# Craftsmanship Limpo

## Disciplinas, Padrões e Ética

**Robert C. Martin**

AMOSTRA

ALTA BOOKS
EDITORA
Rio de Janeiro, 2022

# Sumário

Preâmbulo xvii
Prefácio xxi
Agradecimentos xxvii
Sobre o Autor xxix

**1. Craftsmanship ........ 1**

**I. AS DISCIPLINAS**

Extreme Programming (XP) 15
Ciclo de Vida 16
Desenvolvimento Orientado a Testes (TDD) 17
Refatoração 18
Design Simples 19
Programação Colaborativa 20
Testes de Aceitação 20

AMOSTRA

**2. Desenvolvimento Orientado a Testes ........ 21**
Visão Geral 23
Software 26
As Três Regras TDD 27
A Quarta Lei 38
Princípios Básicos 40
Exemplos Simples 40
Pilha (Stack) 41
Fatores Primos 56
Jogo de Boliche 66
Conclusão 83

**3. TDD Avançado ........ 85**
Sort 1 86
Sort 2 90
Ficando Empacado 98
Padrão de Teste Triple A 106
BDD 107
Máquinas de Estados Finitos 108
De Novo: BDD 110
Dublê de Testes 110
Dummy 113
Stub 117
Spy 119
Mock 122
Fake 125
O Princípio da Incerteza TDD 127
Escola de Londres versus Escola de Chicago 140
O Problema da Certeza 141
Escola de Londres 142
Escola de Chicago 143

AMOSTRA

Síntese 144
Arquitetura 144
Conclusão 147

**4. Testando o Design ....................................................................149**
Testando o Banco de Dados 150
Testando as GUIs 152
GUI Input 155
Padrões de Teste 156
Subclasse Específica de Teste 157
Padrão Self-Shunt 158
Padrão de Objeto Humble 159
Testando o Design 162
O Problema do Teste Frágil 163
A Correspondência Biunívoca 163
Quebrando a Correspondência 165
A Videolocadora 166
Especificidade versus Generalidade 185
Premissa de Transformação Prioritária (TPP) 186
{} → Nil 188
Nil → Constante 189
Incondicional → Seleção 190
Valor → Lista 191
Instrução → Recursão 191
Seleção → Iteração 192
Valor → Valor Mutável 192
Exemplo: Sequência de Fibonacci 193
Premissa de Transformação Prioritária (TPP) 197
Conclusão 198

AMOSTRA

**5. Refatoração ........ 199**

O Que É Refatoração? 200

O Toolkit Básico 202

Renomear 202

Refatoração Extract Method 203

Extract Variable 205

Extract Field 206

Cubo Mágico 219

As Disciplinas 219

Testes 220

Testes Rápidos 220

Quebrando Correspondências Biunívocas Profundas 220

Refatore Continuamente 221

Refatore sem Dó nem Piedade 221

Mantenha os Testes Passando! 221

Tenha uma Saída 222

Conclusão 223

**6. Design Simples ........ 225**

YAGNI 229

Cobertura de Testes 230

Cobertura 232

Meta Assintótica 233

Design? 234

Mas, Espere... Isso Não É Tudo 234

Maximize a Expressividade 235

Abstração Subjacente 237

Testes: A Outra Metade do Problema 238

Minimize a Duplicação 239

Minimize o Tamanho 241

Design Simples 241

AMOSTRA

7. **Programação Colaborativa ........................................243**
8. **Testes de Aceitação ........................................249**
A Disciplina 252
O Build Contínuo 253

## II. OS PADRÕES

Seu Novo CTO 256

9. **Produtividade ........................................257**
Nós Nunca Entregaremos M***A 258
Adaptabilidade Acessível 260
Estaremos Sempre Prontos 261
Produtividade Estável 263

10. **Qualidade ........................................265**
Melhoria Contínua 266
Competência Destemida 267
Qualidade Extrema 268
Não Vamos Jogar a QA no Lixo 269
A Doença da QA 270
O Pessoal de QA Não Encontrará Nada 270
Automatização de Testes 271
Testes Automatizados e Interfaces de Usuário 272
Testando a Interface do Usuário 274

11. **Coragem ........................................275**
Damos Cobertura Uns aos Outros 276
Estimativas Honestas 278
Você Precisa Dizer Não 280

AMOSTRA

Aprendizagem Determinante Contínua 281
Mentoria 282

## III. ÉTICA

O Primeiro Programador 284
Setenta e Cinco Anos 285
Nerds e Salvadores 290
Exemplos de Inspiração e de Vilões 293
Nós Comandamos o Mundo 294
Catástrofes 296
O Juramento do Programador 297

**12. Prejuízo ........ 299**
Primeiro, Não Prejudique 300
Não Prejudique a Sociedade 301
Não Prejudique Seu Ofício 303
Não Prejudique a Estrutura 305
Flexibilidade 307
Testes 308
Meu Melhor Trabalho 310
Fazendo o Certo 311
O Que É uma Boa Estrutura? 312
Matriz de Eisenhower 314
Programadores e Desenvolvedores São Stakeholders 316
Seu Melhor 318
Prova Reproduzível 320
Dijkstra 320
Provando a Precisão 321
Programação Estruturada 323

AMOSTRA

Decomposição Funcional 326
Desenvolvimento Orientado a Testes 327

13. Integridade ....331
Ciclos Curtos 332
A História Sobre o Controle do Código-fonte 332
Git 338
Ciclos Curtos 339
Integração Contínua 340
Branches versus Toggles 341
Implementação Contínua 343
Build Contínuo 345
Melhoria Implacável 346
Cobertura de Teste 346
Teste de Mutação 347
Estabilidade Semântica 348
Limpeza 349
Criações 350
Mantenha Alta Produtividade 350
Viscosidade 351
Gerenciando as Distrações 354
Gerenciamento de Tempo 357

14. Trabalho em Equipe ....359
Trabalhe como uma Equipe 360
Trabalho Remoto ou Virtual 360
Faça uma Estimativa Honesta e Justa 362
Mentiras 363
Honestidade, Acurácia e Precisão 364
História 1: Vetores 365

AMOSTRA

História 2: pCCU 368
Lição Aprendida 369
Acurácia 369
Precisão 371
Agregação 373
Honestidade 374
Respeito 376
Nunca Pare de Aprender 377

Índice 379

AMOSTRA

# 1 Craftsmanship

ANOSTRA

Sem dúvidas, o sonho de voar é quase tão imemorial quanto a humanidade. O antigo mito grego que retrata a fuga de Dédalo e de Ícaro data de cerca de 1550 a.C. Nos milênios que se seguiram, diversas almas corajosas, ainda que tolas, amarraram geringonças desajeitadas a seus corpos, pulando de penhascos e torres rumo à morte certa em busca desse sonho.

As coisas começaram a mudar há cerca de quinhentos anos, quando Leonardo Da Vinci esboçou projetos de máquinas que, embora não fossem capazes de voar, demonstravam certo raciocínio fundamentado. Foi Da Vinci quem percebeu que voar seria possível, visto que a resistência do ar funciona em ambas as direções. Ao pressionar o ar para baixo, a resistência ocasionada pelo movimento cria uma elevação na mesma proporção. Graças a esse mecanismo, todos os aviões modernos conseguem voar.

Até meados do século XVIII, não tivemos contato com as ideias de Da Vinci. Mas então iniciou-se uma exploração quase alucinada sobre a possibilidade de voar. Os séculos XVIII e XIX foram uma época de intensa pesquisa e experimentação aeronáutica. Construíram-se protótipos sem motores que foram testados, descartados e aprimorados. A ciência aeronáutica começou a tomar forma e a ganhar vida. As forças de sustentação, arrasto, empuxo e gravidade foram identificadas e compreendidas. Algumas almas corajosas tentaram voar.

AMOSTRA

E algumas se chocaram contra o solo e morreram.

Nos últimos anos do século XVIII, e durante o meio século que se seguiu, Sir George Cayley, o pai da aerodinâmica moderna, construiu plataformas experimentais, protótipos e modelos em tamanho real, o que resultou no primeiro voo tripulado de um planador.

E, mesmo assim, alguns se chocaram contra o solo e morreram.

Então veio a era do vapor e a possibilidade de um voo tripulado e motorizado. Construíram-se dezenas de protótipos e experimentos. Cientistas e entusiastas se juntaram ao enxame de pessoas que exploravam as possibilidades de voar. Em 1890, Clément Ader pilotou uma máquina bimotora a vapor por cinquenta metros.

Mas alguns continuavam a se chocar contra o solo e morrer.

No entanto, o motor de combustão interna foi o verdadeiro divisor de águas. É bem provável que o primeiro voo tripulado e motorizado tenha ocorrido em 1901 por Gustave Whitehead, mas foram os irmãos Wright que, no dia 17 de dezembro de 1903, em Kill Devil Hills, Carolina do Norte, conduziram o primeiro voo tripulado duradouro, motorizado e controlado de uma máquina mais pesada que o ar.

E, mesmo assim, alguns se chocaram contra o solo e morreram.

Entretanto, o mundo mudou da noite para o dia. Onze anos depois, em 1914, biplanos rasgavam os céus de toda a Europa em combates aéreos.

E, apesar de muitas pessoas terem caído e morrido sob fogo inimigo, um número semelhante caiu e morreu aprendendo a voar. Ainda que os princípios de voar tenham sido dominados, as *técnicas* de voo foram mal compreendidas.

Nas duas décadas seguintes, os hediondos caças e bombardeiros da Segunda Guerra Mundial semeavam o caos na França e na Alemanha. Eles atingiam altitudes extremas, eram equipados fortemente com armas e ostentavam um poder destrutivo avassalador.

Durante a guerra, perdeu-se 65 mil aeronaves norte-americanas. Porém, somente 23 mil delas foram perdidas em combate. Os pilotos voavam e morriam em batalha, mas era muito mais comum que voassem e morressem quando ninguém estava atirando. Ainda não sabíamos *como* voar.

Na década seguinte, presenciamos aviões a jato, a quebra da barreira do som e a explosão de companhias aéreas comerciais e viagens aéreas domésticas. Era o início da era do jato, quando pessoas endinheiradas (os famigerados *jet set*) podiam voar de cidade em cidade e de país em país em questão de horas.

AMOSTRA

E os aviões a jato se chocaram contra o solo e despencaram do céu em números aterrorizantes. Havia tanta coisa que ainda não entendíamos sobre a fabricação e o voo de aeronaves.

Isso nos leva à década de 1950. Os Boeing 707 transportariam passageiros de um lugar para outro do mundo até o fim da década. Duas décadas depois, veríamos o primeiro jato jumbo de grande porte, o 747.

A Aeronáutica e as viagens aéreas se consolidaram, tornando-se o meio de viagem mais seguro e eficiente da história mundial. Foi preciso muito tempo e custou muitas vidas, mas, finalmente, aprendemos a construir e a pilotar aviões com segurança.[1]

Chesley Sullenberger nasceu em 1951, em Denison, Texas. Na era do jato, ele não passava de um menino. Aos 16 anos, Sullenberger aprende a pilotar e acaba pilotando um caça-bombardeiro McDonnell Douglas F-4 Phantom II para a Força Aérea dos Estados Unidos. Em 1980, ele se tornou piloto da US Airways.

Em 15 de janeiro de 2009, logo após decolar do Aeroporto LaGuardia, o Airbus A320 que ele pilotava com 155 almas atingiu um bando de gansos e perdeu os dois motores a jato. O Capitão Sullenberger, confiando em suas mais de 20 mil horas de experiência no ar, conduziu o avião avariado para um "pouso na água" no rio Hudson e, por meio de sua competência inabalável, salvou cada uma daquelas 155 almas a bordo. O Capitão Sullenberger se destacou em sua arte de voar. O capitão Sullenberger é um craftsman.

---

E, sem dúvidas, o sonho de processamento computacional e gerenciamento de dados rápidos e seguros é quase tão antigo quanto a humanidade. O ato de fazer contas com os dedos da mão e contar gravetos remonta a milhares de anos. Há mais de 4 mil anos, as

1 A despeito do Boeing 737 Max.

AMOSTRA

pessoas construíam coisas e usavam ábacos. Há cerca de 2 mil anos, artefatos mecânicos foram utilizados para prever o movimento de estrelas e planetas. Há aproximadamente 400 anos, as réguas de cálculo foram inventadas.

Em meados do século XIX, Charles Babbage começou a construir máquinas de calcular à manivela. Estas eram verdadeiros computadores digitais, com memória e processamento aritmético, mas, por conta da tecnologia de metalurgia da época, elas eram difíceis de construir e, embora ele tenha construído alguns protótipos, elas não foram comercialmente bem-sucedidas.

Em meados do século XIX, Babbage tentou construir uma máquina muito mais poderosa. Era um dispositivo a vapor, capaz de executar até programas. Ele a chamou de *A Máquina Analítica*.

AMOSTRA

A filha de Lord Byron, Ada — a condessa de Lovelace — traduziu as notas de uma palestra ministrada por Babbage e percebeu um fato que aparentemente não havia ocorrido a ninguém na época: *os números em um computador não precisam representar números propriamente ditos, mas podem representar coisas no mundo real.* Por conta dessa descoberta repentina, ela é geralmente considerada a primeira programadora do mundo.

Problemas com tecnologias precisas de metalurgia continuaram a frustrar Babbage e, no fim, seu projeto foi por água abaixo. E, durante o resto do século XIX e o início do século XX, não houve mais avanços em relação aos computadores digitais. No entanto, durante esse período, os computadores *analógicos* e mecânicos alcançaram seu apogeu.

Em 1936, Alan Turing demonstrou que não há uma maneira geral de provar que uma dada equação diofantina[2] pode ser solucionada. Ele sustentou sua prova imaginando um computador digital simples, mesmo que infinito, e depois evidenciando que havia números que esse computador não conseguia calcular. Em decorrência de sua prova, ele inventou máquinas de

2 Equações de inteiros.

estados finitos, linguagem de máquina, linguagem simbólica, macros e sub-rotinas primitivas. Alan inventou o que hoje chamamos de software.

Quase ao mesmo tempo, Alonzo Church construiu uma prova totalmente diferente para o mesmo problema e, como resultado, desenvolveu o cálculo lambda — o conceito-chave da programação funcional.

Em 1941, Konrad Zuse construiu o primeiro computador digital programável eletromecânico, o Z3. A máquina tinha mais de 2 mil relés e operava a uma velocidade de clock (taxa de bits transmitida na interface serial) de 5Hz a 10Hz. A máquina usava aritmética binária organizada em palavras de 22 bits.

Durante a Segunda Guerra Mundial, Turing foi recrutado para ajudar os cientistas militares britânicos em Bletchley Park (onde ficava a instalação secreta militar Government Code and Cypher School) a descriptografar os códigos alemães da Enigma. A máquina Enigma era um computador digital simples que randomizava os caracteres de mensagens textuais, normalmente transmitidas por meio de radiotelégrafos. Turing ajudou a criar um mecanismo eletromecânico de busca digital que identificasse as chaves desses códigos.

AMOSTRA

Após a guerra, Turing teve papel fundamental na construção e na programação de um dos primeiros computadores eletrônicos de tubo a vácuo do mundo — o Automatic Computing Engine, ou ACE. O protótipo original usava 1 mil tubos de vácuo e manipulava números binários a uma velocidade de 1 milhão de bits por segundo.

Em 1947, depois de escrever alguns programas para essa máquina e estudar suas funcionalidades, Turing ministrou uma palestra na qual fez as seguintes declarações visionárias:

> *Precisaremos de um grande número de matemáticos habilidosos [para lidar com os problemas] na forma de computação.*
>
> *Uma das nossas dificuldades será a manutenção de uma disciplina adequada a fim de não perdermos a noção do que estamos fazendo.*

E o mundo mudou da noite para o dia.

Em poucos anos, a memória core foi inventada. A possibilidade de ter centenas de milhares, talvez milhões, de bits de memória acessíveis em microssegundos se tornou uma realidade. Ao mesmo tempo, a produção em massa de tubos a vácuo possibilitou que os computadores ficassem cada vez mais baratos e confiáveis. A produção em massa limitada estava se tornando uma realidade. Em 1960, a IBM havia vendido 140 computadores modelo 70x. Estas eram máquinas gigantescas de tubo a vácuo que valiam milhões de dólares.

Turing havia programado sua máquina em binário, porém todo mundo entendia que isso não era nada prático. Em 1949, Grace Hopper cunhou a palavra *compilador* e, em 1952, criou o primeiro sistema A-0. No fim de 1953, John Bachus apresentou a primeira especificação FORTRAN. Em 1958, seguiram as invenções das linguagens ALGOL e LISP.

O primeiro transistor funcional foi criado por John Bardeen, Walter Brattain e William Shockley em 1947. Em 1953, eles começaram a fazer parte dos computadores. Substituir os tubos a vácuo por transistores mudou totalmente o jogo. Os computadores se tornaram menores, mais rápidos, mais baratos e mais confiáveis.

Em 1965, a IBM havia produzido 10 mil computadores do modelo 1401. Eles eram alugados por US$2.500 por mês. Isso estava ao alcance das empresas de médio porte. Essas empresas precisavam de programadores e, assim, a demanda por programadores começou a se intensificar.

Quem estava programando todas essas máquinas? Não havia cursos universitários de computação. Em 1965, ninguém aprendia a programar na escola. Esses programadores vinham dos negócios. Eram pessoas mais velhas, que já trabalhavam em seus negócios há algum tempo. Estavam na casa dos 30, 40 e 50 anos.

Em 1966, a IBM produzia mensalmente 1 mil computadores modelo 360. As empresas não se cansavam dessas máquinas. Elas tinham memórias

AMOSTRA

que atingiam 64kB ou mais, além de poder executar centenas de milhares de instruções por segundo.

Nesse mesmo ano, trabalhando em um Univac 1107 no Norwegian Computer Center, Ole-Johan Dahl e Kristen Nygard inventaram a linguagem Simula 67, uma extensão da ALGOL. Era a primeira linguagem orientada a objetos.

*A palestra de Alan Turing fora somente duas décadas antes!*

Dois anos depois, em março de 1968, Edsger Dijkstra escreveu sua famosa carta à revista *Communications of the ACM (CACM)*. O editor deu a essa carta o título "Go To Statement Considered Harmful",[3] frase que se popularizou por causa da crítica de Dijkstra sobre o uso do comando Goto. Nascia a programação estruturada.

Em 1972, no Bell Labs em Nova Jersey, Ken Thompson e Dennis Ritchie estavam sem projetos para trabalhar. Eles imploraram para usar um PDP 7 de uma equipe de projeto diferente e simplesmente inventaram o UNIX e a C.

Agora o ritmo computacional atingia velocidades quase vertiginosas. Falarei de algumas datas importantes. Pergunte a si mesmo: quantos computadores existem no mundo para cada pessoa? Quantos programadores existem no mundo? E de onde vieram esses programadores? O que comem e como vivem?

1970 — Desde 1965, a Digital Equipment Corporation produziu 50 mil computadores PDP-8.

1970 — Winston Royce escreveu o artigo "waterfall" [Método Cascata], "Managing the Development of Large Software Systems" [Gerenciando o Desenvolvimento de Grandes Sistemas de Software].

1971 — A Intel lançou o microprocessador 4004 em um chip único.

AMOSTRA

3 Edsger W. Dijkstra, "Go To Statement Considered Harmful", Communications of the ACM 11, nº 3 (1968).

1974 — A Intel lançou o microprocessador 8080 em um chip único.

1977 — A Apple lançou o Apple II.

1979 — A Motorola lançou o 68000, um microprocessador de chip único de 16 bits.

1980 — Bjarne Stroustrup inventou *o C com classes* (um pré-processador que faz o C parecer Simula).

1980 — Alan Kay inventou a linguagem de programação Smalltalk.

1981 — A IBM lançou o IBM PC.

1983 — A Apple lançou o Macintosh de 128K.

1983 — Stroustrup renomeou o C com Classes para C++.

1985 — O Departamento de Defesa dos EUA adotou o Método Cascata como processo oficial de desenvolvimento de software (DOD-STD-2167A).

1986 — Stroustrup publicou *The C++ Programming Language*. No Brasil, *Princípios e Práticas de Programação com C++*.

1991 — Grady Booch publicou *Object-Oriented Design with Applications*.

1991 — James Gosling inventou a linguagem de programação Java (chamada de Oak na época).

1991 — Guido Van Rossum lançou a linguagem de programação Python.

1995 — A obra *Padrões de Projetos: Soluções Reutilizáveis de Software Orientados a Objetos* foi escrita por Erich Gamma, Richard Helm, John Vlissides e Ralph Johnson.

1995 — Yukihiro Matsumoto lançou a linguagem de programação Ruby.

1995 — Brendan Eich criou o JavaScript.

1996 — A Sun Microsystems lançou o Java.

1999 — A Microsoft inventou a linguagem C#/.NET (até então chamada de *Cool*).

2000 — O bug Y2K! O Bug do Milênio.

2001 — O Manifesto Ágil foi escrito.

AMOSTRA

Entre 1970 e 2000, a velocidade de clock dos computadores aumentou em três ordens de grandeza. A densidade aumentou em quatro ordens de grandeza. O espaço em disco aumentou em seis ou sete ordens de grandeza. A capacidade RAM aumentou em seis ou sete ordens de grandeza. Os custos caíram de dólares por bit para dólares por gigabit. É difícil visualizar as mudanças de hardware, mas, se somarmos as ordens de grandeza que mencionei, chegaremos a um aumento de processamento computacional em torno de trinta ordens de grandeza.

E tudo isso apenas cerca de cinquenta anos depois da palestra de Alan Turing.

Atualmente, quantos programadores existem no mundo? Quantas linhas de código foram escritas? Qual é o nível de qualidade dessas linhas de código?

Compare essa linha temporal com a linha da Aeronáutica. Percebeu a semelhança? Você consegue ver o aumento teórico gradual, a pressa e o fracasso dos entusiastas, e o aumento gradual da competência? Notou as décadas que passamos sem saber o que estávamos fazendo?

E nos dias atuais, visto que a própria existência da nossa sociedade depende de nossas habilidades, será que temos os Sullenbergers de quem precisamos? Será que preparamos os programadores tão bem quanto os pilotos de avião para entenderem seu ofício? Temos os craftsmen de quem certamente precisaremos?

AMOSTRA

---

O significado da palavra inglesa craftsmanship é saber fazer algo perfeitamente bem. E esse fazer algo perfeitamente bem é fruto de uma boa orientação e de vasta experiência. Até recentemente, o mercado de software tinha pouquíssimo de ambos. Os programadores costumavam não permanecer na carreira de programador, pois enxergavam a programação como um degrau para o gerenciamento. Ou seja, havia poucos programadores que adquiriam experiência suficiente para ensinar o ofício a outras pessoas. Para piorar as coisas, o número de programadores novos

que entram em campo dobra a cada cinco anos ou mais, mantendo a proporção de programadores experientes muito baixa.

O resultado é que a maioria dos programadores nunca aprende as disciplinas, os padrões e a ética que podem definir seu ofício. Durante sua carreira relativamente breve de escrever códigos, eles seguem como principiantes sem formação. E, obviamente, isso significa que boa parte do código escrito por esses programadores inexperientes fica abaixo dos padrões, sendo mal estruturada, insegura, com erros e geralmente uma verdadeira bagunça.

Neste livro, descrevo os padrões, as disciplinas e a ética que acredito que todo programador deve conhecer e adotar a fim de adquirir gradualmente o conhecimento e as habilidades que seu ofício de fato exige.

AMOSTRA

AMOSTRA

# I
# As Disciplinas

AMOSTRA

O que é uma disciplina? Uma disciplina é um conjunto de regras composto de duas partes: a essencial e a arbitrária. A parte essencial é o que faculta à disciplina seu poder; é a razão pela qual a disciplina existe. A parte arbitrária é o que proporciona à disciplina seus aspectos e sentidos. A disciplina não pode existir sem essa arbitrariedade.

Por exemplo, os cirurgiões lavam as mãos antes da cirurgia. Se você observasse, veria que a lavagem das mãos tem aspectos muito específicos. O cirurgião não lava as mãos simplesmente as ensaboando com água corrente, como você e eu fazemos. Ao contrário, o cirurgião adota uma disciplina ritualizada para lavar as mãos. Uma das rotinas que reparei funciona mais ou menos assim:

- Uso do sabonete adequado.
- Uso da escova adequada.
- Lavar cada dedo assim:
  - Escovar a ponta dos dedos no mínimo dez vezes.
  - Escovar dez vezes o lado esquerdo de cada dedo.
  - Escovar dez vezes a parte inferior de cada dedo.
  - Escovar dez vezes o lado direito de cada dedo.
  - Escovar as unhas no mínimo dez vezes.
- E assim sucessivamente.

AMOSTRA

A parte essencial da disciplina deve ser transparente. As mãos do cirurgião devem estar bem limpas. Mas você percebeu a parte arbitrária? Por que dez escovadas, em vez de oito ou doze? Por que dividir as mãos em cinco seções? Por que não três ou sete?

Tudo isso é arbitrário. Não há nenhum motivo real para essa quantidade de vezes a não ser para que sejam consideradas suficientes.

Neste livro, estudaremos cinco disciplinas de desenvolvimento de software craftsmanship. Algumas delas já têm cinco décadas, enquanto outras têm apenas duas. Porém, ao longo dessas décadas, todas elas provaram sua serventia. Sem elas, a própria noção de software como uma profissão seria praticamente inimaginável.

Cada uma dessas disciplinas apresenta as próprias características essenciais e arbitrárias. Talvez, ao ler, você descubra que sua mente refuta uma ou mais disciplinas. Se isso acontecer, preste atenção e procure saber se a rejeição tem a ver com as características essenciais das disciplinas ou com as características arbitrárias. Não se deixe levar erroneamente pela arbitrariedade. Foque as características essenciais. Depois de internalizar a natureza de cada disciplina, a parte arbitrária provavelmente perderá a importância.

Por exemplo, em 1861, Ignaz Semmelweis publicou suas descobertas para a aplicação da disciplina de lavagem das mãos para médicos. Os resultados de sua pesquisa foram impressionantes. Ele conseguiu demonstrar que, quando os médicos lavavam cuidadosamente as mãos com água sanitária antes de examinar as pacientes grávidas, a taxa de mortalidade em relação a infecções graves posteriores, que antes era de uma para cada dez pacientes, praticamente zerava.

Contudo, os médicos da época não separaram o essencial do arbitrário e criticaram a disciplina proposta por Semmelweis. A água sanitária era a parte arbitrária, mas a lavagem era essencial. Os médicos rechaçaram a teoria, pois lavar as mãos com água sanitária era um incômodo para eles. Assim, rejeitaram a evidência da natureza essencial da lavagem das mãos.

Passaram-se muitas décadas até os médicos realmente começarem a lavar as mãos.

AMOSTRA

## Extreme Programming (XP)

Em 1970, Winston Royce publicou o artigo que popularizou o processo de desenvolvimento em cascata. Levou-se quase trinta anos para corrigir esse erro.

Em 1995, os especialistas em software começaram a considerar uma abordagem diferente e mais incremental. Processos como Scrum, Desenvolvimento Orientado à Funcionalidade (FDD), Metodologia de Desenvolvimento de Sistemas Dinâmicos (DSDM) e as Metodologias Crystal foram apresentados. Mas, em geral, pouca coisa na área mudou.

Então, em 1999, Kent Beck publicou o livro *Programação Extrema (XP) Explicada*. A Programação Extrema (XP) foi construída com base nas ideias dos processos anteriores, mas acrescentou algo novo. Ela acrescentou *práticas de engenharia*.

Entre 1999 e 2001, o entusiasmo com a XP cresceu exponencialmente. Foi esse entusiasmo que originou e impulsionou a revolução ágil. Até hoje, a XP continua sendo o mais bem definido e o mais completo de todos os métodos ágeis. As práticas de engenharia que residem em seu centro nevrálgico são o foco desta seção de disciplinas.

## Ciclo de Vida

Na Figura I.1, você vê o *Ciclo de Vida* de Ron Jeffries, que mostra as práticas XP. As disciplinas que abordamos neste livro são as quatro do centro e a da extrema esquerda.

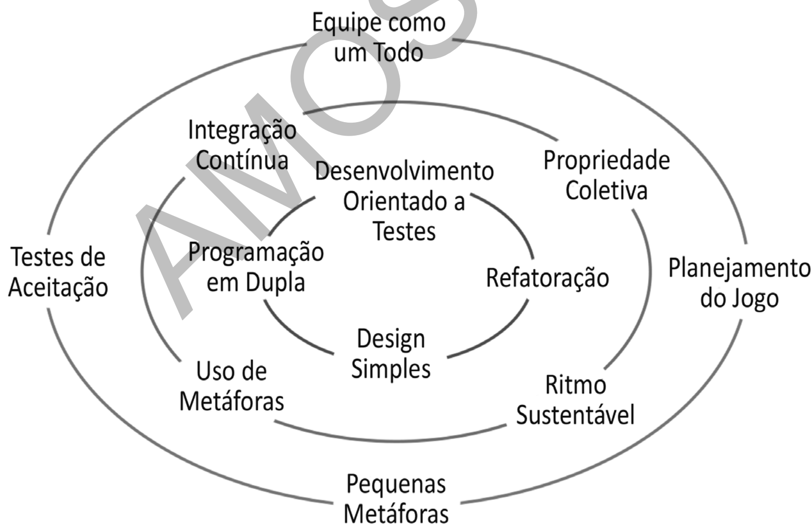


**Figura I.1** Ciclo de vida: práticas XP

As quatro no centro são as práticas de engenharia XP: Desenvolvimento Orientado a Testes (TDD), Refatoração, Design Simples e Programação em Dupla (que aqui neste livro chamaremos de *programação colaborativa*).

AMOSTRA

A prática no canto esquerdo, Testes de Aceitação, é a mais técnica e de engenharia, orientada às práticas de negócios XP.

Essas cinco práticas são as disciplinas indispensáveis do desenvolvimento craftsmanship.

## Desenvolvimento Orientado a Testes (TDD)

TDD é a disciplina crucial. Sem ela, as outras disciplinas são impossíveis ou ineficazes. Por essa razão, as duas próximas seções que abordam o TDD representam quase metade das páginas deste livro e são intensamente técnicas. Talvez essa organização lhe pareça incongruente.
Na verdade, também me parece isso, e me esforcei para saber o que fazer a respeito. Minha conclusão, no entanto, é que essa incongruência é resultado da incongruência correspondente em nossa área. Poucos programadores conhecem essa disciplina a fundo.

AMOSTRA

TDD é a disciplina que governa a forma pela qual um programador trabalha, diariamente e a cada segundo. Não é uma disciplina prévia nem posterior. O TDD está integrado e presente durante todo o processo e bem debaixo do seu nariz. Não há como fazer TDD parcial; é uma disciplina do tipo tudo ou nada.

A essência da disciplina TDD é muito simples. Em primeiro lugar, os pequenos ciclos e testes. Os testes vêm em primeiro lugar em tudo.
Os testes são escritos primeiro. Os testes são limpos primeiro. Em todas as atividades, os testes vêm em primeiro lugar. E todas as atividades são divididas no mais ínfimo dos ciclos.

A duração dos ciclos é mensurada em segundos, não em minutos. É calculada em caracteres, não em linhas. O ciclo de feedback é finalizado quase imediatamente após iniciado.

O objetivo do TDD é criar uma suíte de testes em que você confiaria sua vida. Caso a suíte de testes seja aprovada, você deve se sentir seguro para fazer o deploy do código.

De todas as disciplinas, a TDD é a mais onerosa e a mais complexa. É onerosa porque domina tudo. É a primeira e a última coisa em que você pensa. É a restrição que impacta absolutamente tudo o que você faz. É a disciplina que comanda e mantém o ritmo constante, independentemente da pressão e dos problemas do ambiente.

O TDD é complexo devido à complexidade do código. Para cada estrutura ou formato de código, há uma estrutura e um formato TDD correspondente. O TDD é complexo porque os testes devem ser desenvolvidos para se ajustar ao código, não para ser acoplados, devendo abranger quase tudo e, ainda assim, ser executados em segundos. TDD é uma técnica sofisticada e complexa, muito difícil de ser aprendida, mas imensamente gratificante.

## Refatoração

Refatoração é a disciplina que nos possibilita escrever um código limpo. Refatorar sem TDD[4] é difícil, ou mesmo impossível. Assim sendo, escrever um código limpo sem TDD é absurdamente difícil ou impossível.

A refatoração é a disciplina em que alteramos o código mal estruturado, melhorando sua estrutura interna *sem afetar seu comportamento*. O comportamento é a parte crítica. Ao garantir que o comportamento do código não seja afetado, temos também a garantia de que as melhorias na estrutura são *seguras*.

O motivo pelo qual não limpamos o código — o motivo pelo qual os sistemas de software se deterioram e apodrecem com o tempo — é que temos medo de que a limpeza do código prejudique seu comportamento. Mas, se tivermos uma forma segura de limpar o código, *conseguiremos* limpá-lo e nossos sistemas não apodrecerão.

AMOSTRA

4 Talvez haja outras disciplinas que poderiam apoiar a refatoração tão bem como o TDD. O test && commit || revert de Kent Beck é uma possibilidade. No momento em que este livro foi escrito, no entanto, ele não tinha desfrutado de um alto grau de adoção e continua sendo mais uma curiosidade acadêmica.

Como garantimos que nossas melhorias não prejudicarão o comportamento do código? Temos os testes TDD.

Refatorar também é uma disciplina complexa, pois há muitas maneiras de se escrever um código mal estruturado. Ou seja, existem inúmeras estratégias para limpar esse código. Além do mais, cada uma dessas estratégias deve se ajustar sem impedimento algum e simultaneamente no primeiro ciclo de teste TDD. Na verdade, essas duas disciplinas estão tão arraigadas que são praticamente inseparáveis. É quase impossível refatorar sem TDD, e é basicamente impossível praticar TDD sem praticar a refatoração.

## Design Simples

AMOSTRA

A vida na Terra pode ser descrita em camadas. No topo está a camada da ecologia, o estudo dos sistemas de coisas vivas. Abaixo, está a camada de fisiologia, o estudo dos mecanismos inerentes à vida. A próxima camada pode ser a microbiologia, o estudo das células, dos ácidos nucleicos, das proteínas e de outros sistemas macromoleculares. E, por sua vez, todas essas camadas são abordadas pela ciência química, enquanto esta última é estudada pela mecânica quântica.

Vamos estender essa mesma analogia à programação: se o TDD é a mecânica quântica da programação, logo a refatoração é a química e o design simples é a microbiologia. Prosseguindo com a analogia, os princípios SOLID, o Design Orientado a Objetos e a Programação Funcional representam a fisiologia, e a arquitetura é a ecologia da programação.

O design simples é quase impossível sem a refatoração. Na realidade, o design simples é o objetivo final da refatoração, e a refatoração é o único meio prático de alcançar esse objetivo. Tal objetivo é a produção de grãos atômicos simples de design que se encaixam bem em grandes estruturas de programas, sistemas e aplicativos.

O design simples não é uma disciplina complexa. Ele é orientado por quatro regras bem simples. No entanto, ao contrário do TDD e da refatoração, o design simples é uma disciplina imprecisa. Ela depende de discernimento e de experiência. Quando bem-feito, é o primeiro indício

que separa um aprendiz que conhece as regras de um artesão que entende os princípios. É o começo do que Michael Feathers chamou de *senso de design*.

## Programação Colaborativa

A programação colaborativa é a disciplina e a arte de trabalhar junto em uma equipe de software. Isso inclui subdisciplinas como programação em duplas, programação mob, revisões de código e brainstorms. A programação colaborativa envolve todos na equipe, programadores e não programadores. Esse é o principal meio pelo qual compartilhamos conhecimento, asseguramos consistência e integramos a equipe em um todo funcional.

De todas as disciplinas, a programação colaborativa é a menos técnica e a menos prescritiva. Mas talvez seja a mais importante das cinco disciplinas, visto que a formação de uma equipe eficaz é algo raro e valioso.

AMOSTRA

## Testes de Aceitação

O teste de aceitação é a disciplina que vincula a equipe de desenvolvimento de software ao segmento de negócio. O objetivo do negócio é a especificação dos comportamentos desejados do sistema. Esses comportamentos são codificados em testes. Se esses testes forem aprovados, o sistema se comportará conforme especificado.

Os testes devem ser legíveis e passíveis de alteração pela equipe de negócio. É por meio da escrita, da leitura e da aprovação desses testes que a equipe de negócios sabe como o software funciona e faz o que o segmento de negócio precisa que ele faça.