? Grande quantidade. A arquitetura estava condenada desde o início. O desenvolvedor original que escreveu o núcleo do aplicativo nunca havia feito programação de simultaneidade, portanto, a base - todo o design do sistema - tornou-se extremamente difícil de entender depois que ele deixou a empresa.

Além disso, para cortar custos e tornar o código mais fácil de raciocinar, os desenvolvedores removeram os *recursos de segurança de hardware* dos modelos anteriores, substituindo-os para serem controlados inteiramente *por recursos de segurança de software*. Isso significava que, quando as coisas davam errado por causa de bugs de software e a máquina expelia muito mais radiação do que deveria, não havia detector de hardware para desligar a máquina. Os desenvolvedores estavam muito confiantes. Eles não escreveram e executaram testes capazes de testar a moeda. Mesmo quando os operadores descobriram que as pessoas estavam morrendo devido ao que parecia ser um bug crítico, os engenheiros repetidamente transferiram a culpa para si mesmos. Eles insistiram que as mortes foram devidas a “erro do operador”.

Por fim, um dedicado físico da equipe cavou fundo, encontrou o bug e o reproduziu. AECL, a empresa por trás da máquina, lançou atualizações para corrigir o bug. Mas mesmo depois de consertarem, alguns meses depois,

alguém morreu de um bug causado por *um problema totalmente diferente*: um estouro de contador.

Há um certo legado para este evento. É uma demonstração perfeita de como práticas ruins de software, falta de excelência técnica e *profissionalismo para assumir responsabilidades* podem resultar em resultados terríveis (às vezes fatais).

Leia mais sobre o Therac-25 neste artigo intitulado "["Killed By A Machine"](#)".

Bolha pontocom, OOP e Extreme Programação (1995 – 2001)

Chamamos os anos 90 de boom **das pontocom** porque a internet assumiu completamente o controle. Em algum momento, as empresas perceberam que, se pudessem entregar produtos baseados na Internet mais rapidamente, teriam uma enorme vantagem competitiva.

Infelizmente, muitas equipes ainda estavam *usando metodologias iterativas* de desenvolvimento de software como Waterfall, que, nesse novo clima de necessidade de mudanças rápidas, rápidas e ágeis (que continuamos a esperar hoje), geralmente levavam a baixa *produtividade e qualidade*.

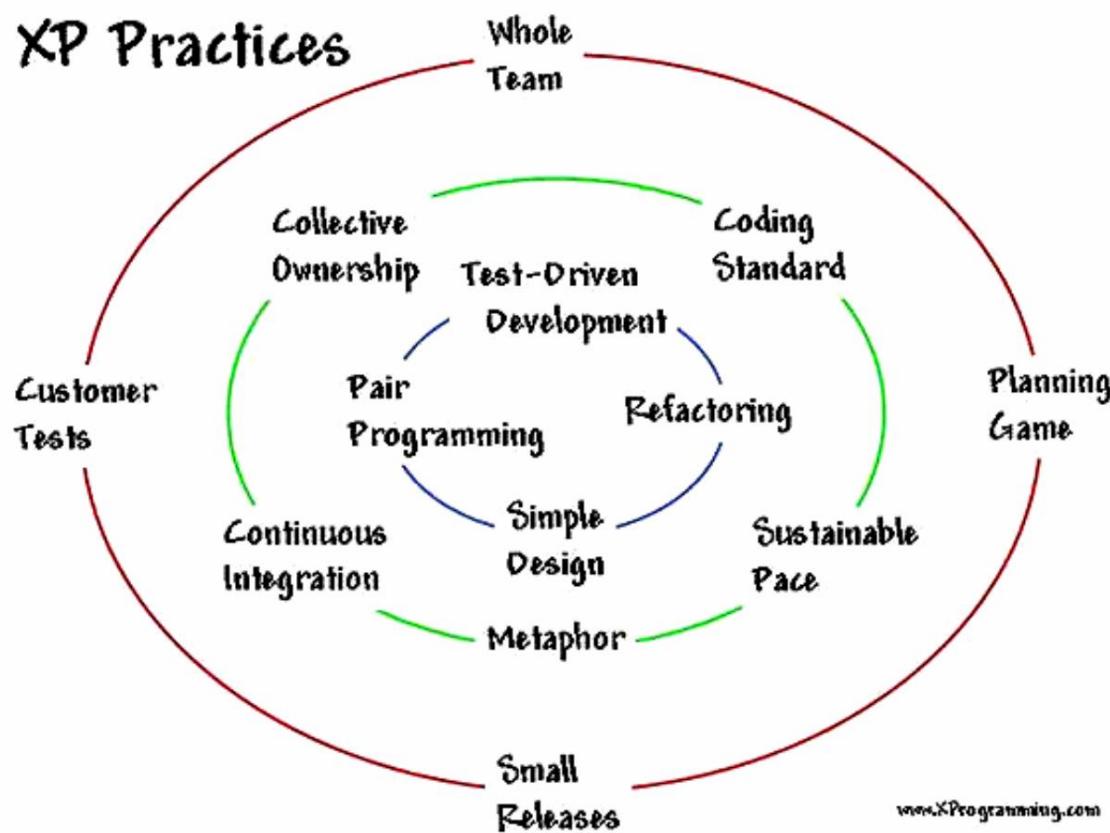
Muitos projetos Waterfall começaram fortes, mas conforme os requisitos do cliente mudaram, os projetos pareciam mais [marchas da morte](#).

Quando os desenvolvedores avançavam da fase de Planejamento para Análise, Projeto e Implementação, se novos requisitos surgissem tarde, não sabíamos como lidar melhor com isso, a não ser terminar o que estávamos trabalhando e começar tudo de novo. Abordagens iterativas não nos ajudaram a acompanhar as demandas de nossos clientes. Freqüentemente, entregávamos coisas incompletas que não eram realmente o que nossos clientes queriam.

Duas outras coisas significativas aconteceram nessa época nos anos 90.

1. A programação orientada a objetos tornou-se o paradigma de programação mais popular, superando a programação procedural 2. Kent Beck formalizou a Extreme Programming

OOP tornou-se o objetivo de muitos desenvolvedores. Havia muitos benefícios da experiência do desenvolvedor, mas um grande deles era que agora era mais fácil criar *modelos de domínio* que capturavam de forma mais precisa a essência do negócio. Foram introduzidas técnicas que nos permitiram alterar estrategicamente o código quando novos requisitos surgiram.



O “Círculo da Vida” — práticas técnicas envolvidas na implementação de Programação Extrema (XP).

E foi nessa época que Extreme Programming (XP), o **processo Agile mais bem definido e completo** (antes mesmo de Agile existir), foi formalizado por Kent Beck. A XP nasceu na era ponto-com de requisitos que mudam rapidamente.

Ágil (2001 - hoje)

Ágil é entregar valor de forma incremental em vez de tudo de uma vez.



Embora o XP fosse muito promissor, também usamos e contamos com outras metodologias (ou seja, Scrum, o *Modelo de Desenvolvimento de Sistemas Dinâmicos (DSDM)*, *Desenvolvimento de Software Adaptativo*, Crystal, *Desenvolvimento Orientado a Recursos* e *programação pragmática*).

Em fevereiro de 2001, dezessete desenvolvedores influentes, incluindo aqueles que criaram essas metodologias, concordaram em se encontrar em uma estação de esqui para discutir uma melhor entrega de software. O objetivo era ver se eles concordavam em algo a ser escrito.

A lista de desenvolvedores era Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C.

Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland e Dave Thomas.



A reunião da viagem de esqui de 2001 em Utah que deu origem ao Manifesto Ágil e deu início a uma nova era de desenvolvimento de software.

Milagrosamente, ao final da viagem, o grupo havia chegado a um acordo sobre o que seria chamado de **Manifesto Ágil**, formalizando assim o Agile e mudando a trajetória de nossa indústria para sempre.

O Manifesto Ágil

O Manifesto Ágil é o seguinte:

“Indivíduos e interações sobre processos e ferramentas

Software funcional sobre documentação abrangente

Colaboração do cliente em vez de negociação de contratos

Responder à mudança ao invés de seguir um plano

Ou seja, enquanto há valor nos itens da direita, valorizamos mais os itens da esquerda.”

A era (enganada) do Agile

Houve mudanças notáveis na indústria por causa do Agile.

A primeira é **a introdução de Agile Coaches**. As empresas contratavam treinadores ágeis para ir ao local e ajudá-los a realizar *uma transformação ágil*, introduzindo processos e estruturas ágeis como o Scrum. É importante observar que esses treinadores ágeis não fizeram nada para *ajudar os desenvolvedores a mudar a forma como escrevem o código* — eles apenas introduziram o processo.

A segunda mudança é **que os desenvolvedores se tornaram generalistas**. Agile introduziu a necessidade de desenvolvedores completos. Hoje, os desenvolvedores precisam saber como projetar, desenvolver em toda a pilha, configurar integração contínua, implantação, bancos de dados e saber como falar com os clientes e coletar feedback.

Enquanto o Agile prometia a resolução de *problemas de qualidade e produtividade*, muitas empresas mergulharam de cabeça, consertando *seus processos*, mas ainda **se sentiam atormentadas**

com problemas de qualidade e produtividade. E assim vieram os inimigos do Agile, mais marchas da morte e mais projetos fracassados.

Por que o Agile não funcionou?

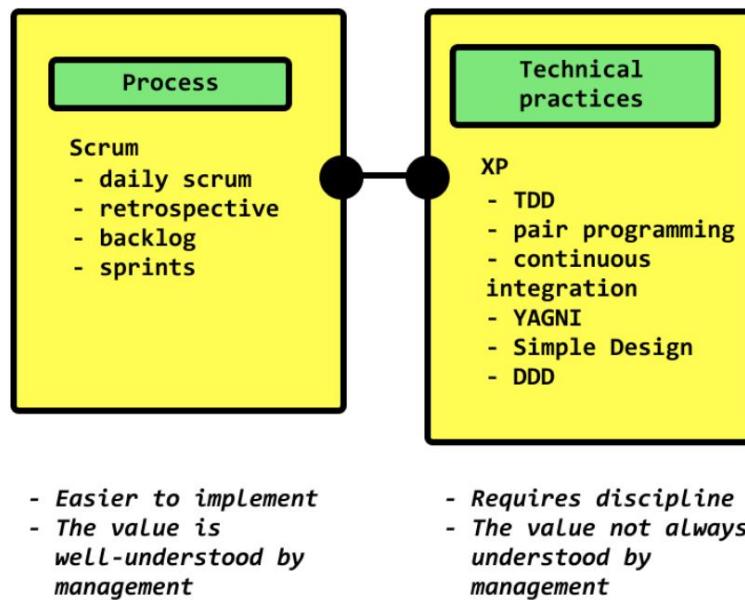
Muitas empresas não perceberam que, para o Agile funcionar, duas coisas são necessárias:

1. processo
2. e práticas técnicas

Se o Agile não funcionou, é porque faltava o número 2 — práticas técnicas. Mas o que realmente faltava era artesanato.

Agile

Both process & technical practices are mandatory to deliver value incrementally instead of all at once



As equipes estavam muito focadas em estruturas orientadas a processos como o Scrum, mas prestavam pouca atenção às práticas técnicas do XP. Não apenas isso, mas os gerentes não técnicos não viam valor nas práticas XP, como programação em pares e desenvolvimento orientado a testes .

“Por que ter dois desenvolvedores trabalhando na mesma coisa quando você poderia fazer mais trabalhando separadamente? ”.

“E você está me dizendo que pode demorar mais para você *inicialmente* se você fizer TDD? Então não faça isso.”

A atitude de cima para baixo, eu sou *mais inteligente do que você, então faça o que eu digo, industrial*, a atitude do trabalhador de fábrica aplicada aos desenvolvedores de software permitiu que líderes não técnicos adiassem com sucesso as coisas nas quais os desenvolvedores veem o valor a longo prazo, mas eles, eles próprios não valorizam tanto quanto *entregam no prazo*.

Para que as empresas migrassem adequadamente para o Agile, elas precisavam se **comprometer** com as melhores práticas técnicas. Precisávamos de Software Craftsmanship: o elo que faltava.

Software Craftsmanship (2006 - hoje)

Houve alguma conversa sobre *Software Craftsmanship* antes. O primeiro livro a sugerir que o desenvolvimento de software pode ser mais um comércio do que uma ciência foi “*The Pragmatic Programmer: From Journeyman to Master*”. Ele fez comparações entre como os desenvolvedores aprendem e ganham antiguidade com o modelo de aprendizado na Europa medieval e foi um dos primeiros textos a tentar destilar algum senso de profissionalismo em relação aos desenvolvedores de software.

Em 2006, Peter McBreen publicou o *livro Software Craftsmanship*, solidificando o nome e explicando muitas das práticas profissionais de hoje. Infelizmente para a comunidade de artesãos nessa época, o Agile estava no centro das atenções. Consideramos o Agile a solução para nossos maiores problemas, dando pouca necessidade para essa coisa chamada *Software Craftsmanship*.

Para trazer o artesanato de volta aos holofotes, em 2008, Uncle Bob propôs um quinto valor para o Manifesto Ágil: “**artesanato acima da execução**”.

Como as coisas não estavam indo do jeito que queríamos com o Agile, e porque nos desviamos dos frameworks técnicos do Agile como o XP, em 2009, um grupo de desenvolvedores pretendia produzir algo a ser escrito que pudesse descrever de forma concisa o movimento artesanal. Por fim, criamos o **Software Craftsmanship Manifesto**. O manifesto, impresso abaixo, pegou os valores do *Manifesto Ágil* e o levou além, promovendo excelência técnica e profissionalismo.

O Manifesto de Artesanato de Software

O manifesto é o seguinte:

"Como aspirantes a artesãos de software, estamos elevando o nível do desenvolvimento de software profissional praticando-o e ajudando outras pessoas a aprender o ofício.

Através deste trabalho passamos a valorizar o seguinte:

- **Não apenas software funcional**, mas também software bem elaborado
- **Não apenas respondendo à mudança**, mas também agregando valor constantemente
- **Não apenas indivíduos e interações**, mas também uma comunidade de profissionais
- **Não apenas colaboração com o cliente**, mas também parcerias produtivas"

De volta ao básico (XP)

Ainda há muito trabalho a ser feito, mas hoje estamos vendo algumas mudanças positivas na indústria de software em torno de Agile e Software Craftsmanship.

A comunidade de artesanato de software está crescendo. Desenvolvedores em todo o mundo estão organizando encontros, assinando [o manifesto original](#), e espalhando a palavra de profissionalismo. Essa comunidade pode muito bem ser aquela que se reúne para popularizar e padronizar a abordagem para resolver esse problema de produtividade e qualidade de 60 anos.

Além disso, em 2019, um tio Bob frustrado publicou um livro intitulado "*Clean Agile: Back to Basics*", que é quase inteiramente um **passo a passo das práticas técnicas do XP**.

Se você me perguntar sobre minhas suposições futuras, eu diria que na próxima década veremos o artesanato e as comunidades ágeis convergirem. É apenas uma questão de tempo até que isso aconteça porque eles compartilham o mesmo objetivo: entregar **software de alta qualidade no prazo**. Posso ver o que será necessário para termos essa transformação ágil completa. Vimos agora que não podemos deixar de lado as práticas técnicas e que precisamos ser mais disciplinados em aderir a essas práticas.

Agora cabe a nós, os artesãos do software, fazer a nossa parte e dar o exemplo.

Artesanato: Profissionalismo no desenvolvimento de software

Definição

Repetiremos a *melhor* definição para artesanato de software.

Artesanato é profissionalismo no desenvolvimento de software.

Software Craftsmanship é uma mentalidade *de profissionalismo*. Não é sobre se você tem uma certificação, se é um desenvolvedor sênior ou se pratica TDD o tempo todo.

Trata-se de **assumir a responsabilidade por** nossas carreiras, clientes e comunidade. Significa também **ter orgulho de** nossas soluções, trabalhar **com pragmatismo** e buscar melhorar cada vez mais nossa reputação com o bom trabalho que fazemos.

Você é um artesão de software?

Se você fizer as coisas mencionadas acima, você já é um artesão de software. Não há mágica nisso. Mesmo que você não goste do rótulo e não queira se chamar de artesão, tudo bem. O importante é que você faça essas coisas.

Entendendo o manifesto

Não apenas software funcional, mas também software bem elaborado

É importante lembrar que um **software funcional** também pode ser um software que:

- difícil de mudar
- difícil de entender lento
-

Mas um software bem elaborado é um software testável, flexível e passível de manutenção. Também é um software que fica melhor com o tempo — crescendo em valor à medida que você dedica mais tempo e cuidado a ele.

Não apenas respondendo à mudança, mas também agregando valor constantemente

Agregar valor não é apenas adicionar novos recursos ou corrigir bugs. Você pode considerar agregar valor para melhorar continuamente a estrutura do código, mantendo-o limpo, testável, flexível e de fácil manutenção. Custa muito dinheiro realizar reescritas e, embora todo aplicativo tenha um tempo de vida, geralmente - se essa for a única opção viável em um aplicativo relativamente jovem, isso significa que falhamos.

Não apenas indivíduos e interações, mas também uma comunidade de profissionais

Não estamos sozinhos nisso. Seus colegas desenvolvedores de software estão passando pelos mesmos desafios que você. Para impulsionar esta indústria, precisamos compartilhar

conhecimento, aprender com os outros, inspirar, orientar e preparar a próxima geração de artesãos.

Como o desenvolvimento de software é muito jovem e ainda não há *um padrão para* muitas coisas, o compartilhamento de conhecimento e a orientação são incrivelmente importantes. Não queremos perder o que descobrimos nos últimos 40 anos. Queremos manter a qualidade alta e ensinar desenvolvedores menos experientes a trabalhar melhor é uma forma de alavancagem que valerá a pena nos próximos anos. Todos nós temos essa responsabilidade.

Não apenas colaboração com o cliente, mas também parcerias produtivas

Não acreditamos em uma relação empregador-empregado. Isso não existe na fabricação de software. Tudo o que está escrito em seu contrato é uma formalidade.

Em vez disso, trate **seu empregador como seu cliente, da mesma forma que faria se fosse apenas um consultor ou um empreiteiro**. Algumas coisas importantes a serem lembradas:

- Seu empregador não é responsável por garantir que você cresça profissionalmente - você é.
- Seu empregador não é responsável por garantir que você receba um orçamento de livro, treinamento ou enviado para conferências e seminários - você é.
- Sua responsabilidade é **tratar seu empregador como um cliente e fornecer a ele um serviço excelente (aconselhamento, consultoria, desenvolvimento) e ajudá-lo a atingir seus objetivos realizando atividades de alto valor, mesmo que às vezes isso não signifique codificação.**

Às vezes, significa dar uma palestra, uma apresentação ou escrever uma postagem no blog.

Infelizmente, nem todas as empresas têm uma estrutura de gerenciamento que permite que você trabalhe dessa maneira, e pode ser difícil rejeitar o que você acredita ser o melhor para o negócio quando a outra parte não o vê como um parceiro.

Princípios de artesanato

Mantendo-nos fiéis ao manifesto, estamos prontos para discutir coisas açãoáveis que você pode fazer para se tornar um artesão de software melhor *hoje*.

Para escrever um software bem elaborado...

Preocupe-se com o que você faz. O mundo depende do trabalho que você e eu fazemos. Existem pelo menos 4 a 10 sistemas de computador diferentes (seus fones de ouvido, telefone, computador, aparelhos) em qualquer sala. Coloque amor e cuidado no que você está fazendo, porque apenas gostar do trabalho que você faz aumenta a probabilidade de fazer um bom trabalho.

Use as melhores práticas técnicas ágeis. Há algumas coisas que todos os desenvolvedores de software profissionais devem saber como fazer e *quando*. TDD, refatoração, programação em pares e design simples são **práticas técnicas ágeis essenciais do XP**. Aprendemos como dominar cada um deles neste livro. Em seguida, melhoramos nossas habilidades de design com princípios de design, padrões e vários estilos arquitetônicos, etc. Também abordaremos cada um deles ao longo do livro.

Esteja sempre se aprimorando. Há sempre espaço para melhorias. Depois de ler este livro, encorajo-o a trabalhar gradualmente com os recursos e leituras que listei em cada capítulo. Aprenda regularmente novas técnicas, fornecedores e troque as ferramentas em sua caixa de ferramentas por outras melhores.

Conheça a sua indústria. Discutimos a breve história do desenvolvimento de software, Agile e o surgimento da habilidade de software. Serei o primeiro a admitir que não tinha ideia do que era o Agile, quais eram os objetivos originais e o que significava para nós ser *Agile* no trabalho. É importante saber por que fazemos o que fazemos. Fique de olho nas tendências emergentes e nas atualizações das bibliotecas, estruturas e linguagens que você usa para desenvolver software.

Aprenda o domínio. Seu trabalho é entender o espaço do problema (o que seu cliente deseja realizar) e criar o espaço da solução (software). Se você conhece o negócio do seu cliente, entenderá melhor os requisitos; isso permite que você faça perguntas melhores, entenda as maneiras prováveis pelas quais o código pode ou não precisar ser alterado e contribua para discussões com especialistas do domínio. Quando você conhece o domínio, o código se torna *uma metáfora para o que acontece na vida real*. No Domain-Driven Design, chamamos isso de *Linguagem Ubíqua*, para usar as mesmas palavras da vida real nos negócios, no código.

Se Agile é sobre fazer melhorias incrementais e ciclos de feedback curtos, podemos manter o código simples e evitar designs excessivos projetando o que acontece na vida real no código.

Simplicidade implacável. Crie exatamente o que é necessário para que o recurso funcione. Não mais. Sinta quando você está divergindo disso, adicionando mais código ou abstrações desnecessárias, e se segure. Se você pratica TDD, quanto mais código você tiver que considerar, mais maneiras ele pode dar errado e mais testes você precisará escrever para garantir que ele se comporte da maneira que deveria. Lembre-se de YAGNI - você não vai precisar.

Prática. Se você não usá-lo, você o perde. À medida que avanço em minha carreira, frequentemente me pego escrevendo mais, falando, orientando e menos codificando. Você precisa manter suas habilidades afiadas. Minha maneira favorita de praticar é trabalhar em projetos paralelos com base em seus interesses. Escolha algo com base em seus interesses, para ficar motivado e motivado a melhorá-lo o máximo possível. Gosto de música, então criei um aplicativo de negociação de vinil. Outra opção é contribuir para o código aberto. Escolha uma biblioteca ou estrutura que você usa com frequência, execute os testes, leia os documentos e dê uma olhada nos problemas em aberto. Essa não é apenas uma ótima maneira de praticar, mas você também pode ver como outras pessoas desenvolvem software no mundo real.

Para agregar valor continuamente...

Aplique a regra do escoteiro. A regra do escoteiro é deixar o acampamento mais limpo do que quando o encontramos. Se você se deparar com algum código confuso, confuso e errado ou com variáveis não utilizadas, reserve um tempo

para refatorá-lo imediatamente. É um excelente exemplo para os outros, promovendo a ideia de que esta base de código é um local de *limpeza e qualidade*.

Esperançosamente, outros irão perceber isso e dar-lhe o respeito que merece também. Alguns chamam isso de “teoria da janela quebrada”.

Refatorar, protegido por testes. Se você for refatorar algo para ficar mais limpo, primeiro você deve protegê-lo com testes. Simplesmente não podemos alterar o código com segurança sem a presença de testes. Outro bom momento para adicionar testes e refatorar é quando encontramos um bug. Para corrigir bugs, escreva o teste que prova a existência do bug e refatore o código para que o teste passe e o bug não exista mais. Com o tempo, você fortalecerá seu aplicativo e dormirá bem sabendo que pode alterar o código com uma quantidade cada vez maior de segurança.

Seja corajoso. Vindo das duas declarações anteriores, quando você vê o código que tem medo de mudar porque tem medo de que ele quebre e tem medo do que pode acontecer com você se o quebrar, este é o começo da podridão do software. Todo mundo também está com medo. Seja o corajoso. Seja corajoso para cercá-lo de testes e refatorá-lo em algo compreensível. Os testes são sua graça salvadora e permitirão que você aja como um profissional.

Aprenda e aplique XP. É a maneira ágil mais extensa e bem pensada de entregar um projeto de software. Aprenda e **desafie a burocracia e a gestão não técnica a** aplicar suas práticas técnicas no seu trabalho do dia-a-dia. É a coisa profissional a fazer. Consulte “Ajude-os a ver o valor das práticas técnicas” abaixo para saber como vendê-lo.

Encantar os clientes, ajudando-os a alcançar o que desejam. Seu objetivo é ganhar dinheiro, economizar dinheiro e preservar os fluxos de receita. Seu objetivo é ajudá-los a fazer exatamente isso, pegando seus requisitos difusos, aplicando as melhores práticas ágeis (XP) e transformando-as *na coisa certa*.

Envolva-se na comunidade...

Aprenda com os outros. Há tanta coisa que você pode aprender sozinho. Leia blogs de desenvolvedores, livros, assista a cursos, vídeos do YouTube e faça perguntas.

Faça programação em par. Alguns desenvolvedores temem a programação em par porque o outro desenvolvedor pode perceber que cometeu um erro. O melhor conselho é superar esse medo. Expor-se a como outra pessoa pensa e trabalha pode ajudá-lo a refinar ou reconstruir *sua maneira de trabalhar*.

Mentor de desenvolvedores menos experientes. Infelizmente, não temos mais aprendizados de software. Quando você se forma na faculdade, universidade ou campo de treinamento, você é um aprendiz. Você tem pouca experiência em entregar software de qualidade no prazo. Você pode ter passado algum tempo aprendendo uma biblioteca ou estrutura JavaScript front-end, mas as práticas profissionais (TDD, BDD, DDD, etc) estão faltando. Os artesãos observam outros ofícios, como encanamento, e arranjam tempo para estimular os recém-chegados à maneira profissional de fornecer produtos de software. Na Amazon, os PRs dos desenvolvedores juniores de software estão sujeitos a muitos escrutínios, mas os engenheiros mais experientes deixam comentários e feedback incrivelmente úteis (às vezes mais de 100 comentários em um PR). Eventualmente, os desenvolvedores juniores chegam ao ponto em que seus PRs exigem muito menos feedback e seu código é mesclado em 1 ou 2 revisões em vez de 10.

Compartilhe o que você sabe. Ensinar é aprender. Aprendi mais do que pensei ser possível compartilhando o que já sabia na internet. Se você estiver errado, alguém acabará por corrigi-lo. Isso é importante como indústria porque precisamos preservar o que aprendemos ao longo do tempo para que outros possam evitar os mesmos erros. Portanto, escreva postagens de blog, codifique, compartilhe suas vitórias, falhas, ideias, dicas - isso ajuda os desenvolvedores em diferentes estágios de sua jornada.

Socialize com os outros. Fiquei surpreso por poder pedir o conselho de Vaughn Vernon no Twitter e realmente obtê-lo. Também fiquei surpreso por poder fazer pesquisas, perguntando o que os desenvolvedores acham que significa “código limpo” e receber um fluxo de opiniões para informar melhor o meu. O Twitter pode ser um lugar um pouco estranho às vezes, mas como desenvolvedor, há muito valor em estar na plataforma. Você pode se *relacionar* com outras pessoas interagindo com seu conteúdo e ideias e tendo discussões técnicas. Offline, confira encontros e conheça outros desenvolvedores usando ferramentas, tecnologias e abordagens de seu interesse.

Considere-se um parceiro...

Assuma a responsabilidade pelo seu próprio aprendizado. Seu empregador não é responsável por sua educação. Imagine seu médico dizendo: "sim, posso ajudar a curá-lo, mas vou precisar que você pague para eu ler esses livros também". Novamente, este é um lembrete de que o trabalho que fazemos é uma troca e cabe a você arranjar tempo para melhorar.

Assuma a responsabilidade pelos sucessos/fracassos. É chato quando as coisas dão errado, mas é nobre assumir a responsabilidade e agir para consertar as situações. Assumir a responsabilidade por suas deficiências é uma excelente maneira de preservar seus relacionamentos e reputação. Você vai estragar tudo. Confie em mim. E a reputação é importante, para que você possa conseguir melhores empregos e oportunidades. Veja desta forma, quando as coisas *vão bem*, você também pode receber elogios por isso, e isso é bom.

Ajude-os a ver o valor das práticas técnicas. Como você faz seu chefe ver o valor em TDD ou refatoração? Não promova as *práticas técnicas*; promova o valor em seu lugar. As pessoas não mudarão de ideia a menos que possam ver o valor. Então, qual é o valor do TDD?

Bem, você poderia dizer ao seu chefe, que é incrivelmente agressivo em tentar fazer com que você termine o projeto no menor tempo possível, que o TDD reduz o tempo necessário para testar o sistema, resulta em menos bugs, o que significa menos horas de codificação . E isso significa que *eles economizam dinheiro*. Você também pode dizer a eles que poderíamos lançar software de maneira muito mais confiável. Eles podem perguntar: "qual é o custo disso?". E essa é uma boa pergunta a se fazer, porque tudo tem um custo. O *custo do TDD* é que todos precisam saber como fazê-lo, e pode não ser o caso. Você pode se oferecer para aprendê-lo e dedicar uma hora para ensinar sua equipe a fazer algo que os poupará centenas de horas corrigindo bugs.

Não se sobrecarregue. A maioria das pessoas não faz um bom trabalho quando está apressada. A maioria das pessoas também não faz um bom trabalho quando está cansada. Pessoalmente, assim que começo a sentir fadiga cerebral, afasto-me do computador e vou dar uma caminhada, fazer exercícios ou encerrar o dia. Por que? Porque eu sei que qualquer código que sair de minhas mãos a seguir provavelmente será um passivo, em vez

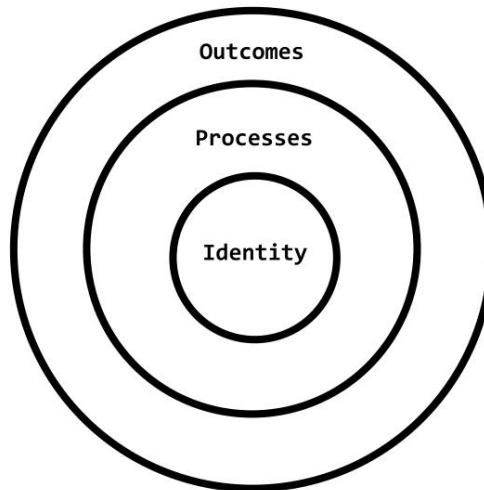
do que um ativo. Você é apenas humano. Minha dica? Faça o trabalho mais importante sempre que se sentir mais alerta durante o dia. Para mim, isso é logo de manhã. Pode ser diferente para você embora.

Forneça valor, mesmo quando não estiver codificando. Esperamos que os desenvolvedores em uma era ágil de desenvolvimento de software ajam muito mais como generalistas. Portanto, procure fornecer o maior valor possível, mesmo que não envolva codificação. Por exemplo, você pode ver uma oportunidade de melhorar um processo, lance isso. Questione os requisitos e certifique-se de que estamos construindo o que precisamos. Gaste tempo entendendo o negócio e como ele ganha dinheiro. Ajude os líderes a priorizar o trabalho mais crítico. Dê *-lhes opções*. Esperançosamente, você pode ver como essas coisas são valiosas.

Como criar hábitos a partir dos princípios do artesanato

As pessoas têm objetivos elevados e coisas que adorariam transformar em hábitos o tempo todo. Por exemplo, eu adoraria ler um novo livro todo mês, fazer exercícios e beber 8 copos de água todos os dias. O que está por trás de tornar isso uma realidade à qual me atenho - não apenas por algumas semanas, mas por toda a vida?

Na pesquisa de James Clear sobre hábitos em *Atomic Habits*, ele argumenta que a principal maneira de construir hábitos não é *apenas tentar muito*, mas sim começar **identificando quem queremos ser**.

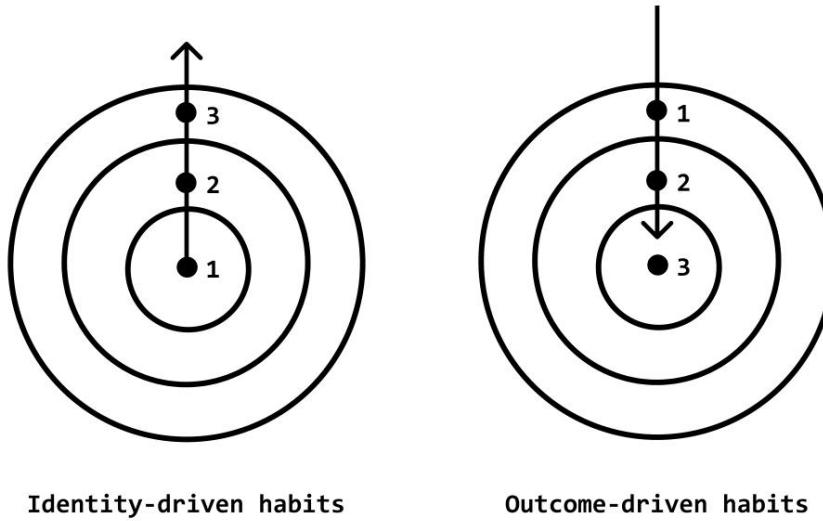


A maioria das pessoas quer **atingir uma meta** ou criar um conjunto de **resultados**, então elas compõe um conjunto de **hábitos** que seguirão **até** atingirem seu(s) objetivo(s). Somente depois que as pessoas atingem seus objetivos, elas decidem que **são a pessoa que desejam ser**.

por exemplo: se eu escrevo um código testável, flexível e sustentável, então sou um bom desenvolvedor de software e sou um artesão de software

Isso é o que ele chama de **hábitos orientados a resultados**. O problema dos hábitos orientados para resultados é que só nos sentimos motivados a continuar realizando os hábitos se sentirmos que estamos chegando mais perto de nosso objetivo. Isso significa que qualquer pequena falha, um passo para trás em vez de um passo para a frente, tem muito poder em nossas vidas. Tem o poder de destruir todo o nosso sistema. É por isso que as pessoas às vezes desistem de suas rotinas de dieta e perda de peso quando percebem que estabilizaram ou ganharam peso novamente.

Em vez disso, o que Clear recomenda é a **abordagem de hábitos baseada em identidade**.



A abordagem baseada em identidade para hábitos é:

1. Decidir quem queremos ser
2. Provar que somos aquela pessoa com pequenas vitórias, repetidamente

De acordo com a pesquisa de ciência comportamental e psicologia de Clear, **a verdadeira mudança de comportamento vem da mudança de identidade**. Se quisermos escrever um livro, trabalhemos de trás para frente, vendo a nós mesmos como escritores. A partir daí, nos perguntamos “quais são os tipos de valores e princípios que os escritores têm”? Pensamento claro, cumprimento de prazos, etc. E que tipo de hábitos eles têm? Eles provavelmente acordam e fazem um diário todas as manhãs, certo?

Quando fizermos este exercício e começarmos a *acreditar* que somos a pessoa que queremos ser, mais fácil será manter os hábitos.

Vamos trabalhar para trás agora. Qual é o nosso objetivo? Para escrever código testável, flexível e sustentável? Quem é o tipo de pessoa que faz isso? Isso é um artesão de software. E o que eles têm o hábito de fazer?

ex.: Sou um artesão de software. Os artesãos de software escrevem código testável, flexível e sustentável por **[INSERT SOFTWARE CRAFTSMANSHIP PRINCIPLE HERE]**.

ex.: Sou um artesão de software. Os artesãos de software escrevem código testável, flexível e sustentável porque “**se preocupam com o que fazem**”.

ex.: Sou um artesão de software. Os artesãos de software escrevem código testável, flexível e sustentável “**usando as melhores práticas técnicas ágeis**”.

Quanto mais orgulhoso você estiver de sua identidade, mais fácil será para você manter os hábitos associados a ela. Portanto, se você tem orgulho de ser um artesão de software - o tipo de pessoa que sai de sua mesa sentindo que fez um ótimo trabalho, será muito mais fácil para você dominar técnicas desafiadoras como TDD e Domain Driven Design.

Resumo

- Até agora, o maior problema que enfrentamos como indústria é não saber o que devemos fazer, é ter disciplina para fazê-lo.
- Artesanato de software é sobre profissionalismo no desenvolvimento de software.
- Os artesãos de software se preocupam com o software bem elaborado, agregando valor constantemente, sua comunidade de profissionais e construindo parcerias produtivas com os clientes.
- Para transformar princípios artesanais em hábitos, você já deve acreditar que é um artesão de software. Concentre-se em colocar as repetições e os resultados virão.

exercícios

Em breve

Referências

livros

- Padrões de aprendizado: orientação para o aspirante a artesão de software por Dave Hoover
- Artesanato de software: o novo imperativo por Pete McBreen O codificador limpo por Robert C. Martin O programador pragmático por Andrew Hunt e David Thomas O artesão de software: profissionalismo, pragmatismo, orgulho por Sandro Mancuso Hábitos atômicos de James
- Clear

3. Uma visão de 5000 pés de design de software

Quanto mais *incógnitas desconhecidas*, maior a chance de complexidade acidental. Aqui, esmagamos muitas incógnitas desconhecidas no mundo do design de software por meio do conceito de *camadas de design* e uma visão de 5000 pés de como tudo está conectado. O objetivo deste capítulo é obter uma compreensão de alto nível do que precisamos saber para dominar o design de software.

Objetivos do capítulo

- Entenda a diferença entre design e arquitetura de software
- Entenda o panorama geral do design de software e como os vários níveis de design se encaixam uns nos outros

Níveis de design

A palavra *arquitetura* carrega consigo uma conotação de estar relacionada a coisas que são “grandes” e de “alto nível”.

A palavra também muitas vezes chama a atenção para paralelos entre a construção de edifícios, carros ou outras coisas vitais e caras para mudar.

No entanto, os profissionais responsáveis pelos projetos **de alto nível** dessas criações, como casas, aviões ou Teslas, também entendem os detalhes de implementação (as coisas **de baixo nível**) que precisarão ser realizadas para cumprir o projeto.

O que estou tentando dizer é que você **não pode ser um designer de alto nível** (arquiteto de software), sem conhecimento dos detalhes **de baixo nível** (escrever código limpo, usar um paradigma de programação de forma eficaz, aderir aos princípios de design). O diabo está nos detalhes.

Você pode, no entanto, ser um codificador **de baixo nível**, despejar o código em um produto para fazer o próximo recurso funcionar, tudo sem respeitar o design **de alto nível** e, possivelmente, escapar impune algumas vezes.

Mas lembre-se do objetivo nº 2? Ser capaz de satisfazer *consistentemente as necessidades dos usuários?*

Quando os **detalhes de baixo nível** vão contra a **política de alto nível**, é apenas uma questão de tempo até que tenhamos um sistema legado que não seja mais fácil (ou valioso) de manter em nossas mãos.

Ambos os níveis de design de software (alto e baixo) são essenciais. Eles formam um relacionamento simbiótico entre si que, quando sincronizados, pode levar a um software de alta qualidade fácil de manter e alterar.

Isso significa que **todos na equipe** têm a responsabilidade compartilhada de entender a arquitetura de alto nível e como os detalhes de baixo nível a suportam.

O Design e Arquitetura de Software Pilha e Roteiro

Vamos tomar um segundo para recapitular o que acabamos de descobrir. Faremos isso regularmente ao longo do livro:

- Os **objetivos do software** são:
 - Objetivo nº 1: Satisfazer as necessidades dos usuários enquanto minimiza o esforço necessário para fazê-lo.
 - Objetivo nº 2: Cumprir consistentemente o Objetivo nº 1 à medida que os requisitos mudam.
- **Arquitetura** é identificar os **atributos de qualidade do sistema** que aumentarão nossas chances de realizar com sucesso a Meta nº 1 e nossa escolha do **padrão de arquitetura** que acomodará melhor o projeto (podemos nos aprofundar muito aqui).
- Por fim, **o design de software** não é muito diferente da arquitetura, além do fato de terem diferentes **níveis de design** em que aparecem.

Ótimo, é um bom começo. Definimos pelo que estamos lutando e entendemos em alto nível como chegar lá.

Vamos continuar a descida para chegar ao fundo do que está envolvido em projetar um bom software.

Gostaria de apresentar a vocês **dois artefatos** nos quais passei muito tempo pensando para visualizar o escopo e a amplitude do design e da arquitetura de software.

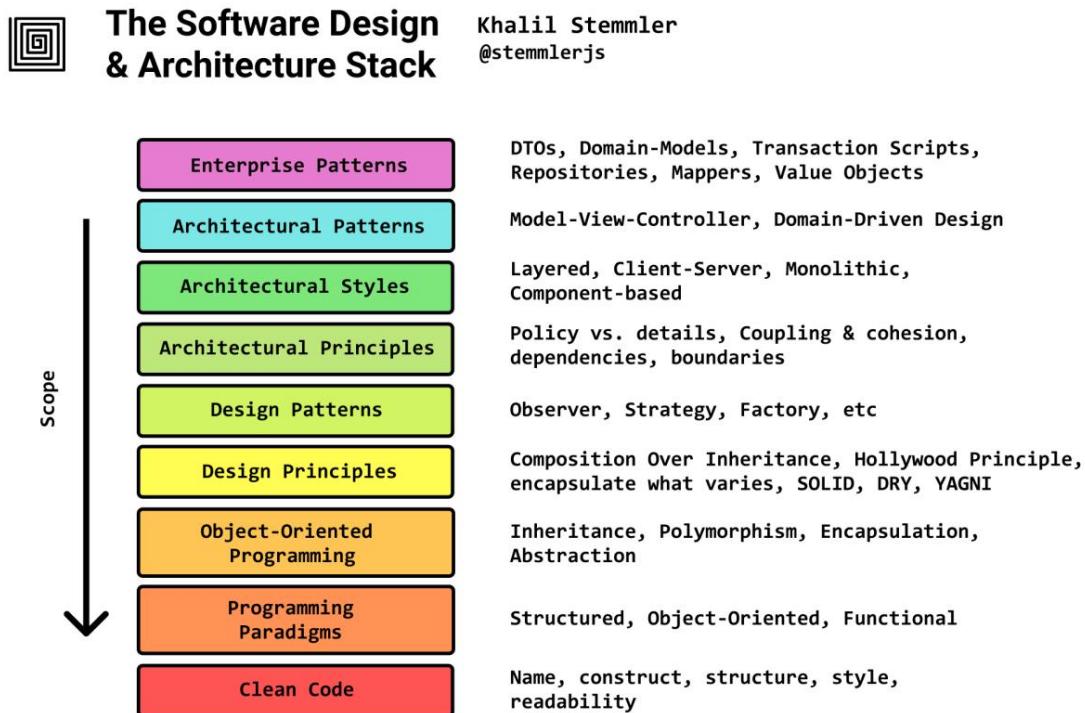
Eles são a **pilha** e o **roteiro**.

Recurso: A Pilha

A pilha de design e arquitetura de software

A *pilha*. Ele descreve o escopo do aprendizado desde os detalhes mais íntimos do padrão corporativo que você escolheu até a maneira como você escreve *código limpo*.

O conhecimento necessário para chegar ao topo da pilha está em camadas. Da mesma forma que o modelo OSI em rede, cada camada da pilha é construída sobre a base da anterior.



A pilha de design e arquitetura de software descreve as camadas de design e arquitetura de software.

Na pilha de gráficos, incluí exemplos de *alguns* dos conceitos mais importantes em cada camada respectiva. Como existem muitos conceitos em cada camada, não incluí todos eles.

Recurso: O Mapa

O roteiro de design e arquitetura de software

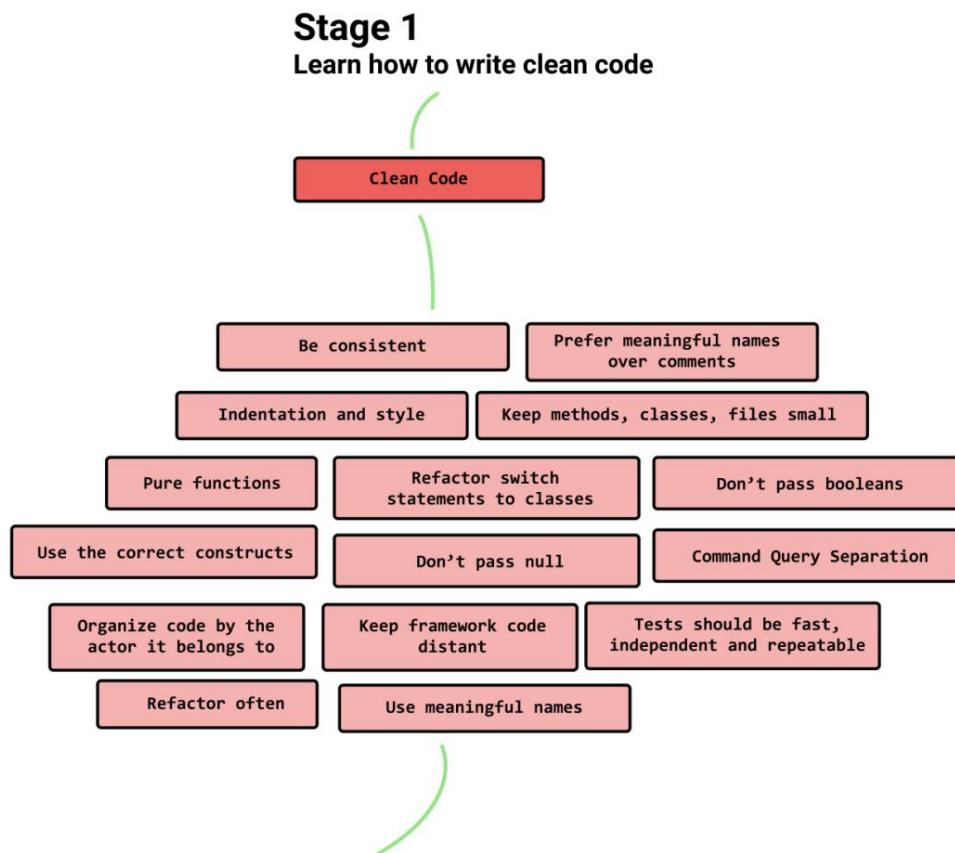
Confira **o mapa**. Embora eu ache que a pilha é boa para ver a imagem maior do que temos que cobrir de relance, *o mapa* é um pouco mais detalhado e, como resultado, acho que é mais útil.

Do código limpo aos conceitos de Domain-Driven Design, o mapa é indicativo do caminho que vamos percorrer neste livro para que você se familiarize com o mundo do design e arquitetura de software.

Você pode visualizar o mapa inteiro (é bastante grande e não está incluído aqui, apenas para leitores EPUB e PDF) [aqui através deste link](#).

Passo 1: Código Limpo

Objetivo: aprender a escrever código limpo.



O primeiro passo para criar um software duradouro é descobrir como escrever um código limpo.

Se você perguntar a alguém o que eles acham que constitui *um código limpo*, provavelmente obterá uma resposta diferente a cada vez. Muitas vezes, você ouvirá que *código limpo* é um código fácil de entender e alterar. No nível inferior, isso se manifesta em algumas opções de design, como:

- ser consistente
- preferindo nomes de variáveis, métodos e classes significativos em vez de escrever comentários
- garantindo que o código seja recuado e espaçado corretamente
- garantindo que todos os testes possam ser executados escrevendo funções puras sem efeitos colaterais não passando nulo

Podem parecer coisas pequenas, mas pense nisso como um jogo de Jenga. Para manter a estrutura do nosso projeto estável ao longo do tempo, coisas como indentação, classes e métodos pequenos e nomes significativos compensam muito a longo prazo.

Se você me perguntar, esse aspecto *do código limpo* é ter boas convenções de codificação e segui-las.

Acredito que esse seja apenas *um aspecto* de escrever *código limpo*.

Minha explicação definitiva sobre código limpo consiste em:

- Sua mentalidade de desenvolvedor (empatia, habilidade, mentalidade de crescimento, pensamento de
- design) & Suas convenções de codificação (nomear coisas, refatoração, teste etc.)
- Suas habilidades e conhecimento (de padrões, princípios e como evitar cheiros de código e antipadrões)

Muito do que torna o software excelente acontece antes mesmo de tocarmos no teclado.

Um requisito é que você se preocupe o suficiente para aprender sobre o negócio no qual está escrevendo o código. Se não nos importamos com o domínio o suficiente para entendê-lo, como podemos ter certeza de que estamos usando bons nomes para representar conceitos de domínio? Como podemos ter certeza de que capturamos com precisão os requisitos funcionais?

Se não nos importamos com o código que estamos escrevendo, é muito menos provável que implementemos convenções de codificação essenciais, tenhamos discussões significativas e peçamos feedback sobre nossas soluções.

Muitas vezes pensamos que o código foi escrito apenas para atender às necessidades do *usuário final*, mas nos esquecemos das outras pessoas para quem escrevemos código: nós, nossos colegas de equipe e os futuros mantenedores do projeto. Ter uma compreensão dos princípios *do design* e de como a **psicologia humana decide o que é um design bom e ruim** nos ajudará a escrever um código melhor.

Então, essencialmente, a melhor palavra que descreve esta etapa da sua jornada? Empatia.

Assim que entendermos isso, aprenda os *truques do comércio* e continue a aprimorá-los ao longo do tempo, aprimorando seu conhecimento dos padrões e princípios essenciais de desenvolvimento de software.

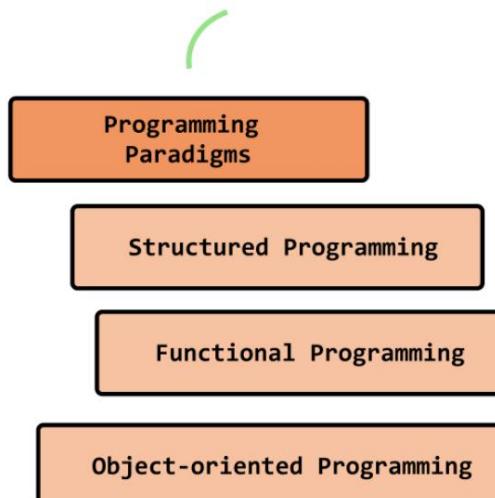
Na [Parte II: Humanos e Código](#), discutimos o que é código limpo e como escrever um código limpo.

Passo 2: Paradigmas de programação

Objetivo: entender as diferenças entre cada paradigma de programação convencional, o que cada um traz de maneira exclusiva e quando usá-los.

Stage 2

Understand the differences between each mainstream programming paradigm, what each uniquely brings to the table, and when to use them.



Agora que estamos escrevendo um código legível que é fácil de manter, seria uma boa ideia realmente entender os 3 paradigmas de programação dominantes e a maneira como eles influenciam como escrevemos o código.

No livro de Uncle Bob, *Clean Architecture*, ele chama a atenção para o fato de que:

- **A Programação Orientada a Objetos** é a ferramenta mais adequada para definir como cruzamos os limites da arquitetura com polimorfismo e plugins
- **A programação funcional** é a ferramenta que usamos para enviar dados para as bordas de nossos aplicativos e lidar elegantemente com o fluxo do programa
- e a **programação estruturada** é a ferramenta que usamos para compor algoritmos

Isso implica que o software robusto usa um híbrido de todos os 3 estilos de paradigmas de programação em momentos diferentes.

Embora você possa adotar uma abordagem estritamente funcional ou estritamente orientada a objetos para escrever código em um projeto, entender onde cada um se destaca melhorará a qualidade de seus projetos.

É um daqueles cenários em que:

...se tudo que você tem é um martelo, tudo parece um prego.

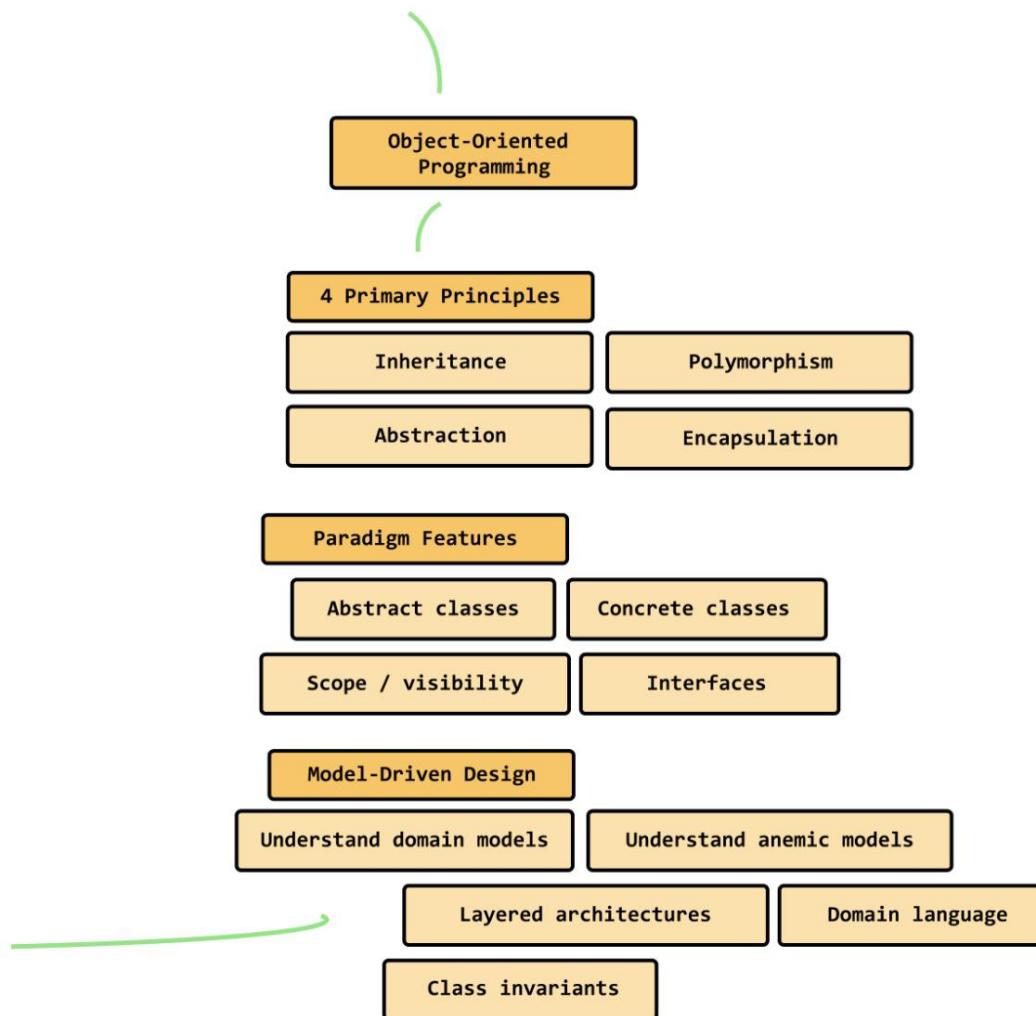
Em [23. Paradigmas de Programação](#), discutimos essas declarações bastante ousadas sobre paradigmas de programação.

Passo 3: Programação Orientada a Objetos e Modelagem de Domínio

Objetivo: Reaprender programação orientada a objetos, mas desta vez, com o design orientado a modelo em mente.

Stage 3

Re-learn object-oriented programming but this time with model-driven design in mind



Em um livro sobre design e arquitetura de software, a Programação Orientada a Objetos será muito apreciada porque é a **ferramenta certa para a arquitetura**.

A programação orientada a objetos não apenas nos permite criar uma **arquitetura de plug-in** e criar flexibilidade em nossos projetos, mas a OOP vem com os 4 princípios da OOP (encapsulamento, herança, polimorfismo e abstração) que nos ajudam a criar **modelos de domínio ricos**.

A maioria dos desenvolvedores que aprende Programação Orientada a Objetos nunca chega a esta parte: aprender como criar uma *implementação de software do domínio do problema* e permitir que ele viva no centro de um aplicativo da Web **em camadas**.

A popularidade da programação funcional parece estar crescendo recentemente, e espero que tenha muito a ver com o React e o ecossistema JavaScript, mas não seja tão rápido em descartar OOP, design orientado a modelo e design orientado a domínio.

Na [Parte V: Design Orientado a Objetos \(Com Testes\)](#), gastamos algum tempo para entender o quadro geral de como os modeladores de objetos encapsulam *todo um negócio e seus processos* em um modelo de domínio de dependência zero.

Por que isso é um grande negócio?

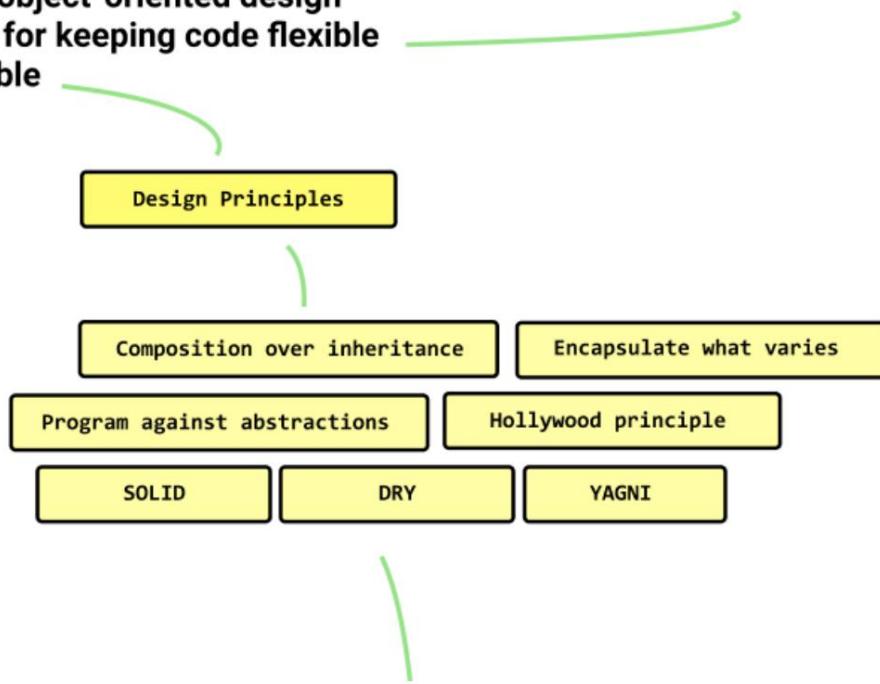
Porque se podemos criar um modelo mental de um negócio, podemos criar a implementação de software do negócio.

Passo 4: Princípios de Design

Objetivo: aprender os princípios de design orientado a objetos para manter o código flexível, testável e de fácil manutenção.

Stage 4

Learn the object-oriented design principles for keeping code flexible and testable



A Programação Orientada a Objetos é benéfica para encapsular modelos de domínio ricos e resolver o terceiro tipo de “Problemas de Software Difícil” – [Domínios Complexos](#), mas pode introduzir alguns desafios de projeto.

Quando devo usar extensões e herança?

Quando devo usar uma interface?

Quando devo usar uma classe abstrata ?

Princípios de design são práticas recomendadas de orientação a objetos bem estabelecidas e testadas em batalha que podemos usar como proteção.

Exemplos de princípios de design comuns com os quais nos familiarizaremos são:

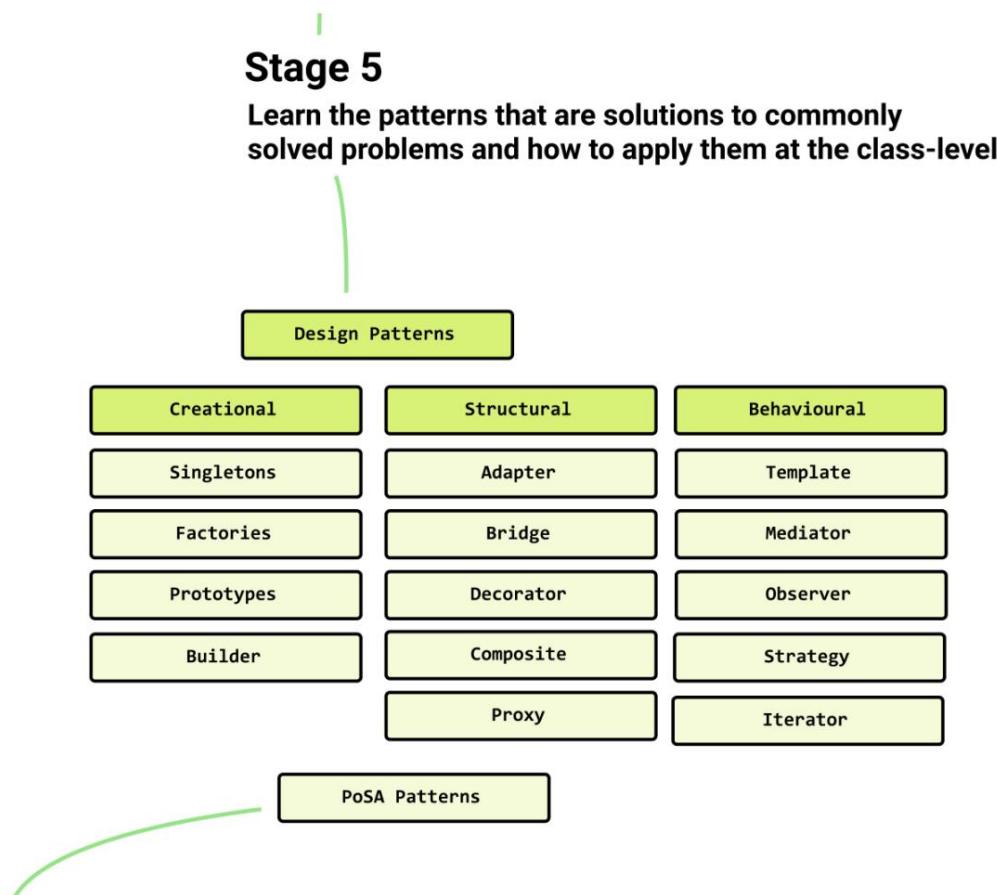
- Composição sobre herança
- Encapsule o que varia
- Programe contra abstrações, não concreções

- O princípio de Hollywood: “Não nos ligue, nós ligaremos para você.”
- Os princípios SOLID, especialmente o princípio da responsabilidade única DRY (não se repita)
- [YAGNI \(você não vai precisar\)](#)

Estes são apenas alguns dos muitos princípios de projeto OO que podem nos ajudar a melhorar nossos projetos. Nós os discutimos em detalhes na Parte VI: Princípios de Design.

Passo 5: Padrões de Projeto

Objetivo: Aprender os padrões que são soluções para problemas comumente resolvidos e como aplicá-los em nível de classe.



Versões genéricas dos problemas mais comuns no desenvolvimento de software já foram categorizadas e resolvidas. Nós os chamamos

padrões. *Padrões de design*, na verdade.

Existem 3 categorias de padrões de design: **criacional**, **estrutural** e **comportamental**.

Padrões criacionais são padrões que controlam como os objetos são criados.

Exemplos de padrões de criação incluem:

- O **padrão Singleton** para garantir que apenas uma única instância de uma classe possa existir
- O **padrão Abstract Factory**, para criar uma instância de várias famílias de classes
- O **padrão Prototype**, para começar com uma instância que é clonada de uma existente

Padrões estruturais são padrões que simplificam a forma como definimos as relações entre os componentes. Exemplos de padrões de projeto estrutural incluem:

- O **padrão Adapter**, para criar uma interface para permitir que classes que geralmente não podem trabalhar juntas, trabalhem juntas.
- O **padrão Bridge**, para dividir uma classe que na verdade deveria ser uma ou mais, em um conjunto de classes que pertencem a uma hierarquia, permitindo que as implementações sejam desenvolvidas independentemente umas das outras.
- O **padrão Decorator**, para adicionar responsabilidades aos objetos dinamicamente.

Padrões comportamentais são padrões comuns para facilitar a comunicação elegante entre objetos. Exemplos de padrões comportamentais são:

- O **padrão Template**, para adiar as etapas exatas de um algoritmo para uma subclasse.
- O **padrão Mediator**, para definir os canais de comunicação exatos permitidos entre as classes.
- O **padrão Observer**, para permitir que as classes assinem algo de interesse e sejam notificadas quando ocorrer uma alteração.

Críticas aos padrões de design

Os padrões de design são ótimos e tudo mais, mas às vezes eles podem adicionar complexidade adicional aos nossos designs. É essencial lembrar de YAGNI e tentar manter nossos designs o mais simples possível. Só use padrões de projeto quando tiver certeza de que precisa deles. Você saberá quando quiser.

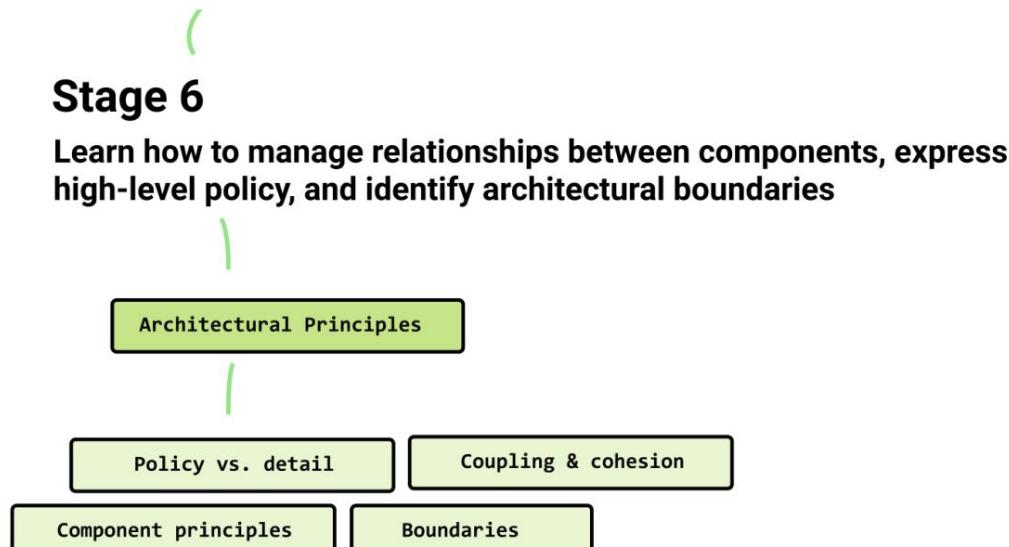
Se soubermos o que é cada um desses padrões, quando usá-los e quando nem *mesmo nos incomodar* em usá-los, estaremos em boas condições para começar a entender como arquitetar sistemas maiores.

A razão por trás disso é porque **os padrões de arquitetura** ([10. Padrões de arquitetura](#)) **são apenas padrões de design ampliados em escala para o alto nível**, onde os padrões de design são implementações de baixo nível (mais próximas de classes e funções).

Discutimos os padrões de projeto na [Parte VII: Padrões de Projeto](#).

Passo 6: Princípios de Arquitetura

Objetivo: aprender como gerenciar relacionamentos entre componentes, expressar políticas de alto nível e identificar limites arquitetônicos.



Agora estamos em um nível mais alto de pensamento logo acima do nível da classe.

Neste ponto de nossa jornada, entendemos que as relações entre os componentes terão um impacto significativo na capacidade de manutenção, flexibilidade e testabilidade de nosso projeto.

Em [8. Princípios de Arquitetura](#), abordaremos os princípios orientadores que nos ajudam:

- Melhorar a flexibilidade em nossa base de código para poder reagir a novos recursos e requisitos
- preocupações separadas
- Melhore a legibilidade e a capacidade de verificação organizando nosso código em módulos coesos ditados pelos casos de uso de nosso aplicativo

Aqui está um vislumbre do que estamos interessados em aprender:

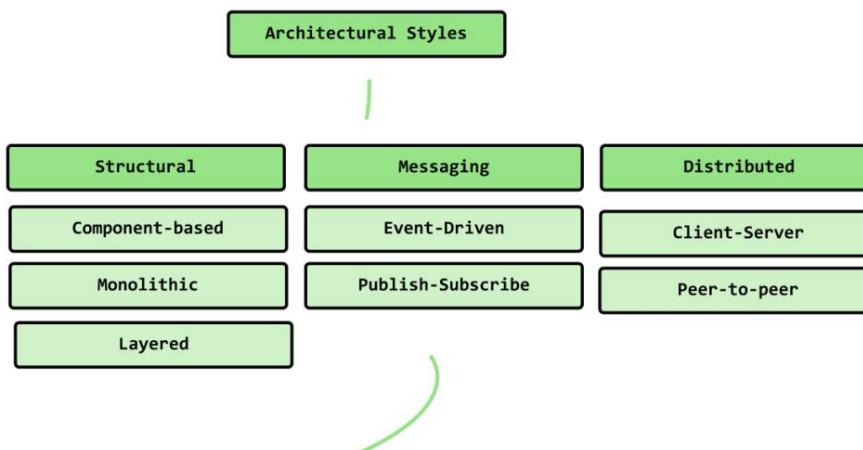
- **Princípios de design de componentes:** [Princípio de Abstração Estável](#), [Princípio de Dependência Estável](#) e [Princípio de Dependência Acíclica](#), para saber como organizar componentes, suas dependências, quando acoplá-los e as implicações de criar acidentalmente ciclos de dependência e depender de componentes instáveis.
- [Política vs. Detalhe](#), para entender como separar as regras do seu aplicativo dos detalhes de implementação.
- **Limites** e como identificar os [subdomínios](#) que os [recursos do seu aplicativo](#) pertencem.

Passo 7: Estilos Arquitetônicos

Objetivo: Aprender as diferentes abordagens para organizar nosso código em módulos de alto nível e definir as relações entre eles.

Stage 7

Learn the different approaches to organizing our code into high-level modules and defining the relationships between them.



Atributos de qualidade do sistema (SQAs) são métricas que precisamos proteger para aumentar as probabilidades de sucesso da arquitetura de nosso aplicativo.

Estilos arquitetônicos são agrupamentos de todos os diferentes tipos de arquiteturas que você pode empregar. Cada um desses estilos tem efeitos positivos exclusivos na manutenção da saúde de um ou mais SQAs.

Por exemplo, um sistema com muita **complexidade de lógica de negócios** se beneficiaria do uso de uma **arquitetura em camadas** para encapsular essa complexidade.

Um sistema como o Uber precisa ser capaz de lidar com muitos **eventos em tempo real** ao mesmo tempo e atualizar as localizações dos motoristas, portanto, a arquitetura **de publicação-assinatura** ou de estilo **orientado a eventos** pode ser mais eficaz.

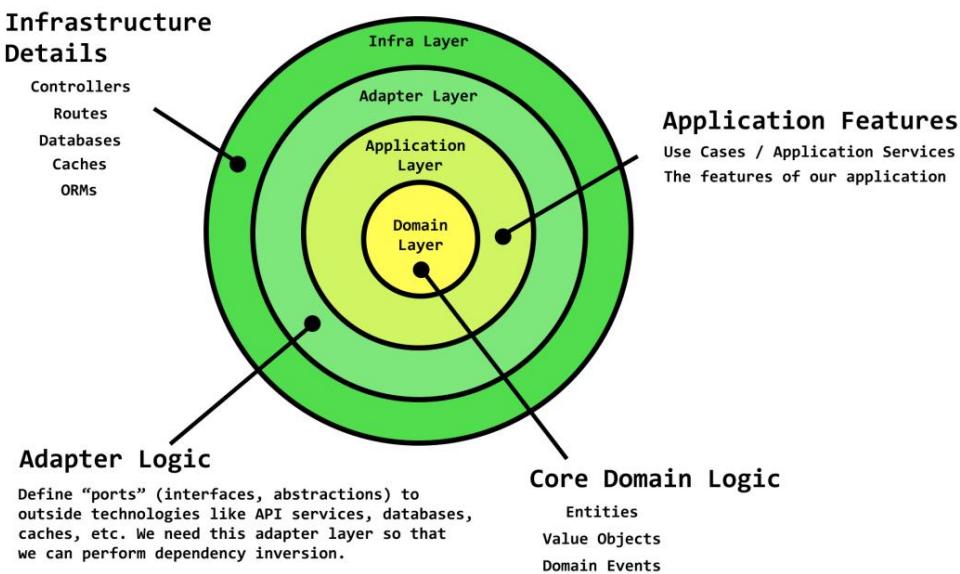
Vou me repetir aqui porque é importante observar que as 3 categorias de estilos arquitetônicos são semelhantes aos 3 grupos de padrões de projeto porque os **estilos arquitetônicos são apenas padrões de projeto de alto nível**.

estrutural

Projetos com *vários níveis* de componentes e ampla funcionalidade geralmente procuram - **flexibilidade** como um SQA. Os estilos arquitetônicos estruturais tornam mais fácil estender e separar as preocupações de sistemas complexos.

Aqui estão alguns exemplos:

- As arquiteturas **baseadas em componentes** enfatizam a *separação de preocupações* entre os *componentes individuais* dentro de um sistema. Pense **no Google** por um segundo. Considere quantos aplicativos eles têm em sua empresa (Google Docs, Google Drive, Google Maps etc.). Para plataformas com muitas funcionalidades, as arquiteturas baseadas em componentes dividem as preocupações em componentes independentes fracamente acoplados. Esta é uma separação *horizontal*.
- **Monolítico** significa que o aplicativo é combinado em uma única plataforma ou programa, implantado em conjunto. *Observação: você pode ter uma arquitetura monolítica E baseada em componentes se separar seus aplicativos adequadamente e, ainda assim, implantar tudo como uma única peça.*
- As arquiteturas **em camadas** separam as preocupações dividindo o software em camadas de infraestrutura, aplicativo e domínio. Esta é uma separação *vertical*.



Um exemplo de reduzir as preocupações de um aplicativo *verticalmente* usando uma arquitetura em camadas.

baseado em mensagem

O envio de mensagens pode ser um componente crucial para o sucesso do sistema.

As arquiteturas baseadas em mensagens são construídas sobre princípios de programação funcional e padrões de design comportamentais, como o padrão observador.

Aqui estão alguns exemplos de estilos arquitetônicos baseados em mensagens:

- As arquiteturas **orientadas a eventos** visualizam todas as mudanças significativas no estado como eventos. Por exemplo, em um [aplicativo de negociação de vinil](#), o estado de uma Oferta pode mudar de “pendente” para “aceita” quando ambas as partes concordam com a negociação. Comandos e eventos tornam-se os principais mecanismos para invocar e reagir a mudanças no sistema.
- As arquiteturas **Publish-Subscribe** fazem uso intenso do padrão de projeto Observer, permitindo que os assinantes ouçam algo de interesse (uma sala de bate-papo ou fluxo de eventos) e publiquem eventos para todos os assinantes apropriados. Um assinante pode ser algo dentro do próprio sistema, usuários finais/ clientes e outros sistemas e componentes.

distribuído

Uma arquitetura distribuída significa simplesmente que os componentes do sistema são implantados separadamente e operam se comunicando por meio de um protocolo de rede. Os sistemas distribuídos podem ser úteis para dimensionar a taxa de transferência, dimensionar equipes e delegar (tarefas potencialmente caras ou) responsabilidade a outros componentes.

Alguns exemplos de estilos arquitetônicos distribuídos são:

- **Arquitetura cliente-servidor.** Uma das arquiteturas mais comuns, onde dividimos o trabalho a ser feito entre o cliente (apresentação) e o servidor (lógica de negócio).
- As arquiteturas **peer-to-peer** distribuem tarefas da camada de aplicação entre participantes igualmente privilegiados, formando uma rede peer-to-peer.

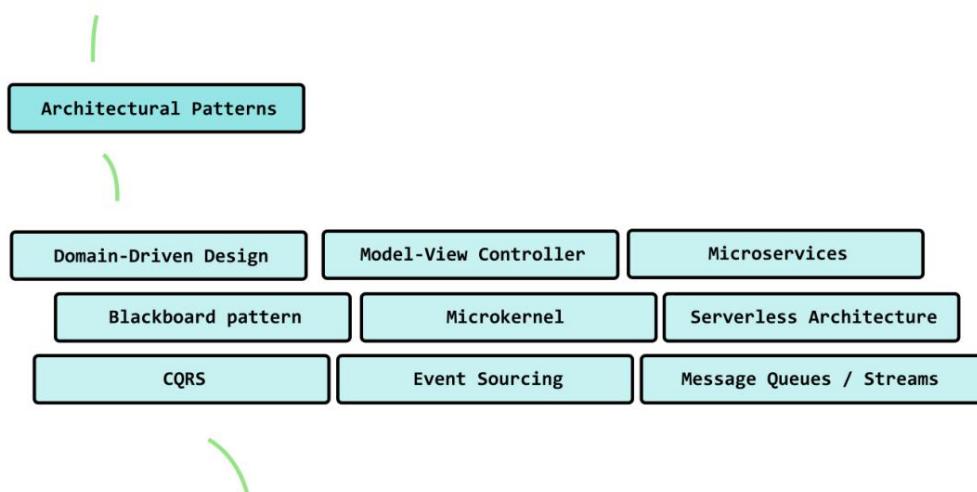
Discutimos os estilos arquitetônicos com mais detalhes em [9. Estilos arquitetônicos.](#)

Passo 8: Padrões de Arquitetura

Objetivo: Aprender os padrões de arquitetura que implementam um ou mais estilos de arquitetura para resolver um problema.

Stage 8

Learn the architectural patterns that implement one or more architectural styles to solve a problem.



Padrões arquitetônicos são implementações táticas de um ou mais *estilos arquitetônicos*.

E quando digo implementação “tática”, realmente quero dizer isso. Esses são padrões exatos que você pode usar para criar a arquitetura que protege seus SQAs.

Aqui estão alguns exemplos de padrões arquitetônicos, além dos estilos que eles herdam:

- Design Orientado ao Domínio é uma abordagem para o desenvolvimento de software contra domínios de problemas realmente complexos. Para que o DDD seja mais bem-sucedido, precisamos implementar uma **arquitetura em camadas (estilo estrutural)** para separar as preocupações de um modelo de domínio dos detalhes de infraestrutura que fazem o aplicativo realmente rodar, como bancos de dados, servidores web, caches, etc.
- Model-View-Controller é provavelmente o padrão de arquitetura *mais conhecido* para o desenvolvimento de aplicativos baseados em interface de usuário. Estilisticamente, é uma **arquitetura distribuída**. Funciona dividindo o aplicativo em 3 componentes: modelo, visualização e controlador. O MVC é incrivelmente útil quando você está começando e ajuda você a pegar carona em outras arquiteturas, mas chega um ponto em que percebemos que o **MVC não é suficiente** para problemas com muita lógica de negócios.
- **Event sourcing** é uma abordagem funcional em que armazenamos apenas as transações e nunca o estado. Se precisarmos do estado, podemos aplicar todas as transações desde o início dos tempos. Você provavelmente adivinhou, mas essa é uma abordagem de arquitetura **orientada a eventos** .

Discutimos isso na [Parte VIII: Arquitetura Essencial](#).

Passo 9: Padrões Corporativos

Objetivo: aprender os prós e contras dos conceitos envolvidos na implementação do padrão de arquitetura escolhido.

Dependendo do padrão de arquitetura que você escolheu que melhor atende às suas necessidades, haverá muitas novas construções e *jargões técnicos* para entender.

Por exemplo, quando você decide que o Domain-Driven Design é o padrão de arquitetura que faz mais sentido para o seu projeto, você precisa aprender sobre:

- Entidades: eles descrevem modelos que têm uma identidade.
- Objetos de valor: esses são modelos que não têm identidade e podem ser usados para encapsular a lógica de validação.
- Eventos de domínio: esses são eventos que significam a ocorrência de algum evento de negócios relevante e podem ser assinados a partir de outros componentes.

E se você decidir que Event Sourcing faz sentido, você terá um conjunto totalmente novo de conceitos para aprender como:

- Eventos Retroativos: Corrija automaticamente as consequências de um evento incorreto que já foi processado.
- Consistência eventual: uma maneira de obter alta disponibilidade que garante informalmente que, se nenhuma nova atualização for feita em um determinado item de dados, eventualmente todos os acessos a esse item retornarão o último valor atualizado. - via [Wiki](#)

Observação: é muito comum combinar **vários** padrões de arquitetura em uma arquitetura que atenda aos seus SQAs. Considere os desafios de DDD + Event Sourcing ou MVC + Message Queues/Streams.

Dependendo do estilo arquitetônico que você escolheu, haverá uma tonelada de conceitos para você aprender a implementar esse padrão em seu potencial máximo.

Na [Parte IX](#), construímos um aplicativo do mundo real com design orientado a domínio, arquitetura hexagonal e CQRS.

Resumo

- O design e a arquitetura de software são fundamentalmente a mesma coisa - eles apenas têm conotações de níveis ligeiramente diferentes de design.
- A arquitetura é sobre o esqueleto de nossos aplicativos e as coisas difíceis de mudar, como a maneira como a estruturamos, a maneira como lidamos com erros, representamos recursos no código etc. É o conjunto de padrões que se juntam para realizar nosso aplicativo .

- Muito do que foi descoberto no passado sobre design e arquitetura de software é relevante e relacionado de alguma forma. Por exemplo, padrões de design de software, que são tradicionalmente usados no nível de classe, influenciam novas ferramentas, tecnologias e padrões de arquitetura em um nível mais alto de abstração.

exercícios

Em breve

Referências

Artigos

- [Wikipedia: Lista de estilos e padrões arquitetônicos Estilos](#)
- [arquitetônicos vs. padrões arquitetônicos vs. padrões de design A](#)
- [Arquitetura Limpa Como e](#)
- [por que os cientistas compartilham resultados](#)
- [O que é código limpo e por que você deveria se importar?](#)

livros

- Manual do Arquiteto de Software: Torne-se um arquiteto de software de sucesso implementando conceitos de arquitetura eficazes por Joseph Ingino

Parte II: Humanos e Código

O código tem dois consumidores: usuários e desenvolvedores. As práticas técnicas **do XP nos ajudarão a nos tornar mais estruturados com a maneira como escrevemos software, mas como podemos saber se nossos projetos são agradáveis de se trabalhar? Precisamos entender como **entendemos**. Esta seção do livro apresenta os princípios do Human-Centered Design e as técnicas para desenvolver bases de código mais amigáveis ao ser humano.

4. Desmistificando o Código Limpo

De acordo com o “Simple Design”, código limpo é o código que passa em todos os testes, não possui duplicação, maximiza a clareza e possui menos elementos. Para conseguir isso, as práticas técnicas do XP promovem uma espécie de design *emergente* — ou seja, adotar uma abordagem gradual em vez de inicial. O design emergente nos dá a oportunidade de corrigir com frequência designs ruins no início, mas prosperamos mais quando podemos julgar um design de perspectivas *técnicas e centradas no ser humano*.

Objetivos do capítulo

- Saiba o que o *código limpo* significa para a comunidade geral de desenvolvimento de software e para os especialistas
- Discuta como práticas técnicas XP como TDD promovem *Emergent Design* como uma forma de construir a estrutura
- Entenda a relação que a estrutura tem com a experiência do desenvolvedor

Afinal, o que é código limpo?

O código limpo faz parte da nossa definição do objetivo do software. É um código que “**atende às necessidades dos usuários** e pode ser **alterado de forma econômica pelos desenvolvedores**”.

A limpeza é predominantemente determinada pela propriedade na segunda metade dessa definição - mudança econômica.

Poderíamos dizer que o código limpo **** tem pouca *complexidade acidental*, como **ondulação, carga cognitiva, baixa capacidade de descoberta e baixa compreensão** de 1. Complexidade e o mundo do design de software que possível.

Em vez de assumir, usaremos novamente a abordagem *aristotélica* para dominar o design de software. Para responder melhor a essa pergunta, vamos ver o que a comunidade pensa e o que os especialistas acham que é um código limpo.

De acordo com a comunidade

Deixe-os agir como uma espécie de *quadro de humor* para o que significa código limpo.



Khalil Stemmler 🇨🇴
@stemmlerjs

What constitutes "clean code" to you? Every developer tends to have a different opinion of this.

3:26 PM · Feb 15, 2020 · Twitter Web App

Veja o tópico [aqui](#).

“Código limpo é fácil de ler e modificar, revela a finalidade pretendida sem qualquer ofuscação e fala uma linguagem de domínio clara e consistente.”

Fácil de ler pode ser um comentário sobre [como nomear as coisas](#). Também pode ter a ver com [com formatação e estilo](#).

Fácil de modificar pode ser um comentário sobre acoplamento fraco, efeitos de ondulação mínimos e pode implicar na existência de testes, pois somos [capazes de refatorar o código](#) sem quebrá-lo.

Para mim, *revelar o propósito* pretendido fala diretamente sobre o uso do elemento de Design Simples que diz para usar a metáfora, mas acho que também significa que o código é *declarativo* e *funcionalmente completo*. Isso é fácil de conseguir se praticarmos TDD e escrevermos [unit](#) e [testes de aceitação](#) de forma a documentar que os recursos orientados ao cliente estão funcionando conforme o esperado.

“[O código limpo é] fácil de ler. Também não tem acoplamento entre bibliotecas.”

O segundo comentário é fascinante. Aponta para algo *mais arquitetônico*. É certo que é um pouco como uma toca de coelho - mas eles estão certos. Acoplamento, empacotamento, inversão de dependência, separação de responsabilidades e decomposição. Trata-se de [impôr limites arquitetônicos](#) e manter as dependências à distância. É engraçado. Observamos isso antes, mas o acoplamento não é uma das primeiras coisas em que você pensa na conversa sobre *código limpo*, mas não deve ser excluído. Para manter as bibliotecas ou módulos desacoplados, nós, como indústria, estabelecemos abordagens bem conhecidas para lidar com esse cenário exato.

“Ele conta uma boa história sobre o domínio e pode evoluir.”

Contar a *história do domínio* significa que qualquer novo desenvolvedor pode aprender como o negócio funciona lendo os testes.

“Eu posso lê-lo sem ser doloroso. Eu posso mudá-lo sem quebrar tudo.”

Legibilidade! De novo! Para mim, se eu tiver que alternar entre muitos arquivos e pastas, e para cima e para baixo em muitas camadas de abstração - isso é *doloroso*. E sim, testes novamente. Falando naqueles...

“De um modo geral, código limpo é código testável!”

E na minha experiência, a maneira mais fácil de escrever Mais fácil falar do que fazer! O código de escrita inherentemente testável não é óbvio. Pelo menos, não era para mim quando comecei. Eu certamente concordo com esta afirmação, no entanto.

"Para mim, é como arte. Quando criança, eu olhava para Picasso e pensava: eu posso fazer essas coisas. Agora, eu olho para ele, e é genial. Eu realmente não sei dizer por quê... claro que aprendi mais sobre formas, cores, etc. Mas é mais do que apenas técnica. A codificação é a mesma."

Sou cuidadoso ao comparar código limpo com arte, mesmo que essa seja uma maneira charmosa de ver isso. Fazer algo parecer simples geralmente leva muito mais tempo, esforço e experiência do que fazer algo parecer complexo.

O código limpo é uma arte tanto quanto um encanador instalando um banheiro ou um mecânico de automóveis realizando uma troca de óleo. Código limpo é código escrito **profissionalmente**. O tema principal do profissionalismo? *Assumir a responsabilidade*.

"Sem mágica. Simplicidade."

Isso ecoa. Pergunte a si mesmo se a complexidade está relacionada à complexidade *essencial* do problema ou *accidentalmente* baseada na maneira que você escolheu para resolvê-lo.

"KISS, DRY, YAGNI, POLA, DIP, idealmente facilitado por TDD"

Mantenha-o simples, bobo, não se repita, você não vai precisar , princípio do menor espanto, princípio de inversão de dependência e o bom e velho desenvolvimento orientado a testes. Sim, isso é *definitivamente* uma sopa de princípios de design. Mas, honestamente, se você souber o que cada um deles é, mesmo que opte por não segui-los, tê-los em mente enquanto **codifica pode melhorar** o resultado de seus designs.

"Facilmente substituível."

Como você projeta código para ser substituível? Tornando-o simples, legível e fornecendo testes.

E o último,

“Código que não amaldiçoo seu criador”

Sim, todos nós já estivemos lá...

De acordo com os especialistas

Tudo bem, agora vamos ver como alguns dos especialistas em nosso setor descrevem o código limpo.

“O custo de propriedade de um programa inclui o tempo que as pessoas gastam para entendê-lo. Os seres humanos são caros, então otimize para compreensibilidade”.
— Mathias Verraes

Mathias tem um ponto fantástico aqui. Você já começou um novo projeto em um novo domínio e teve que aumentar em uma base de código existente? A quantidade de tempo que você leva para contribuir com o código está diretamente relacionada à sua compreensão. Compreensibilidade, como aprenderemos — é outra maneira de dizer *descoberta*, na terminologia **designer**.

“Código é como humor. Quando tem que explicar, é ruim”. — Cory House

Talvez Cory esteja se posicionando no tópico de *deixar comentários no código*. O código que requer comentários está limpo? Está sujo? Praticantes de Programação Extrema acreditam que muitos comentários são *sujos*. Você ouvirá sobre isso em breve em [9. Comentários](#).

“O código limpo sempre parece ter sido escrito por alguém que se importa. Não há nada óbvio que você possa fazer para torná-lo melhor.” —Michael Penas

Talvez a primeira coisa a discutir não seja realmente como escrever um código limpo. Em vez disso, talvez devêssemos começar determinando se você está no espaço certo para defender a limpeza do código, a habilidade e seus semelhantes. Se você está apático sobre a qualidade do código e futuras

capacidade dos mantenedores de manter seu código, talvez queiramos chegar ao fundo disso primeiro.

"Se você deseja que seu código seja fácil de escrever, torne-o fácil de ler" — Robert C. Martin

Não é fascinante que a maioria das opiniões sobre código limpo dos **especialistas** tenham mais a ver com humanos do que com o código? Chame de acaso, mas essas pessoas passaram décadas fazendo essas coisas.

Considere o fato de que você e eu gastamos cerca de 60% do nosso *tempo lendo código em vez de escrevê-lo* (eu inventei esse número). Estamos lendo (e escrevendo) o livro do espaço de soluções da nossa empresa, então não se apresse e escreva bem.

Padrões de codificação limpos

Na seção anterior, procuramos conhecer o inimigo: a complexidade.

No mundo real, lidar com clientes, requisitos em constante mudança e a miríade de abordagens que podemos adotar para criar produtos usando código, talvez a melhor abordagem de imagem completa que descobrimos *como uma indústria* para lidar com essa complexidade são o conjunto de práticas descritas em Extreme Programming (XP).

Um dos principais valores do XP é construir um *Entendimento Compartilhado* da base de código com sua equipe. Isso significa:

- Um *padrão de codificação* compartilhado (ou seja: padrão de codificação)
- Uma *propriedade compartilhada da base de código* (ou seja: propriedade coletiva do código)
- Uma aspiração compartilhada para usar o *design mais simples* possível (ou seja: Simples Projeto)
- Um *entendimento compartilhado do domínio* e uma preferência para usar a *linguagem do domínio* para informar os nomes de nossas variáveis, métodos, funções e classes (ou seja: metáfora do sistema/design orientado ao domínio)

Examinaremos abordagens para cada uma delas ao longo do livro, mas neste capítulo, vamos nos concentrar especificamente nas ideias por trás de um bom padrão de codificação e design simples. Eles vão nos ajudar a elucidar qualquer confusão que possamos ter sobre o que é *código limpo* e como escrevê-lo.

O que é um padrão de codificação?

Por definição, **um padrão de codificação é uma coleção de regras que empurra o código para um estilo e abordagem consistentes em uma base de código.**

É aqui que tomamos decisões sobre:

- como estruturaremos e organizaremos nosso projeto
- como implementaremos novos recursos
- como lidaremos com erros
- como nomearemos as coisas
- a formatação e estilo por trás de arquivos, pastas, variáveis, métodos e breve
- como o código é submetido à produção (todo o código deve ter testes de acompanhamento, primeiro deve ser verificado quanto ao estilo, etc) e
- qualquer outra coisa sobre a qual os desenvolvedores possam ter opiniões variadas (comentários, tamanho do arquivo etc.)

Por que precisamos de padrões de codificação?

Padrões de codificação (às vezes também chamados de *convenções de codificação*) são úteis em qualquer projeto, mas são mais essenciais quando estamos trabalhando em um projeto que é ou será mantido ou desenvolvido por mais de um desenvolvedor (e isso é a maioria dos projetos com economia valor associado).

A grande palavra mágica por trás da importância dos padrões de codificação é **consistência**.

Das quatro maneiras pelas quais você pode detectar a complexidade (*ondulação, carga cognitiva, baixa capacidade de descoberta e baixa compreensão*), a consistência sozinha pode melhorar pelo menos três das quatro (exceto a *ondulação*, que é em geral um problema de acoplamento).

Nós, como humanos, somos máquinas de correspondência de padrões. Quando vemos como algo é feito uma vez, algo uma vez, tomamos uma nota subconsciente disso. Quando o vemos duas vezes, temos boas razões para acreditar que é assim que as coisas são feitas nesses parques. Quando o vemos três vezes, bem - deve sinalizar que é tão bom quanto a lei.

Em suma, consistência:

- Promove a
- reutilização Reduz a
- duplicação Resultados em componentes coesos, legíveis e
- focados ... e menos código

Design simples

Os humanos tendem a querer adicionar elementos em vez de subtraí-los para resolver problemas. Isso transforma a programação em algo como um quebra-cabeça ou uma atividade criativa, com experimentos em vez de construções criadas por meio de ação disciplinada.

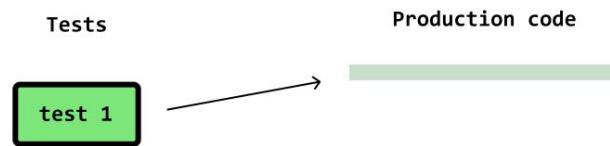
Design simples é outra ideia principal do XP que **representa os princípios de mais alto nível para um bom design.** Os quatro elementos do design simples especificam esse código limpo:

- Executa todos os testes
- Não contém duplicação
- Maximiza a clareza
- Tem menos elementos

Esta é provavelmente a **melhor definição que os especialistas concordam sobre o que é código limpo** e como alcançá-lo. Uma coisa é lutar pela legibilidade,

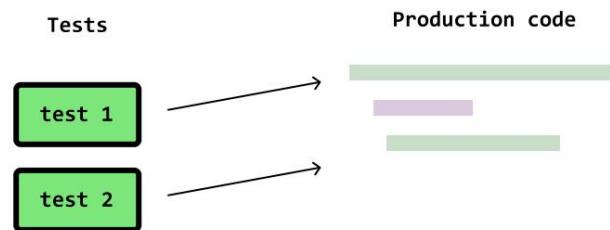
mas também precisamos emparelhar isso com práticas técnicas para que funcione. Veja como isso funciona na prática.

Aplicamos essas regras em ordem.

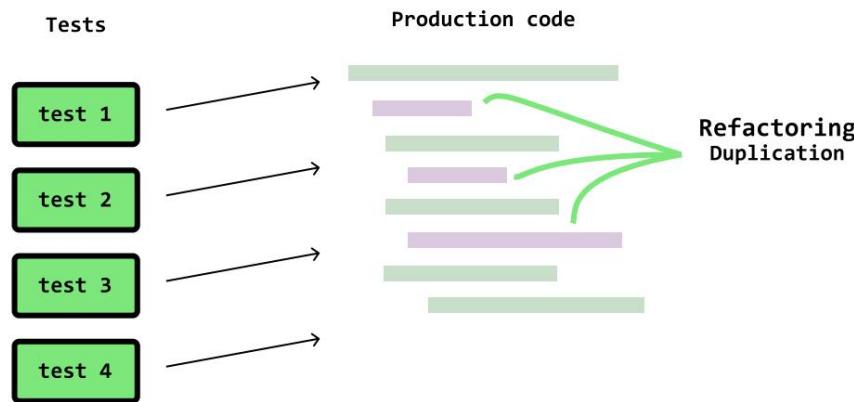


Praticando o TDD, começamos escrevendo um teste que falhou, depois a quantidade mínima de código **até que os testes passem**, então olhamos para o design em busca de *code smells*, refatoramos se você encontrar algum e fazemos o próximo teste.

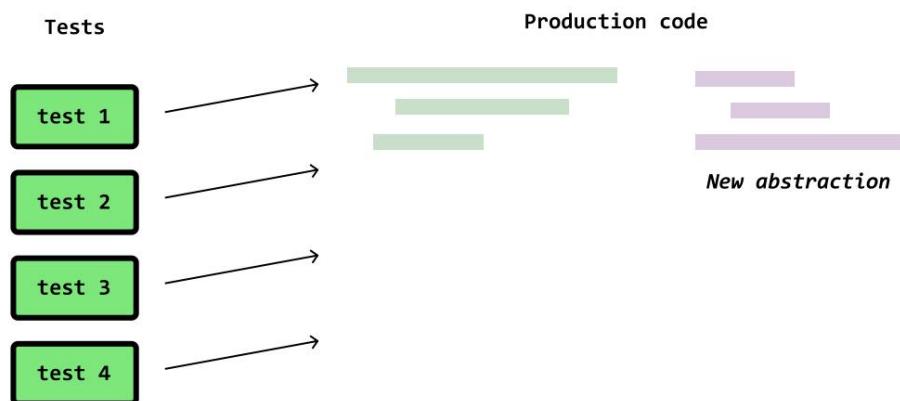
Neste livro, você dominará o TDD e ele se tornará a forma padrão de existência do código de produção.



Quando escrevermos testes suficientes, inevitavelmente acabaremos com **a duplicação**.



A duplicação é a próxima coisa a priorizar a correção. Acontece que é um dos primeiros problemas de design que surgem. No TDD, aplicamos técnicas de *refatoração*, mas somente quando os testes estão passando. Para corrigir isso, criamos uma nova abstração para expressar o conceito subjacente.



Trocando a duplicação por uma abstração executa a próxima etapa do design simples: **maximizar a clareza**. Bem usado, isso torna o código mais simples e expressivo.

Finalmente, se estamos fazendo TDD, só introduzimos novos elementos (linhas de código, métodos, funções, classes) apenas quando for absolutamente necessário. E é aí que encontramos duplicação, cheiros de código e coisas do gênero.

Revisitamos o [Simple Design](#) com mais detalhes na [Parte VI: Princípios de Design](#).

Emergência

Técnicas de feedback como Pair Programming e TDD promovem **design emergente** – uma abordagem para design que significa que as decisões de design emergem *gradualmente*.

Fazemos isso em vez de muitas decisões de design iniciais massivas.

O maior benefício disso é que temos **várias oportunidades** de corrigir um design ruim antes que se torne muito difícil de mudar. E ao fazer isso, vemos o design *surgir*.

Nenhuma UML inicial obrigatória criada pelo arquiteto e entregue aos desenvolvedores (isso é chamado de *Ivory Tower Architect* e esse processo é chamado de Big Design Up Front — geralmente tentamos evitar fazer isso).

Estrutura x experiência do desenvolvedor

O design é um empurrão e puxão de prioridades.

No mundo real, os fabricantes de móveis de escritório precisam produzir itens da maneira mais econômica possível. Por outro lado, os consumidores querem móveis de boa qualidade, mas a um preço razoável. Os atributos em desacordo entre si são **custo versus qualidade**.

Exemplos de outros equilíbrios importantes a manter:

- Anotações = contexto vs. compressão.
- Sistema de segurança = segurança vs. facilidade de configuração.

O design de software não é diferente. As duas forças em jogo são **a estrutura e a experiência do desenvolvedor**.

O design de software é a estrutura versus a experiência do desenvolvedor.

Software Design

=



Structure

vs.

Developer Experience

Estrutura

Depois de algum tempo cuidando de seus designs com TDD, teremos um pouco de estrutura desenvolvida.

Estrutura é sobre **as decisões que você tomou sobre como vamos abordar certos problemas em nossa base de código**. São coisas como decidimos que iniciaremos o aplicativo, como aplicaremos as regras de autenticação e autorização, onde empacotaremos as coisas (e com o que as empacotaremos). Isso é estrutura.

Temos que ter cuidado com a forma como implementamos a estrutura. Mais estrutura apresenta um desafio para desenvolvedores novos na base de código. Encontrar mais código,

mais coisas pelas quais prestar contas e mais regras para aprender — em projetos com muita estrutura, diz-se que a curva de aprendizado é alta.

Uma curva de aprendizado mais alta geralmente significa que leva mais tempo para aprender suas opções para concluir uma tarefa, como executar a opção e como saber se você está fazendo as coisas corretamente.

Esta não é uma ótima *experiência de desenvolvedor*, pelo menos inicialmente até descobrirmos as coisas fora.

Experiência do desenvolvedor

Onde [o design UX \(experiência do usuário\)](#) trata-se de projetar aplicativos para usuários finais, [o design DX \(experiência do desenvolvedor\)](#) trata de projetar APIs, ferramentas, linguagens, estruturas e *bases de código para desenvolvedores*.

No DX, estamos mais preocupados com os objetivos do desenvolvedor do usuário de nível superior.

De levantar e correr:

- Como posso executar isso localmente?
- Como executá-lo no modo de depuração?

Para se tornar produtivo com a base de código:

- Quais são todas as coisas que posso fazer?
- Quão fácil é entender o que esse código faz?
- Com que rapidez posso encontrar a(s) área(s) no código que preciso alterar para adicionar esse novo recurso?
- Como foi a curva de aprendizado?

E assim por diante.

A experiência do desenvolvedor é uma grande parte do motivo pelo qual usamos algumas ferramentas e não usamos outras. Junto com outros fatores como prova social (que outros