

# Introdução ao Clean Craftsmanship

Apresentação

Série Robert C. Martin



# Craftsmanship Límpio

Disciplinas, Padrões e Ética



Robert C. Martin

Referência mundial em desenvolvimento de software e métodos ágeis

*Preâmbulo de Stacia Heimgartner Viscardi, CST e Mentora Ágil*

# Prefácio

# Origem do Termo ‘Craftsman’

- Domina bem seu ofício
- Tem conhecimento especializado
- Referência de profissionalismo
- Tem orgulho do código feito



Ser rápido exige pecar  
na qualidade? VISÃO  
ERRÔNEA.

Craftsman é o único  
caminho? Quem deve  
saber?

# Não! Encontre o seu, mas conhecer algum é importante

- Ache o seu caminho!
- Depende da criticidade do segmento de software em questão
  - Como diferenciar software crítica do não-crítico?
    - Não existe resposta simples.
- Investir em padrões, disciplinas e ética: servem para todos os programadores
  - Acho minha visão superior a do fulano tal. Será sensato afirmar isso?
- É um livro para programadores e gerentes

# Padrões, disciplinas e ética... Qual objetivo exato?

- Criação de softwares robustos, tolerantes a falhas e eficientes.
- Só saber criar e fazer funcionar é insuficiente
  - Tempo e recursos são escassos

# Programas dominam o mundo!

- A maioria dos programadores ainda são amadores, ocupados demais e desorganizados
- Desastres causados por falta de profissionalismo
  - Eleições no Iowa Caucus 2020
  - Acidentes com Boing 737 Max
  - Knight Capital Group perdendo milhões
  - Aceleração involuntária em carros da Toyota nos EUA
- Erro ou falta de profissionalismo?
  - Erro é um indício de falta de profissionalismo. Minimização é ignorada.

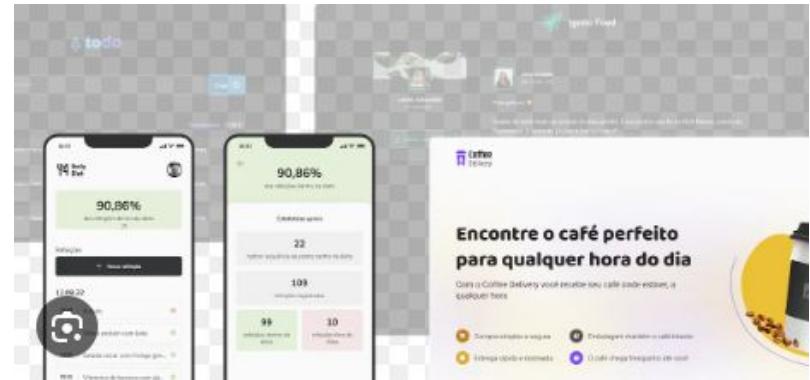
Toy Projects não  
refletem a realidade.



# Exemplos de Toy Programs



- Encontrar raízes de equação do 1º, 2º grau (fórmula de Bhaskara)
- Fatorial, Fibonacci
- Calculadora simples
- Prime numbers
- Bootcamp apps, Summer apps
- Course apps, demos
- A vida real é mais complexa
  - Lidamos com pessoas, diferentes formas de pensar e etc
- São interessantes como introdução apenas



Entrei no mercado...  
profissão? Hã?

# Muitos sequer têm ideia de que existe profissionalismo

- Muitos são jogados no mercado bastante crus, sem ideias de profissionalismo
- Não tem consciência de que estão montando castelo de cartas frágeis
  - Violação de princípios da Engenharia de Software por desconhecê-los



Falta de  
profissionalismo  
gerando desconfiança  
social e excessos de  
regulamentos.

# Quebra da confiança e regulamentações excessivas

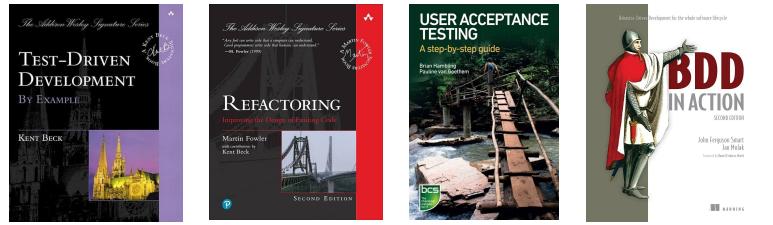
- A sociedade naturalmente vai reagir, criando regras, limitando a adoção de tecnologias.
- Redução de oportunidades.
- Fidelização e industrialização do trabalho criativo.
- Fidelizar é muito ruim. Por quê?
  - Não somos o cliente.
  - Ideia de substituição fácil
  - Negociação desequilibrada

Programar é uma  
Arte! Documentos não  
funcionam. Código  
funcionando e  
confiável é tudo.

# As três partes do livro

# Três partes

- Disciplina
  - Mais baixo nível
  - Pragmática
  - TDD, Refatoração
  - Leitura para programadores
- Padrões
  - Médio nível
  - Leitura para gerentes sobre o que esperar dos programadores
- Ética
  - Alto nível
  - Juramento que todo programador deve fazer
  - Leitura para programadores e gerentes



# Unidade I - As Disciplinas

O que é disciplina?

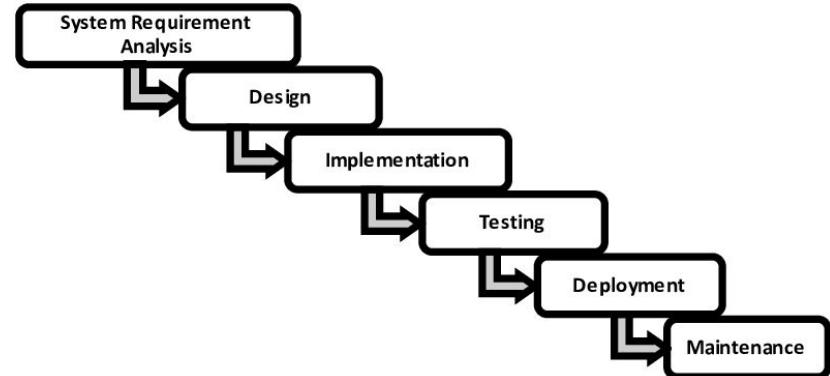
# Disciplina?

- Conjunto de regras
- Partes: Essencial e Arbitrária
- Essencial: dá aspectos e sentido
  - Cirurgia precisa ter mãos limpas
- Arbitrária: etapas, rituais
  - Cirurgião movimenta os dedos debaixo de corrente de água
  - Passível de ser questionada e demora para “cair na graça” dos profissionais

# XP (Extreme Programming)

# Antes do XP.... Waterfall

- Royce declara Waterfall em 1970. Bob considera maior erro da história. 30 anos para ser corrigida
- Em 1995 surgem: Scrum, FDD (Feature-driven Development), DSDM (Dynamic Systems Development Method) e Crystal

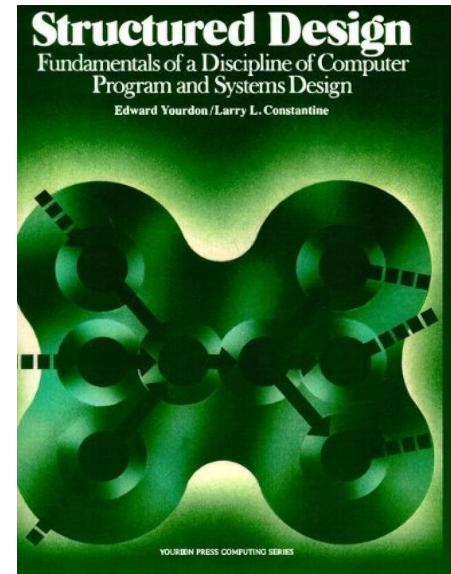


# 1970

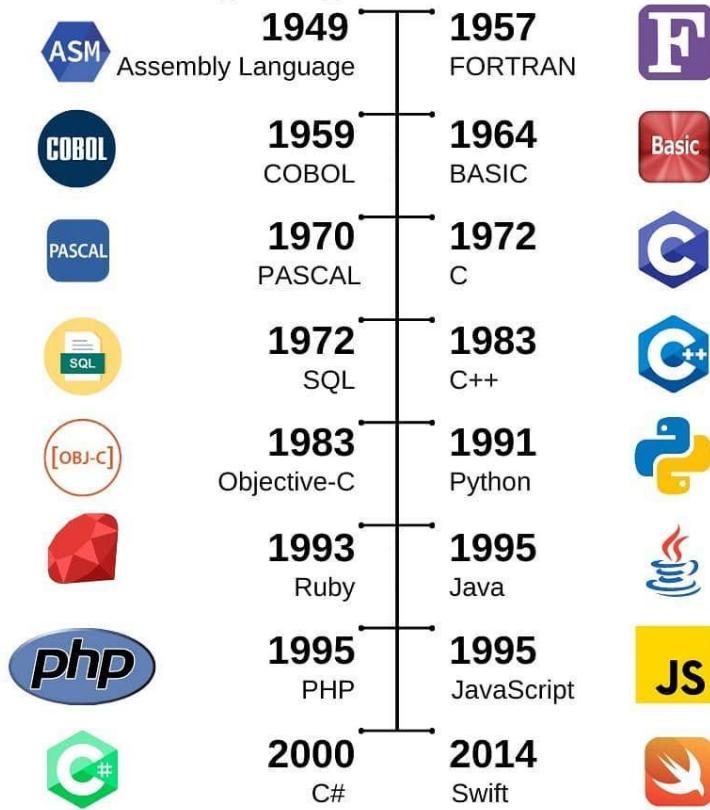
- 1960s Structured Programming
- 1970s Structured Design
- 1980s Structured Analysis



# 1970... computadores só para corporações



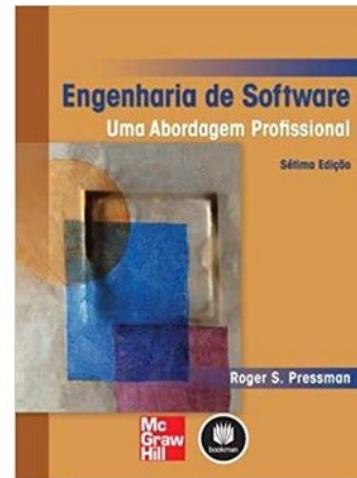
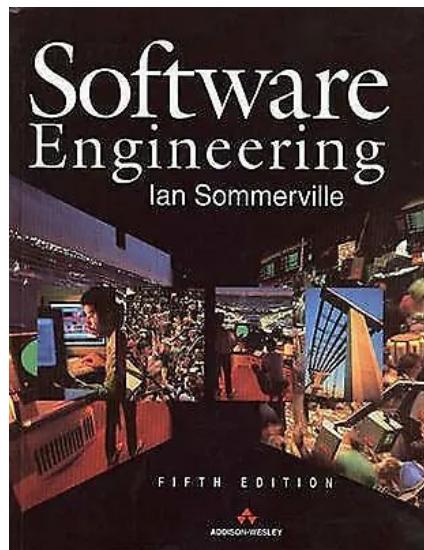
# History of Popular Programming Languages: A timeline



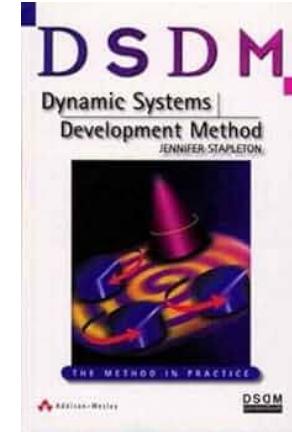
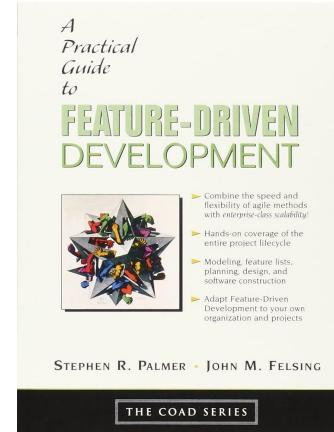
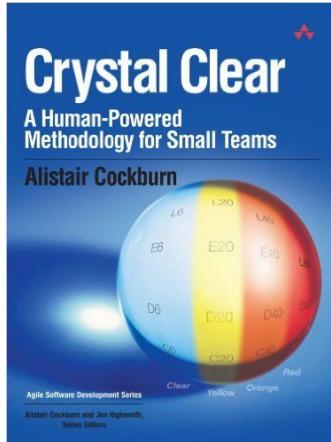
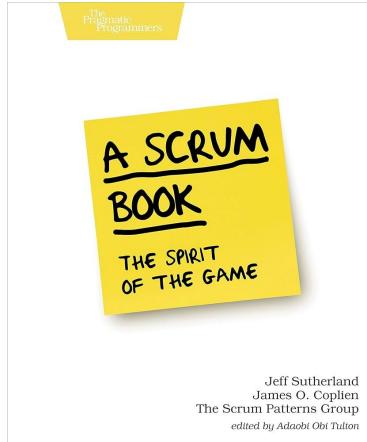
# 1995.... nasce XP e a tecnologia se populariza



1995 na Engenharia de Software Tradicional (forte influência do Waterfall, longo planejamento e design)

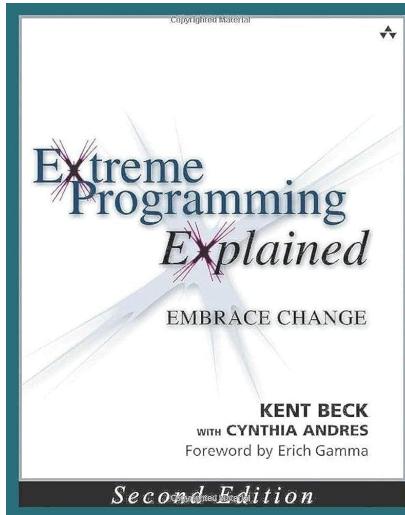


# 1995... consultores de TI subvertem a visão tradicional



Muito focados em aspectos culturais e processos mais leves. Práticas para gerentes.

1999.... Kent Beck apresenta XP, vindo de Scrum, Crystal, etc

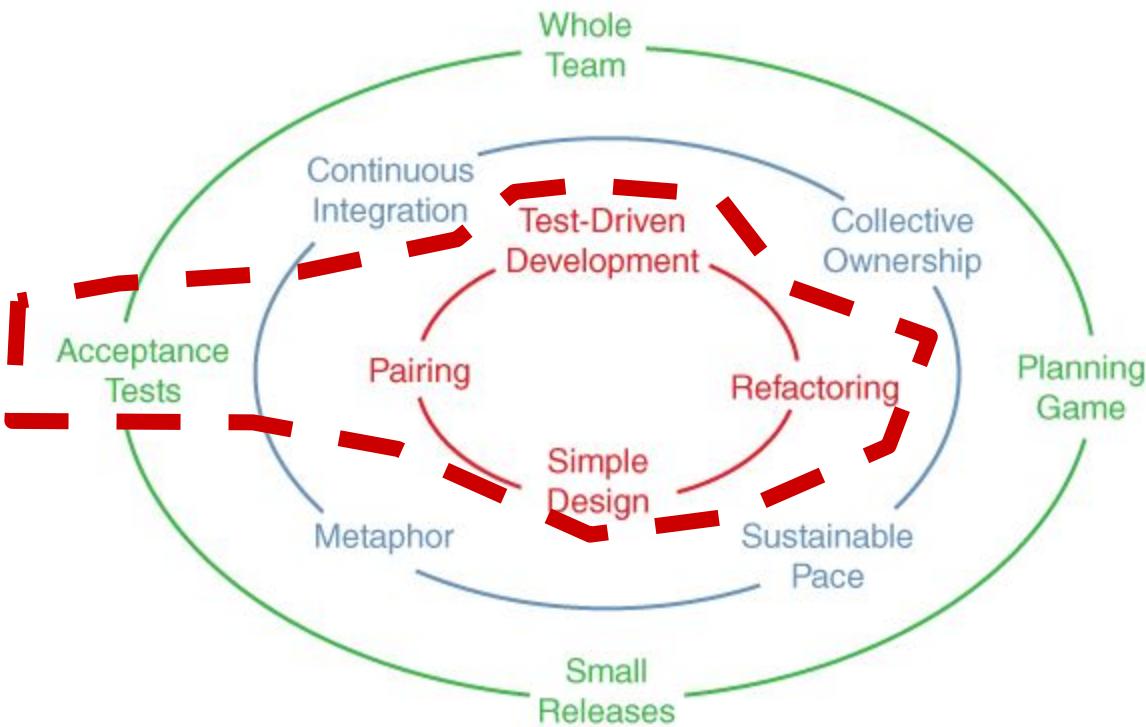


XP traz práticas de engenharia! Nada de papo de rockstar de gerenciamento! Método Ágil mais completo e bem definido que existe.

# Práticas do XP

- Definidos no Ciclo ou “Círculo” de Vida de Ron Jeffries



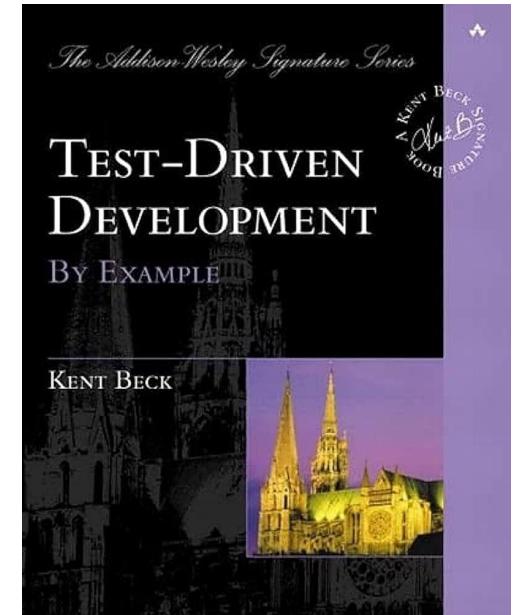


**Figure I.1** The Circle of Life: The practices of XP

TDD (Test-Driven  
Development) | A  
disciplina mais  
importante!

# TDD é a disciplina mais importante!

- Criado por Kent Beck e ensinado para Bob em 1999
- Governa o trabalho do programador
- Objetivo: Criar a suite de testes mais confiável
- Aprovado? Vá para deploy
- Tudo ou nada!
- Parte Essencial
  - Pequenos ciclos e testes
  - Teste primeiro
  - Atividades em ciclos (durando segundos)
  - Feedback imediato
- Complexa e cara! Domina tudo. Deve ser a primeira coisa a ser feita. Fazer mesmo sob pressão. Cada estrutura de código exige formato de TDD diferente. Testes precisam estar desacoplados do código produtivo.

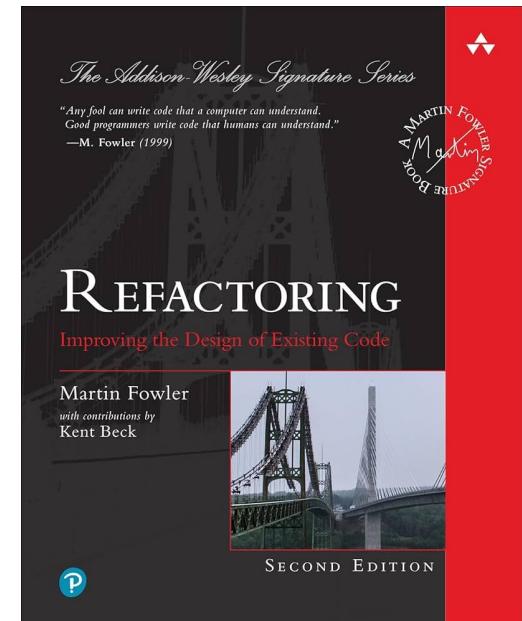


TDD para backend é  
uma coisa. Para front  
é outra. Exigem  
estratégias e formatos  
específicos.

Refactoring | feita para  
Alcançar Design  
Simples

# Disciplina como caminho para o Clean Code

- Objetivo: alcançar design simples
- Única forma de simplificar o design
- Melhorar estrutura interna sem afetar comportamento
  - Por isso a importância dos testes criados com TDD
  - Testes do TDD garantem melhorias seguras



**Refatorar sem TDD ou  
TDD sem refatorar  
são coisas  
IMPOSSÍVEIS!**

Simple Design |  
Objetivo final do  
Craftsmanship

# Analogia da Vida na Terra vs. Programação

Ecologia: Arquitetura



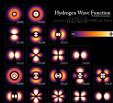
Fisiologia: SOLID, Design OO,  
Programação Funcional



Microbiologia: Design Simples



Química: Refatoração



Mecânica Quântica: TDD

# Design Simples

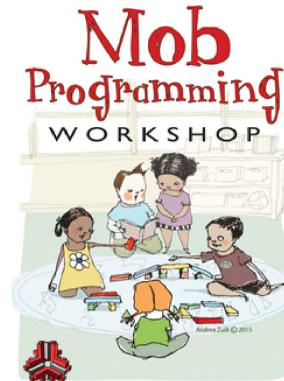
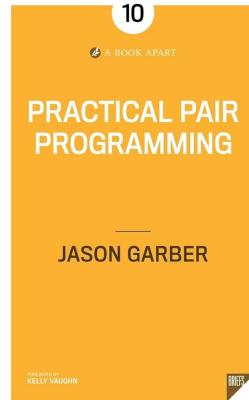
- Impossível sem refatoração
- Feito de 4 regras (p. 231): de Kent Beck e Martin Fowler
  - Passe nos testes
  - Revele intenção
  - Nada de duplicação
  - Tenha menos elementos
    - O sistema deve ter o menor número possível de classes e métodos

# Pair Programming | A Arte de se trabalhar em equipe

Menos técnica e mais  
importante de todas!

# Divisões

- Pair Programming
- Mob Programming
- Code Review
- Brainstorms
- Para quem?
  - Programadores e não-programadores
- Principal meio para compartilhar conhecimento



# Acceptance Tests | devs + business

# Enfim... a disciplina para unir devs e negócio

- O que deve ser feito? Implementar objetivos de negócio
- Como fazer? Através da Especificação de Comportamento do Sistema
  - Ocorre na forma de testes. Chamamos de Testes de Aceitação.
- Aprovado? Quer dizer que o comportamento foi entregue e assim o objetivo de negócio foi alcançado.
- Equipe de negócio deve saber como código de teste funciona (polêmico)
  - Devem saber escrever, ler e verificar a aprovação desses testes
- Na prática, a maioria dos POs sequer sabem escrever boas histórias de usuário, quem dirá testes de aceitação.

QA complementa o  
PO. Entretanto...

# Papel do QA em Acceptance Tests e além

- Muitas empresas no passado incorporaram QA para complementar o papel do PO e ajudar os desenvolvedores a se focarem em testes unitários. Porém, essa “missão” foi mal compreendida por muitas empresas.
- QA além de ficar a cargo de fazer testes manuais, também fazia funcionais e até mesmo unitários, tanto alterando quanto criando novos para aumentar cobertura de testes.
  - Violação do TDD porque escrever um teste EXIGE escrever o código de produção que o faz passar. Escrever um teste que “nasce verde” apenas para satisfazer o code coverage é visto como trabalho desonesto.
- Criou-se um mercado paralelo de tecnologia e metodologias de QA que tornou trabalho de desenvolvimento pesado. Alguns autores apontam quase para um retorno ao Waterfall, com alto consumo de recursos e tempo.

"REGRESSION TESTING"?  
WHAT'S THAT? IF IT COMPILES,  
IT IS GOOD; IF IT BOOTS UP, IT  
IS PERFECT.

- LINUS TORVALDS -



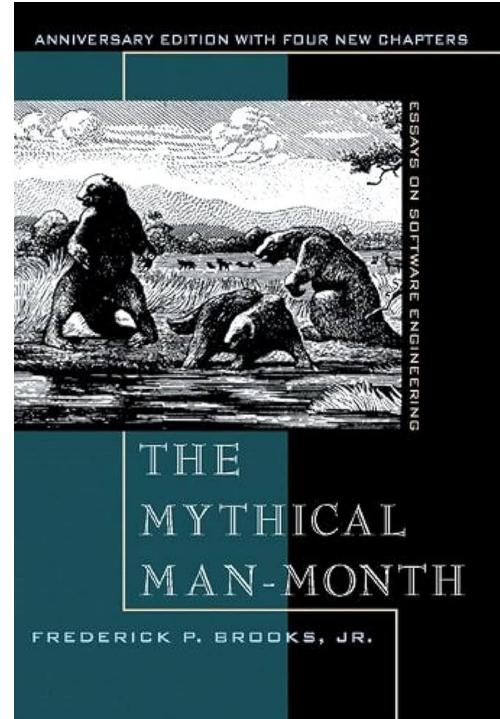
Regression Tests...  
Testes unitários,  
integrados e aceitação  
respondem.

“Also as a consequence of the introduction of new bugs, program maintenance requires far more system testing per statement written than any other programming.

Theoretically, after each fix, one must run the entire batch of test cases previously run against the system to ensure that it has not been damaged in an obscure way.

In practice, such *regression testing* must indeed approximate this theoretical idea, and it is very costly.”

Com Testes Unitários e Aceitação/Funcionais executados em Integração Contínua, perde-se sentido de executar Testes de Recessão manualmente como no passado.

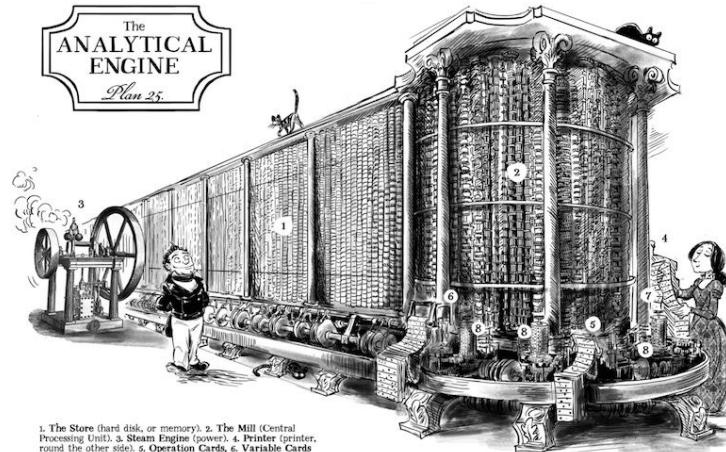


# Cap. 1 - Craftsmanship

Craftsmanship  
significa fazer algo  
“perfeito”. Importância  
da Orientação e  
Experiência.

# Os sonhos da humanidade

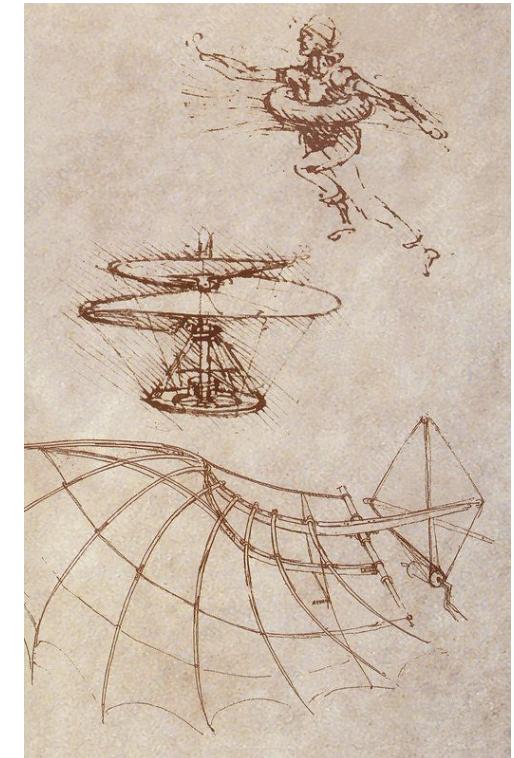
- Sonho de voar
- Sonho de computar, gerenciar e processar dados mais rapidamente



O sonho de voar

# Leonardo da Vinci

- Fundamenta o raciocínio do funcionamento do voo



# Séc 18/19 época de Invenções e Experimentações

- Ciência aeronáutica toma forma
- Pessoas se arriscando, muitas mortes
- Planador de Cailey



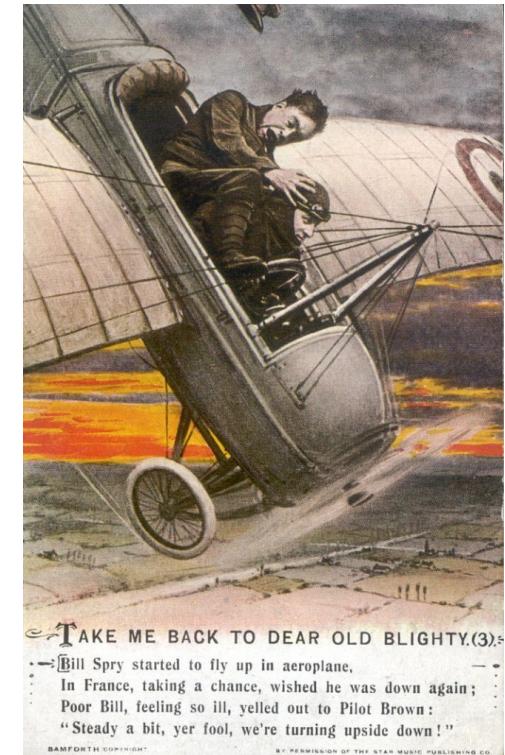
Princípios de voo  
foram dominados.

Mas a técnica era mal  
compreendida.

Primeira Guerra e  
muito mais mortes.

# Combatentes aéreos morrendo em batalhas? Minoria.

- A maioria morria por falta de técnicas de voo
- 64% das mortes ocorriam fora de combate
- Como voar? Isso não era respondido.



TAKE ME BACK TO DEAR OLD BLIGHTY.(3)

Bill Spry started to fly up in aeroplane;  
In France, taking a chance, wished he was down again;  
Poor Bill, feeling so ill, yelled out to Pilot Brown:  
"Steady a bit, yer fool, we're turning upside down!"

BAMFORTH COPYRIGHT

BY PERMISSION OF THE STAR MUSIC PUBLISHING CO.

# Capitão Sullenberger salva vidas em 2009

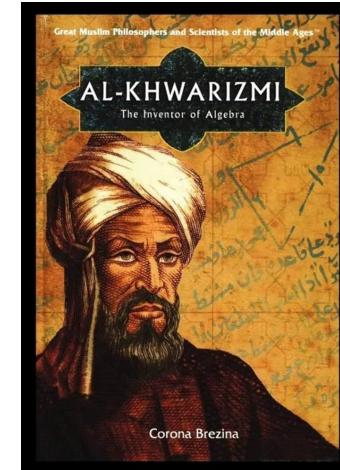
- Domínio da arte de voar
- Grande experiência na Aeronáutica
- Exemplo de craftsman



O sonho da  
computação

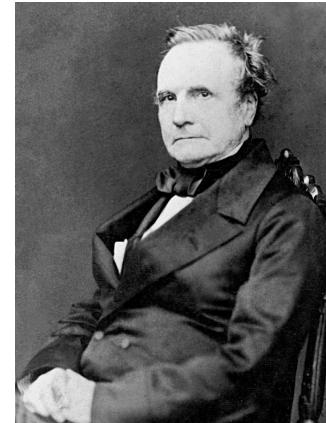
# 2000 anos atrás

- Máquina de Anticitera
- Gregos, Árabes e Indianos tinham intuições de algoritmos
  - Cálculos cosmológicos



# Séc 19 Babbage e Ada iniciam uma jornada séria

- Babbage apresenta a Máquina Analítica
  - Era digital, baseado em mecânica
- Muito difícil de construir, cara. A metalurgia estava atrasada.
- Ada afirma que números podem representar coisas na vida real



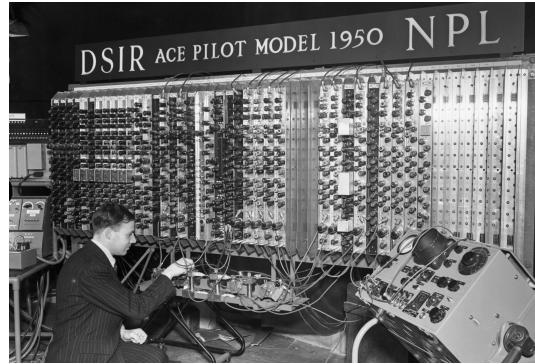
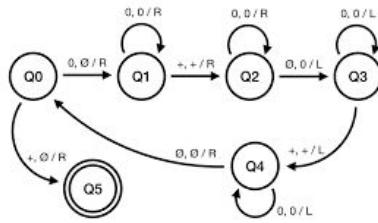
# Representar coisas na vida real? Essa frase torna Ada a primeira programadora da História!

Number of Operation.	Nature of Operation.	Statement of Results.			Working Variables.										Result Variables.			
		Variables acted upon.	Variables receiving results.	Indication of change in the value on any variable.	Data.													
					V <sub>1</sub>	V <sub>2</sub>	V <sub>n</sub>	V <sub>4</sub>	V <sub>5</sub>	V <sub>6</sub>	V <sub>8</sub>	V <sub>9</sub>	V <sub>10</sub>	V <sub>11</sub>	V <sub>12</sub>	V <sub>13</sub>	V <sub>14</sub>	V <sub>15</sub>
1	$\times$	V <sub>2</sub> $\times$ V <sub>3</sub>	V <sub>4</sub> , V <sub>6</sub> , V <sub>16</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	1	2	n	2n	2n	2n								
2	-	V <sub>4</sub> - V <sub>2</sub>	2V <sub>4</sub>	$\begin{cases} V_{12} = 2V_{14} \\ V_{14} = V_{14} \end{cases}$	1	...	...	2n-1	2n									
3	+	V <sub>5</sub> + V <sub>1</sub>	2V <sub>5</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = 2V_5 \end{cases}$	1	...	...	...	2n+1									
4	+	V <sub>4</sub> - 2V <sub>4</sub>	IV <sub>11</sub>	$\begin{cases} V_{12} = 0V_4 \\ V_{14} = V_{14} \end{cases}$	...	...	0	0	...	...	...	...	...	...	2n-1	2n+1		
5	+	V <sub>11</sub> - V <sub>2</sub>	2V <sub>11</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	...	2	...	...	...	...	...	...	...	...	1	2n-1	2n+1	
6	-	V <sub>13</sub> - 2V <sub>11</sub>	IV <sub>12</sub>	$\begin{cases} V_{12} = 0V_{11} \\ V_{14} = V_{14} \end{cases}$	...	...	...	...	...	...	...	...	...	0	...	-\frac{1}{2}	2n-1	2n+1
7	-	V <sub>12</sub> - V <sub>1</sub>	IV <sub>10</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	1	n	...	...	...	...	...	...	...	n-1				
8	+	V <sub>2</sub> + 0V <sub>2</sub>	IV <sub>7</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	...	2	...	...	...	...	2							
9	+	V <sub>6</sub> - V <sub>2</sub>	2V <sub>11</sub>	$\begin{cases} V_{12} = 2V_{11} \\ V_{14} = V_{14} \end{cases}$	...	...	...	...	2n	2	...	...	...	$\frac{2n}{2} = A_1$				
10	$\times$	V <sub>21</sub> $\times$ V <sub>11</sub>	IV <sub>12</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	B <sub>1</sub>	$\frac{2}{2} = B_1 A_1$	...	...	...	...	...	...	...	$\frac{2n}{2} = A_1$	B <sub>1</sub>	$\frac{2n}{2} = B_1 A_1$		B <sub>1</sub>
11	+	V <sub>12</sub> - V <sub>13</sub>	2V <sub>13</sub>	$\begin{cases} V_{12} = 0V_{13} \\ V_{14} = V_{14} \end{cases}$	...	-2	$\frac{2n-1}{2} + B_1$	$\frac{2n}{2}$	...	...	...	...	...	0	{- \frac{1}{2} \cdot \frac{2n-1}{2} + B_1 \cdot \frac{2n}{2}}			
12	-	V <sub>10</sub> - V <sub>1</sub>	IV <sub>10</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	1	...	...	...	...	...	...	...	n-2					
13	-	V <sub>6</sub> - V <sub>1</sub>	IV <sub>8</sub>	$\begin{cases} V_{12} = 2V_{16} \\ V_{14} = V_{14} \end{cases}$	1	...	...	...	2n-1									
14	-	V <sub>1</sub> + IV <sub>2</sub>	2V <sub>7</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	1	...	...	...	3									
15	+	2V <sub>6</sub> - 2V <sub>2</sub>	V <sub>8</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	$\frac{2n-1}{2}$	...	...	2n-1	3	$\frac{2n-1}{3}$								
16	$\times$	V <sub>8</sub> $\times$ V <sub>11</sub>	V <sub>11</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	$\frac{2n}{2} = B_1$	...	...	...	0	...	...	...	$\frac{2n}{2} \cdot \frac{2n-1}{3}$					
17	-	V <sub>6</sub> - V <sub>1</sub>	V <sub>6</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	2n-2	1	...	...	2n-2									
18	+	V <sub>1</sub> + IV <sub>2</sub>	V <sub>7</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	3+1=4	1	...	...	...	4								
19	+	2V <sub>6</sub> - 3V <sub>7</sub>	V <sub>9</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	$\frac{2n-2}{4}$	...	...	2n-2	4	$\frac{2n-2}{4}$	...	$\left\{ \frac{2n}{2}, \frac{2n-1}{3}, \frac{2n-2}{3} \right\}$						
20	$\times$	V <sub>9</sub> $\times$ V <sub>11</sub>	V <sub>11</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	$\frac{2n}{2} = A_2$	...	...	...	0	...	...	...						
21	$\times$	V <sub>22</sub> $\times$ V <sub>11</sub>	V <sub>10</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	$B_1 \cdot \frac{2}{2} = B_2 A_2$	...	...	...	...	...	...	...	0	B <sub>1</sub> A <sub>2</sub>			B <sub>2</sub>	
22	+	V <sub>12</sub> + 2V <sub>13</sub>	IV <sub>13</sub>	$\begin{cases} V_{12} = 0V_{13} \\ V_{14} = V_{14} \end{cases}$	$A_0 + B_1 A_1 + B_2 A_2$	...	...	...	...	...	...	...	0	0	$\left\{ A_2 + B_1 A_1 + B_2 A_2 \right\}$			
23	-	2V <sub>10</sub> - V <sub>1</sub>	V <sub>10</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	$n-3 = (1)$	1	...	...	...	...	...	...	n-3					
Here follows a repetition of Operations thirteen to twenty-three.																		
24	+	V <sub>13</sub> + 0V <sub>24</sub>	V <sub>24</sub>	$\begin{cases} V_{12} = 0V_{13} \\ V_{14} = V_{14} \end{cases}$	B <sub>7</sub>	...	...	...	...	...	...	...	...					B <sub>7</sub>
25	+	V <sub>1</sub> + V <sub>3</sub>	V <sub>3</sub>	$\begin{cases} V_{12} = V_{12} \\ V_{14} = V_{14} \end{cases}$	$n+1=4+1=5$	1	n+1	...	...	0	0	...	...					

by a Variable-card.  
by a Variable-card.

# Turing em 1936

- Criou as ideias de máquina de estados finitos, linguagem simbólica
- Teria sido também primeiro engenheiro de software e hardware a formalizar matematicamente seus métodos
- Simulação mecânica: próximo slide





Com a escassez de chips... Bem, tomara que  
seja somente por curiosidade

“The idea of a digital computer is an old one. Charles Babbage, Lucasian Professor of Mathematics at Cambridge from 1828 to 1839, planned such a machine, called the Analytical Engine, [...]” (Turing, 1950)

Turing faz o primeiro  
chamado por  
profissionais  
programadores (Bob)

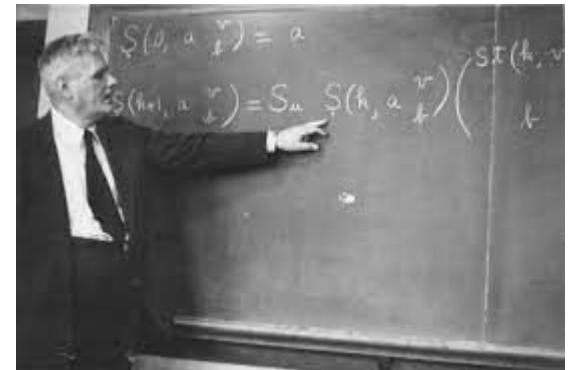
Church cria o Cálculo  
Lambda. Início da  
Programação  
Funcional.

# Alonzo Church, prof. de Turing

- Publica o Cálculo Lambda em 1930
- A programação funcional estava agora fundamentada

Symmetric relation:		
$r + s$	$\rightleftharpoons$	$s + r$ (COMM)
$(r + s) + t$	$\rightleftharpoons$	$r + (s + t)$ (ASSO)
$\lambda x^A.(r + s)$	$\rightleftharpoons$	$\lambda x^A.r + \lambda x^A.s$ (DIST <sub>i<i></i></sub> )
$(r + s)t$	$\rightleftharpoons$	$rt + st$ (DIST <sub>e<i></i></sub> )
$\pi_{A \Rightarrow B}(\lambda x^A.r)$	$\rightleftharpoons$	$\lambda x^A.\pi_B(r)$ (DIST <sub>e<i></i></sub> )
If $r : A \Rightarrow (B \wedge C)$ , $\pi_{A \Rightarrow B}(r)s$	$\rightleftharpoons$	$\pi_B(rs)$ (DIST <sub>ee</sub> )
$rst$	$\rightleftharpoons$	$r(s + t)$ (CURRY)
If $A \equiv B$ , $r$	$\rightleftharpoons$	$r[A/B]$ ( $\alpha$ -TYPES)
If $r =_\alpha s$ , $r$	$\rightleftharpoons$	$s$ ( $\alpha$ -TERMS)
Reductions:		
If $s : A$ , $(\lambda x^A.r)s$	$\rightarrow$	$r[s/x]$ ( $\beta$ )
If $r : A$ , $\pi_A(r + s)$	$\rightarrow$	$r$ ( $\pi_n$ )
If $r : A$ , $\pi_A(r)$	$\rightarrow$	$r$ ( $\pi_1$ )

Table 2: Operational semantics



Em 1965 ninguém  
aprendia a programar  
em universidade  
(Bob).

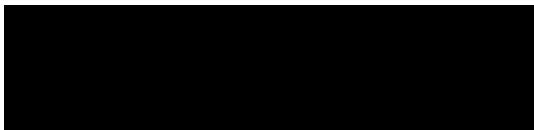
# Primeiros programadores não-cientistas

- Os programadores de aplicações comerciais comuns aprendiam a programar através de manuais dos primeiros computadores digitais eletrônicos
- O “negócio” era a escola deles
- Primeiros graduados em Ciência da Computação chegam no fim da década de 1970

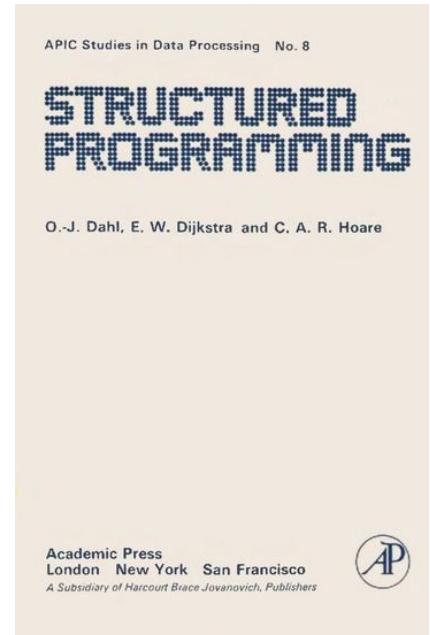
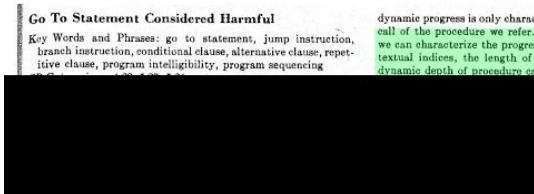


# 1968: Dijkstra e Hoare fundam a Prog. Estruturada

- Começa a se preocupar com códigos confusos (Spaghetti Code)
- Inicia a noção de estética e boas práticas
- Repudia o uso indiscriminado de goto
  - Goto: saltos incondicionais
- Junto a Hoare, criam a Programação Estruturada

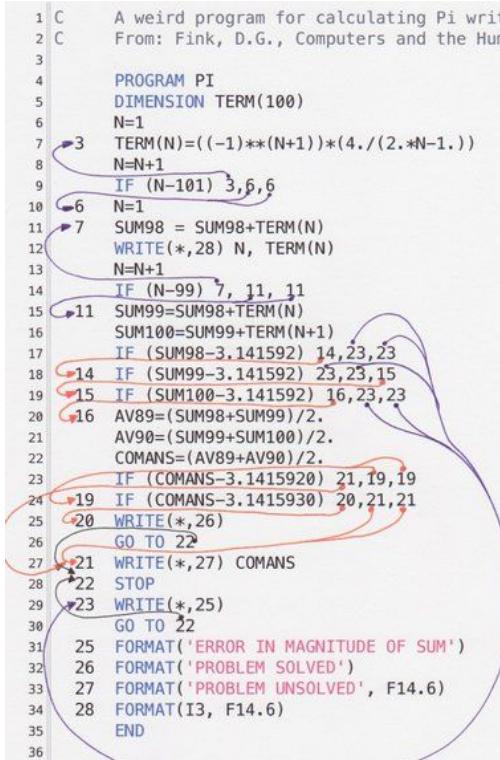


Edgar Dijkstra: Go To Statement Considered Harmful



# Spaghetti Code

```
1 C      A weird program for calculating Pi written in Fortran.  
2 C      From: Fink, D.G., Computers and the Human Mind, Anchor Books, 1966.  
3  
4      PROGRAM PI  
5      DIMENSION TERM(100)  
6      N=1  
7      3 TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))  
8      N=N+1  
9      IF (N>101) 3,6,6  
10     6 N=1  
11     7 SUM98 = SUM98+TERM(N)  
12     WRITE(*,28) N, TERM(N)  
13     N=N+1  
14     IF (N>99) 7, 11, 11  
15     11 SUM99=SUM98+TERM(N)  
16     SUM100=SUM99+TERM(N+1)  
17     IF (SUM98-3.141592) 14,23,23  
18     14 IF (SUM99-3.141592) 23,23,15  
19     15 IF (SUM100-3.141592) 16,23,23  
20     16 AV89=(SUM98+SUM99)/2.  
21     AV90=(SUM99+SUM100)/2.  
22     COMANS=(AV89+AV90)/2.  
23     IF (COMANS-3.1415920) 21,19,19  
24     19 IF (COMANS-3.1415930) 20,21,21  
25     20 WRITE(*,26)  
26     GO TO 22  
27     21 WRITE(*,27) COMANS  
28     22 STOP  
29     23 WRITE(*,25)  
30     GO TO 22  
31     25 FORMAT('ERROR IN MAGNITUDE OF SUM')  
32     26 FORMAT('PROBLEM SOLVED')  
33     27 FORMAT('PROBLEM UNSOLVED', F14.6)  
34     28 FORMAT(I3, F14.6)  
35     END  
36
```

A diagram illustrating the complex control flow of the Fortran code. Purple arrows show various jump points and loops. A large circle encloses the main loop from line 7 to line 22, starting with the assignment of TERM(N) at line 3 and ending with the STOP command at line 22. Other arrows point from lines 11, 15, and 16 back to line 7, and from line 20 back to line 22. There are also several local loops and assignments within the main loop structure.

# Goto Hell

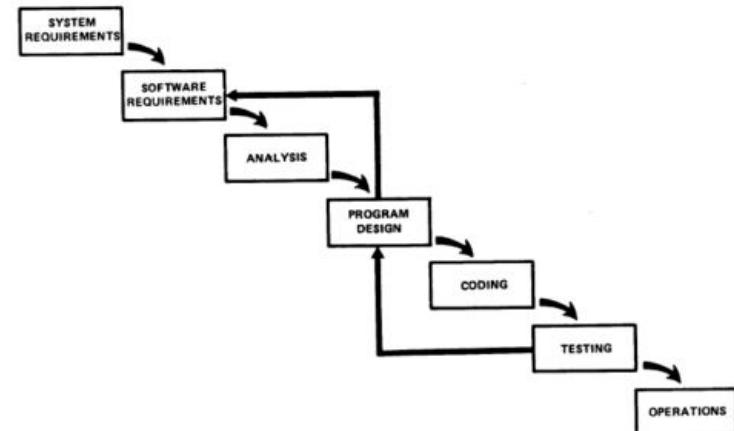
```
10 REMARK THIS PROGRAM COMPUTES INTEREST
20 REMARK EARNED ON SAVINGS
30 PRINT "ENTER AMOUNT SAVED, 0 IF NO MORE DATA"
40 INPUT X
50 IF X = 0 THEN 160
60 ON X/1000 GOTO 90,110,130,130
70 PRINT "VALUE OUT OF RANGE"
80 GOTO 40
90 LET I = .04*X
100 GOTO 140
110 LET I = .045*X
120 GOTO 140
130 LET I = .05*X
140 PRINT "ANNUAL INTEREST ON $">X;" IS $">I
150 GOTO 40
160 END
```

Milhões começam a ser  
queimados. Iniciam-se  
pedidos de ajuda.

Solução? Royce aponta  
Waterfall.

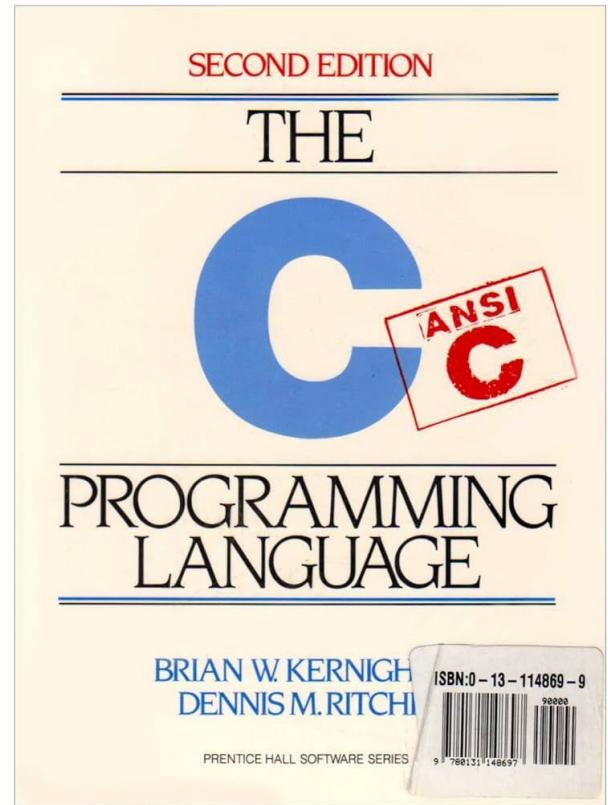
# 1970: Waterfall vai nos salvar!?

- Royce acreditava que a culpa da bagunça do software era somente porque não existia um processo de desenvolvimento organizado e disciplinado
- A ideia de formalismo era considerada “cool”
- Recém formados de Ciência da Computação tinham sido influenciados pelo trabalho de Royce
- Não resolve nada
  - Cientistas e especialistas distantes do cliente
  - Cliente recebia software misterioso com manuais técnicos gigantescos
- Bob: sentiam orgulho de entregar complexidade.



# 1972: C e Unix surgem

- Mudam a indústria do software para sempre
- Torna goto obsoleto
- Permite técnicas sofisticadas com ponteiros
- Ideia do KISS
  - Primeiro senso de design de código
  - Unix introduziu a aplicação de separação de responsabilidades a nível de Infra de Sistema Operacional e forte reuso com Pipes.



## Unix Pipes (Shell Pipes)

```
$ lscpu | grep "CPU op-mode(s)" | grep "16"
$ echo "$?"
1
$ lscpu | grep "CPU op-mode(s)" | grep "64"
CPU op-mode(s):          32-bit, 64-bit
$ echo "$?"
0
```

# 2001: Manifesto Ágil toma forma

- 12 princípios foram escritos pelas lideranças de consultorias de TI
- Fim do Waterfall e reinvenção da programação
- Programar volta a ter protagonismo
  - Waterfall enfatizou disciplina em processos e apelo exagerado ao Design “Perfeito”



2001... foi ontem  
praticamente perto de  
outras invenções da  
humanidade.

# Tecnologia avança, problemas de profissionalismo pioram

- Bob diz que a tecnologia avançou assustadoramente
- Temos mais núcleos de CPU, esbanjamos memória RAM
- Compiladores melhores, linguagens melhores.
- Entretanto, os programadores em imensa maioria estão desorganizados, ocupados demais, “eternos iniciantes”
- Muito código mal escrito, nenhum senso de design simples
- Faculdades e cursos não acompanham a capacitação necessária
  - Ela é difícil. Faltam verdadeiros orientadores e pessoas experientes.
- São os pilotos de caça de 1930 ignorando a descoberta recente dos manuais de “Como voar melhor”.

Faltam profissionais  
craftsmanship.

“Despite this rapid growth, there is an insufficient number of professional developers with the right tech skills to meet the needs of the job market. As of 2021, the tech talent shortage amounted to 40 million qualified developers and engineers worldwide, **expected to reach 85.2 million** by 2030.” ([hatchworks.com](https://hatchworks.com))



# História do Design Orientado a Objeto, RUP, UML, Agile e TDD (Fábio Akita e Uncle Bob)

# LIVE: DISCUTINDO ORIENTAÇÃO A OBJETOS

com Robert Martin  
(Uncle Bob)

#programming



# OBJECT-ORIENTED ANALYSIS AND DESIGN

WITH APPLICATIONS

GRADY BOOCHE

SECOND EDITION

Object-oriented analysis and design, a discipline that has revolutionized software development, allows business professionals to build systems that are easier to maintain, reuse, and extend. This book provides a clear, step-by-step guide to the object-oriented paradigm, showing how it can be used to build systems that are more reliable, efficient, and effective. It covers the basic principles of object-oriented analysis and design, including classes, objects, inheritance, and polymorphism. It also explores advanced topics such as distributed systems, real-time systems, and web-based systems. The book is designed for both novice and experienced programmers, and includes numerous examples and exercises to help readers apply the concepts to real-world problems.



# Elementos Importantes

- Orientação a Objetos nasceu tendo em vista Interfaces Gráficas e não representações da Objetolândia;
- Existia uma crença de que Diagramas de Classe (Diagramas em geral) pudessem expressar completamente o software
  - Podem atuar apenas como esboço
  - Existe uma enorme tendência de se implementar mais classes e métodos do que deveria, violando o Clean Code
  - Crença persistente em muitas empresas
- Diagrama não compila!
- Visão antes de 1995: deixar todas as regras definidas e diagramas prontos e assim iniciar o projeto. Bob diz que naquela época se acreditava que isso permitia criação de projetos poderosos. Isso conduz ao Waterfall.
- Somente após 1995 isso começou a mudar

# LIVE: DISCUTINDO ORIENTAÇÃO A OBJETOS

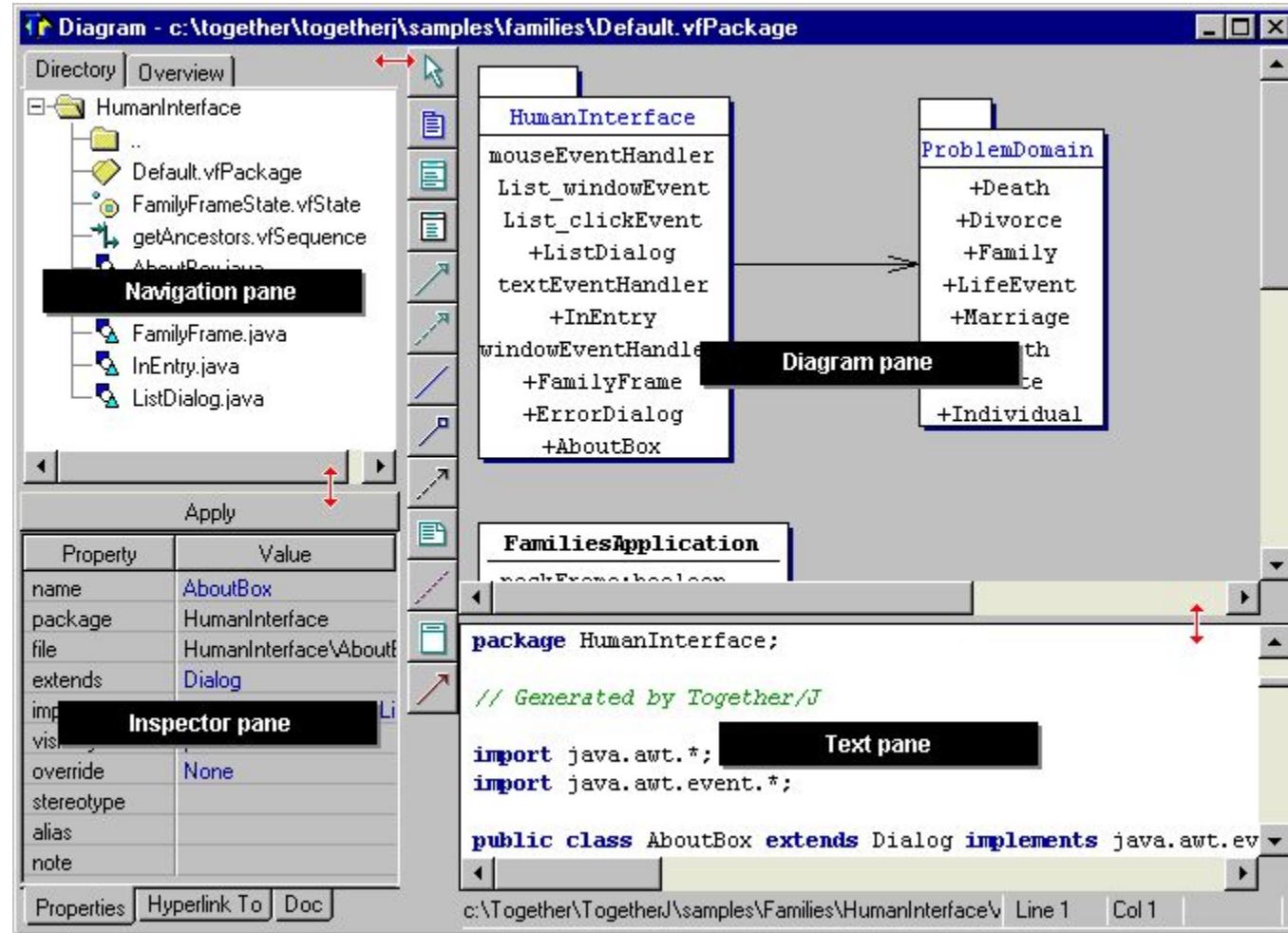
com Robert Martin  
(Uncle Bob)

#programming



# Elementos Importantes

- Extensas etapas de modelagem eram tão complicadas que tiveram que criar o Model Driven Development como atalhos para os processos de Análise, Design e Implementação
- Model Driven Development: transformar Diagramas de Classe em Classes Java ou TADs em C
- Nessa abordagem, só se pode alterar Diagramas. A modelagem precisava ser quase perfeita. Segundo Bob, isso não funcionou.
  - Surge o TogetherJ
  - Criador sai do projeto. Programadores ucranianos entram no projeto e criam o IntelliJ.
- Havia apelo por UML, RAD, Visual Basic (programar em Basic usando UML)



vulp\_example - simple.kt [vulp\_example.main]

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

vulp\_example > src > main > kotlin > simple.kt

simple.kt

BrowserProductionRun in continuous mode

Gradle

Project

vulp\_example ~/ideaProjects/vulp\_example

- > gradle
- > idea
- > build
- > gradle
- < src
  - < main
    - < kotlin
      - simple.kt
    - > resources
    - > test
      - build.gradle.kts
      - gradle.properties
      - gradlew
      - gradlew.bat
      - settings.gradle.kts
- > External Libraries
- < Scratches and Consoles

Fun main() {  
 console.log("Hello, \${greet()}")  
}  
fun greet() = "Vulphere"

Structure

Favorites

Event Log

Gradle sync finished in 6 s 140 ms (11 minutes ago)

5.25 LF UTF-8 4 spaces

# Elementos Importantes

- A razão por adotar Waterfall: escrever software e fazer funcionar era doloroso
- Compiladores eram ruins: podiam-se levar horas
- A ideia para solucionar era poder se antecipar ao máximo com etapa de modelagem extensa e assim evitar reescrita do código
- A partir de 1990: computadores começam a ficar mais rápidos
- Desenvolvedores de SmallTalk redescobrem uma prática alternativa...

# LIVE: DISCUTINDO ORIENTAÇÃO A OBJETOS

com Robert Martin  
(Uncle Bob)

#programming



# Elementos Importantes

- Programadores SmallTalk descobrem o poder do TDD e Refatoração com computadores mais rápidos, com ciclos de compilação e entrega mais rápida
- Galera de SmallTalk migra para Java no fim de 1990
- Inicia-se o questionamento: por que fazer etapas extensas de análise e modelagem em vez de fazer um pouco de código e vê-lo funcionar?
- Nasce a ideia de Ciclos Iterativos

# Por que Waterfall falhou exatamente?

A ideia do software é poder ser mais facilmente alterado. Ser capaz de comportar mudanças. Elas precisam e vão acontecer.

Software = flexível, fácil de alterar

Waterfall cria uma tendência de tornar difícil de alterar ou muito demorado. O software se torna IMUTÁVEL.

O Agile nasceu!

# Authors / Agile Manifesto

From sources across the web



Jim Highsmith



Robert Cecil Martin



Andy Hunt



Jon Kern



Arie van Bennekum



Kevin R. Lowell



Alistair Cockburn



Arie van Bennekum



Ron Jeffries



Brian Marick



Larry Apke



Mike Beedle

1962–2018



Jeff Sutherland



Ward Cunningham



Stephen J. Mellor



Yael Dubinsky



Muitos acreditam que o Agile não mudou  
nada!

# Persistência do medo de alterar

- Segundo Bob, o medo de alterar o software ainda persiste, mesmo depois do Agile ter criado formas de lidar com isso;

# LIVE: DISCUTINDO ORIENTAÇÃO A OBJETOS

com Robert Martin  
(Uncle Bob)

#programming



# Como combater o medo?

- Através de uma suite (conjunto) de testes confiável e rápida
- Necessário coragem!
- Muitos programadores afirmam: não dá tempo, é muito complexo. Não tem retorno. O cliente não pagou por testes unitários. A empresa não exige. Faremos os testes mínimos só para satisfazer a cobertura de teste.

Kent Beck apresenta XP na conferência de Design Patterns (GoF) em 1995. Iniciam-se os movimentos do Manifesto Agile, que irá ocorrer em 2001.

# LIVE: DISCUTINDO ORIENTAÇÃO A OBJETOS

com Robert Martin  
(Uncle Bob)

#programming



# Manifesto para Desenvolvimento Ágil de Software

Estamos descobrindo maneiras melhores de desenvolver software, fazendo-o nós mesmos e ajudando outros a fazerem o mesmo. Através deste trabalho, passamos a valorizar:

**Indivíduos e interações** mais que processos e ferramentas  
**Software em funcionamento** mais que documentação abrangente  
**Colaboração com o cliente** mais que negociação de contratos  
**Responder a mudanças** mais que seguir um plano

Ou seja, mesmo havendo valor nos itens à direita,  
valorizamos mais os itens à esquerda.

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  
Dave Thomas

Agile era realmente novidade? Em 1940 já existiam algumas práticas semelhantes a TDD

# LIVE: DISCUTINDO ORIENTAÇÃO A OBJETOS

com Robert Martin  
(Uncle Bob)

#programming





Jerry Weinberg, um dos engenheiros da NASA, afirmou que a prática na época era de se programar em pequenos ciclos. Não se podia compilar grandes programas devido às limitações dos cartões perfurados. Cada instrução era testada uma após outra até que todo o programa estivesse testável.

Artigo completo:

<https://daveschinkel.medium.com/tdd-on-punch-cards-f17c25282fa4>

# Em 1970 tudo mudou... segundo Bob

- Acredita que os primeiros programadores começaram a envelhecer e se aposentar
- Programação era aprendida “no ambiente produtivo”. Não se ensinava
- Eram vistos como “velhos leitores de manuais de computadores”
- Em 1970 chegam os primeiros bacharéis de Ciência da Computação nos EUA
  - Público majoritariamente masculino (surge aqui o persistente desequilíbrio entre sexos na área)
  - Antes de 1970, os escritórios eram dominados por programadoras
  - De repente, a programação foi dominada por homens jovens
- Iniciou-se a necessidade de colocar processos, formalismos e etc
- Perdemos os fundamentos do Agile. Por isso vivemos agora o Segundo Advento.

Tem mais? Sim. Muito mais.

# Segundo Advento do Agile e novas práticas

- Temos o Mapeamento de História de Usuário
  - Promovendo Indivíduos e Interações e Colaboração com o Cliente em ciclos iterativos
- DevOps como continuação do Agile
- Testes de Aceitação: de acordo com Freeman (2009), é por onde devemos começar antes dos ciclos do TDD
- BDD e ATDD: variações do TDD focados em linguagem de negócio