



CONCEITOS GERAIS E PRINCIPAIS ABORDAGENS DE DESENVOLVIMENTO DO SOFTWARE



Thiago Salhab Alves

**CONCEITOS GERAIS E PRINCIPAIS
ABORDAGENS DE DESENVOLVIMENTO DO
SOFTWARE**

1ª edição

Londrina
Editora e Distribuidora Educacional S.A.
2020

© 2020 por Editora e Distribuidora Educacional S.A.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Editora e Distribuidora Educacional S.A.

Presidente

Rodrigo Galindo

Vice-Presidente de Pós-Graduação e Educação Continuada

Paulo de Tarso Pires de Moraes

Conselho Acadêmico

Carlos Roberto Pagani Junior
Camila Braga de Oliveira Higa
Carolina Yaly
Giani Vendramel de Oliveira
Henrique Salustiano Silva
Juliana Caramigo Gennarini
Mariana Gerardi Mello
Nirse Ruscheinsky Breternitz
Priscila Pereira Silva
Tayra Carolina Nascimento Aleixo

Coordenador

Henrique Salustiano Silva

Revisor

Anderson Paulo Avila Santos

Editorial

Alessandra Cristina Fahl
Beatriz Meloni Montefusco
Gilvânia Honório dos Santos
Mariana de Campos Barroso
Paola Andressa Machado Leal

Dados Internacionais de Catalogação na Publicação (CIP)

A474c Alves, Thiago Salhab
Conceitos gerais e principais abordagens de desenvolvimento
do software / Thiago Salhab Alves, – Londrina:
Editora e Distribuidora Educacional S.A. 2020.
44 p.
ISBN 978-65-87806-07-5

1. Conceitos. 2. Desenvolvimento I. Alves, Thiago Salhab. II. Título.

CDD 005

Jorge Eduardo de Almeida CRB: 8/8753

2020
Editora e Distribuidora Educacional S.A.
Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 — Londrina — PR
e-mail: editora.educacional@kroton.com.br
Homepage: <http://www.kroton.com.br/>

SUMÁRIO

Conceitos gerais da Engenharia de Software	05
Metodologias Clássicas	20
Metodologias Ágeis	35
Metodologia Scrum	49

Conceitos gerais da Engenharia de Software

Autoria: Thiago Salhab Alves

Leitura crítica: Anderson Paulo Avila Santos



Objetivos

- Aprender sobre o surgimento da Engenharia de Software, suas propostas e desafios.
- Aprender sobre o ciclo de desenvolvimento de software e o papel do Engenheiro de Software no processo de desenvolvimento.
- Aprender sobre o modelo de certificação de software CMMI e sua proposta para garantia da qualidade de software.



1. Introdução a Engenharia de Software


Prezado (a) aluno (a), você já parou para pensar como seria nossa vida sem o uso de sistemas de software? Há algumas décadas, toda gestão de um comércio ou indústria, por exemplo, era realizada de forma manual, utilizando livros-caixa, planilhas, cadernos e máquina de escrever. Imagine como era o trabalho dos gestores nessa época? O avanço da tecnologia da informação e o do processo de desenvolvimento de software, facilitou e popularizou o uso de computadores e sistemas de software.

Nesta leitura, será apresentada a Engenharia de Software, o motivo de seu surgimento, suas propostas e desafios. Você aprenderá sobre o ciclo de desenvolvimento do software, o papel do engenheiro de software e o modelo de certificações mais utilizado no mundo: *Capability Maturity Model Integration* (CMMI).

De acordo com Sommerville (2011) todos os países dependem de sistemas complexos baseados em computadores. A manufatura e a distribuição industrial estão completamente automatizadas. Produzir software e manter o software dentro de custos adequados é essencial para o funcionamento da economia nacional e internacional.

Engenharia de Software é área da Engenharia da Computação com foco no desenvolvimento de sistemas de alta qualidade, dentro de custos adequados. Como o software é abstrato e intangível, não sendo limitado por materiais ou controlado pelas leis da física, faz com que o processo possa se tornar extremamente complexo e difícil de ser compreendido.

Segundo Pressman (2006), o termo Engenharia de Software foi proposto em 1968, em uma conferência, na Alemanha, organizada para discutir o que foi chamado de crise de software, resultante do avanço do hardware de computador, baseado em circuitos integrados. Isso fez o




poder computacional crescer, tornando viável o desenvolvimento de determinados softwares até então inviáveis, resultando em softwares com ordem de grandezas maiores e mais complexos que os anteriores.

A construção desses sistemas de computação mostrou que o desenvolvimento informal de software não era suficiente, que o processo estava em crise e que novas técnicas e métodos eram necessários para controlar a complexidade dos grandes sistemas de software. Pressman (2006), aponta que o termo crise de software 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 pudessem ser validados.

As causas da crise do software estão ligadas a complexidade do processo de software e a relativa imaturidade da engenharia de software como profissão. A crise se manifesta de várias formas:

- Projetos estourando o orçamento.
- Projetos estourando o prazo.
- Software de baixa qualidade.
- Softwares que, muitas vezes, não atingiam os requisitos.
- Projetos ingerenciáveis e o código difícil de manter.

Dijkstra (1972) afirma em seu trabalho, *The Humble Programmer*, que a maior causa da crise de software se deu porque as máquinas ficaram mais potentes. Quando as máquinas eram simples, programar não era um problema, mas agora que as máquinas estão mais potentes, programar se tornou um grande problema.



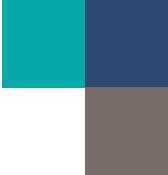
A situação ainda era agravada por outro motivo: programadores da época não possuíam as ferramentas que dispomos hoje. Não existiam técnicas consagradas de trabalho, escolas ou a profissão de programador, e as pessoas aprendiam e exerciam essa atividade por tentativas e erros.

No mesmo trabalho, apresentado num congresso em 1968, os problemas apontados para a construção de software foram:

- Cronogramas não observados.
- Projetos com tantas dificuldades que são abandonados.
- Módulos que não operam corretamente quando combinados.
- Programas que não fazem exatamente o que era esperado.
- Programas tão difíceis de usar, que são descartados.
- Programas que simplesmente param de funcionar.

Os problemas, encontrados em 1968, foram totalmente resolvidos ou ainda se encontram empresas vivendo em crise? Infelizmente sim, e isso mostra que muitas empresas e organizações são imaturas para o processo de desenvolvimento de software. Uma organização imatura apresenta os seguintes aspectos:

- Os processos de software são, geralmente, improvisados pelos praticantes ou gerentes durante o andamento do projeto.
- O processo de software especificado não é seguido rigorosamente.
- Os gerentes estão focados em resolver crises.
- Essas organizações excedem prazos e orçamentos, pois não estão baseadas em estimativas reais.



Dado esse cenário, novas técnicas e métodos eram necessários para controlar a complexidade dos grandes sistemas de software. Essas técnicas tornaram-se parte da engenharia de software, porém, muitas empresas não aplicam de forma efetiva, produzindo softwares de baixa confiabilidade, com atraso e com custo além do orçamento.

Sommerville (2011) descreve que essas novas técnicas, propostas para solucionar os problemas relacionados a crise, tornaram-se parte da engenharia de software, criando um processo de desenvolvimento de software.

Muitas pessoas associam o termo software aos programas de computadores. Essa é uma visão muito restritiva. Software não é apenas o programa, mas todos os dados de documentação e configuração associados, necessários para que o programa opere. Um sistema de software consiste, geralmente, de um conjunto de programas separados; arquivos de configuração; documentação do usuário.

Para um efetivo processo de criação de software é necessário um processo de desenvolvimento de software, que é um conjunto de atividades que produz um produto de software. Existem quatro atividades fundamentais de processo que são comuns a todos os processos de softwares:

- Especificação de software: clientes e engenheiros definem o software a ser produzido e as restrições para sua operação.
- Desenvolvimento de software: o software é projetado e programado.
- Validação de software: o software é verificado para garantir que é o que o cliente deseja.
- Evolução de software: o software é modificado para se adaptar às mudanças dos requisitos do cliente e do mercado.

A Engenharia de Software, segundo Pressman (2016) é uma abordagem sistemática, disciplinada e quantificável no desenvolvimento, operação e manutenção de software, aplicada por uma equipe de desenvolvimento de software.

A Figura 1 ilustra a Engenharia de Software como uma tecnologia em camadas, sendo sua base a camada de processos. O processo é responsável por definir uma metodologia que deve ser usada para a entrega efetiva da tecnologia de engenharia de software, constituindo a base para o controle do gerenciamento de projetos, produzindo artefatos, que são os modelos, documentos, dados, relatórios, formulários etc. Os métodos da engenharia de software estabelecem as informações técnicas para desenvolver software, envolvendo tarefas de comunicação, análise de requisitos, modelagem de projeto, construção de programas, testes e suporte. As ferramentas fornecem suporte automatizado para o processo e para os métodos.

Figura 1 – Camadas de Engenharia de Software



Fonte: Pressman (2016, p. 16).

De acordo com Sommerville (2011), os atributos de um bom software são:

- Facilidade de manutenção: software deve ser escrito de modo que possa evoluir para atender às necessidades de mudança dos clientes.


- **Confiança:** o nível de confiança tem uma série de características, incluindo confiabilidade, proteção e segurança. Um software confiável não deve causar danos físicos ou econômicos, no caso de falha no sistema.
- **Eficiência:** o software não deve desperdiçar os recursos do sistema, como memória e ciclos do processador. A eficiência inclui tempo de resposta, tempo de processamento e utilização de memória.
- **Usabilidade:** o software deve ser usável, sem esforço excessivo, pelo tipo de usuário para o qual foi projetado. Apresentar uma interface com o usuário e documentação adequados.

Os principais desafios da Engenharia de Software, de acordo com Pressman (2016) são:

- **Desafio da heterogeneidade:** cada vez mais é necessário que os sistemas operem como sistemas distribuídos, por meio de redes, que incluem diferentes tipos de computadores, com diferentes tipos de sistemas de apoio.
- **Desafio da entrega:** muitas técnicas tradicionais da engenharia demandam tempo. O tempo que necessitam é necessário para obter a qualidade do software. O desafio é diminuir os tempos da entrega sem comprometer a qualidade.
- **Desafio da confiança:** é essencial que possamos confiar no software. O desafio é desenvolver técnicas que demonstrem que o software pode ter a confiança de seus usuários.

1.1 Ciclo de desenvolvimento e o papel do Engenheiro de Software

De acordo com Paula Filho (2019), o software apresenta um ciclo de vida. A Engenharia de Software trata o software como um produto. Nesse



contexto, há o cliente, que é uma pessoa física ou jurídica que contrata a execução de um projeto e que fará uso do produto (usuário). O usuário pode ser o cliente, um funcionário da organização. O ciclo de vida diz que:

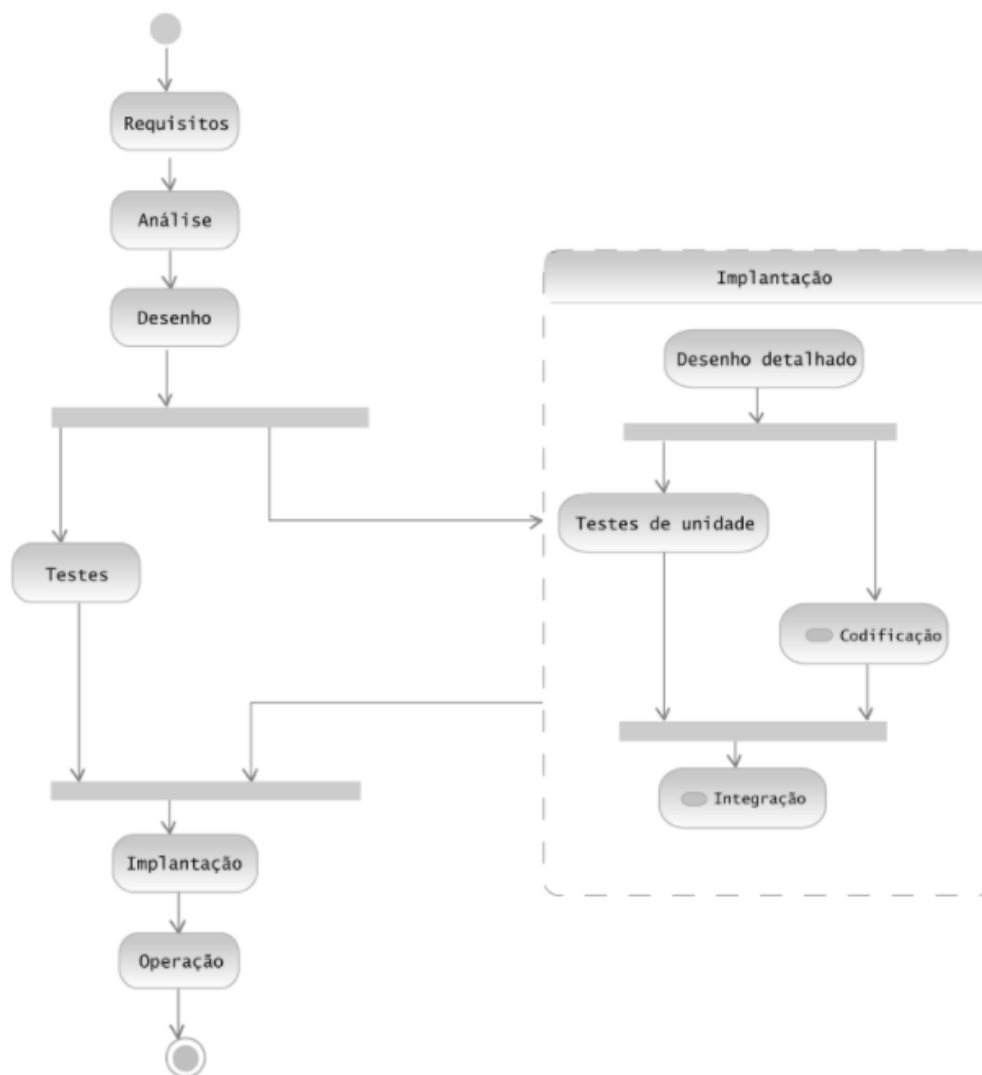
- O software é concebido a partir de uma necessidade.
- É desenvolvido, se transformando em itens que serão entregues ao cliente.
- É colocado em operação, sendo utilizado dentro de algum processo de negócio, sujeito a manutenções, se necessário.
- É retirado de operação ao final de sua vida útil.

A Figura 2 apresenta cada fase do ciclo de vida. O processo se inicia com a definição dos requisitos, que são os serviços, restrições e objetivos do sistema, definidos por meio de consulta aos usuários do sistema que são, portanto, definidos detalhadamente e servem como uma especificação do sistema. Na sequência, há as fases de análise e desenho, que dividem os requisitos em sistemas de hardware ou de software, envolvem a identificação e a descrição das abstrações fundamentais do sistema de software e suas relações. Nas etapas de implantação e testes de unidade, o projeto de software é realizado como um conjunto de programas ou unidades de programa.

O teste unitário envolve a verificação de que cada unidade atende às suas especificações. Nas etapas de integração e testes, as unidades individuais de programa ou os programas são integrados e testados como um sistema completo para garantir que os requisitos de software foram atendidos. Após os testes, o sistema de software é liberado para o cliente. Por fim, o software entra em implantação e operação, geralmente, sendo a fase mais longa do ciclo de vida. O sistema é instalado e colocado em operação. A manutenção envolve a correção de erros não detectados nos estágios anteriores do ciclo de vida, no


aprimoramento da implementação das unidades de sistema e na ampliação dos serviços de sistema à medida em que novos requisitos são identificados.

Figura 2 – Ciclo de vida do software



Fonte: Paula Filho (2019, p. 8).

Segundo Paula Filho (2019), o Engenheiro de Software tem como papel obter domínio de material técnico, aprendendo e aplicando habilidades para entender o problema, projetando soluções, construindo softwares e testando com a finalidade de produzir um produto com a mais alta qualidade.



Além dessas características técnicas, o Engenheiro de Software, de acordo com Paula Filho (2019), necessita desenvolver alguns aspectos humanos que o tornarão competente, tais como:

- Senso de responsabilidade individual: um engenheiro de software competente deve cumprir suas promessas para colegas, para os envolvidos e gerência, significando que fará o que precisar ser feito, quando necessário, executando esforço para obter resultado bem-sucedido.
- Consciência aguçada: consciência das necessidades dos integrantes da equipe, dos envolvidos que solicitaram soluções de software e dos gerentes que têm controle global sobre o projeto, sendo capaz de observar o ambiente em que as pessoas trabalham e adaptar seu comportamento a ele e às próprias pessoas.
- Extremamente honesto: ao ver um projeto falho, deve apontar os defeitos de maneira construtiva, mas honesta. Se for solicitado a distorcer fatos sobre cronogramas, recursos, desempenho ou outra característica do produto ou projeto, deve optar por ser realista e sincero.
- Resiliência sob pressão: o engenheiro é capaz de suportar a pressão de modo que seu desempenho não seja prejudicado.
- Elevado senso de lealdade: compartilha créditos com seus colegas, busca evitar conflitos de interesse e nunca age no sentido de sabotar o trabalho dos outros.
- Atenção aos detalhes: o engenheiro considera atentamente as decisões técnicas que toma em comparação com critérios mais amplos, como, por exemplo, desempenho, custo, qualidade, que foram estabelecidos para o produto e para o projeto.
- Pragmático: reconhece que a engenharia de software é uma disciplina que deve ser adaptada de acordo com as circunstâncias.

1.2 Principais modelos de certificações

Segundo Pressman (2016), os padrões CMMI do *Software Engineering Institute* e as normas ISO são as metodologias de processo mais comumente usadas, possuindo sintaxe e semântica que levarão à implementação de práticas de engenharia de software que melhoram a qualidade do produto. *Capability Maturity Model* (CMM) e *Capability Maturity Model Integration for Development* (CMMI-DEV), iniciam sua trajetória com o CMM, que foi criado no final de década de 1980 e publicado oficialmente em fevereiro de 1993, sendo atualizado para a versão integrada CMMI em agosto de 2002 (versão 1.1) e agosto de 2006 (versão 1.2) CMMI-DEV. Possui foco nas organizações. Foi criado na Universidade Carnegie Mellon, apoiado pelo SEI e Departamento de Defesa Americano.

Segundo Sommerville (2011), dois modelos foram fundamentais para o surgimento do CMMI: o ISO/IEC 15504, que foi criado a partir de 1991 com o projeto *Software Process Improvement and Capability dEtermination* (SPICE); e o ISO/IEC 12207, que provê uma estrutura para que uma organização defina seus processos e não define níveis de maturidade organizacional ou capacidade de processo.

O CMMI, de acordo com Pressman (2016), representa um metamodelo de processo de duas maneiras diferentes: um modelo contínuo e um modelo por estágio. O metamodelo contínuo descreve um processo em duas dimensões, conforme a Figura 3, em que cada área de processo é formalmente avaliada em relação a metas e práticas específicas e classificada de acordo com os seguintes níveis de capacidade:

- Nível 0 – Incompleto: a área de processo não atende as metas e objetivos definidos pelo CMMI para a capacidade nível 1 para a área de processo.


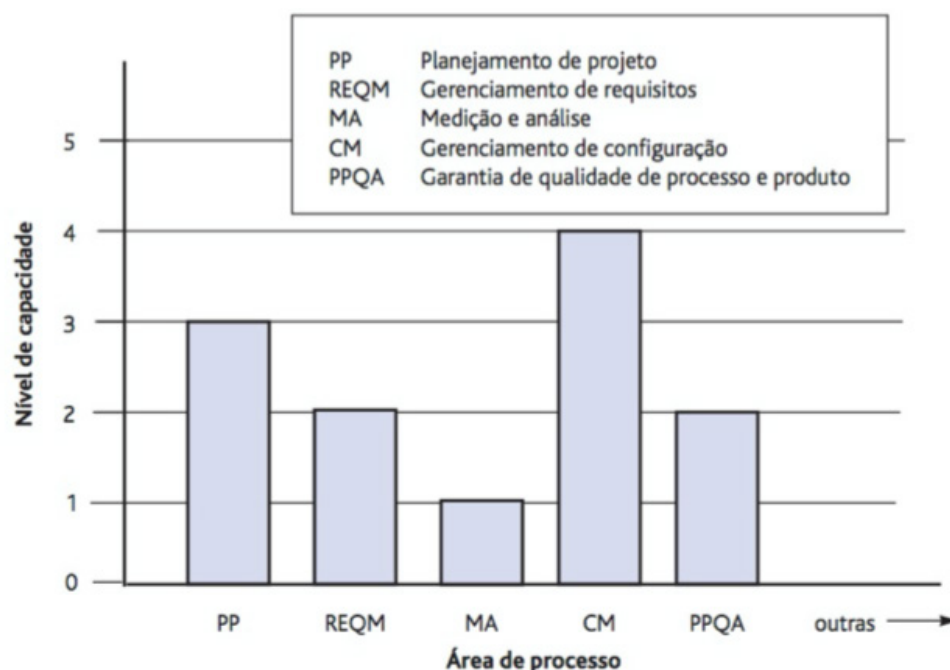
- 
- Nível 1–Executado: atendimento de todas as metas específicas da área de processo. Tarefas necessárias para produzir os artefatos definidos.
 - Nível 2 – Controlada: atendimento total do nível 1 de capacidade. Estabelece-se uma política definida em termos de organização e todas as tarefas e produtos são monitorados, controlados, revisados e avaliados quanto à conformidade com a descrição de processo.
 - Nível 3 – Definido: atendimento total do nível 2 de capacidade e o processo se adapta com base em um conjunto de processos padronizados da organização.
 - Nível 4 – Controlado quantitativamente: atendimento total do nível 3 de capacidade e área de processo é controlada e melhorada usando medição e avaliação quantitativa.
 - Nível 5 – Otimizado: atendimento total do nível 4 de capacidade e a área de processo é adaptada e otimizada usando meios quantitativos (estatísticos) para atender à mudança de necessidades do cliente e melhorar continuamente a eficiência da área de processo.

Figura 3 – Perfil da capacidade da área de processo CMMI



Fonte: Pressman (2016, p. 829).

O modelo CMMI por estágio, de acordo com Pressman (2016), define as mesmas áreas de processo, metas e práticas do modelo contínuo. A principal diferença é que o modelo por estágio define cinco níveis de maturidade, em vez de cinco níveis de capacidade. Para atingir um nível de maturidade, as metas específicas e as práticas associadas a um conjunto de áreas de processos devem ser atingidas. A relação entre níveis de maturidade e áreas de processo é mostrada no Quadro 1.

Quadro 1 – Áreas de processo necessárias para atingir um nível de maturidade

Nível	Foco	Áreas de Processo
Otimizante.	Melhoria contínua do processo.	Inovação organizacional e entrega. Análise causal e resolução.

Controlado quantitativamente.	Gerenciamento quantitativo.	<ul style="list-style-type: none"> - Desempenho de processo organizacional. - Gerenciamento quantitativo de projeto.
Definido.	Padronização de processo.	<ul style="list-style-type: none"> - Desenvolvimento de requisitos. - Solução técnica. - Integração de produto. - Verificação. - Validação. - Foco no processo organizacional. - Definição de processo organizacional. - Treinamento organizacional. - Gerenciamento de projeto integrado. - Gerenciamento de fornecimento integrado. - Gestão de risco. - Análise de decisão e resolução. - Ambiente organizacional para integração. - Equipe integrada.

Repetível.	Gerenciamento básico de projeto.	<ul style="list-style-type: none"> - Gerenciamento de requisitos. - Planejamento de projeto. - Monitoração e controle de projeto. - Gerenciamento de acordo com fornecedor. - Medição e análise. - Garantia de qualidade de processo e produto. - Gerenciamento de configuração.
------------	----------------------------------	---

Fonte: Pressman (2016, p. 831).

Dessa forma, você aprendeu sobre: a Engenharia de Software e o motivo de seu surgimento, suas propostas e desafios; o ciclo de desenvolvimento do software; o papel do engenheiro de software; e o modelo de certificações mais utilizado no mundo, que é o CMMI.

Referências Bibliográficas

DIJKSTRA, E. W. **The Humble Programmer. Communications of the ACM**. N. 10, v. 15, 1972. Disponível em: <https://dl.acm.org/doi/pdf/10.1145/355604.361591?download=true>. Acesso em: 4 jun 2020.

PAULA FILHO, W.P. **Engenharia de software: produtos**. 4. ed. Rio de Janeiro, RJ: Editora LTC, 2019.

PRESSMAN, R. **Engenharia de software**. 6. ed. São Paulo, SP: Makron Books, 2006.

PRESSMAN, R. **Engenharia de software: uma abordagem profissional**. 8. edição. Porto Alegre, RS: AMGH, 2016.

SOMMERVILLE, I. **Engenharia de Software**, 9. ed. Pearson Education do Brasil, 2011.

Metodologias Clássicas

Autoria: Thiago Salhab Alves

Leitura crítica: Anderson Paulo Avila Santos



Objetivos

- Aprender sobre as metodologias clássicas para desenvolvimento de sistemas.
- Aprender sobre as metodologias cascata, prototipação e baseado em componentes.
- Aprender sobre as metodologias incremental e espiral.



1. Metodologias clássicas


Prezado (a) aluno (a), você já parou para pensar como seriam as empresas de desenvolvimento de sistemas, a qualidade dos produtos e serviços de software e o processo de desenvolvimento, se não houvesse as metodologias de desenvolvimento propostas na década de 1970 e 1980? Certamente o processo seria muito mais caótico do que apontado no final da década de 1960, pela demanda por software e serviços e com produtos de baixa qualidade e nada confiáveis.

Nesta leitura, serão apresentadas as metodologias clássicas de desenvolvimento de sistemas, com foco nas metodologias cascata, prototipação, baseado em componentes, incremental e espiral.

As metodologias clássicas, ou orientadas a documentação, foram propostas em um momento em que o uso de *mainframes* era dominante e não existiam ferramentas de apoio ao desenvolvimento de software, depuradores e analisadores de código. Devido a esse cenário, havia uma grande preocupação com planejamento e documentação do projeto antes da sua implementação, segundo Sbrocco e Macedo (2012).

O modelo clássico ou sequencial, por Sbrocco e Macedo (2012), foi o primeiro processo de desenvolvimento de software, composto de um modelo de fácil entendimento, com uma sequência de etapas em que cada uma tem associada ao seu término uma documentação que deve ser aprovada para se iniciar a próxima etapa.

Os modelos de processo foram propostos para trazer ordem ao caos que se instaurou na área de desenvolvimento de software. Esses modelos proporcionam uma considerável contribuição para a estrutura utilizável no trabalho de engenharia de software e fornecem um caminho eficaz para ser seguido pelas equipes de engenharia de software, de acordo com Pressman (2016).



O processo de desenvolvimento de software é um conjunto de atividades que produz um produto de software. Os processos de software são complexos e, como todos os processos intelectuais e criativos, dependem do julgamento humano. Existem quatro atividades fundamentais de processo que são comuns a todos os processos de softwares, segundo Sommerville (2011):

- Especificação de software: clientes e engenheiros definem o software a ser produzido e as restrições para sua operação.
- Desenvolvimento de software: o software é projetado e programado.
- Validação de software: o software é verificado para garantir se atende ao que o cliente deseja.
- Evolução de software: o software é modificado para se adaptar às mudanças dos requisitos do cliente e do mercado.

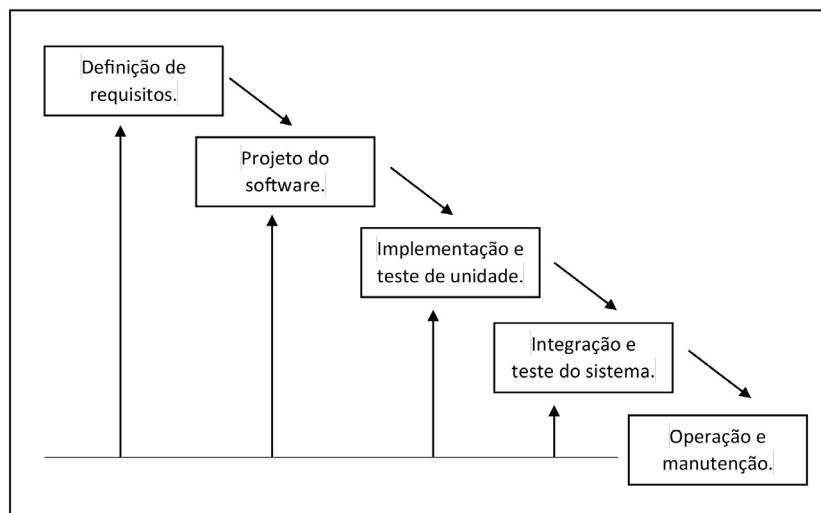
Um modelo de processo prescritivo tem por objetivo estruturar e ordenar o desenvolvimento de software. Todas as atividades e tarefas ocorrem de forma sequencial com orientações claras de seu progresso. Esses processos são chamados de prescritivos, pois prescrevem um conjunto de elementos de processos, tais como atividades metodológicas, ações de engenharia de software, tarefas, artefatos, garantia da qualidade e procedimentos de controle de mudanças para cada projeto, segundo Pressman (2016).

1.1 Metodologias cascata, prototipação e baseado em componentes

1.1.1 Metodologia cascata

O modelo cascata, também conhecido por modelo sequencial linear, foi o primeiro modelo de processo de desenvolvimento de software publicado em 1970, por Winston Walker Royce. Possui esse nome porque o processo é visto como atividades realizadas em sequência, devendo respeitar o término de uma etapa para o início da próxima, e a necessidade de um documento aprovado para a sequência e com foco de aplicação para sistemas de grande porte (*mainframes*), segundo Sommerville (2011). A Figura 1 detalha as etapas do modelo Cascata, segundo Pressman (2016):


Figura 1 – Modelo cascata



Fonte: elaborada pelo autor.

As etapas podem ser descritas como:

- Definição de requisitos: define os serviços, restrições e objetivos do sistema por meio de consulta aos usuários do sistema, sendo




definidos detalhadamente e servem como uma especificação do sistema.

- Projeto do software: realiza o projeto do sistema, dividindo os requisitos de hardware ou de software, realizando a identificação e a descrição das abstrações fundamentais do sistema de software e suas relações.
- Implementação e teste de unidade: codificação dos programas ou unidades de programas, que são testados individualmente para que atendam às especificações.
- Integração e teste do sistema: integração e teste de todas as unidades de sistemas, realizando o teste completo e liberação apenas após a finalização dos testes.
- Operação e manutenção: o sistema é colocado em operação e eventuais manutenções para correção de erros não detectados ou aprimoramento do sistema.

Algumas vantagens são atribuídas ao modelo cascata:

- O processo de desenvolvimento tem uma ordem sequencial de etapas, em que cada uma deve estar terminada para o início da próxima.
- Maior garantia que os requisitos estão completos e os softwares produzidos correspondem ao que foi especificado.
- Processo de desenvolvimento conduzido de forma disciplinada e com atividades definidas e determinadas a partir de um planejamento.
- Documentação completa do sistema.




O modelo cascata é o paradigma mais antigo da engenharia de software e, ao longo das últimas quatro décadas, recebeu muitas críticas, fazendo com que seus mais árdios defensores questionassem sua eficácia. Entre as desvantagens aplicadas ao modelo cascata, segundo Pressman (2016):

- Projetos dificilmente seguem o fluxo sequencial proposto pelo modelo, aplicando o modelo linear de forma indireta, fazendo com que possíveis mudanças atrapalhem o avanço da equipe de projeto.
- O cliente tem dificuldade em determinar todas as suas necessidades e, como o modelo cascata exige isso, tem incertezas que podem prejudicar o andamento do projeto, principalmente, por não apresentar certos requisitos.
- O cliente só receberá a versão final quando o projeto finalizar, não recebendo nenhuma versão antes disso.
- O modelo não prevê a revisão de fases.

1.1.2 Prototipação

Para Sommerville (2011), a prototipação toma por base o desenvolvimento evolucionário, que se baseia na ideia do desenvolvimento de uma descrição inicial, que é submetida à avaliação do usuário, que aponta suas impressões e permite um refinamento por meio de várias versões até que seja desenvolvido um sistema adequado.

A prototipação tem por objetivo realizar uma validação de ideias e ser utilizada para entender os requisitos na forma física de um software (protótipo). É uma etapa que conta com a participação do cliente que, utilizando o protótipo, consegue apontar detalhes não percebidos.

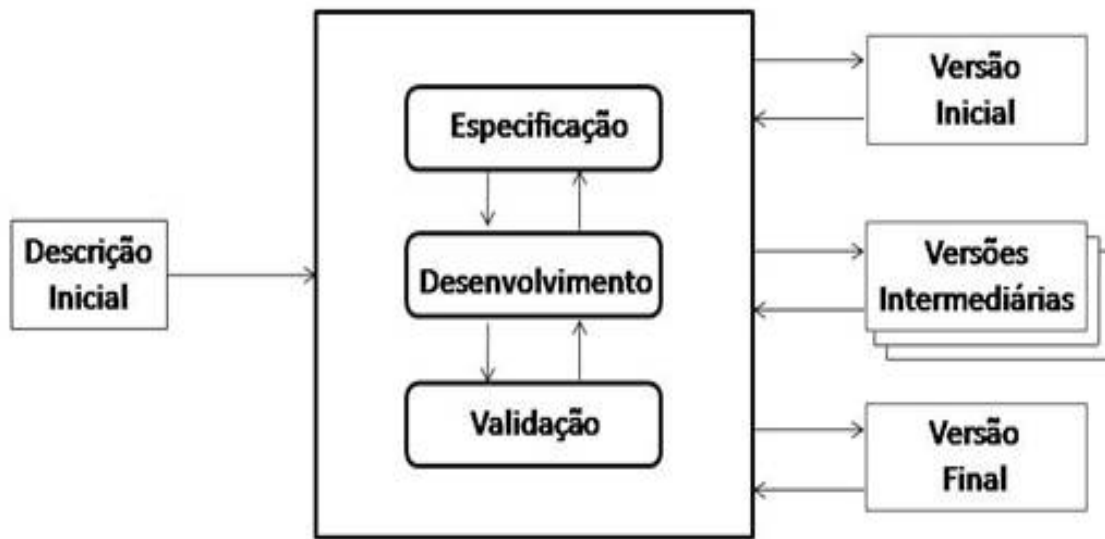


Dois tipos de desenvolvimento evolucionário podem ser considerados, de acordo com Sommerville (2011):

- Exploratório: o objetivo do processo é trabalhar com o cliente para explorar os requisitos e entregar um sistema final. O desenvolvimento começa com as partes do sistema compreendidas e evolui por meio da adição de novas características propostas pelo cliente.
- Prototipação *throwaway*: na qual o objetivo do processo de desenvolvimento evolucionário é compreender os requisitos do cliente e, a partir disso, desenvolver melhor definição de requisitos para o sistema.

A Figura 2 apresenta o funcionamento da prototipação. Nesse modelo, inicia-se com uma ideia inicial do sistema, que gera uma descrição inicial que passa pelo processo de especificação, desenvolvimento e validação. Uma versão inicial é gerada e submetida aos comentários do usuário. Com isso, o usuário avalia os requisitos e realiza propostas de novas funcionalidades. Com as propostas, versões intermediárias são produzidas, também passando pelo processo de especificação, desenvolvimento e validação, até que se tenha a versão final. O protótipo deve ser utilizado para um entendimento das reais necessidades do cliente.

Figura 2 – Modelo Prototipação




Fonte: elaborada pelo autor.

Como vantagens da prototipação evolucionária, pode-se considerar:

- Melhora a comunicação entre os programadores e usuários.
- Utilizado para avaliar a experiência do software pelo usuário.
- Identifica de forma rápida os requisitos que precisam ser atendidos.

O modelo de prototipação evolucionária apresenta algumas desvantagens:

- O processo não é viável, pois os gerentes precisam de produtos regulares para medir o progresso. Se os sistemas são desenvolvidos rapidamente, não é viável economicamente produzir documentos que reflitam cada versão do sistema.
- Os sistemas são frequentemente mal estruturados. A mudança contínua tende a corromper a estrutura do software. A incorporação de mudanças de software torna-se cada vez mais difícil e onerosa.



Para sistemas de pequeno e médio porte (até 500 mil linhas de código), a abordagem prototipação evolucionária é o melhor método de desenvolvimento. Os problemas de desenvolvimento evolucionário tornam-se particularmente graves para sistemas complexos de grande porte e de longo ciclo de vida, nos quais diversas equipes desenvolvem diferentes partes do sistema.

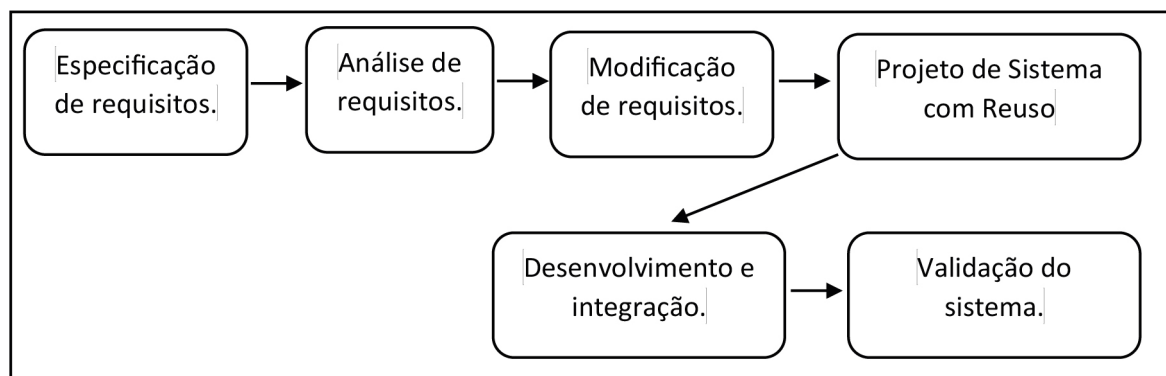
1.1.3 Desenvolvimento baseado em componentes

O desenvolvimento baseado em componentes funciona por meio de uma base de componentes de software que podem ser reutilizados e um *framework* que integra esses componentes, que podem ser, por exemplo, funções de validação de CPF/ CNPJ ou cálculo numérico. O desenvolvimento baseado em componentes apresenta os seguintes estágios, segundo Sommerville (2011):

- Especificação de requisitos: levantamento dos requisitos do sistema realizado junto ao cliente.
- Análise de componentes: dada uma especificação de requisitos, é feita uma busca pelos componentes para implementar essa especificação.
- Modificação de requisitos: são modificados para refletir os componentes disponíveis.
- Projeto de sistema com reuso: o *framework* do sistema é projetado ou um *framework* existente é reusado. Os projetistas levam em consideração os componentes reusados.
- Desenvolvimento e integração: o software que não pode ser adquirido externamente é desenvolvido e os componentes e os sistemas são integrados para criar o novo sistema.

- Validação do sistema: o sistema é validado para verificação de seu correto funcionamento.

Figura 3 – Modelo baseado em componentes



Fonte: elaborada pelo autor.

Como benefícios da metodologia baseada em componentes, pode-se considerar:

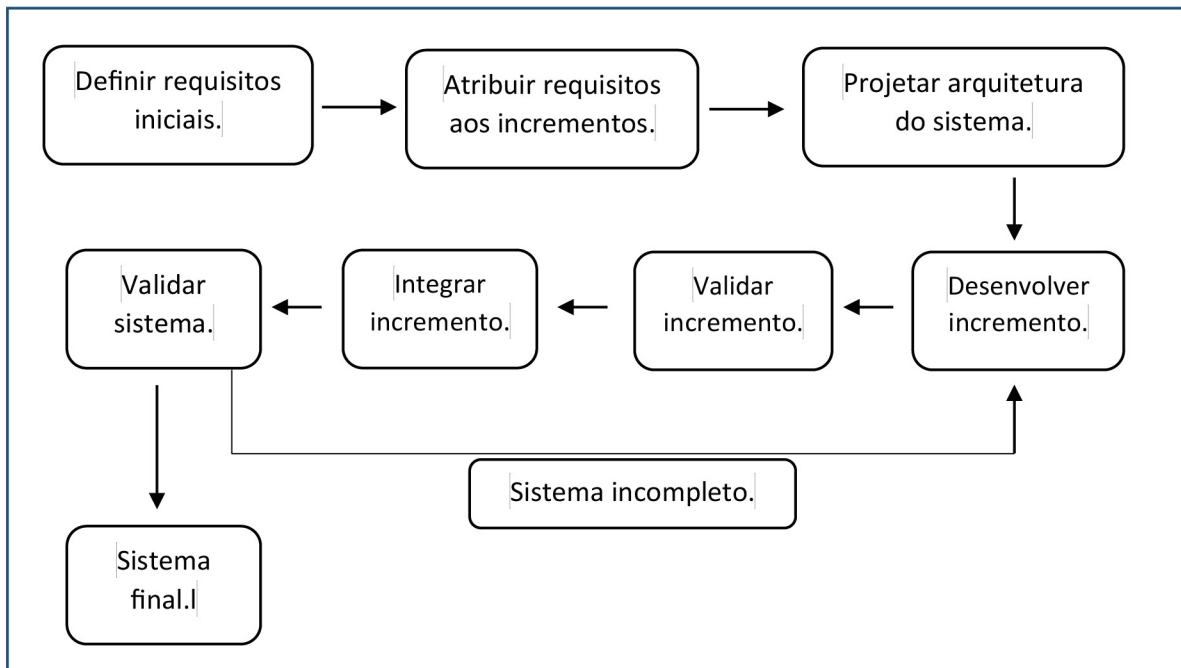
- Aumento da produtividade e economia de tempo de desenvolvimento.
- Maior garantia de funcionamento do produto final que usa componentes, pois já foram testados quando foram inicialmente construídos.

1.2 Metodologias incremental e espiral

1.2.1 Metodologia incremental

A metodologia incremental, por Sommerville (2011), é uma abordagem que combina as vantagens do modelo em cascata. A Figura 4 ilustra o funcionamento do modelo incremental. Em um processo de desenvolvimento incremental, o cliente identifica, em linhas gerais, os serviços a serem fornecidos pelo sistema, que identificam quais serviços são mais importantes e quais são menos importantes.

Figura 4 – Modelo incremental



Fonte: elaborada pelo autor.

Assim, é estabelecido um número de incrementos de entrega e cada um deve possuir um subconjunto das funcionalidades do sistema. Os serviços devem ser priorizados e a alocação de recursos depende da prioridade do serviço. Quando um serviço for concluído e entregue, os clientes podem colocar o incremento em operação, tendo com antecedência a entrega de parte da funcionalidade do sistema.

Quando os incrementos são concluídos, realiza-se a integração, fazendo com que a funcionalidade do sistema receba o incremento. O processo de desenvolvimento incremental tem uma série de vantagens, segundo Sommerville (2011):

- Os clientes não precisam esperar o sistema ser concluído para poder utilizar. Quando os incrementos forem concluídos, já é possível a utilização do software.
- Os incrementos iniciais podem ser usados como protótipos, assim, o cliente pode ir ganhando experiência por meio de seu uso.

- Menor risco de falha geral do projeto. Embora possam ser encontrados problemas em alguns incrementos, é provável que alguns sejam entregues com sucesso aos clientes.

Desvantagens com a entrega incremental:

- Os incrementos devem ser relativamente pequenos.
- A maior parte dos sistemas requer um conjunto de recursos básicos, usados por diferentes partes do sistema.
- Uma variante dessa abordagem incremental é denominada *extreme programming* (XP), que é um modelo de desenvolvimento ágil de sistema.
- Se baseia no desenvolvimento e na entrega de incrementos muito pequenos de funcionalidade, envolvimento do cliente no processo, aprimoramento constante de código e programação em pares.

1.2.2 Modelo espiral

O modelo em espiral do processo de software foi originalmente proposto por Barry Boehm em 1988, sendo um modelo de software evolucionário unindo a parte iterativa da prototipação à parte sistemática e controlada do modelo cascata, segundo Pressman (2016).

Com o modelo espiral, o software é desenvolvido em uma série de versões evolucionárias, sendo nas primeiras iterações, a versão se constitui em um modelo ou protótipo e, nas iterações posteriores, produzidas versões cada vez mais completas do sistema que passa pelo processo de engenharia, de acordo com Paula Filho (2019).

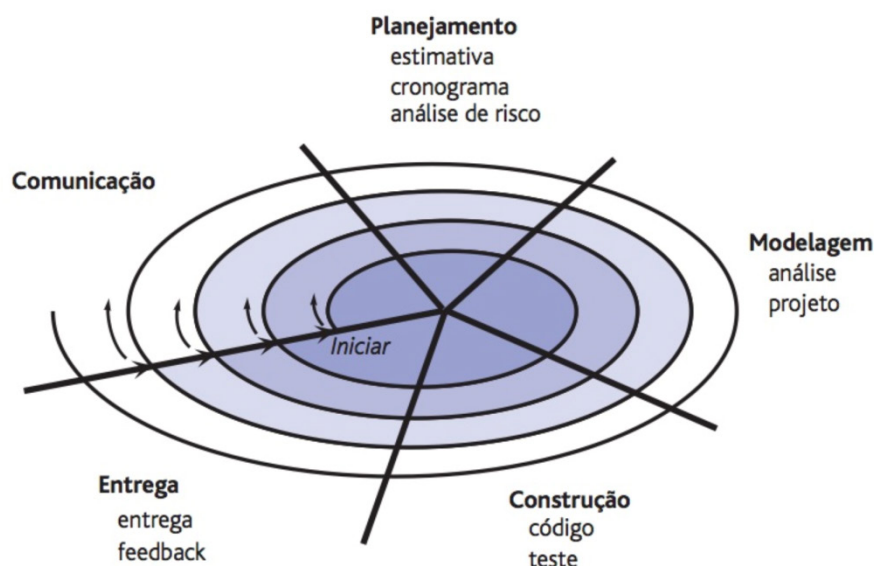
Em vez de apresentar o processo de software como uma sequência de atividades com algum retorno entre uma atividade e outra, o processo

é representado como um espiral. Cada loop na espiral representa uma fase do processo de software.

Quando o processo inicia, a equipe de software realiza as atividades indicadas por um circuito em torno da espiral, no sentido horário, começando pelo seu centro. O primeiro circuito em volta da espiral resulta no desenvolvimento da especificação do produto e as passagens subsequentes em torno da espiral são usadas para desenvolver um protótipo e, progressivamente, versões cada vez mais sofisticadas do software. Cada passagem pela região do planejamento resulta em ajustes no planejamento do projeto. Os custos e cronograma são ajustados de acordo com o feedback do cliente após a entrega, segundo Pressman (2016).

Cada loop na espiral está dividido em cinco setores, conforme a Figura 5:

Figura 5 – Modelo espiral



Fonte: Pressman (2016, p. 48).

- **Comunicação:** os objetivos específicos dessa fase do projeto são definidos. As restrições sobre o processo e o produto são identificadas e um plano detalhado de gerenciamento é elaborado.

- Planejamento: são realizadas as estimativas de prazo e custo e análise de risco. Para cada risco de projeto identificado, uma análise detalhada é realizada. Providências são tomadas para reduzir o risco.
- Modelagem: após a avaliação de risco, um modelo de desenvolvimento para o sistema é selecionado.
- Construção: elaborado a codificação e teste do que foi codificado.
- Entrega: o projeto é revisado e uma decisão é tomada para prosseguimento ao próximo loop da espiral. Se a decisão for pelo prosseguimento, serão elaborados planos para a próxima fase do projeto.

Como vantagem do modelo em espiral sobre outros modelos do processo de software está o reconhecimento explícito do risco. Risco significa simplesmente algo que pode dar errado. Por exemplo, se a intenção for usar uma nova linguagem de programação, o risco é que os compiladores disponíveis não sejam confiáveis ou não produzam código-objeto suficientemente eficaz.

Diferente de outros modelos que terminam quando o software é entregue, o modelo pode ser adaptado para ser aplicado ao longo da vida do software. O primeiro circuito em torno da espiral pode ser um projeto de desenvolvimento de conceitos, que começa no núcleo da espiral e continua por várias iterações até que o desenvolvimento de conceitos seja concluído. Se o conceito for desenvolvido para um produto final, o processo segue na espiral e um novo projeto de desenvolvimento de produto será iniciado. O novo produto vai evoluir, passando por iterações em torno da espiral e, futuramente, se tornar um projeto de aperfeiçoamento de produto, segundo Pressman (2016).

Como desvantagem, o modelo espiral necessita que a equipe tenha habilidades para tratar incertezas ou riscos associados ao projeto. Uma

avaliação equivocada de riscos pode fazer com que o custo do projeto aumente.

Dessa forma, neste tema, você aprendeu sobre as metodologias clássicas para desenvolvimento de sistemas, com foco nas metodologias cascata, prototipação, baseado em componentes, incremental e espiral.

Referências Bibliográficas

PAULA FILHO, W. P. **Engenharia de software: produtos**. 4. ed. Rio de Janeiro, RJ: Editora LTC, 2019.

PRESSMAN, R. **Engenharia de software**. 6. ed. São Paulo, SP: Makron Books, 2006.

PRESSMAN, R. **Engenharia de software: uma abordagem profissional**. 8. ed. Porto Alegre, RS: AMGH, 2016.

SBROCCO, J. H. T.; MACEDO, P. C. **Metodologias ágeis: engenharia de software sob medida**. 1. ed. São Paulo, SP: Érica, 2012.

SOMMERVILLE, I. **Engenharia de software**. 9. ed. Pearson Education do Brasil, 2011.

Metodologias Ágeis

Autoria: Thiago Salhab Alves

Leitura crítica: Anderson Paulo Avila Santos



Objetivos

- Aprender sobre as metodologias ágeis para desenvolvimento de sistemas.
- Aprender sobre o Manifesto Ágil.
- Aprender sobre as metodologias *Extreme Programming* (XP) e *Feature Driven Development* (FDD).



1. Metodologias Ágeis

Prezado (a) aluno (a), você já parou para pensar de que maneira poderíamos acelerar o processo de desenvolvimento de sistemas e manter a qualidade de desenvolvimento? Novas metodologias, com base no sucesso proporcionado pelas metodologias de desenvolvimentos clássicas, com foco em desenvolvimento ágil, certamente seriam muito bem utilizadas.

Nesta leitura, serão apresentados os conceitos das metodologias ágeis para desenvolvimento de sistemas, o manifesto ágil e sua importância para a área de desenvolvimento de sistemas, bem como apresentar as metodologias *Extreme Programming* (XP) e *Feature Driven Development* (FDD).

O custo das alterações do software eleva-se rapidamente à medida que o desenvolvimento progride. Estima-se que as alterações efetuadas quando o produto já está pronto possam custar cem vezes mais que as alterações realizadas na fase de levantamento de requisitos.

As metodologias Ágeis, então, se fazem adequadas, principalmente, em situações em que a mudança de requisitos é frequente. O termo metodologias ágeis tornou-se popular em 2001, quando dezessete especialistas em processos de desenvolvimento de software estabeleceram princípios comuns compartilhados pelos métodos, segundo Sommerville (2011):

- *Extreme Programming* (XP).
- SCRUM.
- Crystal.
- Entre outros.

1.1 Manifesto Ágil


Os conceitos de metodologias ágeis surgiram na década de 1990, motivados por uma reação aos chamados métodos pesados de desenvolvimento de software, devido ao formalismo muito grande nas documentações e regulamentações, sendo em grande maioria, gerenciado pelo método cascata, segundo Sbrocco e Macedo (2012).

Sendo assim, novos *frameworks* para processos de desenvolvimento de software começaram a surgir, sendo chamados inicialmente de métodos leves.

Em 2001, ocorreu uma importante reunião em uma estação de esqui, nas montanhas do estado de Utah, Estados Unidos, marcando o surgimento e propagação de paradigmas de desenvolvimento de software ágeis, de acordo com Pressman (2016).

Kent Beck e outros dezesseis renomados desenvolvedores, autores e consultores da área de software, denominados Aliança Ágil, assinaram o *Manifesto para o Desenvolvimento Ágil de Software (Manifest for Agile Software Development)*. Esses profissionais já trabalhavam, segundo Sbrocco e Macedo (2012), com os chamados métodos leves utilizados na época. Essas pessoas representavam as metodologias já usadas, como SCRUM, *Extreme Programming (XP)*, *Dynamic Systems Development Method (DSDM)*, *Adaptative Software Development*, *Crystal*, *Feature Driven Development (FDD)*, dentre outros.

O objetivo do encontro foi a troca de ideias sobre o que estava sendo feito, bem como discutir formas de melhorar o desempenho de seus projetos. Os participantes trouxeram suas experiências, teorias e práticas, utilizadas no desenvolvimento de projetos de software. Os profissionais trabalham em organizações diferentes, com necessidades diferentes. Apesar da constatação que cada um utilizava uma prática diferente, todos concordaram que existia um conjunto de princípios




básicos que era respeitado por todos quando os projetos eram bem-sucedidos.

Dessa forma, redigiram um documento, cujo nome foi proposto pelo inglês da turma, que usou o termo *agile* para descrever o que estava sendo discutido. Assim, o documento foi chamado de *Manifesto Ágil*, que foi assinado por todos os presentes.


A motivação para criar o manifesto ágil era abandonar os métodos antigos de desenvolvimento, que se mostravam ultrapassados, devido ao uso de hardwares mais avançados, linguagens de programação, ambientes de desenvolvimento e necessidades organizacionais. Foi um documento que encorajou o uso de melhores métodos de desenvolvimento de software, contendo um conjunto de princípios que definem os critérios para os processos de desenvolvimento ágil de sistemas. O manifesto ágil é composto de doze princípios que qualquer metodologia ágil deve ser ajustar, segundo Sbrocco e Macedo (2012, p.88):

1. “A prioridade é satisfazer ao cliente por meio de entregas contínuas e frequentes de software de valor.” O cliente deve receber de forma rápida e eficiente o produto solicitado, sendo o principal foco das equipes de desenvolvimento ágil.
2. “Mudanças de requisitos são bem-vindas, mesmo em uma fase avançada do projeto”. Processos ágeis esperam que a mudança traga uma vantagem competitiva ao cliente.
3. Processos ágeis se adaptam no caso de mudanças nos projetos ao longo de seu desenvolvimento, contudo, as mudanças devem ser aceitas somente com o intuito de agregar valor e vantagens competitivas ao negócio do cliente.
4. “Entregas com frequência de software funcional, sempre na menor escala de tempo, de algumas semanas a alguns meses, preferindo sempre um período curto.” Buscar realizar entrega funcional de



software, dentro de parâmetros de qualidade e o mais breve possível.

5. “As equipes de negócio e de desenvolvimento devem trabalhar juntas diariamente durante o projeto.” Toda a equipe de desenvolvimento deve trabalhar junta e estar alinhada, principalmente com o cliente, que deve estar sempre presente.
6. “A maneira mais eficiente da informação circular entre a equipe de desenvolvimento é por uma conversa cara a cara.” O cliente sempre deve estar presente, realizando reuniões mais frequentes e rápidas, para melhor entendimento das necessidades do cliente e negócio.
7. “Ter um software funcionando é a medida primária de progresso.” Esta preocupação remete ao entendimento de que entregar um software que não funcione é o mesmo que não entregá-lo. Espera-se que, na medida em que as funcionalidades sejam validadas, sejam implementadas e entregues ao cliente.
8. “Processos ágeis promovem o desenvolvimento sustentável. Patrocinadores (quem financia o projeto), desenvolvedores e usuários devem ser capazes de manter um ritmo constante.” Deve-se manter um ritmo de entrega de novas funcionalidades, mantendo a comunicação e foco nas prioridades do projeto.
9. “Atenção contínua à excelência técnica e um bom design aumenta a agilidade.” Manter um padrão de projeto para garantir a qualidade e excelência técnica.
10. “Simplicidade é essencial.” Criar projetos simples para atendimento dos requisitos dos clientes, evitando arquiteturas complexas que podem atrasar as entregas.
11. “As melhores arquiteturas, requisitos e projetos provêm de equipes organizadas.” Formar equipes autoorganizáveis, que se adaptam às mudanças e reinventam e reestruturam o negócio com criatividade.
12. “Em intervalos regulares, a equipe deve refletir sobre como se tornar mais eficaz, buscando sintonizar e ajustar seus



comportamentos.” Realizar reuniões de avaliação de desempenho são importantes para identificar processos falhos e desnecessários e, planejar planos de melhoria.

Por meio do manifesto ágil foi criada a Aliança Ágil, que é uma organização sem fins lucrativos, que tem por objetivo promover o conhecimento e discussões sobre as metodologias ágeis, sendo que muitos dos líderes do manifesto ágil são membros da aliança ágil.

1.2 Abordagens Ágeis – XP e FDD

Extreme Programming (XP)

O *Extreme Programming*, por Sommerville (2011), é uma metodologia ágil para equipes pequenas e médias que desenvolvem software baseado em requisitos vagos e que são modificados rapidamente. Como princípios da XP estão:

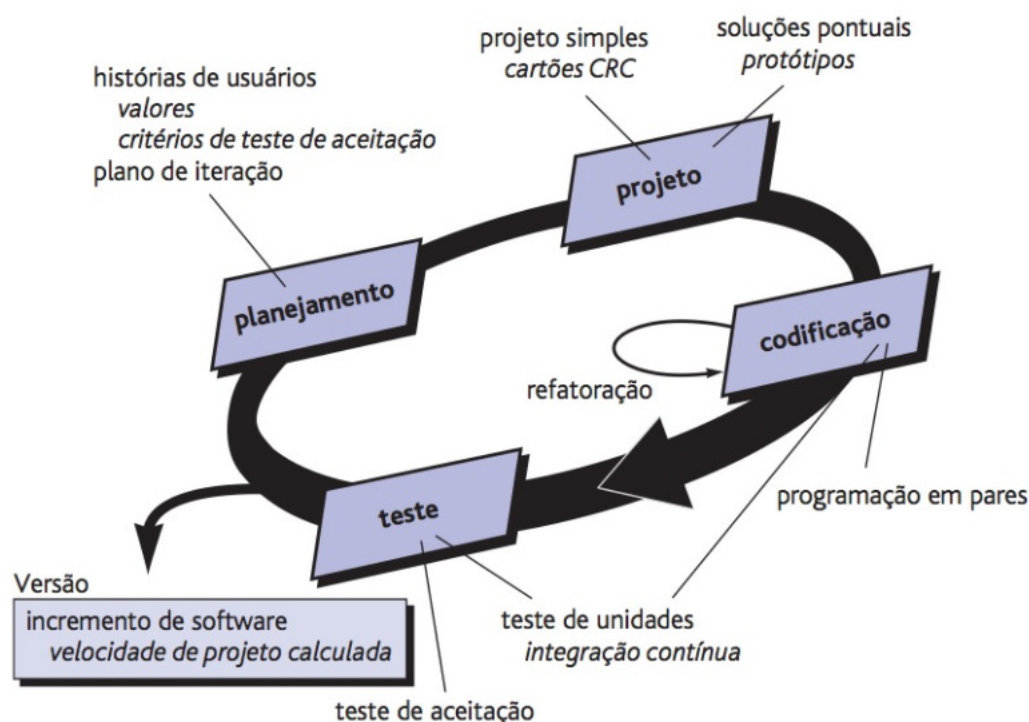
- Feedback constante.
- Abordagem incremental.
- Comunicação entre pessoas é encorajada.
- Simplicidade.

O feedback constante significa que terá, frequentemente, uma parte do software totalmente funcional para avaliar, podendo sugerir novas características e informações aos desenvolvedores. Erros e não conformidades serão rapidamente identificados e corrigidos nas versões seguintes, tendo ao final um produto de acordo com as expectativas do cliente.

O *Extreme Programming* baseia-se em doze práticas, sendo essa metodologia mais comportamental e suas práticas baseadas

atitudes, não sendo necessária a implementação dessas práticas simultaneamente, recomendando-se que sejam aplicadas gradativamente. Algumas não são novas, sendo usadas há muitos anos na indústria. As práticas são ilustradas na Figura 1, segundo Pressman (2016):


Figura 1 – Processo do *Extreme Programming* (XP)



Fonte: Pressman (2016, p. 72).

- Planejamento: consiste em decidir o que é necessário fazer e o que pode ser adiado no projeto. A XP baseia-se em requisitos atuais reais para desenvolvimento de software, não em possíveis requisitos futuros.
- Entregas frequentes: visam a construção de um software simples, atualizado à medida que novos requisitos surgem. Cada versão deve conter os requisitos de maior valor para o negócio.

- Metáfora: são as descrições de um software sem a utilização de termos técnicos, com o intuito de guiar o desenvolvimento do software.
- Projeto simples: o programa deve ser o mais simples possível e satisfazer aos requisitos atuais, sem a preocupação com requisitos futuros. Eventuais requisitos futuros devem ser adicionados assim que realmente existirem.
- Testes: a XP focaliza a validação do projeto durante todo o processo de desenvolvimento. Os programadores desenvolvem o software criando, primeiramente, os casos de testes.
- Programação em Pares: a implementação do código é feita em duplas, ou seja, dois desenvolvedores trabalham em uma única máquina. Um desenvolvedor implementa o código enquanto o outro observa continuamente o trabalho que está sendo feito, procurando por erros sintáticos e semânticos e pensando em como melhorar o código.
- Refatoração: aperfeiçoar o projeto do software em todo o desenvolvimento. A refatoração deve ser feita sempre que for possível simplificar uma parte do software, sem que seja perdida nenhuma funcionalidade.
- Propriedade coletiva: o código do projeto pertence a todos os membros da equipe. Qualquer pessoa pode adicionar valor a um código, mesmo que não o tenha desenvolvido.
- Integração contínua: uma vez testado e validado, o código produzido por uma equipe deve ser integrado ao sistema e este, por sua vez, também deve ser testado.
- Trabalho semanal de 40 horas: a XP assume que não se deve fazer horas extras constantemente. Caso seja necessário trabalhar mais que 40 horas pela segunda semana consecutiva, há um problema



no projeto que deve ser resolvido não com o aumento de horas trabalhadas, mas com melhor planejamento, por exemplo.


- Cliente presente: o cliente deve participar efetivamente de todo o desenvolvimento do projeto, devendo estar disponível para sanar todas as dúvidas sobre requisitos.
- Código-padrão: recomenda-se a adoção de regras de escrita. A padronização favorece o trabalho em equipe e a propriedade coletiva do código.

Assim, o cliente constantemente sugerirá novas características e informações aos desenvolvedores, possuindo um produto que atenda aos requisitos e as expectativas do cliente.

Feature Driven Development (FDD)

A metodologia *Feature Driven Development* (FDD), de acordo com Sbrocco e Macedo (2012), foi criada na década de 1990, em Singapura, sendo utilizada pela primeira vez para o desenvolvimento de um sistema bancário internacional, considerado, inicialmente, inviável de ser desenvolvido em um prazo predeterminado. O FDD é uma metodologia ágil e robusta e apresenta as seguintes características básicas:

- Benefícios a gerentes, desenvolvedores e clientes: apresenta formas de interação e controles fáceis, fazendo com que desenvolvedores fiquem confortáveis com a implementação, pois possui um conjunto de regras de fácil entendimento e resultados rápidos.
- Benefício ao cliente por meio de trabalho significativo: no início da implementação visa agregar valor ao cliente, promovendo a disponibilização daquilo que já foi desenvolvido e testado.
- Atende equipes pequenas, médias ou grandes: fornece estrutura para atender todos os tipos de equipes de trabalho, desde




pequenas até grandes, oferecendo um conjunto de atribuições que podem ser ampliadas.

- Software de qualidade: engloba conjunto de métricas de qualidade a serem aplicadas durante o desenvolvimento do projeto de software, enfatizando que o software deve ter qualidade desde o início da implementação.
- Entrega de resultados frequentes, tangíveis e funcionais: sempre que uma funcionalidade for implementada e testada, deve ser disponibilizada aos usuários.
- Permite acompanhamento do progresso do desenvolvimento do projeto: possui forma simples de visualização do andamento do desenvolvimento do software, por meio de um meio gráfico.

As boas práticas da *Feature Driven Development* (FDD), a serem seguidas e detalhadas por Sbrocco e Macedo (2012), são:

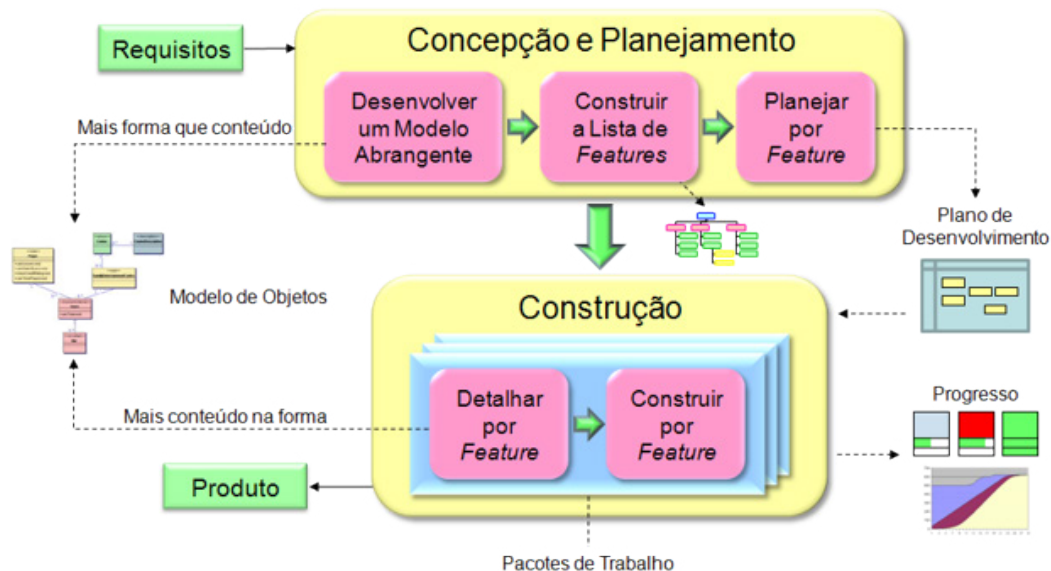
1. Modelagem de objetos do domínio: atividade que deve ser realizada com toda a equipe com a finalidade de estudar, analisar e modelar um sistema. Deve-se produzir um documento de requisitos a partir de um método de coleta de dados, que pode ser uma entrevista, questionários, além de diagramas da *Unified Modeling Language* (UML) que ajudam o desenvolvedor.
2. Desenvolvimento por funcionalidade: essa prática faz a sugestão que se construa uma lista de funcionalidades, decompondo funcionalmente o domínio do negócio em áreas de negócio, atividades do negócio e passos da atividade de negócio.
3. Entregas regulares: o desenvolvedor deve sempre reconstruir o software, incluindo novas funcionalidades, assim desenvolvedores trabalham e utilizam uma versão sempre atualizada do projeto. Os clientes vão se beneficiar, pois sempre estarão com a última versão do projeto.

- 
4. Formação da equipe de projeto—a equipe deve ser composta pelos seguintes profissionais:
 - a. Gerente de projeto: possui contato direto como cliente (*stakeholders*) e capta todos os requisitos e restrições. O gerente monta sua equipe com afinidades e experiências das pessoas. Cabe ao gerente determinar a quantidade de analistas e acompanhar todo o desenvolvimento.
 - b. Equipe de modelagem e planejamento representam um grupo de pessoas definidos pelo gerente para elaborar a lista de funcionalidades do sistema. Esse profissional deve dominar, entre outras habilidades, técnicas de modelagem de sistema, como UML, devendo dividir as funcionalidades para a equipe de funcionalidades (desenvolvedores), por meio do programador chefe, que determina seu sequenciamento.
 - c. Programador chefe: refina a lista de funcionalidades e as transforma em modelo de objetos, organizando os trabalhos, escolhendo a linguagem a ser utilizada, formas de armazenamento, revisão e liberação ao cliente.
 - d. Equipe de funcionalidades: são os desenvolvedores, profissionais que conhecem as ferramentas de desenvolvimento e escreve os métodos e classes (no caso da Programação Orientada a Objetos) para a concepção do projeto. Essa equipe também é responsável por construir testes de unidade para validar o projeto e inspecionar o software em todas as suas fases.
 5. Posse individual do código: elaborar uma lista contendo as funcionalidades do sistema e seus proprietários, de forma que o programador seja responsável pela funcionalidade implementada.

A Figura 2 ilustra o funcionamento do FDD. A partir dos requisitos, em concepção e planejamento, são realizadas atividades de análise orientada a objetos, decomposição funcional e planejamento incremental por funcionalidades. Na construção, são realizadas

atividades de projeto orientado a objetos e programação e testes orientado a objetos, gerando o produto.

Figura 2 - Feature Driven Development (FDD)




Fonte: <https://pt.wikipedia.org/wiki/Ficheiro:Fdd.png>. Acesso em: 8 jun 2020.

1.3 Motivações para Metodologias Ágeis

Uma das grandes motivações para o uso de metodologias ágeis, de acordo com Sbrocco e Macedo (2012), se deve ao fato de serem menos focadas em documentações, mais voltadas ao código-fonte.

Outro motivo se deve ao fato das metodologias ágeis serem adaptativas e não predeterminantes como as metodologias tradicionais, que tendem a fazer um planejamento do processo de desenvolvimento de software para um longo período de tempo, funcionando bem até que se tenha a necessidade de mudanças. Na metodologia ágil, as mudanças ao longo do processo de desenvolvimento, ocorrem de forma natural sempre que solicitadas.

As metodologias ágeis, por Sbrocco e Macedo (2012) são orientadas a pessoas e não a processos. Na engenharia de software tradicional,



um processo é estabelecido independente da habilidade da equipe de desenvolvimento e na metodologia ágil os processos existentes têm por objetivo dar suporte à equipe de desenvolvimento.

Os requisitos são mutáveis e impedir que possam mudar ou congelar requisitos, que é uma tarefa difícil. No caso de projetos com requisitos mutáveis, uma metodologia previsível não funcionará, visto que um bom requisito identificado hoje, pode não ter mais a mesma importância ou prioridade em alguns meses. Dessa forma, uma metodologia adaptativa proporciona um controle da imprevisibilidade, segundo Sbrocco e Macedo (2012).

Para que o desenvolvimento iterativo funcione, é necessário produzir versões do sistema que funcionem e, cada versão, deve ser totalmente testada e integrada como se fosse a entrega final. Os processos adaptativos vão necessitar de diferentes tipos de relacionamento com clientes.

A maioria dos clientes acaba buscando um contrato de custo fixo, porém, requerem requisitos estáveis. Assim, pode-se fixar tempo e preço, deixando variar o escopo de forma controlada, proporcionando ao cliente um controle mais refinado do processo de desenvolvimento e aproximando os clientes dos desenvolvedores, segundo Sbrocco e Macedo (2012).

As metodologias ágeis proporcionam melhor visibilidade do estado real do projeto, ao contrário das metodologias tradicionais, em que a qualidade acaba sendo medida pela análise do planejamento. Nas metodologias ágeis, aplicar um processo adaptativo exige uma equipe de desenvolvimento que possua integrantes qualificados.

Para o conceito tradicional, as pessoas são partes substituíveis e o ponto mais importante são os processos, falhando por não levar em conta fatores relacionados à imprevisibilidade das pessoas durante a definição

de processos e das metodologias utilizadas, o que pode proporcionar todo tipo de trajetórias não planejadas durante a condução de projetos.

Em um cenário no qual processos determinam que pessoas devam dizer como um trabalho deve ser feito, não sendo essas pessoas as mesmas que executam o trabalho de fato, gerentes buscam alguma forma de medir a efetividade (produtividade) de seus colaboradores.

Usar uma metodologia ágil não é para todos, embora essas metodologias sejam amplamente aplicáveis e devessem ser utilizadas por mais pessoas, pois existe a vantagem de ser um passo bem mais largo que usar uma metodologia pesada. Um dos limitadores, dessas novas metodologias, é o uso com equipes grandes de desenvolvedores, pois tem ênfase em equipes pequenas.

Dessa forma, você pode aprender sobre as metodologias ágeis para desenvolvimento de sistemas, sobre o Manifesto Ágil e as metodologias *Extreme Programming* (XP) e *Feature Driven Development* (FDD).

Referências Bibliográficas

PRESSMAN, r. **Engenharia de software**. 6. ed. São Paulo, SP: Makron Books, 2006.

PRESSMAN, R. **Engenharia de software: uma abordagem profissional**. 8. ed. Porto Alegre, RS: AMGH, 2016.

SBROCCO, J. H. T.; MACEDO, P. C. **Metodologias ágeis: engenharia de software sob medida**. 1. ed. São Paulo, SP: Érica, 2012.

SOMMERVILLE, I. **Engenharia de software**. 9. ed. Pearson Education do Brasil, 2011.

Metodologia Scrum

Autoria: Thiago Salhab Alves

Leitura crítica: Anderson Paulo Avila Santos



Objetivos

- Aprender sobre a metodologia Scrum.
- Aprender sobre os princípios da metodologia Scrum.
- Aprender sobre os papéis, artefatos e cerimônias da metodologia Scrum.



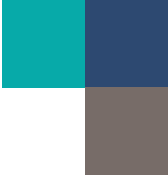
1. Metodologia Scrum

Prezado (a) aluno (a), você já parou para pensar de que forma podemos realizar o processo de desenvolvimento de sistemas de forma ágil e eficiente? Uma metodologia que ganha grande destaque é a Scrum, metodologia que pode ser aplicada a diferentes tipos de organização e apresenta um processo de aplicação focado nas necessidades do cliente.

Nesta leitura, serão apresentados os conceitos da metodologia Scrum, bem como os princípios, papéis, artefatos e cerimônias.

A metodologia SCRUM sofreu influência das boas práticas adotadas pela indústria japonesa, em especial por princípios da manufatura enxuta, implementados pelas companhias Honda e Toyota. O nome dessa metodologia está associado a uma formação típica do jogo rugby que é scrum. No rugby, essa formação é utilizada após determinado incidente ou quando a bola sai de campo, ou seja, é utilizada para reiniciar o jogo, reunindo todos os jogadores. O uso dessa terminologia pareceu adequado, pois, no rugby, cada time age em conjunto, como uma unidade integrada, onde cada integrante desempenha um papel específico e todos se ajudam em busca de um benefício comum, segundo Pressman (2016).

Os responsáveis pela criação da metodologia Scrum, a pedido da *Object Management Group* (OMG), foram Jeff Sutherland e Ken Schwaber, que trabalharam juntos para formalizar e definir o que tinham aprendido, criando, assim, a metodologia SCRUM. A partir de então, a metodologia Scrum tem sido utilizada por todo o mundo, nos mais variados domínios de aplicação e seu sucesso se atribui, principalmente, ao fato dos métodos tradicionais de desenvolvimento terem o foco na geração de documentação sobre o projeto e no cumprimento rígido de processos. As metodologias ágeis, como o Scrum, concentram atenções no produto final e nas interações dos indivíduos, segundo Sbrocco e Macedo (2012).




Scrum é metodologia ágil com uma comunidade grande de usuários. Seu objetivo é fornecer um processo conveniente para projeto e desenvolvimento orientado a objeto. Apresenta uma abordagem empírica que aplica algumas ideias da teoria de controle de processos industriais para o desenvolvimento de software, reintroduzindo a ideia de flexibilidade, adaptabilidade e produtividade, segundo Sommerville (2011).

Os princípios usados pelo Scrum estão alinhados com o manifesto ágil e são usados para orientar as atividades de desenvolvimento de sistemas, dentro de um processo que possui as seguintes atividades metodológicas, segundo Pressman (2006): requisitos, análise, projeto, evolução e entrega.

O Scrum não deve ser utilizado apenas para o desenvolvimento de software, uma vez que sua característica iterativa e incremental pode ser utilizada no desenvolvimento de qualquer produto ou gerenciar qualquer trabalho, sendo adequado, principalmente, quando o produto é construído em partes (iterações). Quando uma parte é finalizada, um novo incremento é produzido até que o desenvolvimento completo, segundo Sbrocco e Macedo (2012).

Scrum é baseado no empirismo, doutrina que defende a ideia de que “o conhecimento vem da experiência e da tomada de decisões baseadas no que é conhecido”, de acordo com Paula Filho (2019, p.93). A abordagem se propõe a usar três pilares: transparência, inspeções e adaptação. Vários tipos de reuniões formais são realizados para garantir o método, segundo Paula Filho (2019).

A transparência indica que os responsáveis pelos resultados devem enxergar os aspectos significativos do processo. Para que os observadores do processo tenham um entendimento comum, os aspectos devem ser definidos por um padrão comum. Por exemplo, o




processo deve ser descrito por uma linguagem comum e o significado do conceito de pronto deve ser compartilhado.

As inspeções devem ser executadas de forma suficientemente frequente, mas não tão frequente que chegue a atrapalhar o andamento do projeto. Os inspetores devem ser proficientes no trabalho a inspecionar.

A adaptação significa que o processo ou produto deve ser ajustado, sempre que as inspeções determinem que aconteceram desvios para fora dos limites aceitáveis. Os ajustes devem ser os mais precoces possíveis.

O foco é encontrar uma forma de trabalho dos integrantes da equipe para produzir o software de forma flexível e em um ambiente em constante mudança. Visa tratar mudanças frequentes de requisitos do software e outras situações, como:

- Trocas de equipes: deve haver um rodízio de equipes ao término do projeto para melhor integração das pessoas.
- Adaptações de cronogramas e orçamento: os cronogramas e orçamentos do projeto podem sofrer alterações e devem ser adaptados.
- Trocas de ferramentas de desenvolvimento ou linguagem de programação: as equipes utilizam diferentes ferramentas e linguagens, principalmente, nas trocas de equipes e início de novos projetos.
- Equipes pequenas: as equipes devem ser compostas de até dez pessoas, sendo o líder (Scrum Master), analistas, engenheiros de software, programadores e equipes de testes.

- 
- Trabalhando com requisitos instáveis ou desconhecidos: os requisitos dos projetos, geralmente, são mutáveis.
 - Utilizando iterações curtas para melhor visibilidade para o desenvolvimento: as iterações trabalhadas não são longas, a fim de que se obtenha melhor visibilidade do andamento do projeto.

O Scrum segue os princípios do manifesto ágil e se baseia em seis características, segundo Sbrocco e Macedo (2012):

- Flexibilidade dos resultados.
- Flexibilidade dos prazos.
- Times pequenos.
- Revisões frequentes.
- Colaboração.
- Orientação a objetos.

O Scrum deve ser utilizado para as seguintes situações, de acordo com Sbrocco e Macedo (2012):

- Desenvolvimento de aplicativos complexos, onde os requisitos mudam rapidamente e constantemente.
- Gerenciamento e controle do desenvolvimento do trabalho.
- Autogerenciamento de equipes.
- Implementação de conceito iterativo e incremental no desenvolvimento de software.
- Identificação de causas de problemas e remoção de impedimentos.

- Valorização dos indivíduos.

Scrum divide o desenvolvimento em ciclos iterativos de até trinta dias. Esses ciclos são formados por equipes pequenas, de até dez pessoas, com analistas, programadores, engenheiros e gerentes de qualidade. Essas equipes trabalham nas funcionalidades (requisitos) definidas no início de cada ciclo.

Na Scrum, há reuniões de acompanhamento diárias, de aproximadamente quinze minutos, onde são discutidos diversos assuntos, como o que foi feito desde a última reunião e o que precisa ser feito até a próxima. As dificuldades encontradas são identificadas e resolvidas. Eventuais problemas no projeto são discutidos e resolvidos diariamente, evitando que sejam prolongados, de acordo com Sommerville (2011).

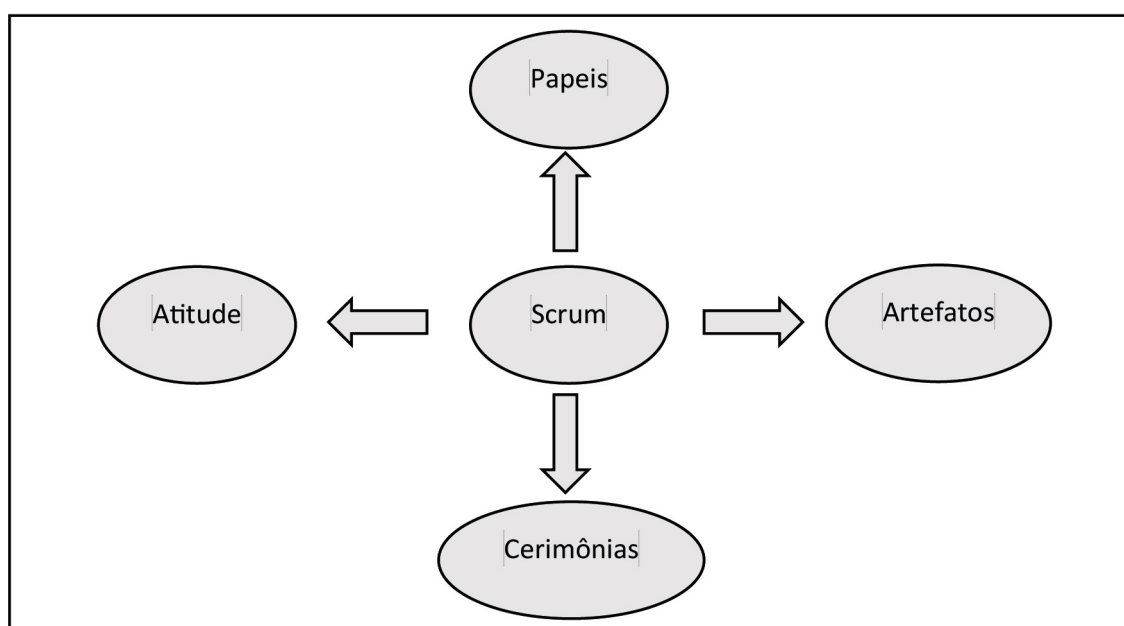
O ciclo de vida da Scrum é baseado em três fases principais, divididas em subfases, segundo Sommerville (2011):

- Pré-planejamento (*pre-game phase*): os requisitos são descritos em um documento chamado *backlog*. Posteriormente, são ordenados por prioridade e, para cada um, são realizadas estimativas de esforço para o desenvolvimento. O planejamento inclui a definição da equipe de desenvolvimento, as ferramentas a serem usadas, os possíveis riscos do projeto e necessidades de treinamento.
- Desenvolvimento (*game phase*): os riscos identificados, previamente, são observados e controlados durante o desenvolvimento. Nesta fase, o software é desenvolvido em ciclos, em que novas funcionalidades são adicionadas. Cada um desses ciclos é desenvolvido de forma tradicional, ou seja, primeiro se faz a análise e, em seguida, o projeto, implementação e testes. Os ciclos são planejados para durar entre uma semana e um mês.

- Pós-planejamento (*post-game phase*): realiza integração do software, os testes finais e a documentação do usuário. A equipe se reúne para analisar o progresso do projeto e demonstrar o software atual para os clientes.


A Figura 1 ilustra os quatro fundamentos que sustentam o Scrum, que são os papéis (são os responsáveis por fazer o Scrum ser aplicado, sendo *Product Owner*, *Scrum Master*, *Team* e Cliente), cerimônias (reuniões que ocorrem em momentos diferentes de um determinada sprint, sendo *Sprint Planning Meeting*, *Daily Meeting* ou *Daily Scrum*, *Sprint Review* e *Sprint Retrospective*), Artefatos (são elementos que são produzidos em momentos diferentes de uma determinada Sprint, sendo o *Product Backlog*, *Sprint Backlog* e o *Burndown Chart*). Todos esses elementos serão detalhados em seguida, durante a explicação da metodologia Scrum.

Figura 1 – Fundamentos básicos do SCRUM



Fonte: adaptada de Sbrocco e Macedo (2012 p. 161).

O ciclo de desenvolvimento da metodologia Scrum, conforme apresentado na Figura 2, inicia o projeto reunindo clientes e desenvolvedores, com o objetivo de definir o *Product Backlog*, que é

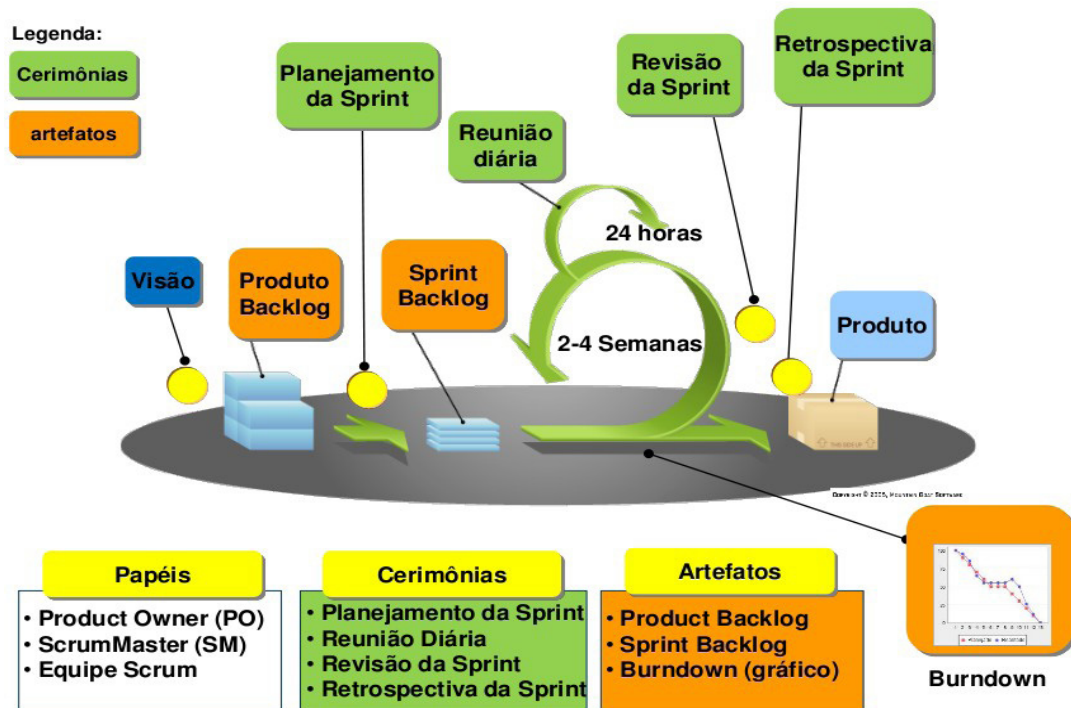


a lista de requisitos. São estimados os custos do projeto e realizada a análise dos riscos, assim como as ferramentas que serão usadas no trabalho, os integrantes da equipe e definidas as datas para entrega dos resultados a partir de priorizações sinalizadas pelo cliente, de acordo com Sbrocco e Macedo (2012).

Define-se também no início do projeto, o *Scrum Master*, que é eleito pela equipe alocada no projeto. Uma vez que o *Product Backlog* é definido, a equipe definirá a *Sprint Backlog*, que contém a lista de atividades que serão realizadas na próxima *Sprint*, onde são definidas as responsabilidades de cada integrante da equipe. Após discutir quais os padrões a serem adotados, as atividades de análise, codificação e testes se iniciam, conforme a Figura 2, e, ao final de cada *Sprint*, um incremento do produto é apresentado ao cliente para avaliação. Se alguma inconsistência for encontrada, deve ser adicionada ao *Product Backlog*. O Scrum deve ser controlado avaliando funcionalidades não entregues, necessidade de mudanças para corrigir inconsistências, problemas técnicos, entre outros, segundo Sbrocco e Macedo (2012).

Os projetos que utilizam Scrum possuem uma série sequencial de *Sprints*, que correspondem às iterações, gerando um produto de valor para o cliente. As *Sprints* ocorrem em um período de duas a quatro semanas, sendo o produto projetado, codificado e testado durante a *Sprint*.

Figura 2 – Fundamentos básicos do SCRUM




Fonte: elaborada pelo autor.

Dentre os papéis da metodologia Scrum, o *Product Owner* representa o cliente, sendo o responsável em garantir que a equipe Scrum entregue o solicitado, desempenhando papel de moderador entre os interesses do cliente e do Team. O *Product Owner* é responsável da seguinte forma, por Pressman (2016):

- Definir as funcionalidades do produto.
- Definir prioridades.
- Elaborar e manter o *Product Backlog*.
- Decidir datas de lançamento do produto.

O *Scrum Master* é o representante do cliente no projeto e desempenha um papel importante de facilitador, responsável pela remoção de impedimentos (problemas técnicos, administração de conflitos, itens não



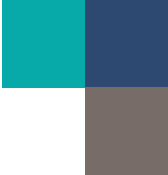
planejados etc.) que, eventualmente, surjam durante o desenrolar do desenvolvimento, segundo Sbrocco e Macedo (2012).

Scrum *Master* desempenha papel de responsabilidade técnica na condução do projeto, devendo proporcionar mecanismos, preferencialmente, informatizados de comunicação entre seus integrantes. Deve proteger a equipe e mantê-la focada em suas tarefas. São suas responsabilidades:

- Liderar o projeto, representando sua gerência.
- Tratar os impedimentos.
- Zelar pela equipe Scrum.
- Auxiliar o *Product Owner* com o *Product Backlog*.
- Ser o facilitador da equipe Scrum, garantindo sua produtividade.
- Aplicar os valores e práticas Scrum.

O *Team*, por Sbrocco e Macedo (2012), é composto por um grupo de cinco a nove integrantes, que possuem características multifuncionais. Essa equipe deve contar com analistas, programadores, testadores, que devem trabalhar com dedicação integral ao projeto. O *Team* Scrum é responsável por:

- Definir as tarefas que serão realizadas.
- Fazer estimativas.
- Desenvolver o produto.
- Garantir a qualidade do produto.
- Apresentar o produto ao cliente.



As cerimônias são as reuniões que ocorrem em momentos diferentes dentro de uma *Sprint*. O *Sprint Planning Meeting* é a primeira reunião do projeto que conta com a participação de todos, sendo uma reunião de no máximo oito horas. Nessa reunião, o *Product Owner* planeja e elabora a lista de prioridades que devem ser cumpridas no projeto, com a reunião dividida em duas partes, segundo Sommerville (2011):

- *Product Owner* define suas prioridades, selecionando os itens do *backlog* e a meta da *Sprint*.
- A equipe define a *Sprint Backlog*, documento que contém as tarefas que devem ser executadas para se cumprir a meta da *Sprint*.

A *Daily Scrum* é uma reunião simples, mas muito importante para a *Scrum*. Na reunião, cada integrante do time responde o que já fez, o que pretende fazer e se há algum impedimento para a conclusão das tarefas. Essa reunião deve ser rápida, com duração de aproximadamente quinze minutos, contendo o *Team* e o *Scrum Master*, em que os integrantes devem ficar em pé, ao lado de um *task board* (quadro de tarefas). Cada membro do *Team* deve responder a três perguntas, de acordo com SBROCCO e MACEDO (2012):

- O que fiz desde a última reunião?
- O que vou fazer até a próxima?
- Tive ou estou tendo algum impedimento? Qual (is)?

A *Sprint Review* é uma reunião sobre tudo o que foi feito durante uma *Sprint*. Nessa reunião, o *Team* deve apresentar os resultados da *Sprint* ao *Product Owner*, contendo apenas as atividades que estiverem cem por cento prontas. Essa reunião deve ter uma estimativa de quatro horas e contar com o *Product Owner*, *Scrum Master* e equipe *Scrum*, sendo marcada sempre no final da *Sprint*. Espera-se os seguintes objetivos após a reunião, de acordo com Sbrocco e MAcedo (2012):

- Apresentar o que a equipe fez durante a *Sprint*.
- Entregar o software funcionando ao *Product Owner* (a parte que foi implementada).

O *Product Owner* tem o direito de aceitar ou rejeitar a *Sprint*. Qualquer necessidade de mudança ou inserção de novas funcionalidades será incorporado ao *Product Backlog*.

A *Sprint Retrospective*, por Sbrocco e Macedo (2012), é uma reunião de fechamento, buscando identificar os pontos positivos e negativos durante a *Sprint*. São avaliados o trabalho em equipe e melhorias nas estratégias de melhoria que podem ser adotadas. Participam dessa reunião o *Team* e *Scrum Master*, podendo se convidar o *Product Owner*. Essa reunião deve acontecer após a revisão da *Sprint* com duração aproximada de três horas. Responde-se basicamente a duas perguntas:

- O que foi bom durante a *Sprint*?
- O que se pode fazer para melhor a próxima?

Dentre os artefatos da metodologia Scrum, por Sbrocco e Macedo (2012), são produzidos em momentos diferentes de uma *Sprint*, três artefatos, que são o *Product Backlog*, o *Sprint Backlog* e o *Burndown Chart*. O *Product Backlog* é um documento que contém todos os itens que devem ser desenvolvidos durante o projeto, sendo uma lista de prioridades que são produzidas no início do projeto, contendo tudo o que deve ser entregue ao cliente. O *Product Backlog* deve ser criado e mantido pelo *Product Owner*, que pode alterar esse documento quando quiser.

O Quadro 1 ilustra um exemplo de conteúdo de um *Product Backlog*. A tabela apresenta o nível de prioridade, com base no *Return of Investment* (ROI) para o cliente, o Ator (quem usará o item), Requisitos Funcionais (número de identificação do requisito, por exemplo, RF001, a Descrição

do Item (descrição do que deve ser realizado), Tamanho estimado pelo *Team*, Release (versão que está sendo desenvolvida), *Sprint* (a qual *Sprint* se relaciona) e o Status, que pode ser *To Do* (para fazer), *Doing* (fazendo), *Done* (feito) ou *To Verify* (para verificar).

Quadro 1 – Exemplo de conteúdo do *Product Backlog*

Nível de Prioridade (ROI)	Ator	Requisitos funcionais	Descrição do Item	Tamanho	Release	<i>Sprint</i>	Status
Essencial	Usuário	RF001	Cadastro de usuário.	24	1	1	<i>To Do</i>
Essencial	Usuário	RF002	Cadastro de currículo.	24	1	1	<i>To Do</i>
Essencial	Usuário	RF003	Cadastro de interesse.	24	1	2	<i>To Do</i>
Essencial	Usuário	RF004	Busca de oportunidades de emprego.	16	1	2	<i>To Do</i>
Essencial	Empresas	RF005	Cadastro de empresas.	24	1	3	<i>To Do</i>
Essencial	Empresas	RF006	Busca de candidato a emprego.	16	1	3	<i>To Do</i>

Fonte: adaptado de Sbrocco e Macedo (2012 p. 168).

O *Sprint Backlog* representa todas as tarefas que devem ser desenvolvidas durante uma *Sprint*. O Quadro 2 mostra o exemplo de uma *Sprint Backlog*. Cada item deve ser detalhado em Tarefas e cada uma dessas tarefas deve ter uma estimativa de esforço, que, no caso, é em horas. Por exemplo, a tarefa Criar Base de Dados será a primeira a ser executada e levará quatro horas para ser executada. O objetivo é

dar visibilidade ao andamento das tarefas. O *Team* só deve começar a trabalhar o segundo Item após a conclusão do primeiro.

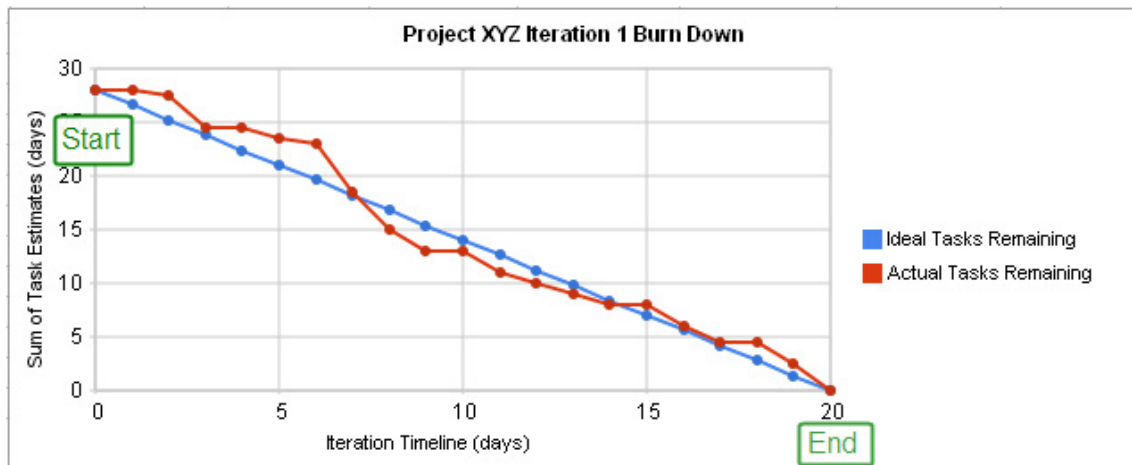
Quadro 2 – Exemplo de conteúdo do *Sprint Backlog*

Item	Tarefa	1	2	3	4	5	6	7	8	9	10	Total
ES006	Criar base de dados.	4	0	0	0	0	0	0	0	0	0	4
	Desenvolver modelo.	8	8	0	0	0	0	0	0	0	0	16
	Desenvolver controle.	0	0	4	4	0	0	0	0	0	0	8
	Desenvolver view.	0	0	0	4	4	0	0	0	0	0	8
	Teste de unidade.	0	0	0	0	4	2	2	0	0	0	8
	Teste funcional.	0	0	0	0	0	4	4	2	0	0	10
	Documentação técnica.	0	0	0	0	0	0	4	4	2	2	12

Fonte: adaptado de Sbrocco e Macedo (2012 p. 169).

O *Burndown Chart* tem por objetivo apresentar o quão próximo está de se atingir a meta. A coluna vertical representa a quantidade de esforço (quantidade de trabalho a ser realizada) e a coluna horizontal representa os dias de uma iteração. A linha azul indica o fluxo ideal de trabalho e a linha vermelha, o fluxo de trabalho atual. Se a linha vermelha está acima da linha azul, indica atraso do *Team*. Em cima da linha, o *Team* está no fluxo ideal de trabalho e abaixo da linha azul, o *Team* está superando as expectativas. O *BurnDown Chart* é usado, principalmente, pelo *Scrum Master* para o acompanhamento do trabalho.

Figura 3 – Exemplo de uso de *Burndown Chart*



Fonte: https://upload.wikimedia.org/wikipedia/commons/8/8c/Burn_down_chart.png.
Acesso em: 8 jun 2020.

Dessa forma, você pode aprender sobre os conceitos da metodologia Scrum, bem como os princípios, papéis, artefatos e cerimônias.

Referências Bibliográficas

- PAULA FILHO, W. P. **Engenharia de software: produtos**. 4. ed. Rio de Janeiro, RJ: Editora LTC, 2019.
- PRESSMAN, R. **Engenharia de software**. 6. ed. São Paulo, SP: Makron Books, 2006.
- PRESSMAN, R. **Engenharia de software: uma abordagem profissional**. 8. ed. Porto Alegre, RS: AMGH, 2016.
- SBROCCO, J. H. T.; MACEDO, P. C. **Metodologias ágeis: engenharia de software sob medida**. 1. ed. São Paulo, SP: Érica, 2012.
- SOMMERVILLE, I. **Engenharia de software**. 9. ed. Pearson Education do Brasil, 2011.



Bons estudos!