

# Análise e Levantamento de Requisitos de Software

Análise do Software

**Aluno:** Rafael Magno da Silva

**Registro Acadêmico:** 2232801

**Turma:** 002.0067.B.20223

**E-mail:** raphael.magno.rj@gmail.com

## Sumário

INTRODUÇÃO.....	3
DESENVOLVIMENTO.....	8
1 – Década de 50.....	8
2 – Década de 60: Engenharia de Software e a Crise do Software.....	9
3 – Década de 70: Modelo em Cascata.....	20
4 – Década de 80.....	24
4.1 - Modelo de Desenvolvimento Incremental.....	25
4.2 - Engenharia de software orientada a reúso.....	27
4.3 - Modelo Espiral de Boehm.....	28
5 – Década de 90 e o Processo Unificado/RUP.....	30
6 – Anos 2000.....	36
6.1 - Desenvolvimento Ágil de Software vs. Pós Crise de Software.....	36
6.2 - Extreme Programming – XP.....	39
6.3 - Método Scrum.....	44
6.4 - Método Lean.....	48
6.5 - Método Kanban.....	51
CONCLUSÃO.....	54
REFERENCIAS BIBLIOGRÁFICAS.....	57

## NOTA DE ESCLARECIMENTO

Primeiramente, gostaria de registrar a minha completa **INSATISFAÇÃO COM ESSE** complicado **SISTEMA** da Descomplica!

Ato contínuo; o presente trabalho foi inspirado pelo tema do Pensar & Responder. Realizou-se assim, uma **COMPILAÇÃO DE DADOS TÉCNICOS** eivados de dois livros, e, da Wikipédia. Trata-se portanto, de uma pseudo-coletânea de estudo de material didático citado nas referências bibliográficas, objetivando traçar um lapso temporal de desenvolvimento do software e da engenharia de software com intermeio da Crise do Software.

Ocorre que, o estudo fora realizado no Writer, e, salvo em PDF. No entanto, o sistema da faculdade **NÃO ACEITA NENHUM ARQUIVO DE PRODUÇÃO DE TEXTO!** Horrível! =/

Honestamente... Essa é a última vez que parei para estudar além do material didático! Nunca mais irei perturbar esse *glorioso sistema* da Descomplica baseado na **LEI DO MENOR ESFORÇO!**

**Me desculpem!!!** De agora em diante, irei me ater a escrever meia dúzia de frases para ganhar os pontos e acabou... É assim que vocês querem, certo?

Assim será feito!

**#MassiveFail #Shame #Disappointed**

# INTRODUÇÃO

A presente pesquisa acadêmica visa descrever de forma sucinta possível, sem no entanto, perder o caráter histórico, e, absolutamente relevante sobre os eventos vividos para culminar a Crise do Software ocorrido na década de 60-70, interpolando-se a este, a evolução do conceito de software, linguagens de programação; e, a evolução conceito de engenharia de software e modelos de processo de software até os dias atuais. Frise-se que, a presente pesquisa se desenvolveu na forma de **compilação de dados técnicos**.

Ou seja, a intenção da presente é corroborar com o ensino universitário e ampliar o conhecimento do tema adjunto ao material didático alocado nas referências bibliográficas. Portanto, não se coaduna refletir-se sobre o “plágio” em quaisquer parágrafos, uma vez que a presente é, em verdade, uma compilação técnica para estudos universitários.

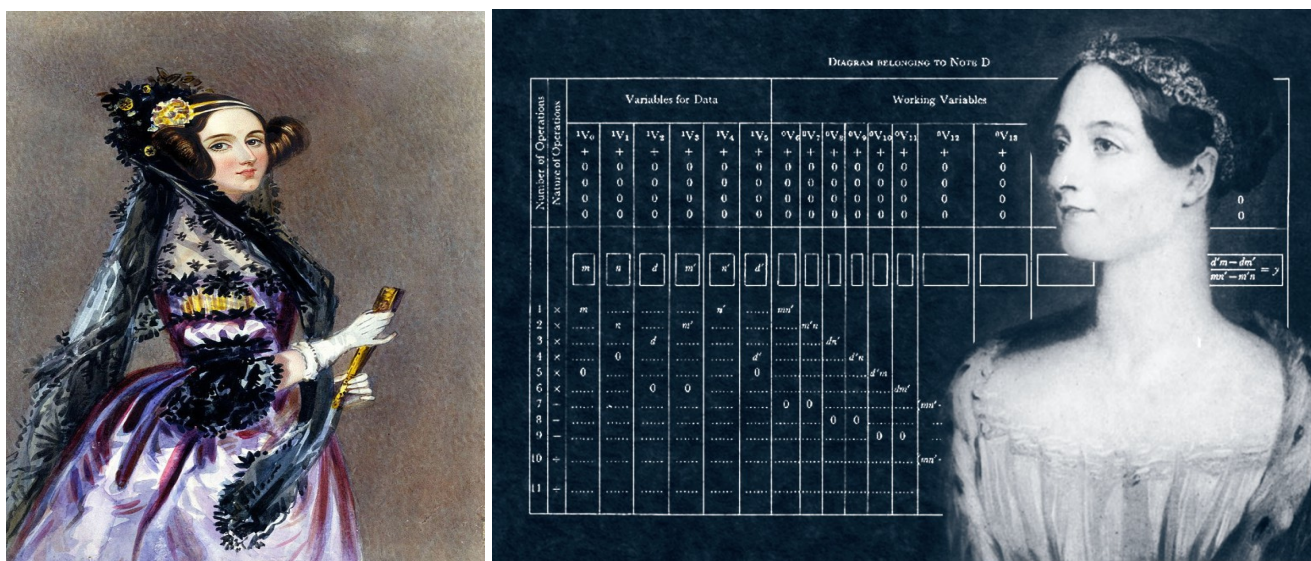
Tendo-se isto, primordialmente, antes que se fale em uma “crise”, mister se faz compreender o assunto tema da presente: **O Software**.

Afinal, **o que é um software? Como surgiu? Para quê serve?**

Tantas são as perguntas de uma coisa que, ironicamente, hoje, não vivemos sem!

Comecemos assim, com um pequeno passeio pelo ano de 1842... Este pequeno grande momento da história começou com Charles Babbage na Itália. Charles foi convidado a ministrar um seminário na Universidade de Turim sobre sua máquina analítica. Luigi Menabrea, um jovem engenheiro italiano e futuro Primeiro-ministro da Itália, publicou a palestra de Babbage em francês, e, esta transcrição foi posteriormente publicada na Bibliothèque Universelle de Genève, em 1842.

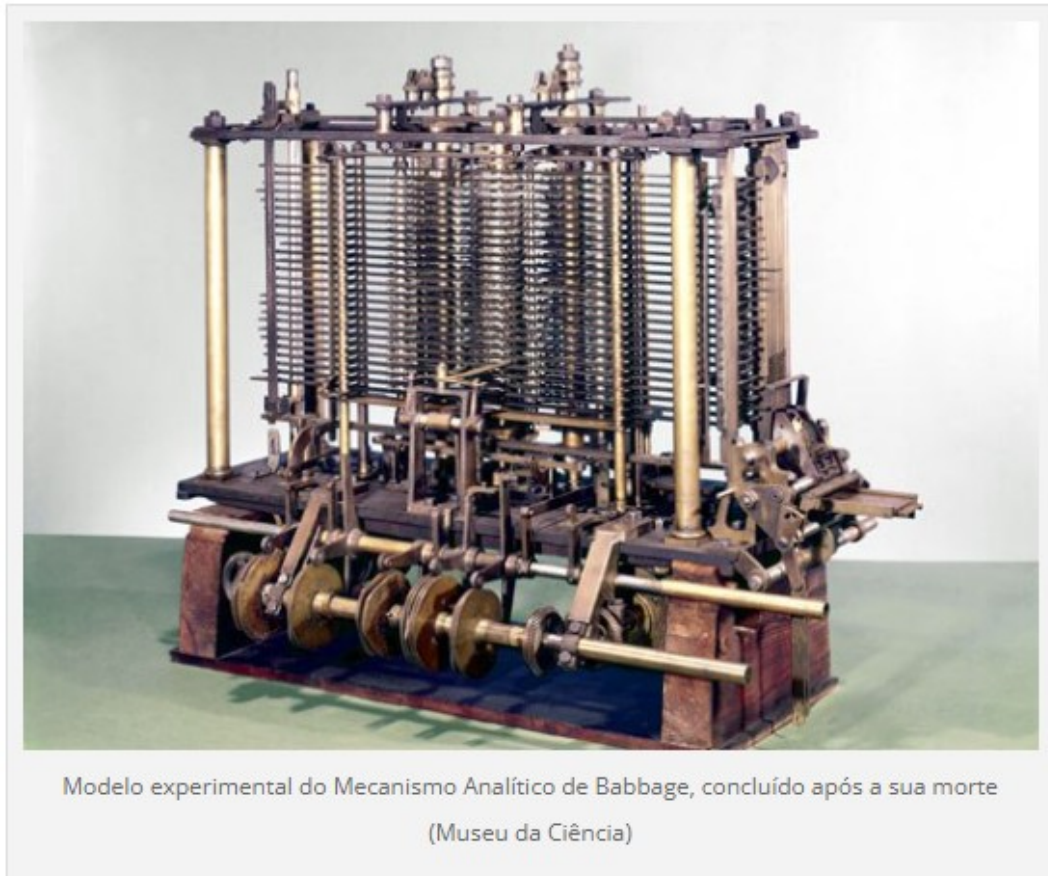
Charles Babbage pediu a **Augusta Ada Lovelace – Condessa Lovelace** -, para traduzir o artigo de Menabrea para o inglês, adicionando depois a tradução com as anotações que ela mesma havia feito. Ada Lovelace levou grande parte do ano nesta tarefa. Estas notas, que são mais extensas que o artigo de Menabrea, foram então publicados no The Ladies' Diary e no Memorial Científico de Taylor sob as iniciais "AAL".



Ada King, Condessa de Lovelace (Retrato de Alfred Edward Chalon, 1840)

As notas de Lovelace sobre a máquina analítica de Babbage foram, posteriormente, reconhecida como o **primeiro modelo de computador**, e, as notas de Lovelace como **a descrição de um computador e um software**.

As notas de Ada Lovelace foram classificadas alfabeticamente de A a G. Na nota G ela descreve **o algoritmo para a máquina analítica computar a Sequência de Bernoulli**. Este, é considerado **o primeiro algoritmo especificamente criado para ser implementado num computador**, e, Ada Lovelace é recorrentemente citada como **a primeira pessoa programadora por esta razão**.

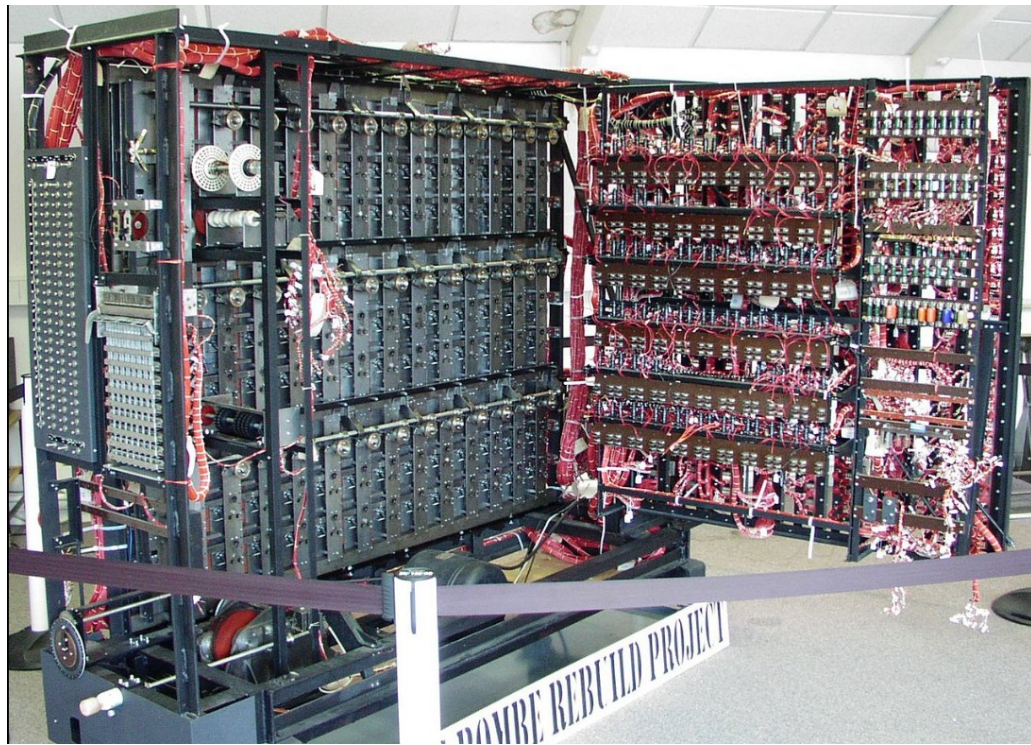


Não obstante o brilhante intelecto da Condessa Lovelace, capaz de modificar a história; temos em seguida, **Alan Mathison Turing**, matemático britânico, que modificou a história da 2ª Guerra Mundial, e, fez nascer conceitos até hoje respeitados.

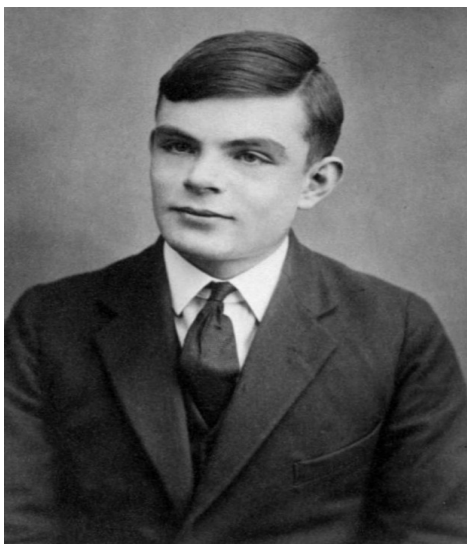
Durante a Segunda Guerra Mundial – 1939 a 1945 – **Alan Turing** trabalhou para a Escola de Código e Cifras do Governo (GC&CS) em Bletchley Park, o centro britânico de criptoanálise que produzia ultra inteligência. Por um tempo ele liderou a Hut 8, a seção responsável pela análise criptográfica naval alemã.

Lá ele desenvolveu várias técnicas para acelerar a quebra das cifras alemãs, incluindo melhorias no método de bombardeio polonês antes da guerra, bem como, uma máquina eletromecânica que poderia encontrar configurações para a máquina Enigma.





**Turing desempenhou um papel crucial na quebra de mensagens codificadas interceptadas que permitiram aos Aliados derrotar os nazistas em muitos compromissos cruciais, incluindo a Batalha do Atlântico, e, ao fazê-lo, os ajudou a vencer a guerra. Ao final, foi estimado que este trabalho encurtou a guerra na Europa em mais de dois anos e salvou mais de 14 milhões de vidas.**



**Alan Turing 1912 – 1954**



Em suma, esse belo passeio pelo tempo, nos forneceu importantes dados históricos para iniciar os nossos estudos. Mas, conceitualmente falando, **o que é um software?**

De acordo Ian Sommerville (9ª Edição, página 4): ***“Softwares são programas de computador e documentação associada. Produtos de software podem ser desenvolvidos para um cliente específico ou para o mercado em geral. (...) Um bom software deve prover a funcionalidade e o desempenho requeridos pelo usuário; além disso, deve ser confiável e fácil de manter e usar.”***

Então, ficam as dúvidas...

O que é um programa?

Qualquer um pode fazer?

Por que há uma necessidade de construção em nível de engenharia?

Vejamos o que nos diz Sommerville a respeito desse mister:

*“Quando falamos sobre a qualidade do software profissional, devemos levar em conta que o software é usado e alterado pelas pessoas, além de seus desenvolvedores. A qualidade, portanto, implica **não apenas o que o software faz**. Ao contrário, **ela tem de incluir o comportamento do software enquanto ele está executando**, bem como a estrutura e a organização dos programas do sistema e a documentação associada.*

*(...)*

*Portanto, um sistema bancário deve ser seguro, um jogo interativo deve ser ágil, um sistema de comutação de telefonia deve ser confiável, e assim por diante.”*

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 4).

Na tabela abaixo podemos observar mais especificamente os atributos essenciais de um bom software:

Características do Produto	Descrição
Manutenibilidade	O software deve ser escrito de forma que possa evoluir para atender às necessidades dos clientes. Esse é um atributo crítico, porque a mudança de software é um requisito inevitável de um ambiente de negócio em mudança.
Confiança e proteção	A confiança do software inclui uma série de características como confiabilidade, proteção e segurança. Um software confiável não deve causar prejuízos físicos ou econômicos no caso de falha de sistema. Usuários maliciosos não devem ser capazes de acessar ou prejudicar o sistema.
Eficiência	O software não deve desperdiçar os recursos do sistema, como memória e ciclos do processador. Portanto, eficiência inclui capacidade de resposta, tempo de processamento, uso de memória etc.
Aceitabilidade	O software deve ser aceitável para o tipo de usuário para o qual foi projetado. Isso significa que deve ser compreensível, usável e compatível com outros sistemas usados por ele.

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 5).

Finaliza, Sommerville com belíssimas alusões ao tema:

*“O mundo moderno não poderia existir sem o software. Infraestruturas e serviços nacionais são controlados por sistemas computacionais, e a maioria dos produtos elétricos **inclui um computador e um software que o controla**. A manufatura e a distribuição industriais são totalmente informatizadas, assim como o sistema financeiro. A área de entretenimento, incluindo a indústria da música, jogos de computador, cinema e televisão, **faz uso intensivo de***

*software. Portanto, a engenharia de software é essencial para o funcionamento de sociedades nacionais e internacionais.*

*(...)*

*Engenheiros de software têm o direito de se orgulhar de suas conquistas. É claro que ainda temos problemas em desenvolver softwares complexos, mas, **sem a engenharia de software, não teríamos explorado o espaço, não teríamos a Internet ou as telecomunicações modernas.** Todas as formas de viagem seriam mais perigosas e caras. A engenharia de software contribuiu muito, e tenho certeza de que suas contribuições no século XXI serão maiores ainda.”*

*(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 2-3).*

Com o exposto acima, podemos iniciar nossos estudos na seara desta disciplina, sabendo agora o conceito de software, podemos estudá-lo mais profundamente a seguir; como o faremos.

# DESENVOLVIMENTO

Após essas notáveis mentes nos agradecerem com suas criações, o universo da tecnologia iniciou sua longa viagem evolutiva desde os primórdios maquinais da era industrial até os dias atuais.

Nas décadas de 50 e 60, o software passou de um conceito teórico e abstrato, a ser reconhecido como um fator indispensável para a invenção e desenvolvimento de dispositivos, ou, máquinas.

Vejamos sua evolução do tempo.

## 1 – Década de 50

Na década de 1950 posiciona-se sobre as primeiras três linguagens de programação modernas, cujos descendentes ainda estão em uso difundido nos dias de hoje, quais sejam:

**1. FORTRAN (1954)**, a "FORmula TRANslator", inventada por John Backus e outros.

**2. LISP**, a "LISt Processor", inventada por John McCarthy e outros.

**3. COBOL**, a COmmon Business Oriented Language, criada pelo Short Range Committee, com grande influência de Grace Hopper.

Outro marco na década de 1950 foi a publicação, por um comitê de cientistas americanos e europeus, de "**uma nova linguagem para os algoritmos**", a **ALGOL 60** através da publicação do relatório "**The ALGOL 60 Report** (the "ALGO<sup>r</sup>ithmic Language)".

Este relatório consolidou muitas ideias que circulavam na época e apresentou duas inovações chave quanto ao projeto de linguagens:

**(1) Estrutura de blocos aninhados:** pedaços significativos de código poderiam ser agrupados em bloco de instruções, sem ter que ser transformados em procedimentos separados e ser explicitamente chamados.

**(2) Escopo léxico:** um bloco podia ter suas próprias variáveis não acessíveis fora do bloco, e muito menos manipuláveis de fora do bloco.

Panorama geral de linguagens de programação ao longo da década 50:

**1951** - Regional Assembly Language

**1952** - Autocode

**1954** - FORTRAN



1955 - FLOW-MATIC (antecessor do COBOL)

1957 - COMTRAN (antecessor do COBOL)

1958 - LISP

1958 - ALGOL 58

1959 - FACT (antecessor do COBOL)

1959 – COBOL

Com isto, encerramos os principais aspectos que marcaram tecnologicamente falando o período da década de 50. Daqui, continuaremos a evolução cronológica do assunto – na medida do que nos é possível averiguar –, de acordo com sua evolução demarcada no tempo.

## 2 – Década de 60: Engenharia de Software e a Crise do Software

No mundo moderno, tudo é software. Hoje em dia, por exemplo, empresas de qualquer tamanho dependem dos mais diversos sistemas de informação para automatizar seus processos. Empresas vendem por meio de sistemas de comércio eletrônico, uma gama imensa de produtos, diretamente para os consumidores. Software está também embarcado em diferentes dispositivos e produtos de engenharia, incluindo automóveis, aviões, satélites, etc.

Portanto, devido a sua relevância no nosso mundo, existe uma área da Computação destinada a investigar os desafios e propor soluções que permitam desenvolver sistemas de software — principalmente aqueles mais complexos e de maior tamanho, de forma produtiva e com qualidade: **a Engenharia de Software.**

Vejamos o que nos diz Sommerville a respeito da matéria:

*“A engenharia de software tem por objetivo **apoiar o desenvolvimento profissional de software, mais do que a programação individual.** Ela inclui **técnicas que apoiam especificação, projeto e evolução de programas,** que normalmente não são relevantes para o desenvolvimento de software pessoal.*

*(...)*

... quando falamos de engenharia de software, **não se trata apenas do programa em si, mas de toda a documentação associada e dados de configurações necessários para fazer esse programa operar corretamente.** Um sistema de software desenvolvido profissionalmente é, com frequência, mais do que apenas um programa; ele normalmente consiste em **uma série de programas separados e arquivos de configuração que são usados para configurar esses programas.** Isso pode incluir documentação do sistema, que descreve a sua estrutura; documentação do usuário, que explica como usar o sistema; e sites, para usuários baixarem a informação recente do produto.

(...)

Engenharia tem a ver com **obter resultados de qualidade requeridos dentro do cronograma e do orçamento.**

(...)

Em geral, os engenheiros de software adotam uma abordagem sistemática e organizada para seu trabalho, pois essa costuma ser a maneira mais eficiente de produzir software de alta qualidade. No entanto, **engenharia tem tudo a ver com selecionar o método mais adequado para um conjunto de circunstâncias, então uma abordagem mais criativa e menos formal pode ser eficiente em algumas circunstâncias.**”

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 3-5).

Tendo-se um conceito de software, e, uma ótica arrojada sobre o conceito da engenharia de software; podemos assim, começar a trabalhar a ideia do **processo de software.** Este último, forma-se por uma sequência de atividade que leva à produção de um produto de software.

Para compreender melhor esse processo, vamos listar abaixo as atividades fundamentais inerentes a um processo de software:

<b>Especificação de Software</b>	Clientes e engenheiros definem o software a ser produzido e as restrições de sua operação.
<b>Desenvolvimento de Software</b>	O software é projetado e programado.
<b>Validação de Software</b>	O software é verificado para garantir que é o que o cliente quer.
<b>Evolução de Software</b>	O software é modificado para refletir a mudança de requisitos do cliente e do mercado.

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 5-6).

Vale observar que, a forma como essa abordagem sistemática é realmente implementada varia de acordo com a organização que esteja desenvolvendo o software, o tipo de software e as pessoas envolvidas no processo de desenvolvimento. **Não existem técnicas e métodos universais na engenharia de software adequados a todos os sistemas e todas as empresas. Em vez disso, um conjunto diverso de métodos e ferramentas de engenharia de software tem evoluído exponencialmente em um lapso de 50 anos.**

O fator mais significativo em determinar quais técnicas e métodos de engenharia de software, são absolutamente relevantes ao tipo de aplicação que se deseja construir:

**1. Aplicações stand-alone:** Essas são as aplicações executadas em um computador local, como um PC.

**2. Aplicações interativas baseadas em transações:** São aplicações que executam em um computador remoto, acessadas pelos usuários a partir de seus computadores ou terminais.

**3. Sistemas de controle embutidos:** São sistemas de controle que controlam e gerenciam dispositivos de hardware.

**4. Sistemas de processamento de lotes:** São sistemas corporativos projetados para processar dados em grandes lotes.

**5. Sistemas de entretenimento:** São sistemas cuja utilização principal é pessoal e cujo objetivo é entreter o usuário.

**6. Sistemas para modelagem e simulação:** São sistemas que incluem vários objetos separados que interagem entre si, desenvolvidos por cientistas e engenheiros para modelar processos ou situações físicas.

**7. Sistemas de coleta de dados:** São sistemas que coletam dados de seu ambiente com um conjunto de sensores e enviam esses dados para outros sistemas para processamento.

**8. Sistemas de sistemas:** São sistemas compostos de uma série de outros sistemas de software. Alguns deles podem ser produtos genéricos de software, como um programa de planilha eletrônica. Outros sistemas do conjunto podem ser escritos especialmente para esse ambiente.

Após compreendermos estas importantes questões-base a respeito do software e da engenharia de software, podemos traçar agora, fundamentos que devem existir e ser aplicados a todos os tipos de sistemas de software:

- 
- 1** Eles devem ser desenvolvidos em um processo gerenciado e compreendido. A organização que desenvolve o software deve planejar o processo de desenvolvimento e ter ideias claras do que será produzido e quando estará finalizado. É claro que processos diferentes são usados para tipos de software diferentes.
  - 2** Confiança e desempenho são importantes para todos os tipos de sistema. O software deve se comportar conforme o esperado, sem falhas, e deve estar disponível para uso quando requerido. Deve ser seguro em sua operação e deve ser, tanto quanto possível, protegido contra ataques externos. O sistema deve executar de forma eficiente e não deve desperdiçar recursos.
  - 3** É importante entender e gerenciar a especificação e os requisitos de software (o que o software deve fazer). Você deve saber o que clientes e usuários esperam dele e deve gerenciar suas expectativas para que um sistema útil possa ser entregue dentro do orçamento e do cronograma.
  - 4** Você deve fazer o melhor uso possível dos recursos existentes. Isso significa que, quando apropriado, você deve reusar o software já desenvolvido, em vez de escrever um novo.
- 

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 8).

Nesta seara, nos ensina Marco Tulio Valente:

*“Historicamente, a área surgiu no final da década de 60 do século passado. Nas duas décadas anteriores, os primeiros computadores modernos foram projetados e começaram a ser usados principalmente para resolução de problemas científicos. Ou seja, **nessa época software não era uma preocupação central, mas sim construir máquinas que pudessem executar***

*alguns poucos programas. Em resumo, computadores eram usados por poucos e para resolver apenas problemas científicos.*

*No entanto, progressos contínuos nas tecnologias de construção de hardware mudaram de forma rápida esse cenário. No final da década de 60, computadores já eram mais populares, já estavam presentes em várias universidades norte-americanas e europeias e já chegavam também em algumas grandes empresas. Os cientistas da computação dessa época se viram diante de um novo desafio: como os computadores estavam se tornando mais populares, novas aplicações não apenas se tornavam possíveis, mas começavam a ser demandadas pelos usuários dos grandes computadores da época. Na verdade, os computadores eram grandes no sentido físico e não em poder de processamento, se comparado com os computadores atuais. Dentre essas novas aplicações, as principais eram sistemas comerciais, como folha de pagamento, controle de clientes, controle de estoques, etc.”*

(Engenharia de Software Moderna, Marco Tulio Valente)

Vale lembrar que, os primeiros processos de desenvolvimento de software — do tipo Waterfall, propostos na década de 70 — eram estritamente sequenciais, começando com uma fase de especificação de requisitos até chegar às fases finais de implementação, testes e manutenção do sistema.

Observação: Os modelos específicos serão vistos em capítulo próprio ao longo desse trabalho.

Se considerarmos o contexto histórico, essa primeira visão de processo era natural, visto que projetos de Engenharia tradicional também são sequenciais e precedidos de um planejamento detalhado. Todas as fases também geram documentações detalhadas do produto que está sendo desenvolvido. Por isso, nada mais natural que, a - até então - frágil Engenharia de Software, se espelhasse nos processos de áreas mais tradicionais, como Engenharia Eletrônica, Civil, Mecânica, Aeronáutica, etc.

No entanto, **começou-se a perceber que software é diferente de outros produtos de Engenharia.** Essa percepção foi ficando clara devido aos problemas frequentes enfrentados por projetos de software.

Neste período – anos 70 – os cronogramas e orçamentos dos projetos não eram, muitas vezes, obedecidos. Não raro, projetos inteiros eram cancelados, após anos de trabalho, sem entregar um sistema funcional para os clientes, o que levou ao objeto desse estudo. A famigerada:

### **Crise do Software**

A aludida crise, foi em verdade, um termo que surgiu nos anos 70. O termo expressava as dificuldades do desenvolvimento de software frente ao rápido crescimento da demanda por software, da complexidade dos problemas a serem resolvidos, e, da inexistência de técnicas estabelecidas para o desenvolvimento de sistemas que funcionassem adequadamente, ou, que pudessem ser validados.

No Início dos anos 70, quando vivia-se a terceira era do software, houve **muitos problemas de prazo e custo no desenvolvimento de software, devido a baixa produtividade, baixa qualidade e difícil manutenção do software.**

### **Os problemas mais comuns no desenvolvimento de software eram:**

- Estimativas de prazo e de custo imprecisas;
- Produtividade da área de software não acompanhava a demanda do mercado;
- Prazos ultrapassados;
- Custos acima do previsto;
- Difícil manutenção;
- Não atendimento dos requisitos do usuário, e, decepção do produto final.

Estes importantes fatores ensejaram, assim, uma vital necessidade à evolução da tecnologia: **a criação da Engenharia de Software.**

Esta, a Engenharia de Software, surge então numa tentativa de contornar a crise do software, e, dar um tratamento de engenharia - mais sistemático e controlado - ao desenvolvimento de sistemas de software complexos.

O termo Engenharia de Software, tornou-se conhecido após a conferência da OTAN; quando as dificuldades e armadilhas de projetar sistemas complexos foram discutidas francamente. Esta, ocorreu em **outubro de 1968**, um grupo de cerca de 50 renomados Cientistas da Computação se reuniu durante uma semana em Garmisch, na Alemanha, em uma conferência patrocinada por um comitê científico da OTAN, a organização militar que congrega os países do Atlântico Norte.

O objetivo da conferência era chamar a atenção para um problema crucial do uso de computadores, **o chamado software**. A conferência produziu um relatório, com mais de 130 páginas, que afirmava a necessidade de que software fosse construído **com base em princípios práticos e teóricos, tal como ocorre em ramos tradicionais e bem estabelecidos da Engenharia.**



FOTO: Conferência OTAN – 1968.



Para deixar essa proposta mais clara, decidiu-se cunhar o termo **Engenharia de Software**. Por isso, a Conferência da OTAN é considerada **o marco histórico de criação da área de Engenharia de Software**.

Deste momento em diante, a busca de soluções começou. Ela se concentrou em melhores metodologias e ferramentas. As mais importantes, foram as linguagens de programação que refletem os estilos procedimental, modular; e, em seguida, orientado a objeto.

A engenharia de software está intimamente ligada ao aparecimento e aperfeiçoamento desses estilos. Também importantes foram os esforços de sistematização, automatização da documentação do programa e testes. Por último, a verificação analítica e provas de correção deveriam substituir os testes. Somente após a conferência, as dificuldades foram discutidas abertamente e confessadas com franqueza incomum, e os termos “engenharia de software” e “crise de software” tornaram-se comuns.

A partir da insurgência dessa problemática, o universo tecnológico sentiu uma vasta necessidade de crescer e evoluir com o fim de acompanhar a demanda que se lhe impunha naquele momento, e assim, inicia-se o movimento evolutivo constante das linguagens e modelos.

O período compreendido à década de 1970, trouxe um grande florescimento de linguagens de programação. A maioria dos principais paradigmas de linguagem agora em uso, foram inventados durante este período:

**Simula:** inventada nos anos 1960 por Nygaard e Dahl como um super conjunto de Algol 60, foi a primeira linguagem a suportar os conceitos programação orientada a objetos.

**C:** uma das primeiras linguagens de programação de sistemas, foi desenvolvido por Dennis Ritchie e Ken Thompson nos laboratórios da Bell entre 1969 e 1973.

**Smalltalk:** (meados de 1970) forneceu uma base completa para o projeto de uma linguagem orientada a objetos.

**Prolog:** projetada em 1972 por Colmerauer, Roussel, e Kowalski, foi a primeira linguagem de programação do paradigma lógico.

**ML:** inventada por Robin Milner em 1973, é uma linguagem funcional, baseada em Lisp, estaticamente tipada.

Cada uma dessas línguas gerou toda uma família de descendentes, e linguagens mais modernas contam, pelo menos, com uma delas em sua ascendência.

Algumas linguagens mais importantes desenvolvidas durante este período, incluem:

1970 - **Pascal**

1970 - **Forth**

1972 - **C**

1972 - **Smalltalk**

1972 - **Prolog**

1973 - **ML**

1978 - **SQL** (inicialmente apenas uma linguagem de consulta, mais tarde estendido com construções de programação)

Assim, pudemos compreender melhor a questão da crise em si. Mas, quando se trata de engenharia de software e toda sua problemática, mister se faz, aprofundar o estudo objetivando conhecer, agora, os processos de software; e, os modelos adequados ao seu surgimento cronológico. Portanto, o estudo iniciado pelo conceito de software, objetivou explicar a crise do software, e, deste momento em diante, buscará explicitar o “procedimento” da construção do software. Ao final, encerrar-se-á na forma preterida à introdução do presente: o conceito de software, a evolução deste, e, a construção do software.

Portanto, continuaremos a tratar o assunto da Engenharia de Software - por didática necessária à conclusão da famigerada crise do software -, e, ao melhor entendimento dos capítulos vindouros. Nos valem assim, das lições de Marco Tulio Valente acerca do tema:

*“Desenvolvimento de software é diferente de qualquer outro produto de Engenharia, principalmente quando se compara software com hardware. Frederick Brooks, Prêmio Turing em Computação (1999) e um dos pioneiros da área de Engenharia de Software, foi um dos primeiros a chamar a atenção para esse fato. **Em 1987, em um ensaio intitulado Não Existe Bala de Prata: Essência e Acidentes em Engenharia de Software**, ele discorreu sobre as particularidades da área de Engenharia de Software.*

*Segundo Brooks, **existem dois tipos de dificuldades em desenvolvimento de software: dificuldades essenciais e dificuldades acidentais.***

*As **essenciais são da natureza da área** e dificilmente serão superadas por qualquer nova tecnologia ou método que se invente.*

*Segundo Brooks, as dificuldades **essenciais são as seguintes:***

**Complexidade:** *dentre as construções que o homem se propõe a realizar, software é uma das mais desafiadoras e mais complexas que existe. Na verdade, como dissemos antes, mesmo construções de engenharia tradicional, como um satélite, uma usina nuclear ou um foguete, são cada vez mais dependentes de software.*

**Conformidade:** *pela sua natureza, software tem que se adaptar ao seu ambiente que muda a todo momento no mundo moderno. Por exemplo, se as leis para recolhimento de impostos mudam, normalmente espera-se que os sistemas sejam rapidamente adaptados à nova legislação. Brooks comenta que isso não ocorre, por exemplo, na Física, pois as leis da natureza não mudam de acordo com os caprichos dos homens.*

**Facilidade de mudanças:** *que consiste na necessidade de evoluir sempre, incorporando novas funcionalidades. Na verdade, quanto mais bem sucedido for um sistema de software, mais demanda por mudanças ele recebe.*

**Invisibilidade:** *devido à sua natureza abstrata, é difícil visualizar o tamanho e consequentemente estimar o esforço de construir um sistema de software.*

*Desenvolvimento de software enfrenta também **dificuldades acidentais**. No entanto, **elas estão associadas a problemas tecnológicos, que os Engenheiros de Software podem resolver, se devidamente treinados e caso tenham acesso às devidas tecnologias e recursos**. Como exemplo, **podemos citar as seguintes dificuldades**: um compilador que produz mensagens de erro obscuras, uma IDE que possui muitos bugs e frequentemente sofre travamentos, um framework que não possui documentação, uma aplicação Web com uma interface pouco intuitiva, etc. Todas essas dificuldades dizem respeito à solução adotada e, portanto, não são uma característica inerente dos sistemas.”*

(Engenharia de Software Moderna, Marco Tulio Valente)

Objetivando aprofundar a análise sobre a engenharia de software em sua essência, nos valem, outrossim, do **Guide to the Software Engineering Body of Knowledge**, também conhecido pela sigla **SWEBOK**. Trata-se de um documento, organizado pela **IEEE Computer Society**, com o apoio de diversos pesquisadores e de profissionais da indústria. O objetivo do SWEBOK é precisamente documentar o corpo de conhecimento que caracteriza a área, que assim chamamos, Engenharia de Software.

O SWEBOK define **12 áreas de conhecimento** em Engenharia de Software:

1. Engenharia de Requisitos
2. Projeto de Software
3. Construção de Software
4. Testes de Software
5. Manutenção de Software
6. Gerência de Configuração
7. Gerência de Projetos
8. Processos de Software
9. Modelos de Software
10. Qualidade de Software
11. Prática Profissional
12. Aspectos Econômicos

De posse dessas informações, podemos iniciar um aprofundamento da disciplina começando a falar sobre: **Processos de Software**.

Neste diapasão, nos valem das lições de Sommerville:

***“Um processo de software é um conjunto de atividades relacionadas que levam à produção de um produto de software. Essas atividades podem envolver o desenvolvimento de software a partir do zero em uma linguagem padrão de programação.”***

(Engenharia de Software, Ian Sommerville, 9ª Ed, Página 18)

Para compreender melhor esse processo, vamos listar abaixo as atividades fundamentais inerentes a um processo de software:

<b>Especificação de Software</b>	Clientes e engenheiros definem o software a ser produzido e as restrições de sua operação.
<b>Desenvolvimento de Software</b>	O software é projetado e programado.
<b>Validação de Software</b>	O software é verificado para garantir que é o que o cliente quer.
<b>Evolução de Software</b>	O software é modificado para refletir a mudança de requisitos do cliente e do mercado.

(Engenharia de Software, Ian Sommerville, 9ªEd, Página 18)

Marco Tulio Valente reforça a lição:

***“Processo é o conjunto de passos, etapas e tarefas que se usa para construir um software. Toda organização usa um processo para desenvolver seus sistemas, o qual pode waterfall, por exemplo. Ou talvez, esse processo pode ser caótico. Porém, o ponto que queremos reforçar é que sempre existe um processo.***

*Já método, no nosso contexto, define e especifica um determinado processo de desenvolvimento (a palavra método tem sua origem no grego, onde significa caminho para se chegar a um objetivo).*

*Assim, XP, Scrum e Kanban são métodos ágeis ou, de modo mais extenso, são métodos que definem práticas, atividades, eventos e técnicas compatíveis com princípios ágeis de desenvolvimento de software.*

***Aproveitando que estamos tratando de definições, frequentemente usa-se também o termo metodologia quando se fala de processos de software.***

*Por exemplo, é comum ver referências a metodologias para desenvolvimento de software, metodologias ágeis, metodologia orientada a objetos, etc. A palavra metodologia, no sentido estrito, denota o ramo da lógica que se ocupa dos métodos das diferentes ciências, segundo o Dicionário Houaiss. No entanto, a palavra também pode ser usada como sinônimo de método, segundo o mesmo dicionário.”*

(Engenharia de Software Moderna, Marco Tulio Valente)

Ainda sobre processos de software, completa a lição o autor:

*“ A adoção de um processo é menos importante. Ou, dizendo de outra forma, o processo em tais projetos é pessoal, composto pelos princípios, práticas e decisões tomadas pelo seu único desenvolvedor; e que terão impacto apenas sobre ele mesmo.*

***Porém, os sistemas de software atuais são por demais complexos para serem desenvolvidos por uma única pessoa.***

*(...)*

*Na prática, os sistemas modernos são desenvolvidos em equipes.*

*E essas equipes, para produzir software com qualidade e produtividade, precisam de um ordenamento, mesmo que mínimo. **Por isso, empresas dão tanto valor a processos de software. Eles são o instrumento de que as empresas dispõem para coordenar, motivar, organizar e avaliar o trabalho de seus desenvolvedores, de forma a garantir que eles trabalhem com produtividade e produzam sistemas alinhados com os objetivos da organização.** Sem um processo — mesmo que simplificado e leve — existe o risco de que os times de desenvolvimento passem a trabalhar de forma descoordenada, gerando produtos sem valor para o negócio da empresa. Por fim, processos são importantes não apenas para a empresa, mas também para os desenvolvedores, pois **permitem que eles tomem consciência das tarefas e resultados que se esperam deles. Sem um processo, os desenvolvedores podem se sentir perdidos, trabalhando de forma errática e sem alinhamento com os demais membros do time.**”*

*(Engenharia de Software Moderna, Marco Tulio Valente)*

Desta forma, podemos exemplificar, *in casu* que, como a produção de um carro em uma fábrica de automóveis segue um processo bem definido; primeiro, as chapas de aço são cortadas e prensadas, para ganhar a forma de portas, tetos e capôs. Depois, o carro é pintado e instalam-se painel, bancos, cintos de segurança e a fiação. Por fim, instala-se a parte mecânica, incluindo motor, suspensão e freios.

Software, também é produzido de acordo com um processo, embora certamente menos mecânico e mais dependente de esforço intelectual. **Um processo de desenvolvimento de software define um conjunto de passos, tarefas, eventos e práticas que devem ser seguidos por desenvolvedores de software, na produção de um sistema.**

*“In software development, perfect is a verb, not an adjective. There is no perfect process. There is no perfect design. There are no perfect stories. You can, however, perfect your process, your design, and your stories.” – Kent Beck.*

*“No desenvolvimento de software, perfeito é um verbo, não um adjetivo. Não existe processo perfeito. Não existe projeto perfeito. Não existem histórias perfeitas. Você pode, no entanto, aperfeiçoar seu processo, seu design e suas histórias.” - tradução Google Translator.*



Objetivando, portanto, uma melhor compreensão desses processos, passaremos nos capítulos a seguir, a analisar cada modelo de processo de software individualmente, e assim, buscaremos apresentá-los em ordem cronológica crível à sua aceitação e uso.

A partir deste ponto, adentraremos os processos de software de forma técnica sob um ponto de vista cirúrgico a todas as questões inerentes a formação de um software. Assim, vamos tratar um assunto que custará basicamente todas as páginas restantes desta pesquisa: **Os modelos de processos de software.**

Nos ensina Sommerville:

*“Um modelo de processo de software é uma representação simplificada de um processo de software. Cada modelo representa uma perspectiva particular de um processo e, portanto, fornece informações parciais sobre ele.*

(...)

*...modelos genéricos, não são descrições definitivas dos processos de software. Pelo contrário, são abstrações que podem ser usadas para explicar diferentes abordagens de desenvolvimento de software. Podem ser vistos como frameworks de processos que podem ser ampliados e adaptados para criar processos de engenharia de software mais específicos.*

(...)

*Esses modelos não são mutuamente exclusivos e muitas vezes são usados em conjunto, especialmente, para o desenvolvimento de sistemas de grande porte.”*

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 19, e, 20).

Esses modelos são:

- (a) O modelo em cascata; Waterfall.
- (b) Desenvolvimento incremental; Mills.
- (c) Engenharia de software orientada a reuso.

Vamos agora esmiuçar cada um deles separadamente, e, avaliá-los em suas íntimas características.

### 3 – Década de 70: Modelo em Cascata

Um processo de desenvolvimento define quais atividades e etapas devem ser seguidas para construir e entregar um sistema de software. Uma analogia pode ser feita, por exemplo, com a construção de prédios, que ocorre de acordo com algumas etapas: fundação, alvenaria, cobertura, instalações hidráulicas, instalações elétricas, acabamento, pintura, etc.

Historicamente, existem dois grandes tipos de processos que podem ser adotados na construção de sistemas de software:

**(a) Processo Tradicional:** Waterfall, Mills, Boehm.

**(b) Processo Ágil:** XP, Scrum, KanBan.

O primeiro modelo do processo de desenvolvimento de software a ser publicado, foi derivado de processos mais gerais da engenharia de sistemas (ROYCE, **1970**). Por causa do encadeamento entre uma fase e outra, esse modelo é conhecido como “**modelo em cascata**”, ou, **Waterfall**.

O modelo em cascata é um exemplo de um processo dirigido a planos — em princípio, deve ser planejado e programado todas as atividades do processo antes de começar a trabalhar nelas. Esse processo, foi inspirado nos processos usados em engenharias tradicionais, os quais são largamente sequenciais, como no exemplo de uma construção predial.

Processos Waterfall foram muito usados até a década de 1990, e, grande parte desse sucesso deve-se a uma padronização lançada pelo Departamento de Defesa Norte-Americano, em 1985. Basicamente, eles estabeleceram que todo software comprado, ou, contratado pelo Departamento de Defesa, deveria ser construído usando o método Waterfall.

Seguindo as lições de Sommerville:

*“Os principais estágios do modelo em cascata refletem diretamente as atividades fundamentais do desenvolvimento:*

**1. Análise e definição de requisitos.** *Os serviços, restrições e metas do sistema são estabelecidos por meio de consulta aos usuários. Em seguida, são definidos em detalhes e funcionam como uma especificação do sistema.*

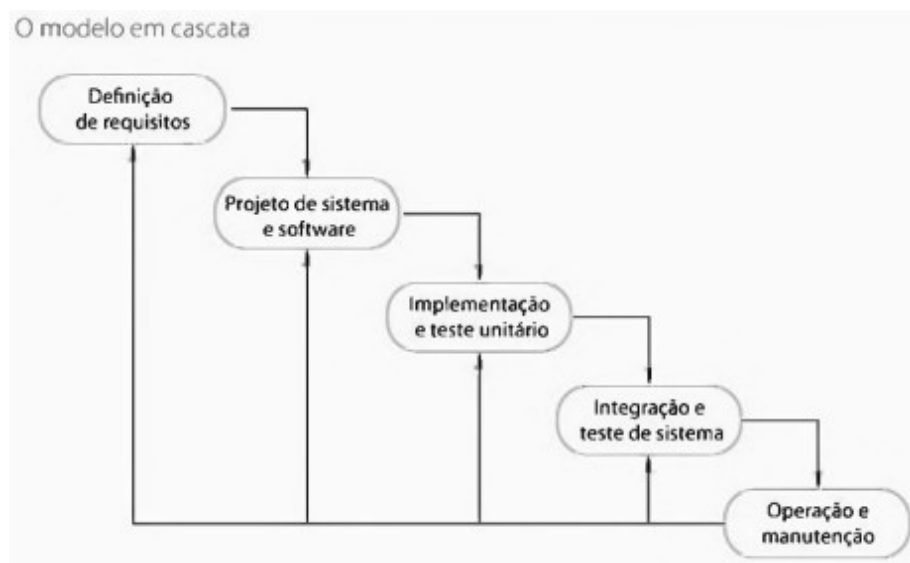
**2. Projeto de sistema e software.** *O processo de projeto de sistemas aloca os requisitos tanto para sistemas de hardware como para sistemas de software, por meio da definição de uma arquitetura geral do sistema. O projeto de software envolve identificação e descrição das abstrações fundamentais do sistema de software e seus relacionamentos.*

**3. Implementação e teste unitário.** *Durante esse estágio, o projeto do software é desenvolvido como um conjunto de programas ou unidades de programa. O teste unitário envolve a verificação de que cada unidade atenda a sua especificação.*

**4. Integração e teste de sistema.** As unidades individuais do programa ou programas são integradas e testadas como um sistema completo para assegurar que os requisitos do software tenham sido atendidos. Após o teste, o sistema de software é entregue ao cliente.

**5. Operação e manutenção.** Normalmente (embora não necessariamente), essa é a fase mais longa do ciclo de vida. O sistema é instalado e colocado em uso. A manutenção envolve a correção de erros que não foram descobertos em estágios iniciais do ciclo de vida, com melhora da implementação das unidades do sistema e ampliação de seus serviços em resposta às descobertas de novos requisitos.”

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 20-21).



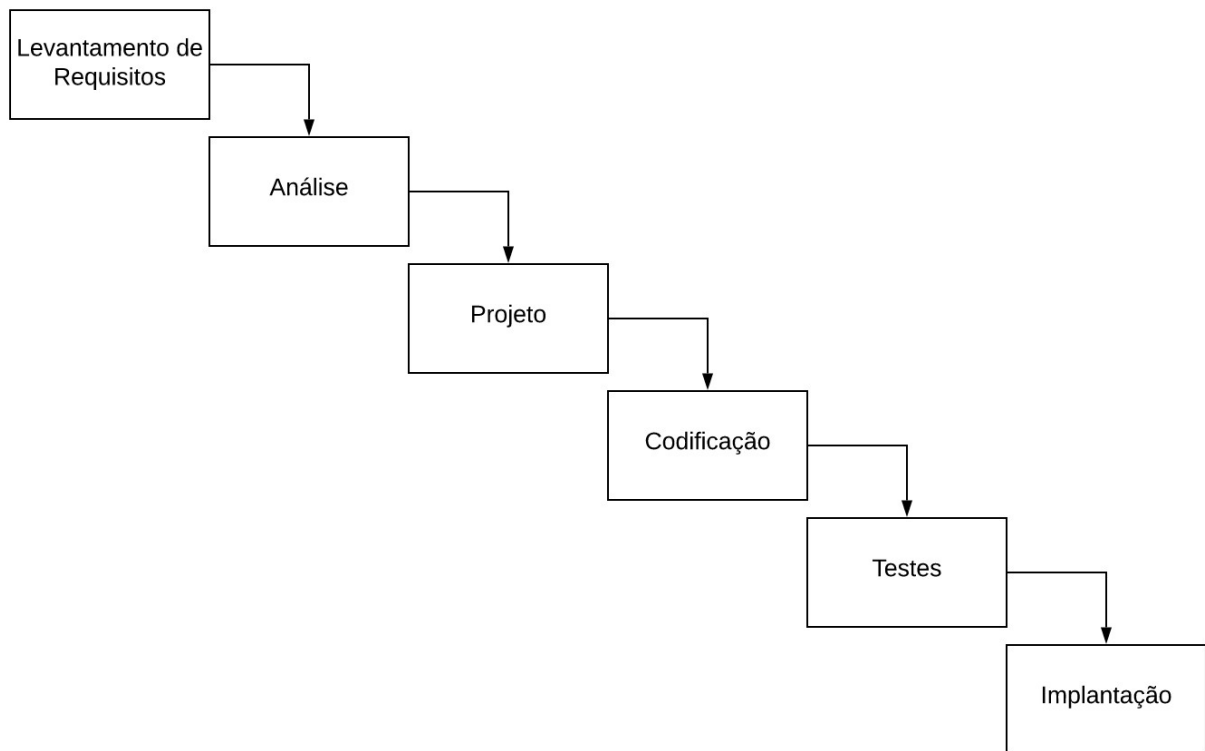
Para ilustrar o tema, nos ensina Marco Tulio Valente:

**“Processos Waterfall** — também chamados de processos dirigidos por planejamento (plan-driven) — propõem que a construção de um sistema deve ser feita em etapas sequenciais, como em uma cascata de água, na qual a água vai escorrendo de um nível para o outro.

Essas etapas são as seguintes: **levantamento de requisitos, análise (ou projeto de alto nível), projeto detalhado, codificação e testes.**

Finalizado esse pipeline, o sistema é liberado para produção, isto é, para uso efetivo pelos seus usuários, conforme ilustrado na próxima figura.”

Engenharia do Software Moderna, Marco Tulio Valente).



### **Problemas:**

Devido aos custos de produção e aprovação de documentos, as iterações podem ser dispendiosas e envolver significativo retrabalho. Assim, após um pequeno número de iterações, é normal se congelarem partes do desenvolvimento, como a especificação, e dar-se continuidade aos estágios posteriores de desenvolvimento.

A solução dos problemas fica para mais tarde, ignorada ou programada, quando possível. Esse congelamento prematuro dos requisitos pode significar que o sistema não fará o que o usuário quer. Também pode levar a sistemas mal estruturados, quando os problemas de projeto são contornados por artifícios de implementação.

Durante o estágio final do ciclo de vida (operação e manutenção), o software é colocado em uso. Erros e omissões nos requisitos originais do software são descobertos. Os erros de programa e projeto aparecem e são identificadas novas necessidades funcionais. O sistema deve evoluir para permanecer útil. Fazer essas alterações, manutenção do software, pode implicar repetição de estágios anteriores do processo.

Seu maior problema é a divisão inflexível do projeto em estágios distintos. Os compromissos devem ser assumidos em um estágio inicial do processo, o que dificulta que atendam às mudanças de requisitos dos clientes.

EVANS (2017), defensor de uma linguagem de arquitetura que reflita as regras de negócio do cliente, nos afirma que:

*“No Modelo em Cascata, os especialistas de um negócio conversam com os analistas, que passam o resultado para os programadores. No entanto, essa abordagem falha porque lhe falta feedback. Em outras palavras, o software resultante pode atender às necessidades do cliente, mas o código não reflete as regras de negócio, uma vez que os programadores ‘só programam’, não sendo possível identificar os termos de negócio olhando no código.”*

Já, PRESSMAN, 2006:

*“A natureza linear do modelo leva a estados de bloqueio, nos quais alguns membros da equipe de projeto precisam esperar que outros membros completem as tarefas dependentes.”*

Nos encerra a lição, Marco Tulio Valente com fortes alusões ao tema:

*“Processos Waterfall, a partir do final da década de 90, passaram a ser muito criticados, devido aos atrasos e problemas recorrentes em projetos de software, que ocorriam com frequência nessa época.*

***O principal problema é que Waterfall pressupõe um levantamento completo de requisitos, depois um projeto detalhado, depois uma implementação completa, etc. Para só então validar o sistema com os usuários, o que pode acontecer anos após o início do projeto.***

*No entanto, nesse período, o mundo pode ter mudado, bem como as necessidades dos clientes, que podem não mais precisar do sistema que ajudaram a especificar anos antes”*

(Engenharia do Software Moderna, Marco Tulio Valente)

Em suma, o cliente deve ter muita paciência, pois, uma versão executável do software somente é disponibilizada em etapa de finalização. Durante sua construção, é muito difícil retornar para as fases anteriores a fim de corrigir problemas detectados posteriormente, uma vez que, o processo segue o fluxo de uma cascata de água. Assim, muito tempo é gasto para garantir que as fases sejam executadas corretamente.

O cliente, no final, pode se decepcionar drasticamente ao receber o produto. Nesse ínterim, a equipe de desenvolvimento está sempre “quase acabando” um produto que pode se tornar completamente obsoleto. Tal questão, faz com que os riscos sejam retardados por muito tempo, criando sérios atrasos à sua resolução.

Por vezes, projetos grandes e complexos acabam sendo cancelados, ou, não entregues por falta de orçamento, ou, por estarem completamente desatualizados ao próprio negócio do cliente solicitante.



## 4 – Década de 80

A década de 1980 foi um tempo de relativa consolidação das linguagens imperativas. A linguagem C++ combinou orientação a objetos e programação de sistemas. O governo dos Estados Unidos padronizou a Ada, uma linguagem de programação de sistemas destinados à utilização por parte dos contratantes de defesa.

No Japão e em outros lugares, vastas somas foram gastas investigando as chamadas linguagens de programação de quinta geração que incorporavam a programação lógica em suas construções. A comunidade de linguagens funcionais se dedicou a padronizar a ML e o Lisp. Ao invés de inventar novos paradigmas, todos estes esforços, visaram aperfeiçoar as ideias inventadas na década anterior.

A década de 80 também trouxe avanços na implementação das linguagens de programação. O movimento RISC, em arquitetura de computadores postulou que o hardware deveria ser concebido para os compiladores ao invés de ser voltado a programadores assembly. Ajudado por melhorias na velocidade dos processadores que permitiu cada vez mais técnicas agressivas de compilação, o movimento RISC despertou maior interesse na tecnologia de compilação para linguagens de alto nível.

Algumas linguagens importantes que foram desenvolvidas durante este período, incluem:

1983 - **Ada**

1983 - **C++**

1984 - **MATLAB**

1985 - **Eiffel**

1986 - **Objective-C**

1987 - **Perl**

1989 - **FL** (Backus)

#### 4.1 - Modelo de Desenvolvimento Incremental

O desenvolvimento incremental **é baseado na ideia de desenvolver uma implementação inicial, expô-la aos comentários dos usuários e continuar por meio da criação de várias versões até que um sistema adequado seja desenvolvido.** Atividades de especificação, desenvolvimento e validação **são intercaladas, e não separadas,** com rápido feedback entre todas as atividades.

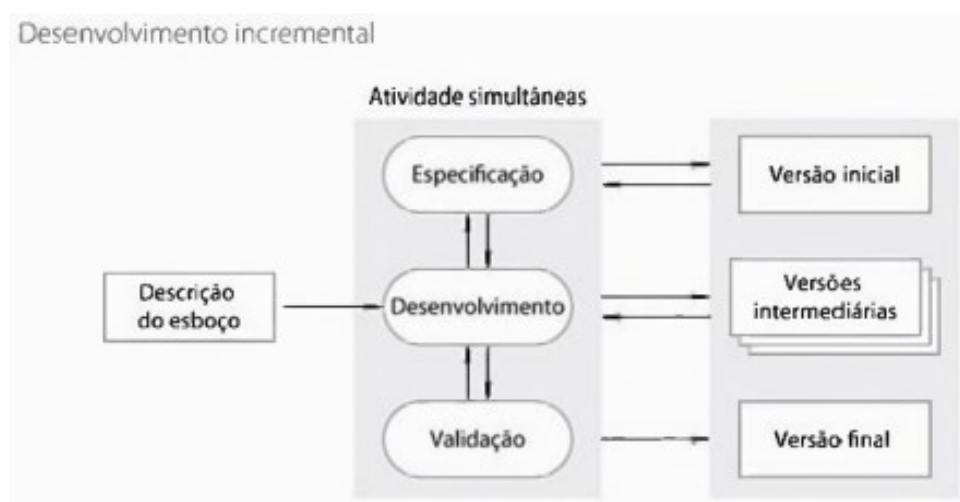
Cada incremento, ou, versão do sistema, incorpora alguma funcionalidade necessária para o cliente. Frequentemente, os incrementos iniciais incluem a funcionalidade mais importante, ou, mais urgente. **Isso significa que o cliente pode avaliar o sistema em um estágio relativamente inicial do desenvolvimento para ver se ele oferece o que foi requisitado. Em caso negativo, só o incremento que estiver em desenvolvimento no momento precisará ser alterado** e, possivelmente, nova funcionalidade deverá ser definida para incrementos posteriores.

Sobre o tema, nos ensina Sommerville:

*“O desenvolvimento incremental tem três vantagens importantes quando comparado ao modelo em cascata:*

1. O custo de acomodar as mudanças nos requisitos do cliente é reduzido. A quantidade de análise e documentação a ser refeita é muito menor do que o necessário no modelo em cascata.
2. É mais fácil obter feedback dos clientes sobre o desenvolvimento que foi feito. Os clientes podem fazer comentários sobre as demonstrações do software e ver o quanto foi implementado. Os clientes têm dificuldade em avaliar a evolução por meio de documentos de projeto de software.
3. É possível obter entrega e implementação rápida de um software útil ao cliente, mesmo se toda a funcionalidade não for incluída. Os clientes podem usar e obter ganhos a partir do software inicial antes do que é possível com um processo em cascata.

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 22).



Do ponto de vista do gerenciamento, a abordagem incremental tem dois problemas:

**1. O processo não é visível.** Os gerentes precisam de entregas regulares para mensurar o progresso. Se os sistemas são desenvolvidos com rapidez, não é economicamente viável produzir documentos que reflitam cada uma das versões do sistema.

**2. A estrutura do sistema tende a se degradar com a adição dos novos incrementos.** A menos que tempo e dinheiro sejam dispendidos em refatoração para melhoria do software, as constantes mudanças tendem a corromper sua estrutura. Incorporar futuras mudanças do software torna-se cada vez mais difícil e oneroso.

Encerra a lição com enfática crítica o autor:

*“Os problemas do desenvolvimento incremental, são particularmente críticos para os sistemas de vida-longa, grandes e complexos, nos quais várias equipes desenvolvem diferentes partes do sistema. **Sistemas de grande porte necessitam de um framework ou arquitetura estável, e, as responsabilidades das diferentes equipes de trabalho do sistema precisam ser claramente definidas, respeitando essa arquitetura. Isso deve ser planejado com antecedência, e não, desenvolvido de forma incremental.***

*Você pode desenvolver um sistema de forma incremental e expô-lo aos comentários dos clientes, sem realmente entregá-lo e implantá-lo no ambiente do cliente. **Entrega e implantação incremental significa que o software é usado em processos operacionais reais. Isso nem sempre é possível pois experimentações com o novo software podem interromper os processos normais de negócios.***”

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 23).

## 4.2 - Engenharia de software orientada a reuso

Na maioria dos projetos de software, há algum reuso de software. Isso acontece muitas vezes informalmente, quando as pessoas envolvidas no projeto sabem de projetos, ou, códigos semelhantes ao que é exigido. Elas os buscam, fazem as modificações necessárias, e, incorporam-nos a seus sistemas.

Esse reuso informal ocorre independentemente do processo de desenvolvimento que se use. Abordagens orientadas a reuso dependem de uma ampla base de componentes reusáveis de software e de um framework de integração para a composição desses componentes.

De acordo com Sommerville, os estágios em um processo orientado a reuso são:

<b>Análise de componentes</b>	Dada a especificação de requisitos, é feita uma busca por componentes para implementar essa especificação. Em geral, não há correspondência exata, e os componentes que podem ser usados apenas fornecem alguma funcionalidade necessária.
<b>Modificação de requisitos</b>	Durante esse estágio, os requisitos são analisados usando-se informações sobre os componentes que foram descobertos. Em seguida, estes serão modificados para refletir os componentes disponíveis. No caso de modificações impossíveis, a atividade de análise dos componentes pode ser reinserida na busca por soluções alternativas.
<b>Projeto do sistema com reuso</b>	Durante esse estágio, o framework do sistema é projetado ou algo existente é reusado. Os projetistas têm em mente os componentes que serão reusados e organizam o framework para reuso. Alguns softwares novos podem ser necessários, se componentes reusáveis não estiverem disponíveis.
<b>Desenvolvimento e integração</b>	Softwares que não podem ser adquiridos externamente são desenvolvidos, e os componentes e sistemas COTS são integrados para criar o novo sistema. A integração de sistemas, nesse modelo, pode ser parte do processo de desenvolvimento, em vez de uma atividade separada.

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 23).

Finaliza, Sommerville, a lição:

*“Engenharia de software orientada a reuso tem a vantagem óbvia de reduzir a quantidade de software a ser desenvolvido e, assim, reduzir os custos e riscos. Geralmente, também proporciona a entrega mais rápida do software. No entanto, **compromissos com os requisitos são inevitáveis, e isso pode levar a um sistema que não atende às reais necessidades dos usuários. Além disso, algum controle sobre a evolução do sistema é perdido, pois as novas versões dos componentes reusáveis não estão sob o controle da organização que os está utilizando.**”*

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 23 e 24).



### 4.3 - Modelo Espiral de Boehm

Criado por Barry Boehm em **1988**, o Modelo em Espiral é uma melhoria do Modelo Incremental, e, possui esse nome por causa de sua representação onde **cada volta no espiral, percorre todas as fases do processo de software**. As voltas devem ser repetidas quantas vezes forem necessárias até que o software possa ser completamente entregue. O modelo em espiral combina prevenção e tolerância a mudanças, assume que mudanças são um resultado de riscos de projeto, e, inclui atividades explícitas de gerenciamento de riscos para sua redução.

É um processo evolucionário, ou seja, adequado para softwares que precisam passar por inúmeras evoluções na medida que o desenvolvimento acontece.

Diferente do Modelo Incremental, que entrega partes prontas uma de cada vez, o Modelo Espiral é mais interativo e tenta fazer sucessivos refinamentos. Outras novidades, são os novos conceitos de Prototipagem e Gerenciamento de Riscos.

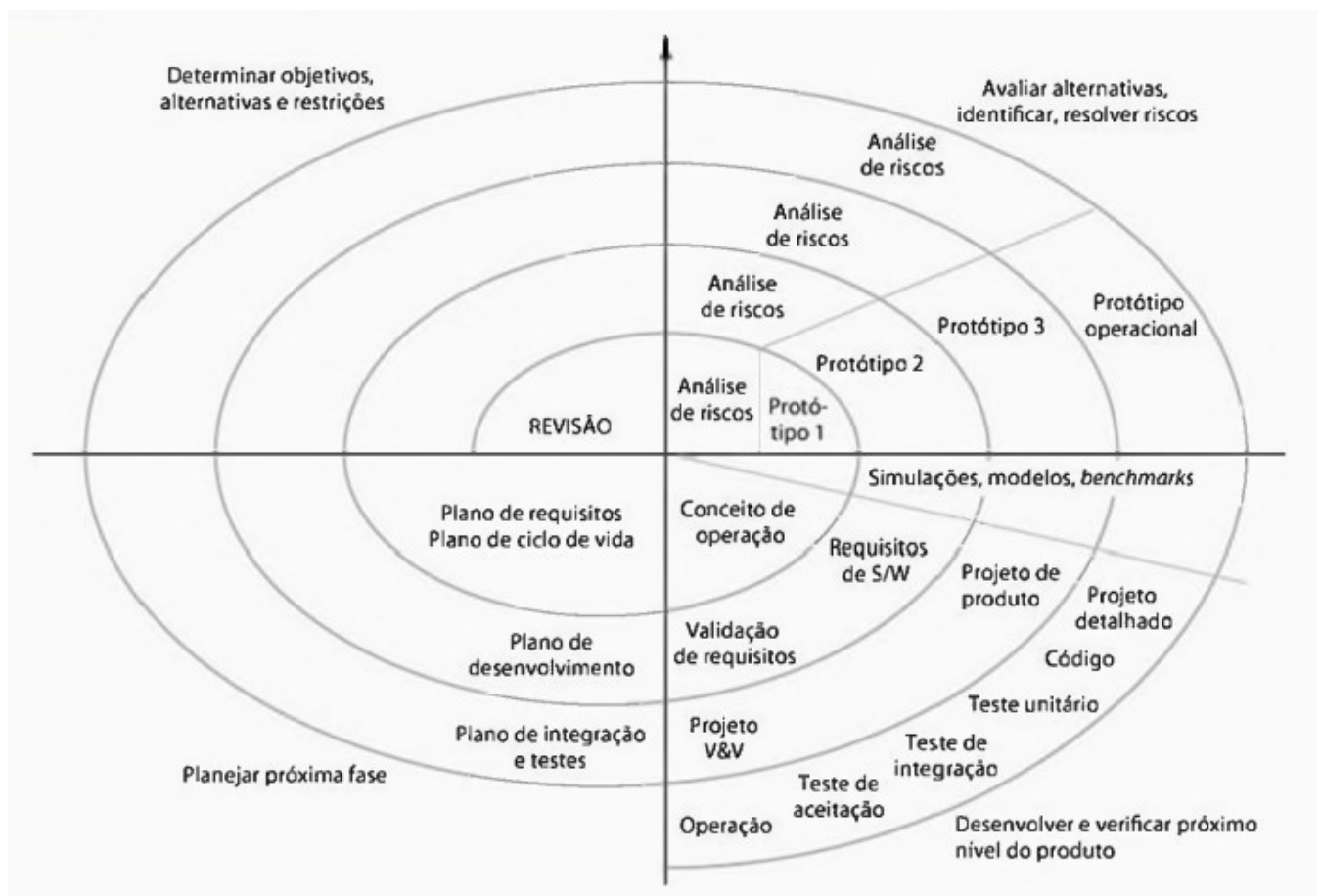
Na espiral que representa o modelo, a volta mais interna pode preocupar-se com a viabilidade do sistema; o ciclo seguinte, com definição de requisitos; o seguinte, com o projeto do sistema, e assim por diante.

De acordo com Sommerville, cada volta é dividida em quatro setores:



<b>Definição de objetivos</b>	Objetivos específicos para essa fase do projeto são definidos; restrições ao processo e ao produto são identificadas, e um plano de gerenciamento detalhado é elaborado; os riscos do projeto são identificados. Podem ser planejadas estratégias alternativas em função desses riscos.
<b>Avaliação e redução de riscos</b>	Para cada um dos riscos identificados do projeto, é feita uma análise detalhada. Medidas para redução do risco são tomadas. Por exemplo, se houver risco de os requisitos serem inadequados, um protótipo de sistema pode ser desenvolvido.
<b>Desenvolvimento e validação</b>	Após a avaliação dos riscos, é selecionado um modelo de desenvolvimento para o sistema. Por exemplo, a prototipação descartável pode ser a melhor abordagem de desenvolvimento de interface de usuário se os riscos forem dominantes. Se os riscos de segurança forem a principal consideração, o desenvolvimento baseado em transformações formais pode ser o processo mais adequado, e assim por diante. Se o principal risco identificado for a integração de subsistemas, o modelo em cascata pode ser a melhor opção.
<b>Planejamento</b>	O projeto é revisado, e uma decisão é tomada a respeito da continuidade do modelo com mais uma volta da espiral. Caso se decida pela continuidade, planos são elaborados para a próxima fase do projeto.

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 33).



Finaliza, Sommerville, a lição:

*“A principal diferença entre o modelo espiral e outros modelos de processo de software é seu reconhecimento explícito do risco. Um ciclo da espiral começa com a definição de objetivos, como desempenho e funcionalidade. Em seguida, são enumeradas formas alternativas de atingir tais objetivos e de lidar com as restrições de cada um deles. Cada alternativa é avaliada em função de cada objetivo, e as fontes de risco do projeto são identificadas. O próximo passo é resolver esses riscos por meio de atividades de coleta de informações, como análise mais detalhada, prototipação e simulação.”*

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 33).

## 5 – Década de 90 e o Processo Unificado/RUP

Primeiramente, objetivando elucidar a evolução da tecnologia, vamos listar algumas das linguagens mais importantes que foram desenvolvidas durante este período:

1990 - **Haskell**

1991 - **Python**

1991 - **Java**

1993 - **R**

1993 - **Lua**

1994 - **ANSI Common Lisp**

1995 - **Delphi** (Object Pascal)

1995 - **JavaScript**

1995 - **PHP**

1995 – **Ruby**

Ato contínuo, passaremos a desenvolver o tópico acerca do Processo Unificado (Unified Process).

O **Processo Unificado** foi criado em 1995 por Jim Rumbaugh, Grady Booch e Ivar Jacobson para suportar o paradigma de Orientação a Objetos. Faziam parte da **Rational Software Corporation**, onde concluíram a elaboração da **UML (Unified Modeling Language)** e a lançaram em 1996.

Em 2003, a IBM adquiriu o **Rational Unified Process**, ou, **RUP**; como passou a ser chamada a versão melhorada do Processo Unificado, que fornecia “**uma forma sistemática para se obter vantagens no uso da UML**”.

**A principal novidade é o acréscimo da UML para modelagem do sistema.**

Esta metodologia, objetiva realizar um maior controle sobre os resultados obtidos, gerenciar mudanças, e, fomentar um produto de qualidade estável.

Os princípios essenciais do RUP são:

- atacar os riscos cedo e continuamente;
- entregar algo de valor ao cliente;
- focar em um software que possa ser utilizado o quanto antes;
- realizar mudanças cedo;
- liberar protótipos da aplicação;
- utilizar componentes reutilizáveis;
- trabalhar como um time;
- fazer da qualidade um estilo de vida.

Tratemos aqui de uma discussão dos criadores do processo objetivando sua apreciação:

*“Hoje, a tendência em software é em direção a sistemas maiores, mais complexos. Isso se deve, em parte, ao fato de que os computadores têm se tornado mais potentes a cada ano, levando os usuários a esperar mais deles. Essa tendência tem também sido influenciada pelo uso da internet, que está se expandindo, para trocar toda espécie de informação... Nosso apetite por softwares cada vez mais sofisticados cresce à medida que aprendemos de uma versão de produto para a seguinte como o produto poderia ser aperfeiçoado. Desejamos softwares que sejam melhor adaptados às nossas necessidades, mas que, por sua vez, não torne o software somente mais complexo. Em resumo, desejamos mais.”*

(JACONSON, 1999)

O RUP, reconhece que os modelos de processo convencionais apresentam uma visão única do processo. Em contrapartida, o RUP é normalmente descrito em três perspectivas.

Nos ensina Sommerville:

<b>PERSPECTIVA DINÂMICA</b>	Mostra as fases do modelo ao longo do tempo.
<b>PERSPECTIVA ESTÁTICA</b>	Mostra as atividades realizadas no processo.
<b>PERSPECTIVA PRÁTICA</b>	Sugere boas práticas a serem usadas durante o processo.

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 34).

O RUP, **é um modelo constituído de fases que identifica quatro fases distintas no processo de software.** No entanto, ao contrário do modelo em cascata, no qual as fases são equalizadas com as atividades do processo, **as fases do RUP são estreitamente relacionadas ao negócio, e não, a assuntos técnicos.**

Continuando com Sommerville, são elas:

<b>Concepção</b>	O objetivo da fase de concepção é estabelecer um business case para o sistema. Você deve identificar todas as entidades externas (pessoas e sistemas) que vão interagir com o sistema e definir as interações. Então, você deve usar essas informações para avaliar a contribuição do sistema para o negócio. Se essa contribuição for pequena, então o projeto poderá ser cancelado depois dessa fase.
<b>Elaboração</b>	As metas da fase de elaboração são desenvolver uma compreensão do problema dominante, estabelecer um framework da arquitetura para o sistema, desenvolver o plano do projeto e identificar os maiores riscos do projeto. No fim dessa fase, você deve ter um modelo de requisitos para o sistema, que pode ser um conjunto de casos de uso da UML, uma descrição da arquitetura ou um plano de desenvolvimento do software.
<b>Construção</b>	A fase de construção envolve projeto, programação e testes do sistema. Durante essa fase, as partes do sistema são desenvolvidas em paralelo e integradas. Na conclusão dessa fase, você deve ter um sistema de software já funcionando, bem como a documentação associada pronta para ser entregue aos usuários.
<b>Transição</b>	A fase final do RUP implica transferência do sistema da comunidade de desenvolvimento para a comunidade de usuários e em seu funcionamento em um ambiente real. Isso é ignorado na maioria dos modelos de processo de software, mas é, de fato, uma atividade cara e, às vezes, problemática. Na conclusão dessa fase, você deve ter um sistema de software documentado e funcionando corretamente em seu ambiente operacional.

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 34).

No RUP, a iteração é apoiada de duas maneiras:

(a) Cada fase pode ser executada de forma iterativa com os resultados desenvolvidos de forma incremental.

(b) A visão estática do RUP prioriza as atividades que ocorrem durante o processo de desenvolvimento. Na descrição do RUP, essas são chamadas workflows. Existem seis workflows centrais, identificadas no processo, e três workflows de apoio. O RUP foi projetado em conjunto com a UML, assim, a descrição do workflow é orientada em torno de modelos associados à UML, como modelos de sequência, modelos de objetos etc.



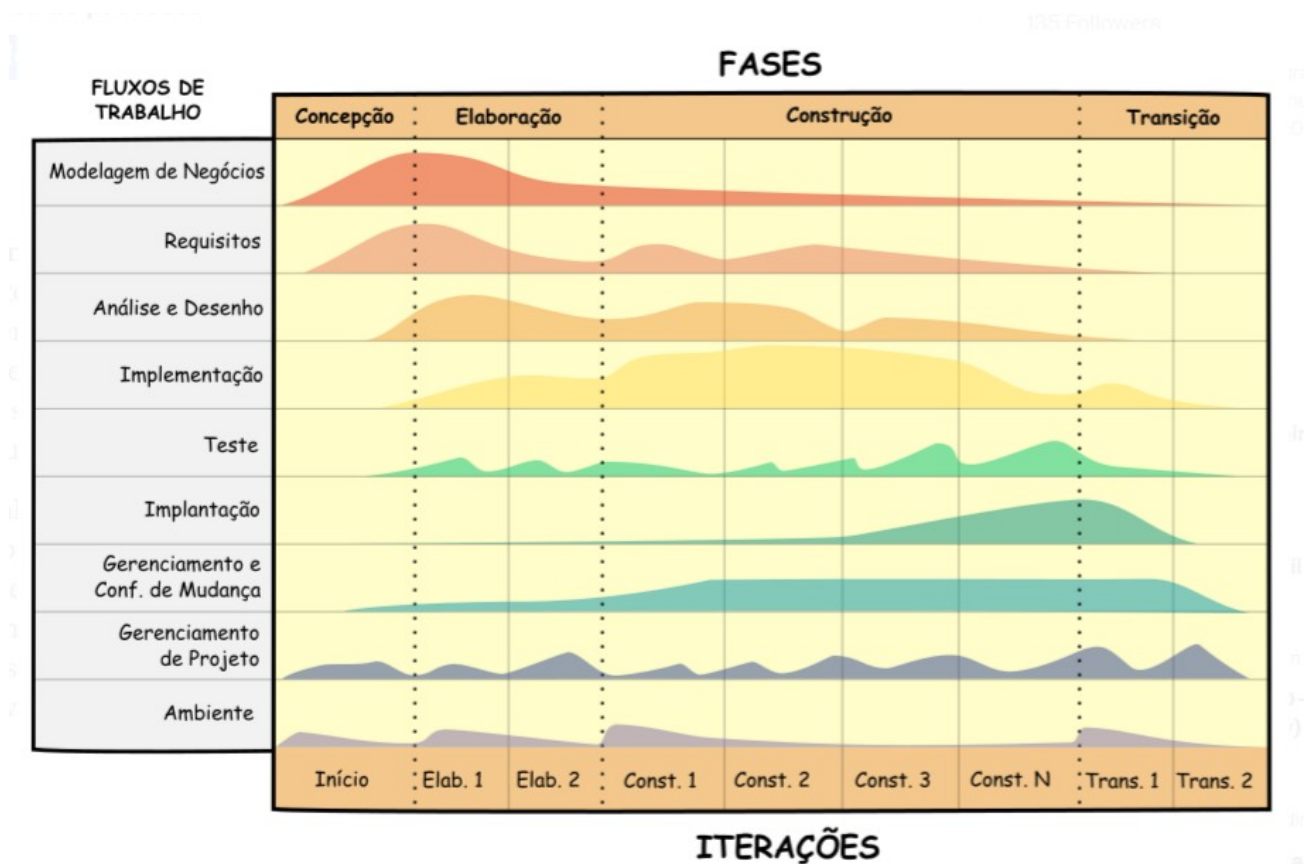
A perspectiva prática sobre o RUP descreve seis boas práticas fundamentais para uso no desenvolvimento de sistemas. Assim, as descreve Sommerville:

<b>Desenvolver software iterativamente</b>	Planejar os incrementos do sistema com base nas prioridades do cliente e desenvolver os recursos de alta prioridade no início do processo de desenvolvimento.
<b>Gerenciar os requisitos</b>	Documentar explicitamente os requisitos do cliente e acompanhar suas mudanças. Analisar o impacto das mudanças no sistema antes de aceitá-las.
<b>Usar arquiteturas baseadas em componentes</b>	Estruturar a arquitetura do sistema em componentes, conforme discutido anteriormente neste capítulo.
<b>Modelar o software visualmente</b>	Usar modelos gráficos da UML para apresentar visões estáticas e dinâmicas do software.
<b>Verificar a qualidade do software</b>	Assegurar que o software atenda aos padrões de qualidade organizacional.
<b>Controlar as mudanças do software</b>	Gerenciar as mudanças do software, usando um sistema de gerenciamento de mudanças e procedimentos e ferramentas de gerenciamento de configuração.

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 35).

**O RUP não é um processo adequado para todos os tipos de desenvolvimento, no entanto, ele representa uma abordagem que potencialmente combina os três modelos de processo genéricos: cascata, incremental, e, reúso.**

As inovações mais importantes do RUP são a separação de fases e workflows e o reconhecimento de que a implantação de software em um ambiente do usuário é parte do processo. As fases são dinâmicas e têm metas. Os workflows são estáticos e são atividades técnicas que não são associadas a uma única fase, mas podem ser utilizadas durante todo o desenvolvimento para alcançar as metas específicas.



Essas fases servem de apoio para a execução das tarefas de cada um dos Fluxos de Trabalho, sendo seis da Engenharia de Software e de três para Apoio e Suporte:

<b>Modelagem de negócios</b>	Os processos de negócio são modelados por meio de casos de uso de negócios.
<b>Requisitos</b>	Atores que interagem com o sistema são identificados e casos de uso são desenvolvidos para modelar os requisitos do sistema.
<b>Análise e projeto</b>	Um modelo de projeto é criado e documentado com modelos de arquitetura, modelos de componentes, modelos de objetos e modelos de sequência.
<b>Implementação</b>	Os componentes do sistema são implementados e estruturados em subsistemas de implementação. A geração automática de código a partir de modelos de projeto ajuda a acelerar esse processo.
<b>Teste</b>	O teste é um processo iterativo que é feito em conjunto com a implementação. O teste do sistema segue a conclusão da implementação.
<b>Implantação</b>	Um release do produto é criado, distribuído aos usuários e instalado em seu local de trabalho.

<b>Gerenciamento de configuração e mudanças</b>	Esse workflow de apoio gerencia as mudanças do sistema.
<b>Gerenciamento de projetos</b>	Esse workflow de apoio gerencia o desenvolvimento do sistema.
<b>Meio ambiente</b>	Esse workflow está relacionado com a disponibilização de ferramentas apropriadas para a equipe de desenvolvimento de software.

---

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 35).

### **Vantagens**

- É um dos modelos mais usados em projetos modernos de desenvolvimento de software;
- É extremamente robusto e bem estruturado;
- Por fazer uso da UML, a facilidade de compreensão dos requisitos aumenta, tanto para os clientes, como para os analistas e desenvolvedores;
- Os riscos que poderiam ser mais preocupantes são resolvidos no início, minimizando o retrabalho, e, conseqüentemente, a chance de fracasso do projeto.

### **Desvantagens**

- Exige uma equipe experiente e alinhada com os processos;
- Exigem um bom gerente de projetos;
- Complexo e trabalhoso para projetos pequenos.

## 6 – Anos 2000

### 6.1 - Desenvolvimento Ágil de Software vs. Pós Crise de Software

Como visto, o Processo Unificado, inovou no desenvolvimento do software orientado a objetos. No entanto, embora seu formato fosse estruturado, e, a introdução da UML tenha sido um grande avanço; muitas dificuldades ainda persistiam na tentativa de obter o controle do projeto. A maior delas era a quantidade de artefatos para controlar o processo que dificultava a adoção por pequenos cases.

Antes de aprofundar em novos modelos de processo de software, é importante lembrar da situação em que a engenharia de software se encontrava nesse momento, e, como os eventos desta época impactaram no futuro da engenharia.

Na virada do milênio, começaram a ganhar visibilidade vários processos alternativos, que prometiam uma entrega mais rápida de software, e, ao mesmo tempo, focavam num melhor relacionamento com o cliente.

Neste cenário, em 2001, um grupo de profissionais da indústria, se reuniu na cidade de Snowbird no estado norte-americano de Utah para discutir e propor uma alternativa aos processos do tipo Waterfall, que até então predominavam. Essencialmente, eles passaram a defender que software é diferente de produtos tradicionais de Engenharia. Por isso, software também demanda um processo de desenvolvimento diferente.

Este pensamento, foi precursor do lançamento das bases para um novo conceito de processo de software, as quais foram registradas em um documento que chamado: **Manifesto para Desenvolvimento Ágil de Software**.

Formando assim, uma declaração com os princípios que regem o **desenvolvimento ágil**.



Da esquerda, Stephen Mellor, Ward Cunningham, Dave Thomas, Andrew Hunt, Martin Fowler e James Grenning.



O Manifesto se tornou o embasamento filosófico para os chamados **Métodos Ágeis**, universalmente, baseiam-se em uma abordagem incremental para a especificação, o desenvolvimento, e, a entrega do software. Eles são mais adequados ao desenvolvimento de aplicativos, nos quais os requisitos de sistema, mudam rapidamente durante o processo de desenvolvimento.

Porém, temos que fazer uma pausa e refletir o momento para compreender essa mudança tão radical no processo de software.

As empresas, operam em um ambiente global com mudanças rápidas. Assim, precisam responder a novas oportunidades, novos mercados, à mudanças nas condições econômicas, e, ao surgimento de produtos e serviços concorrentes.

Softwares, fazem parte de quase todas as operações de negócios; assim, novos softwares são desenvolvidos rapidamente para obterem proveito de novas oportunidades, e, responderem às pressões competitivas. O desenvolvimento e entrega rápidos são, portanto, o requisito mais crítico para o desenvolvimento de softwares.

Desta forma, processos de software que planejam especificar completamente os requisitos, e, em seguida, projetar, construir e testar o sistema, não estão adaptados ao desenvolvimento rápido de software. Com as mudanças nos requisitos, ou, a descoberta de problemas de requisitos; o projeto do sistema ou sua implementação precisa ser refeito, ou ainda, retestado. Como consequência, um processo convencional em cascata, ou, baseado em especificações, costuma ser demorado, e, o software final, acaba sendo entregue ao cliente bem depois do prazo.

**Os métodos ágeis são métodos de desenvolvimento incremental em que os incrementos são pequenos e, normalmente, as novas versões do sistema são criadas e disponibilizadas aos clientes a cada duas ou três semanas. Elas envolvem os clientes no processo de desenvolvimento para obter feedback rápido sobre a evolução dos requisitos. Assim, minimiza-se a documentação, pois, se utiliza mais a comunicação informal, do que reuniões formais com documentos escritos.**

Tendo-se o exposto, apresenta-se a filosofia do **Manifesto Ágil**:

*“Estamos descobrindo melhores maneiras de desenvolver softwares, fazendo-o e ajudando outros a fazê-lo. A través desse trabalho, valorizamos mais:*

***Indivíduos e interações*** do que processos e ferramentas.

***Software em funcionamento*** do que documentação abrangente.

***Colaboração do cliente*** do que negociação de contrato.

***Respostas a mudanças*** do que seguir um plano.

*Ou seja, embora itens à direita sejam importantes, valorizam os mais os que estão à esquerda.”*

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 40).

Nos ensina Sommerville, acerca da metodologia trazida com o Manifesto Ágil:

Princípios	Descrição
Envolvimento do cliente	Os clientes devem estar intimamente envolvidos no processo de desenvolvimento. Seu papel é fornecer e priorizar novos requisitos do sistema e avaliar suas iterações.
Entrega incremental	O software é desenvolvido em incrementos com o cliente, especificando os requisitos para serem incluídos em cada um.
Pessoas, não processos	As habilidades da equipe de desenvolvimento devem ser reconhecidas e exploradas. Membros da equipe devem desenvolver suas próprias maneiras de trabalhar, sem processos prescritivos.
Aceitar as mudanças	Deve-se ter em mente que os requisitos do sistema vão mudar. Por isso, projete o sistema de maneira a acomodar essas mudanças.
Manter a simplicidade	Focalize a simplicidade, tanto do software a ser desenvolvido quanto do processo de desenvolvimento. Sempre que possível, trabalhe ativamente para eliminar a complexidade do sistema.

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 40).

Modelos de Processo de Software em Método Ágil:

- **Extreme Programming** (BECK, 1999; BECK, 2000)
- **Scrum** (COHN, 2009; SCHWABER, 2004; SCHWABER e BEEDLE, 2001),
- **Crystal** (COCKBURN, 2001; COCKBURN, 2004),
- **Desenvolvimento de Software Adaptativo - ASD** (Adaptative Software Development), (HIGHSMITH, 2000),
- **DSDM** (STAPLETON, 1997; STAPLETON, 2003)
- **Desenvolvimento Dirigido a Características** (Feature Teste Driven Development - TDD, Desenvolvimento Orientado a Teste), (PALMER e FELSING, 2002).

Após a exposição supra, pudemos ter uma visão mais clara das consequências da Crise do Software. Como pôde ser observado, esta, não foi apenas um momento no tempo ao qual se observa a década de 70 como exclusiva no índice cronológico. Ao contrário, toda a cadeia evolutiva dos processos que ocorreram após a crise de 70, desencadearam um efeito como “bola de neve” lançada do alto da montanha. Até a base, a pequena bolinha ganhou proporções exponenciais, vindo assim, a trazer uma nova forma de pensar processos de engenharia, envolver o cliente, e, focar na sua satisfação.

Os principais métodos ágeis, serão a seguir, intimamente detalhados. Com isto, ao final, teremos uma visão panorâmica de grande valia a compreender os acontecimentos neste lapso temporal com maior exatidão, e, assim, as consequências geradas pela crise de 70.

## 6.2 - Extreme Programming – XP

Extreme Programming – XP, é talvez o mais conhecido e mais utilizado dos métodos ágeis. O nome foi cunhado por Beck em 2000, pois, a abordagem foi desenvolvida para impulsionar práticas como o desenvolvimento iterativo a níveis extremos. Por exemplo, em XP, **várias novas versões de um sistema podem ser desenvolvidas, integradas e testadas em um único dia por programadores diferentes.**

Neste processo, os requisitos são expressos como cenários, chamados de histórias do usuário. Estas, são implementadas diretamente como uma série de tarefas. Os programadores trabalham em pares e desenvolvem testes para cada tarefa antes de escreverem o código. Quando o novo código é integrado ao sistema, todos os testes devem ser executados com sucesso. Havendo, outrossim, um curto intervalo entre os releases do sistema.

Vejamos o que Sommerville nos dita acerca do tema:

*“1. O desenvolvimento incremental é sustentado por meio de pequenos e frequentes releases do sistema. Os requisitos são baseados em cenários ou em simples histórias de clientes, usadas como base para decidir a funcionalidade que deve ser incluída em um incremento do sistema.*

*2. O envolvimento do cliente é sustentado por meio do engajamento contínuo do cliente com a equipe de desenvolvimento. O representante do cliente participa do desenvolvimento e é responsável por definir os testes de aceitação para o sistema.*

*3. Pessoas — não processos — são sustentadas por meio de programação em pares, propriedade coletiva do código do sistema e um processo de desenvolvimento sustentável que não envolve horas de trabalho excessivamente longas.*

*4. As mudanças são aceitas por meio de releases contínuos para os clientes, do desenvolvimento test-first, da refatoração para evitar a degeneração do código e integração contínua de nova funcionalidade.*

*5. A manutenção da simplicidade é feita por meio da refatoração constante que melhora a qualidade do código, bem como por meio de projetos simples que não antecipam desnecessariamente futuras mudanças no sistema.”*

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 44).

Em um processo XP, os clientes estão intimamente envolvidos na especificação e priorização dos requisitos do sistema. Os requisitos não estão especificados como uma lista de funções requeridas do sistema. Pelo contrário, **o cliente do sistema é parte da equipe de desenvolvimento e discute cenários com outros membros da equipe.** Juntos, eles desenvolvem um **cartão de história**, englobando as necessidades do cliente. A equipe de desenvolvimento, então, tenta implementar esse cenário em um release futuro do software.

**Os cartões de história são as principais entradas para o processo de planejamento em XP, ou, jogo de planejamento.** Uma vez que tenham sido desenvolvidos, a equipe de desenvolvimento os divide em tarefas e estima o esforço, e, os recursos necessários para a realização de cada tarefa. Esse processo geralmente envolve discussões com o cliente para refinamento dos requisitos. O cliente, então, prioriza as histórias para implementação, escolhendo aquelas que podem ser usadas imediatamente para oferecer apoio aos negócios.

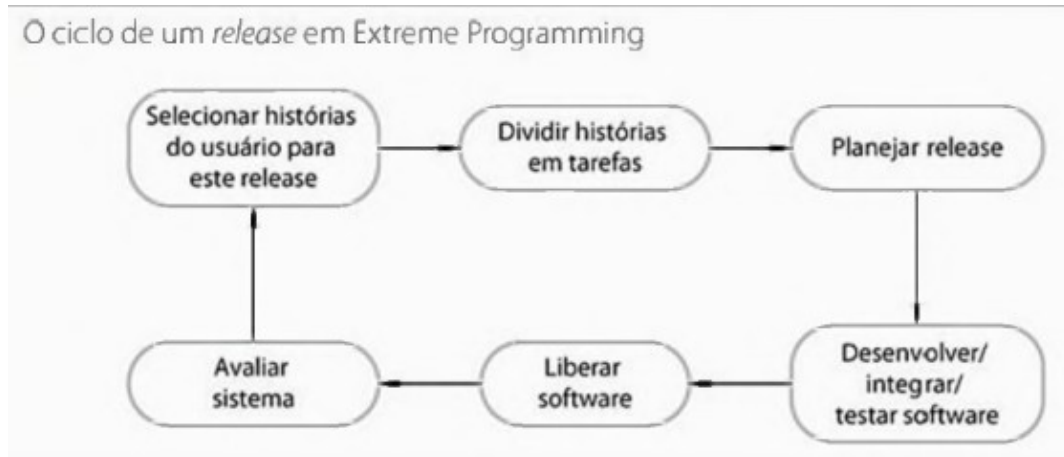
A intenção é identificar funcionalidade útil que possa ser implementada em cerca de duas semanas, quando o próximo release do sistema é disponibilizado para o cliente. Extreme Programming leva uma abordagem extrema para o desenvolvimento incremental. **Novas versões do software podem ser construídas várias vezes por dia, e, releases são entregues aos clientes a cada duas semanas, aproximadamente. Prazos de releases nunca são desrespeitados;** se houver problemas de desenvolvimento, o cliente é consultado, e, a funcionalidade é removida do release planejado.

**Na prática, muitas empresas que adotaram XP não usam todas as práticas da Extreme Programming listadas abaixo. Elas escolhem de acordo com sua organização.** No entanto, a maioria das empresas que adotaram uma variante de XP usa releases de pequeno porte, desenvolvimento do test-first e integração contínua.

Prática	Descrição
Planejamento incremental	Os requisitos são gravados em cartões de história e as histórias que serão incluídas em um release são determinadas pelo tempo disponível e sua relativa prioridade. Os desenvolvedores dividem essas histórias em Tarefas.
Pequenos releases	Em primeiro lugar, desenvolve-se um conjunto mínimo de funcionalidades útil, que fornece o valor do negócio. Releases do sistema são frequentes e gradualmente adicionam funcionalidade ao primeiro release.
Projeto simples	Cada projeto é realizado para atender às necessidades atuais, e nada mais.
Desenvolvimento test-first	Um framework de testes iniciais automatizados é usado para escrever os testes para uma nova funcionalidade antes que a funcionalidade em si seja implementada.
Refatoração	Todos os desenvolvedores devem refatorar o código continuamente assim que encontrarem melhorias de código. Isso mantém o código simples e manutenível.
Programação em pares	Os desenvolvedores trabalham em pares, verificando o trabalho dos outros e prestando apoio para um bom trabalho sempre.
Propriedade coletiva	Os pares de desenvolvedores trabalham em todas as áreas do sistema, de modo que não se desenvolvam ilhas de expertise. Todos os conhecimentos e todos os desenvolvedores assumem responsabilidade por todo o código. Qualquer um pode mudar qualquer coisa.
Integração contínua	Assim que o trabalho em uma tarefa é concluído, ele é integrado ao sistema como um todo. Após essa integração, todos os testes de unidade do sistema devem passar.
Ritmo sustentável	Grandes quantidades de horas-extra não são consideradas aceitáveis, pois o resultado final, muitas vezes, é a redução da qualidade do código e da produtividade a médio prazo.

Cliente no local	Um representante do usuário final do sistema (o cliente) deve estar disponível todo o tempo à equipe de XP. Em um processo de Extreme Programming, o cliente é um membro da equipe de desenvolvimento e é responsável por levar a ela os requisitos de sistema para implementação.
------------------	--

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 45).



Vejamos as lições de Marco Tulio Valente sobre o tema ora apresentado:

*“XP é um método leve recomendado para desenvolver software com requisitos vagos ou sujeitos a mudanças; isto é, basicamente sistemas comerciais. Sendo um método ágil, XP adota ciclos curtos e iterativos de desenvolvimento, concede menos ênfase para documentação e para planos detalhados, propõe que o design de um sistema também seja definido de forma incremental e sugere que as equipes de desenvolvimento sejam pequenas.*

*Porém, XP não é um método prescritivo, que define um passo a passo detalhado para construção de software. Em vez disso, XP é definido por meio de um conjunto de valores, princípios e práticas de desenvolvimento.*

*Ou seja, XP é inicialmente definido de forma abstrata, usando-se de valores e princípios que devem fazer parte da cultura e dos hábitos de times de desenvolvimento de software. Depois, esses valores e princípios são concretizados em uma lista de práticas de desenvolvimento.*

*Frequentemente, quando decidem adotar XP, desenvolvedores e organizações concentram-se nas práticas. Porém, os valores e princípios são componentes chaves do método, pois **são eles que dão sentido às práticas propostas em XP.***

**Valores:** comunicação, simplicidade, feedback, coragem, respeito e qualidade de vida.

**Princípios:** humanidade, economicidade, benefícios mútuos, melhorias contínuas, falhas acontecem, baby steps e responsabilidade pessoal.

Resolvemos organizá-las em três grupos: **práticas sobre o processo de desenvolvimento, práticas de programação e práticas de gerenciamento de projetos.**

- > **Práticas sobre o Processo de Desenvolvimento:** representante dos clientes, histórias dos usuários, iterações, releases, planejamento de releases, planejamento de iterações, planning poker, slack.
- > **Práticas de Programação:** design incremental, programação pareada, desenvolvimento dirigido por testes (TDD), build automatizado, integração contínua.
- > **Práticas de Gerenciamento de Projetos:** métricas, ambiente de trabalho, contratos com escopo aberto.

XP defende que projetos de software devem seguir um conjunto de princípios. A imagem que se apresenta é de um rio: de um lado estão os valores e de outro as práticas. Os princípios fazem o papel de uma ponte ligando esses dois lados. Alguns dos principais princípios de XP são:

- > **Humanidade (humanity, em inglês).** Software é uma atividade intensiva no uso de capital humano.
- > **Economicidade (economics, em inglês).** Se por um lado, peopleware é fundamental, por outro lado software é uma atividade cara, que demanda a alocação de recursos financeiros consideráveis.
- > **Benefícios Mútuos.** XP defende que as decisões tomadas em um projeto de software têm que beneficiar múltiplos stakeholders.
- > **Melhorias Contínuas (no livro de XP, o nome original é improvements):** Como expressa a frase de Kent Beck que abre este capítulo, nenhum processo de desenvolvimento de software é perfeito.
- > **Falhas Acontecem.** Desenvolvimento de software não é uma atividade livre de riscos.
- > **Baby Steps.** É melhor um progresso seguro, testado e validado, mesmo que pequeno, do que grandes implementações com riscos de serem descartadas pelos usuários.
- > **Responsabilidade Pessoal** (que usamos como tradução para accepted responsibility).

XP — como outros métodos ágeis — **recomenda o envolvimento dos clientes com o projeto.** Ou seja, além de desenvolvedores, os times incluem pelo menos um representante dos clientes, que deve entender do domínio do sistema que será construído.

**Uma das funções desse representante é escrever as histórias de usuário (user stories), que é o nome que XP dá para os documentos que descrevem os requisitos do sistema a ser implementado.** No entanto, histórias são documentos resumidos, com **apenas duas ou três sentenças**, com as quais o representante dos clientes define o que ele deseja que o sistema faça, usando sua própria linguagem.

Depois de escritas pelo representante dos clientes, as histórias são estimadas pelos desenvolvedores. Ou seja, **são os desenvolvedores que definem, mesmo que preliminarmente, quanto tempo será necessário para implementar as histórias escritas pelo representante dos**

**clientes.** Frequentemente, a duração de uma história é estimada em story points, em vez de horas ou homens/hora.

**A implementação das histórias ocorre em iterações, as quais têm uma duração fixa e bem definida,** variando de uma a três semanas, por exemplo. **As iterações, por sua vez, formam ciclos mais longos, chamados de releases,** de dois a três meses, por exemplo. **A velocidade de um time é o número de story points que ele consegue implementar em uma iteração.** Sugere-se que o representante dos clientes escreva histórias que requeiram pelo menos uma release para serem implementadas. Ou seja, em XP, o horizonte de planejamento é uma release, isto é, alguns meses.

Em resumo, para começar a usar XP precisamos de:

- > Definir a duração de uma iteração.
- > Definir o número de iterações de uma release.
- > Um conjunto de histórias, escritas pelo representante dos clientes.
- > Estimativas para cada história, feitas pelos desenvolvedores.
- > Definir a velocidade do time, isto é, o número de story points que ele consegue implementar por iteração.

Resumindo, um projeto XP é organizado em:

- > **releases**, que são conjunto de iterações, com duração total de alguns meses.
- > **iteraões**, que são conjuntos de tarefas resultantes da decomposição de histórias, com duração total de algumas semanas.
- > **tarefas**, com duração de alguns dias.

Definidas as tarefas, o time deve decidir qual desenvolvedor será responsável por cada uma. Feito isso, começa de fato a iteração, com a implementação das tarefas. Uma iteração termina quando todas as suas histórias estiverem implementadas e validadas pelo representante dos clientes. Assim, ao fim de uma iteração, as histórias devem ser mostradas para o representante dos clientes, que deve concordar que elas, de fato, atendem ao que ele especificou.”

(Engenharia de Software Moderna, Marco Tulio Valente)

## 6.3 - Método Scrum

Como todos os outros processos profissionais de desenvolvimento de software, o desenvolvimento ágil tem de ser gerenciado de modo que se faça o melhor uso com o tempo, e, os recursos disponíveis para a equipe. Isso requer do gerenciamento de projeto uma abordagem diferente, adaptada para o desenvolvimento incremental, e, para os pontos fortes dos métodos ágeis.

A abordagem **Scrum** (SCHWABER, 2004; SCHWABER e BEEDLE, 2001) é um método ágil geral, mas **seu foco está no gerenciamento do desenvolvimento iterativo, ao invés das abordagens técnicas específicas da engenharia de software ágil.**

**Scrum não prescreve o uso de práticas de programação, como programação em pares e desenvolvimento test-first.** Portanto, pode ser usado com abordagens ágeis mais técnicas, como XP, para fornecer um framework de gerenciamento do projeto.

No Scrum, existem três fases:

- **Primeira fase:** planejamento geral, em que se estabelecem os objetivos gerais do projeto e da arquitetura do software.

- **Segunda fase:** ocorre uma série de ciclos de sprint, sendo que cada ciclo desenvolve um incremento do sistema.

- **Terceira fase:** responsável por completar a documentação exigida, como quadros de ajuda do sistema e manuais do usuário, e, avalia as lições aprendidas com o projeto.

A característica inovadora do Scrum é sua fase central, chamada **ciclos de sprint. Um sprint do Scrum é uma unidade de planejamento na qual o trabalho a ser feito é avaliado, os recursos para o desenvolvimento são selecionados, e, o software é implementado. No fim de um sprint, a funcionalidade completa é entregue aos stakeholders.**

Sommerville, nos apresenta as principais características desse processo, vejamos:

- |   |   |
|---|---|
| 1 | Sprints são de comprimento fixo, normalmente duas a quatro semanas. Eles correspondem ao desenvolvimento de um release do sistema em XP   |
| 2 | O ponto de partida para o planejamento é o backlog do produto, que é a lista do trabalho a ser feito no projeto. Durante a fase de avaliação do sprint, este é revisto, e as prioridades e os riscos são identificados. O cliente está intimamente envolvido nesse processo e, no início de cada sprint, pode introduzir novos requisitos ou tarefas.   |
| 3 | A fase de seleção envolve todos da equipe do projeto que trabalham com o cliente para selecionar os recursos e a funcionalidade a ser desenvolvida durante o sprint.  |
| 4 | Uma vez que todos estejam de acordo, a equipe se organiza para desenvolver o software. Reuniões diárias rápidas, envolvendo todos os membros da equipe, são realizadas para analisar os progressos e, se necessário, repriorizar o trabalho. Nessa etapa, a equipe está isolada do cliente e da organização, com todas as comunicações canalizadas por meio do chamado 'Scrum Master'. O papel do Scrum Master é proteger a equipe de desenvolvimento de distrações externas. A maneira como o trabalho é desenvolvido depende do problema e da equipe. Diferentemente do XP, a abordagem |



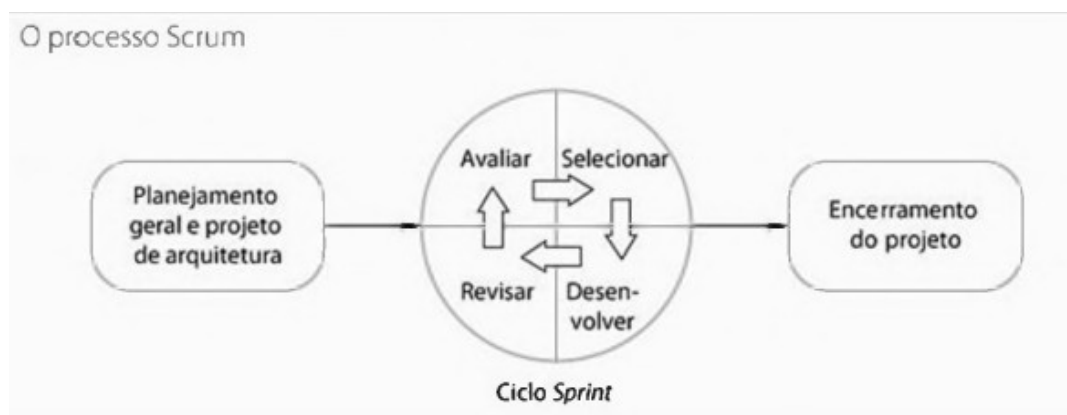
Scrum não faz sugestões específicas sobre como escrever os requisitos ou sobre o desenvolvimento test-first etc. No entanto, essas práticas de XP podem ser usadas se a equipe achar que são adequadas.

- 5 No fim do sprint, o trabalho é revisto e apresentado aos stakeholders. O próximo ciclo sprint começa em seguida.

(Engenharia de Software. Sommerville, Ian – 9ª Edição; página 50).

A ideia por trás do Scrum é que toda a equipe deve ter poderes para tomar decisões, de modo que o termo gerente de projeto tem sido deliberadamente evitado. Pelo contrário, o **Scrum Master é um facilitador**, que **organiza reuniões diárias, controla o backlog de trabalho, registra decisões, mede o progresso comparado ao backlog, e, se comunica com os clientes e a gerência externa à equipe.**

Toda a equipe participa das reuniões diárias; às vezes, estas são feitas com os participantes em stand-up, muito rápidas, para a manutenção do foco da equipe. Durante a reunião, todos os membros da equipe compartilham informações, descrevem seu progresso desde a última reunião, os problemas que têm surgido, e, o que está planejado para o dia seguinte. Isso garante que todos na equipe saibam o que está acontecendo e se surgirem problemas, poderão replanejar o trabalho de curto prazo para lidar com eles. Todos participam desse planejamento de curto prazo; não existe uma hierarquia a partir do Scrum Master.



O Scrum, como originalmente concebido, foi projetado para uso de equipes localizadas, em que todos os membros poderiam se encontrar todos os dias em reuniões rápidas. No entanto, muito do desenvolvimento de software atual envolve equipes distribuídas, ou seja, com membros da equipe situados em diferentes lugares ao redor do mundo. Consequentemente, estão em curso várias experiências para desenvolvimento Scrum em ambientes de desenvolvimento distribuído.

Objetivando ampliar nosso conhecimento acerca do assunto, nos ensina Marco Tulio Valente:

*“Dentre os métodos ágeis, Scrum é também aquele que é melhor definido. Essa definição inclui um conjunto preciso de papéis, artefatos e eventos, que são listados a seguir. No resto desta seção, vamos explicar cada um deles.*

**Papéis:** *Dono do Produto, Scrum Master, Desenvolvedor.*

**Artefatos:** *Backlog do Produto, Backlog do Sprint, Quadro Scrum, Gráfico de Burndown.*

**Eventos:** *Planejamento do Sprint, Sprint, Reuniões Diárias, Revisão do Sprint, Retrospectiva.*

**(a) Papeis:**

Times Scrum são formados por um **Dono de Produto** (Product Owner ou apenas PO), um **Scrum Master** e de três a nove **desenvolvedores**.

> O **Dono do Produto**, como o próprio nome indica, deve possuir a visão do produto que será construído, sendo **responsável também por maximizar o retorno do investimento feito no projeto**. Cabe ao Dono do Produto **escrever as histórias dos usuários** e, por isso, ele deve estar sempre disponível para tirar dúvidas do time.

> O **Scrum Master** é um papel característico e único de Scrum. Trata-se do especialista em Scrum do time, sendo **responsável por garantir que as regras do método estão sendo seguidas**. Para isso, ele deve continuamente treinar e explicar os princípios de Scrum para os demais membros do time.

**(b) Artefatos e Eventos:**

Em Scrum, os dois artefatos principais são o **Backlog do Produto**, e, o **Backlog do Sprint**; e, os principais eventos são **sprints** e o **planejamento de sprints**, conforme descreveremos a seguir.






























> O **Backlog do Produto é uma lista de histórias, ordenada por prioridades**. As histórias são escritas e priorizadas pelo Dono do Produto e constituem uma descrição resumida das funcionalidades que devem ser implementadas no projeto. É importante mencionar ainda que o **Backlog do Produto é um artefato dinâmico, isto é, ele deve ser continuamente atualizado, de forma a refletir mudanças nos requisitos e na visão do produto**. Todas essas atualizações devem ser realizadas pelo Dono do Produto. Na verdade, é o fato de ser o dono do Backlog do Produto que faz o Dono do Produto receber esse nome.

> **Sprint é o nome dado por Scrum para uma iteração**. Ou seja, como todo método ágil, Scrum é um método iterativo, no qual o desenvolvimento é dividido em sprints, de até um mês. Ao final de um sprint, deve-se entregar um produto com valor tangível para o cliente. O resultado de um sprint é chamado de um produto potencialmente pronto para entrar em produção (potentially shippable product).

> O **Planejamento do Sprint é uma reunião na qual todo o time se reúne para decidir as histórias que serão implementadas no sprint que vai se iniciar**. Portanto, ele é o evento que marca o início de um sprint.

> O **Backlog do Sprint** é o artefato gerado ao final do **Planejamento do Sprint**. Ele é uma lista com as tarefas do sprint, bem como inclui a duração das mesmas. Como o Backlog do Produto, o Backlog do Sprint também é dinâmico. Porém, não pode ser alterado o objetivo do sprint (sprint goal), isto é, a lista de histórias que o dono do produto selecionou para o sprint e que o time de desenvolvimento se comprometeu a implementar na duração do mesmo. Assim, Scrum é um método adaptável a mudanças, mas desde que elas ocorram entre sprints.

Ao lado do Backlog do Sprint, costuma-se anexar um **quadro com tarefas** a fazer, em andamento e finalizadas. Esse quadro — também chamado de **Quadro Scrum (Scrum Board)** — pode ser fixado nas paredes do ambiente de trabalho, permitindo que o time tenha diariamente uma sensação visual sobre o andamento do sprint.

Backlog	To Do	Doing	Testing	Done
     	        	       	   	 

Exemplo de Quadro Scrum, mostrando as histórias selecionadas para o sprint e as tarefas nas quais elas foram quebradas. Cada tarefa nesse quadro pode estar em um dos seguintes estados: a fazer, em andamento, em teste ou concluída.

Um outro artefato comum em Scrum é o **Gráfico de Burndown**. A cada dia do sprint, esse gráfico mostra quantas horas são necessárias para se implementar as tarefas que ainda não estão concluídas.

### (c) Reuniões diárias, Revisão do Sprint, e, Retrospectiva:

> Scrum propõe que sejam realizadas **Reuniões Diárias, de cerca de 15 minutos, das quais devem participar todos os membros do time**. Essas reuniões para serem rápidas devem ocorrer com os membros em pé, daí serem também conhecidas como reuniões em pé (standup meetings, ou ainda daily scrum).

> A **Revisão do Sprint (Sprint Review)** é uma reunião para mostrar os resultados de um sprint. Dela devem participar todos os membros do time e idealmente outros stakeholders, convidados pelo Dono do Produto, que estejam envolvidos com o resultado do sprint. Durante essa reunião o time demonstra, ao vivo, o produto para os clientes. Como resultado, todas as histórias do sprint podem ser aprovadas pelo Dono do Produto. Por outro lado, caso ele

*detecte problema em alguma história, ela deve voltar para o Backlog do Produto, para ser retrabalhada em um próximo sprint.*

*> A **Retrospectiva é a última atividade de um sprint**. Trata-se de uma reunião do time Scrum, com o objetivo de refletir sobre o sprint que está terminando e, se possível, identificar pontos de melhorias no processo, nas pessoas, nos relacionamentos e nas ferramentas usadas.”*

(Engenharia de Software Moderna, Marco Tulio Valente)

## **6.4 - Método Lean**

Criado no Japão, o sistema Toyota de produção que também pode ser conhecido como Lean Manufacturing, surgiu logo após a Segunda Guerra mundial na fábrica da empresa automobilística Toyota. Nesta época, a indústria japonesa possuía uma produtividade muito baixa e sofria com a falta de recursos, o que consequentemente impedia de adotar o modelo de produção em massa.

O autor Franco (2007, p. 40) preleciona que:

*“a produção em massa era a forma mais barata de produzir carros, mas significava produzir um grande número de carros iguais e o mercado japonês não era suficiente para consumir uma quantidade grande de veículos iguais”.*

Segundo o Lean Institute Brasil (2010), a metodologia Lean, *é uma estratégia de negócios que busca aumentar a satisfação do cliente através de um melhor aproveitamento dos recursos. A gestão Lean, busca oferecer aos clientes um valor com o custo mais baixo dos seus produtos (propósito) através de melhorias contínuas dos seus fluxos de valor primário, e, de suporte (processos) através de pessoas com iniciativa, motivadas e qualificadas (pessoas).*

A metodologia Lean busca atender a necessidade do cliente da maneira mais simples possível, com um menor valor, aproveitando ao máximo todos os recursos disponíveis para a produção e tendo como consequência um melhor custo-benefício para o cliente.

Para Fadel e Silveira, 2010:

*“O desenvolvimento de software Lean é a aplicação dos conceitos do sistema de produção da Toyota para o desenvolvimento de software. Quando esta aplicação é feita corretamente, tem como consequência um desenvolvimento de alta qualidade que é feito rapidamente e com um baixo custo.”*

Com o intuito de aplicar esta metodologia diretamente em empresas de desenvolvimento de software, em meados de 2002, surgiu a abordagem inicial do desenvolvimento enxuto de software, desenvolvido por Bob Charette.

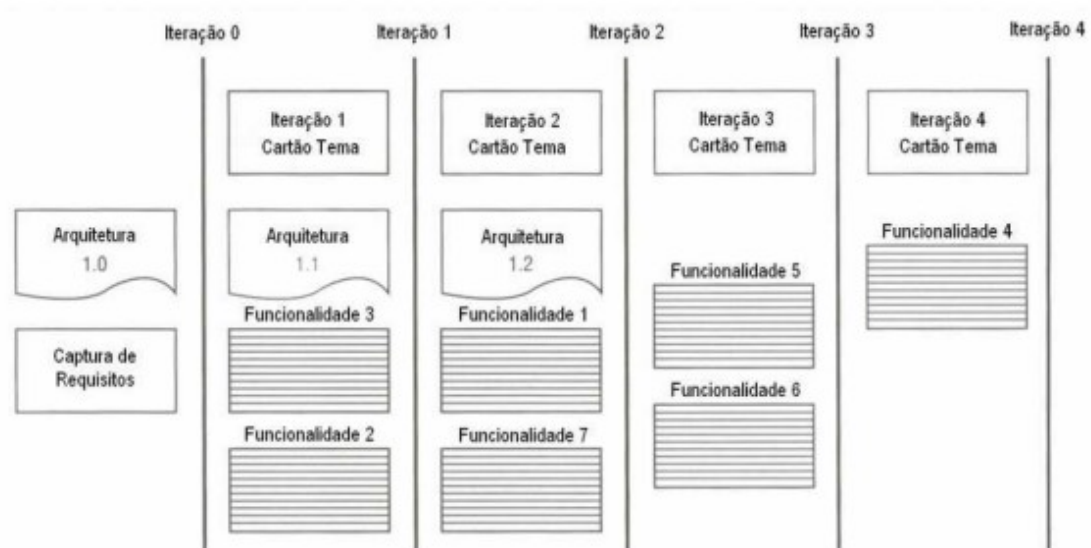
A metodologia Lean é distribuída em sete princípios:

1. eliminar o desperdício;
2. amplificar o aprendizado;
3. adiar comprometerimentos e manter a flexibilidade;
4. entregar rápido;
5. tornar a equipe responsável;
6. construir integridade;
7. visualizar o todo.

Neste sentido, completa Fadel e Silveira, 2010:

*“A metodologia Lean utiliza técnicas de produção puxada (pull) para agendar o trabalho e são dotadas de mecanismos com sinalizações locais, os quais ajudam os outros desenvolvedores a identificarem o trabalho que precisa ser realizado. No desenvolvimento de software Lean, esta técnica de produção puxada é correspondente à entrega de versões refinadas e incrementais do software em intervalos de tempo regulares.”*

O autor Franco (2010, p. 48) aborda que *“a sinalização local é feita através de gráficos visuais, reuniões diárias, integrações frequentes e testes automatizados”*.



As colunas representam a divisão do trabalho que necessita ser realizado a cada iteração. Nesta, existe um cartão tema, onde é definido o objetivo daquela determinada iteração, e, abaixo do cartão tema, são colocados os cartões que definem os requisitos que precisam ser implementados. O nivelamento da produção pode ser feito através da quantidade de trabalho que precisa ser utilizado para implementarem os requisitos descritos nos cartões.

## 6.5 - Método Kanban

A metodologia Kanban foi criada pela Toyota como uma forma de melhorar o processo de fabricação de veículos. Antes da sua aplicação, era comum a cadeia operacional ter um grande número de erros que elevavam os custos, e, reduziam o nível de produtividade. Para solucionar este tipo de problema, a Kanban foi criada. Ela não só otimizou processos, mas também, reduziu o desperdício de recursos, e, tornou a rotina de trabalho mais ágil e eficaz.

Como consequência, a entrega de produtos ganhou mais agilidade e precisão. A empresa foi capaz de garantir um nível de qualidade uniforme para toda a sua cadeia operacional, e assim, aumentar a sua competitividade.

Sobre o tema, nos ensina Marco Tulio Valente:

*“A palavra japonesa kanban significa cartão visual ou cartão de sinalização. Desde a década de 50, o nome também é usado para denotar o processo de produção just-in-time usado em fábricas japonesas, principalmente naquelas da Toyota, onde ele foi usado pela primeira vez. O processo também é conhecido como Sistema de Produção da Toyota (TPS) ou, mais recentemente, por manufatura lean. Em uma linha de montagem, os cartões são usados para controlar o fluxo de produção.*

*No caso de desenvolvimento de software, Kanban foi usado pela primeira vez na Microsoft, em 2004, como parte de um esforço liderado por David Anderson, então um funcionário da empresa (link). Segundo Anderson, Kanban é um método que ajuda times de desenvolvimento a trabalhar em ritmo sustentável, eliminando desperdício, entregando valor com frequência e fomentando uma cultura de melhorias contínuas.*

*Primeiro, Kanban é mais simples do que Scrum, pois não usa nenhum dos eventos de Scrum, incluindo sprints. Também, não existe nenhum dos papéis (Dono do Produto, Scrum Master, etc.), pelo menos da forma rígida preconizada por Scrum. Por fim, não existe nenhum dos artefatos Scrum, com uma única e central exceção: o quadro de tarefas, que é chamado de Quadro Kanban (Kanban Board), e que inclui também o Backlog do Produto.*

*O Quadro Kanban é dividido em colunas, da seguinte forma:*

- A primeira coluna é o backlog do produto. Como em Scrum, usuários escrevem as histórias, que vão para o Backlog.*
- As demais colunas são os passos que devem ser seguidos para transformar uma história do usuário em uma funcionalidade executável. Por exemplo, pode-se ter colunas como Especificação, Implementação e Revisão de Código. A ideia, portanto, é que as histórias sejam processadas passo a passo, da esquerda para a direita, como em uma linha de montagem. Além disso, cada coluna é dividida em duas subcolunas: em execução e concluídas. Por exemplo, a coluna implementação tem duas subcolunas: tarefas em implementação e tarefas*

*implementadas. As tarefas concluídas em um passo estão aguardando serem puxadas, por um membro do time, para o próximo passo. Por isso, Kanban é chamado de um sistema pull.*

*(...)*

*Como em outros métodos ágeis, times Kanban são auto-organizáveis. Isso significa que eles têm autonomia para definir qual tarefa vai ser puxada para o próximo passo. Eles também são cross-funcionais, isto é, devem incluir membros capazes de realizar todos os passos do Quadro Kanban.*

*Por fim, resta explicar o conceito de Limites WIP (Work in Progress). Via de regra, métodos de gerenciamento de projetos têm como objetivo garantir um ritmo sustentável de trabalho. Para isso, deve-se evitar duas situações extremas: (1) o time ficar ocioso boa parte do tempo, sem tarefa para realizar; ou (2) o time ficar sobrecarregado de trabalho e, por isso, não conseguir produzir software de qualidade. Para evitar a segunda situação — sobrecarga de trabalho — Kanban propõe um limite máximo de tarefas que podem estar em cada um dos passos de um Quadro Kanban. Esse limite é conhecido pelo nome Limite WIP, isto é, trata-se do limite máximo de cartões presentes em cada passo, contando aqueles na primeira coluna (em andamento) e aqueles na segunda coluna (concluídos) do passo. A exceção é o último passo, no qual o WIP aplica-se apenas à primeira subcoluna, já que não faz sentido aplicar um limite ao número de tarefas concluídas pelo time de desenvolvimento.”*

Em outras palavras, no ambiente ágil, os times atuarão com alta integração, trocando dados e solucionando problemas de maneira colaborativa. Isso evita conflitos e melhora o nível de inovação, uma vez que as escolhas serão feitas com mais pessoas pensando no melhor caminho a ser tomado.

A principal diferença da implementação da Kanban em relação a outras metodologias ágeis é a sua simplicidade. A maneira de gerenciar a execução de atividades (por meio de um grande quadro com peças que se movimentam de acordo com o progresso das tarefas que forem feitas), agiliza a gestão de rotinas, e, torna o trabalho do gestor mais simples.

Não há a necessidade de realizar múltiplas reuniões. Como consequência, os profissionais podem ficar focados em entregar resultados.

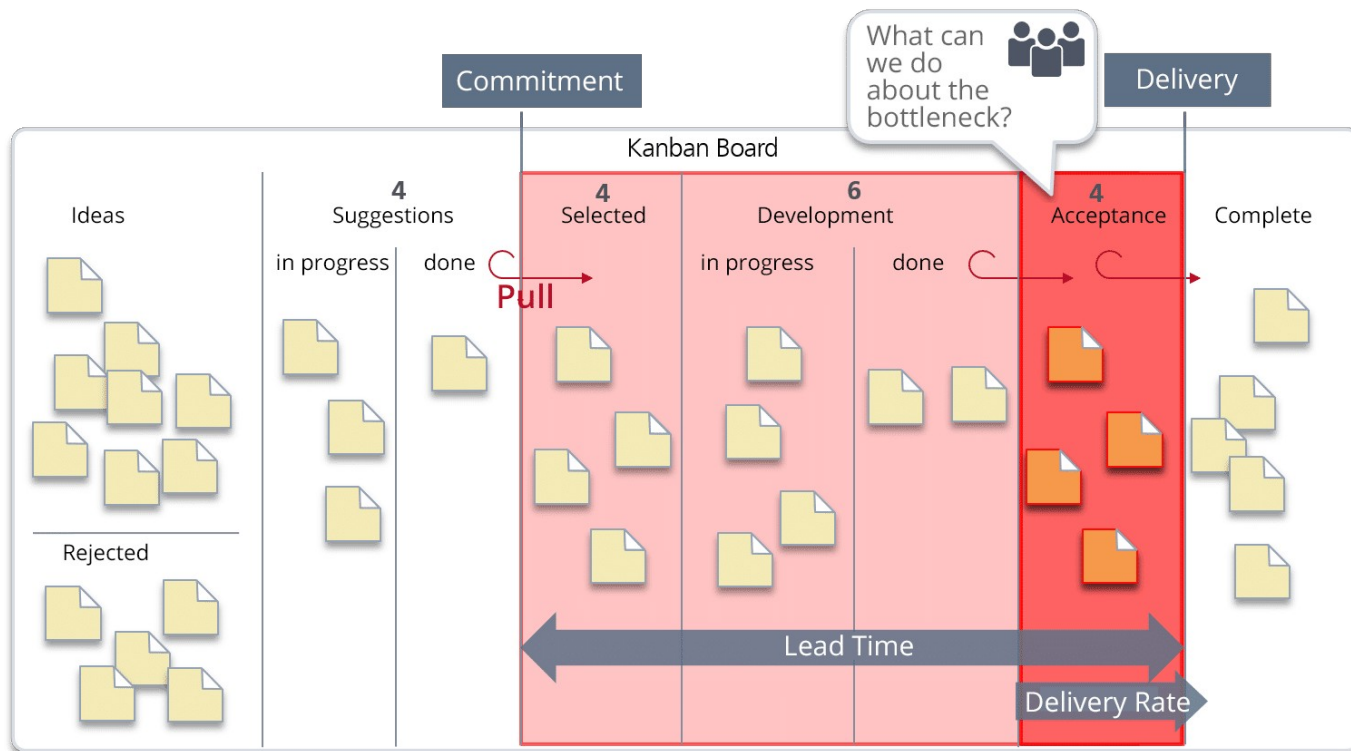
Além disso, o tempo gasto revisando rotinas cai drasticamente. Uma vez que todas as metas estão dispostas em um único local de fácil leitura, os membros do projeto podem identificar as demandas pendentes e em execução rapidamente. Desta forma, profissionais poderão focar mais na entrega de resultados e na busca por um trabalho de qualidade.

Em geral, para fazer uso do Kanban, a empresa deve seguir um conjunto de princípios básicos:

- **Priorizar rotinas**
- **Entender a necessidade de evoluir**
- **Respeitar a hierarquia interna**
- **Incentivar a liderança**
- **Incentivar a autonomia dos profissionais**



A metodologia Kanban deve ser vista como um processo de trabalho contínuo. A melhora destes processos deve ser incentivada sempre, reduzindo o número de erros, e, tornando as atividades mais fluidas. Assim, a equipe pode atingir os resultados esperados, e, conseguir atuar de maneira estratégica para solucionar problemas de clientes com ferramentas inteligentes e inovadoras.



# CONCLUSÃO

A crise do software foi apenas um momento no tempo, um momento onde o mercado queria crescer, um momento onde o mercado precisava de ação; um momento onde a programação deixava de ser um acessório, e, passava a ser uma necessidade ao avanço da própria sociedade.

Ao longo desta compilação acadêmica se pôde observar a linha tempo fluindo desde a década de 50 em um crescimento exponencial de novas linguagens de programação, hardwares, métodos, resultados, etc; e, cada vez mais, a interação entre o homem comum e a máquina, se tornando uma relação que hoje, se pode dizer quase que simbiótica.

O homem do passado, muitas vezes dizia-se avesso à “essas coisas de computador”, não entendia seu funcionamento, serventia, importância, e, assim faziam campanhas buscando a vida simples do campo como refúgio das cidades. Esse homem do campo, hoje, já não mais consegue viver sem os avanços da tecnologia.

Em nossa atualidade, o software desenvolveu-se a níveis tão inteligentes que colaboram drasticamente ao uso correto da terra, a observação da meteorologia, as máquinas que cuidam e tratam de longos hectares, as ferramentas informatizadas de produção agrícola... Enfim, a evolução do software não é apenas uma evolução didática, procedimental, ou, acadêmica. Cada alteração feita, cada evolução gerada, cada avanço realizado na linha tempo dos processos de software desencadeou tecnologias colocando o homem em uma busca incessante por mais.

O software saiu de uma necessidade de sistema de mercado e invadiu as casas, a vida social das pessoas, os veículos, a própria forma hoje como vivemos e escolhemos se usamos a energia solar, ou elétrica difundida. Tudo está de alguma forma, sendo controlado por algum software!

E pensar que tudo começou com uma mente brilhante décadas atrás...

Um pensamento que não parou de crescer e evoluir até hoje!

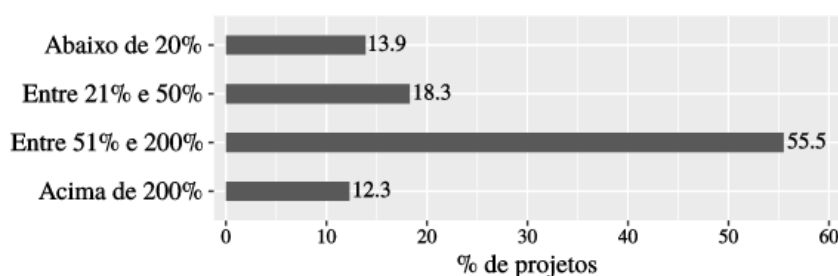
No início chamamos de máquina, hoje, podemos dizer que é praticamente uma parte viva e pulsante de nossa sociedade.

A importância e relevância dessas informações, a nós universitários, é perceber que a evolução do software e da engenharia de software teve um papel crucial nos avanços que essa tecnologia atingiu. Afinal, imagine, se por acaso, a engenharia de hardware estivesse pronta para uso, mas, a engenharia de software estivesse ainda hoje trabalhando com o Modelo Watterfall... Será que o mercado daria credibilidade a essa tecnologia cujas falhas eram tamanhas que não se entregavam os projetos contratados?

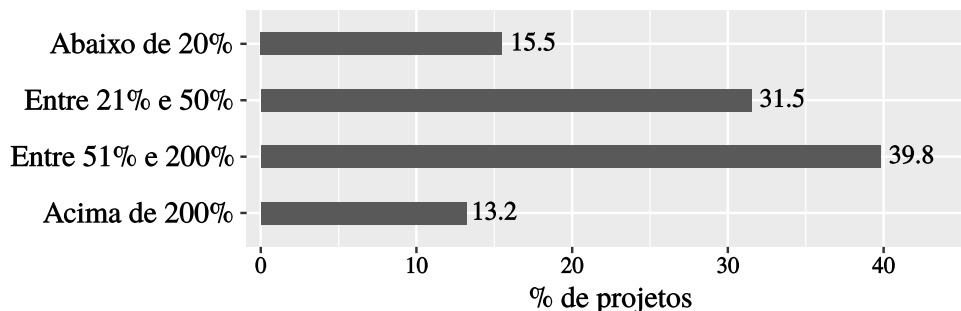
Será que, sem essa evolução na forma de desenvolver os processos de software usando a atual engenharia de software, seria possível atingir todos esses avanços em tecnologia?

Estas reflexões nos pedem uma breve pausa para análise... Em **1994**, um relatório produzido pela empresa de consultoria Standish Group revelou informações mais detalhadas sobre os projetos de software da época.

Por exemplo, o relatório, que ficou conhecido pelo sugestivo nome de **CHAOS Report (1994)**, mostrou que mais de **55% dos projetos estourava os prazos planejados** entre 51% e 200%; pelo menos 12% estouravam os prazos acima de 200%, conforme mostra o próximo gráfico.



Os resultados em termos de custos não eram mais animadores: **quase 40% dos projetos ultrapassava o orçamento entre 51% e 200%**, como mostra o seguinte gráfico:



Agora, vamos lembrar que o Manifesto Ágil é datado de fevereiro de 2001. Ou seja, o modelo Waterfall da década de 70 se arrastava penosamente pelas décadas 80 e 90 causando sérios impactos em um mercado que ansiava crescer muito, e, não tinha soluções inteligentes para conseguir sucesso.

Como dito, “um pensamento que não parou de crescer e evoluir”; é justamente toda a base observada neste trabalho. Desde quando se cunhou o termo Engenharia de Software em 1968, o engenheiro de software vem tratando do assunto com seriedade e atuando de forma específica a torná-lo - o software - sempre o melhor possível.

Como mera referência cronológica e didática, em 2018, o Stack Overflow survey incluiu uma pergunta sobre o método de desenvolvimento mais usado pelos respondentes. Essa pergunta recebeu **57 mil respostas de desenvolvedores profissionais**, e, a grande maioria mencionou métodos ou práticas ágeis; incluindo **Scrum (63% das respostas)**, **Kanban (36%)** e **Extreme Programming (16%)**. Apenas **15%** dos participantes marcaram **Waterfall** como resposta.

**Como seria o resultado – hoje – dos sistemas colossais, se ninguém tivesse pensado em interagir com o cliente, focar na sua satisfação, e, no compromisso em entregar um sistema minimamente funcional?**

Será que essa tecnologia teria atingido o mesmo resultado de hoje se ninguém tivesse pensado:

**“É hora criar uma Engenharia de Software funcional?”**

# REFERENCIAS BIBLIOGRÁFICAS

1. Engenharia de Software. *Wikipedia.org*, 2022. Disponível em: [https://pt.wikipedia.org/wiki/Engenharia\\_de\\_software](https://pt.wikipedia.org/wiki/Engenharia_de_software).
2. SOMMERVILLE, Ian. Engenharia de Software. 9ª Edição. São Paulo: Pearson, 2011.
3. VALENTE, Marco Tulio. Engenharia de Software Moderna. Editora: Independente, 395 páginas, 2020. ISBN: 978-65-00-0077-1 (e-book). Disponível em: <https://engsoftmoderna.info>