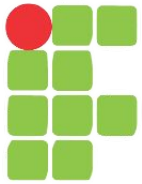


INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SERGIPE

Engenharia de Software

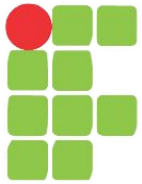
Prof. Christiano Lima Santos

Conteúdo do Curso

- Introdução à Engenharia de Software
- Modelos de Processo
- Desenvolvimento Ágil
- Técnicas de Elicitação de Requisitos
- Unified Modeling Language (UML)
- Gestão de Projetos

Extras

- Princípios na prática de Engenharia de Software
- Teste de Software



INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SERGIPE

Introdução à Engenharia de Software

Parte 01

Sumário

- Por que estudar Engenharia de Software?
- O que é software?
- Características do software
- Evolução e crise do software
- Mitos relativos ao software
- Definição de Engenharia de Software
- Camadas da Engenharia de Software

Por que estudar Engenharia de Software?

- Software afeta todos os aspectos de nossas vidas:
  - Financeiro – sistemas bancários *online*;
  - Educacional – ambientes virtuais de aprendizagem;
  - Social – redes sociais.
- Software tornou-se pervasivo (incorporado) no comércio, na cultura e nas atividades cotidianas;
- Entretanto, softwares são cada vez mais complexos!
- A Engenharia de Software nos capacita para o desenvolvimento de sistemas complexos dentro do prazo e com alta qualidade.

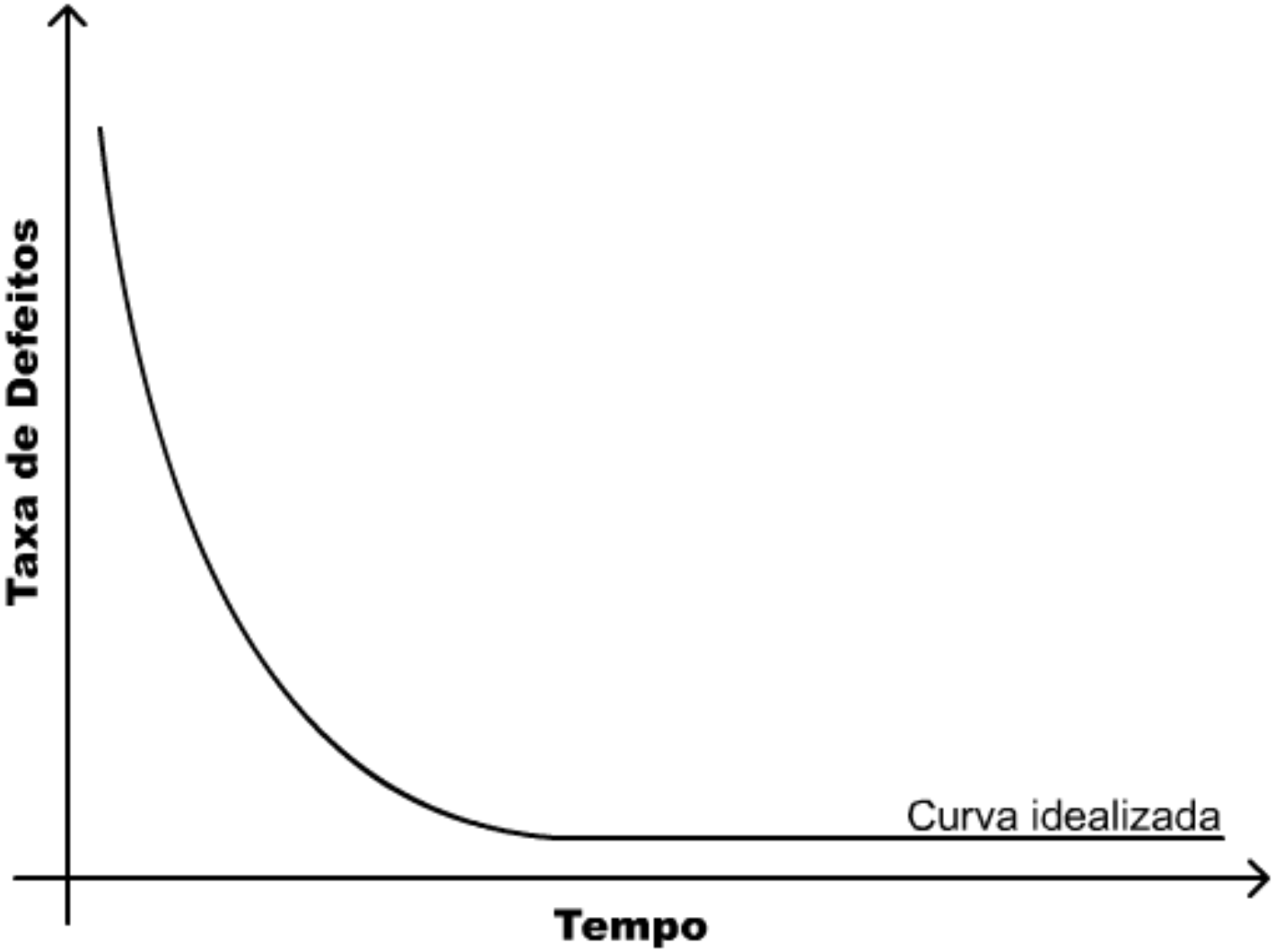
O que é software?

- Um software é composto por (PRESSMAN, 2011):
  - Instruções (programa de computador) que quando executado provê um conjunto de funcionalidades e características desejadas;
  - Estruturas de dados que habilitam os programas a manipularem as informações adequadamente;
  - Documentação que descreve a operação e uso desses programas.

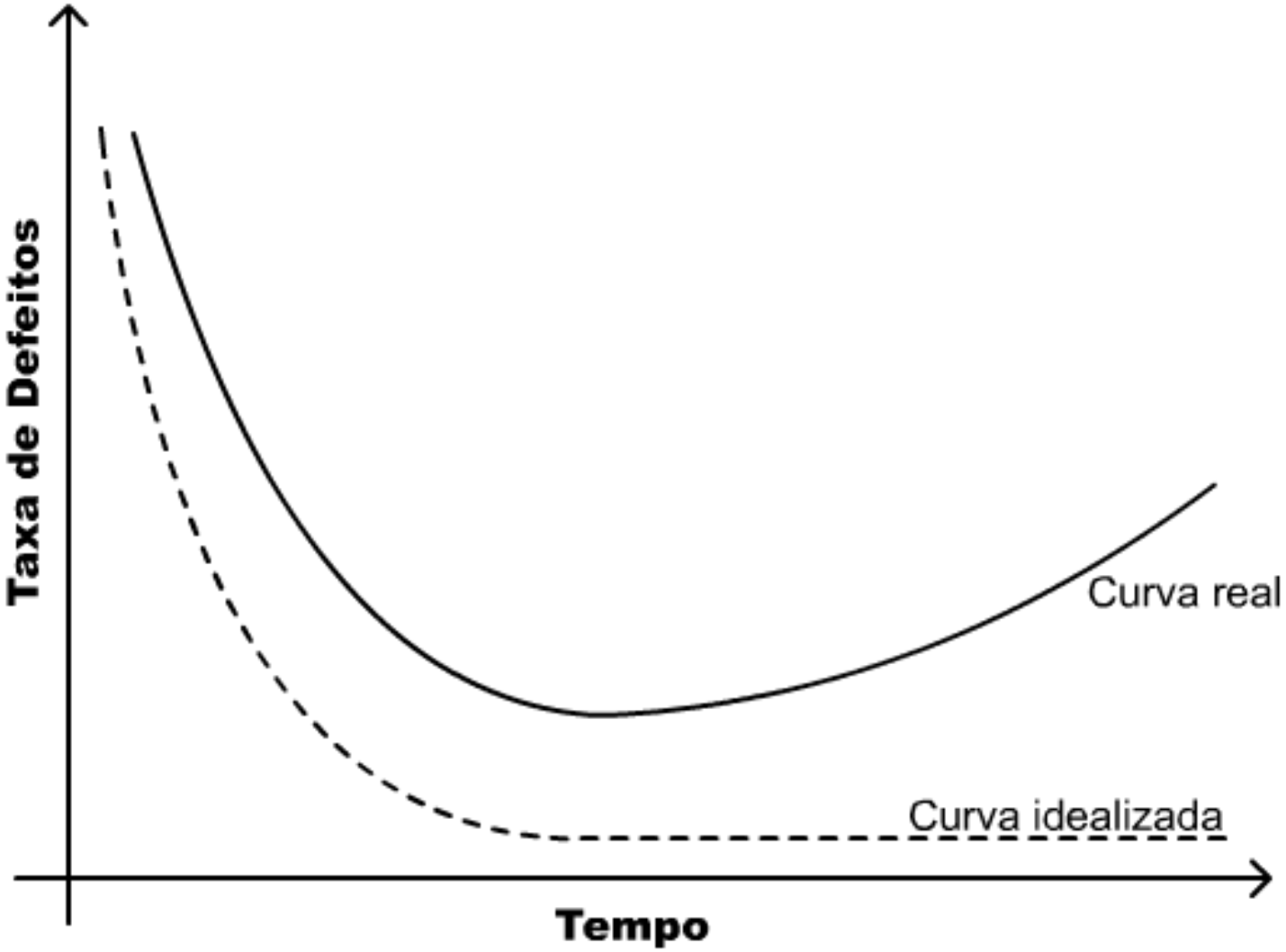
Características do software

- Diferentemente do hardware, o software:
  - É desenvolvido ou passa por um processo de engenharia, mas não é manufaturado (no sentido clássico);
  - Não “se desgasta” (mas se deteriora);
  - Embora a indústria caminhe em direção à construção baseada em componentes, a maioria dos softwares continua a ser construída de forma personalizada (sob encomenda).

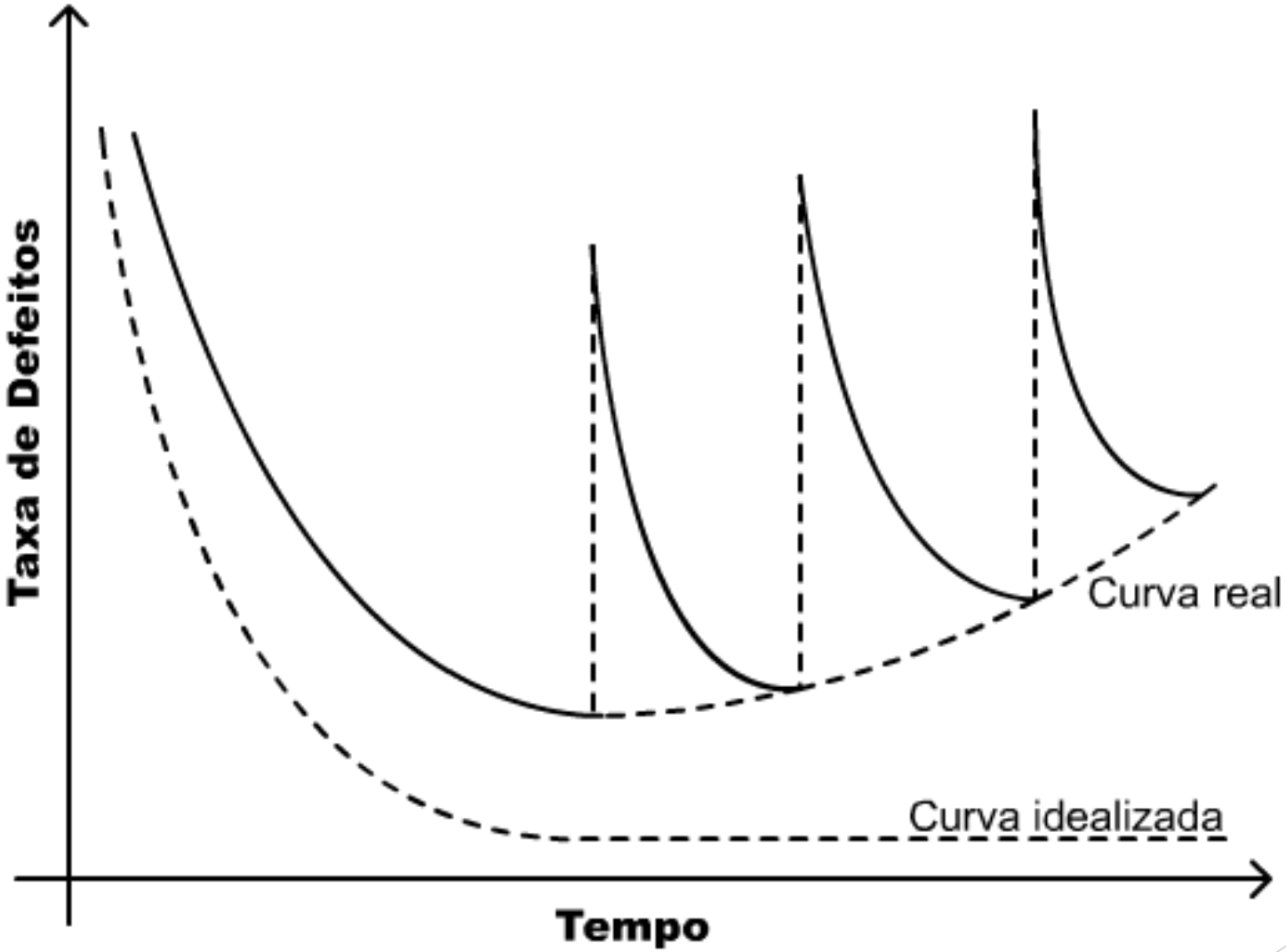
Características do software



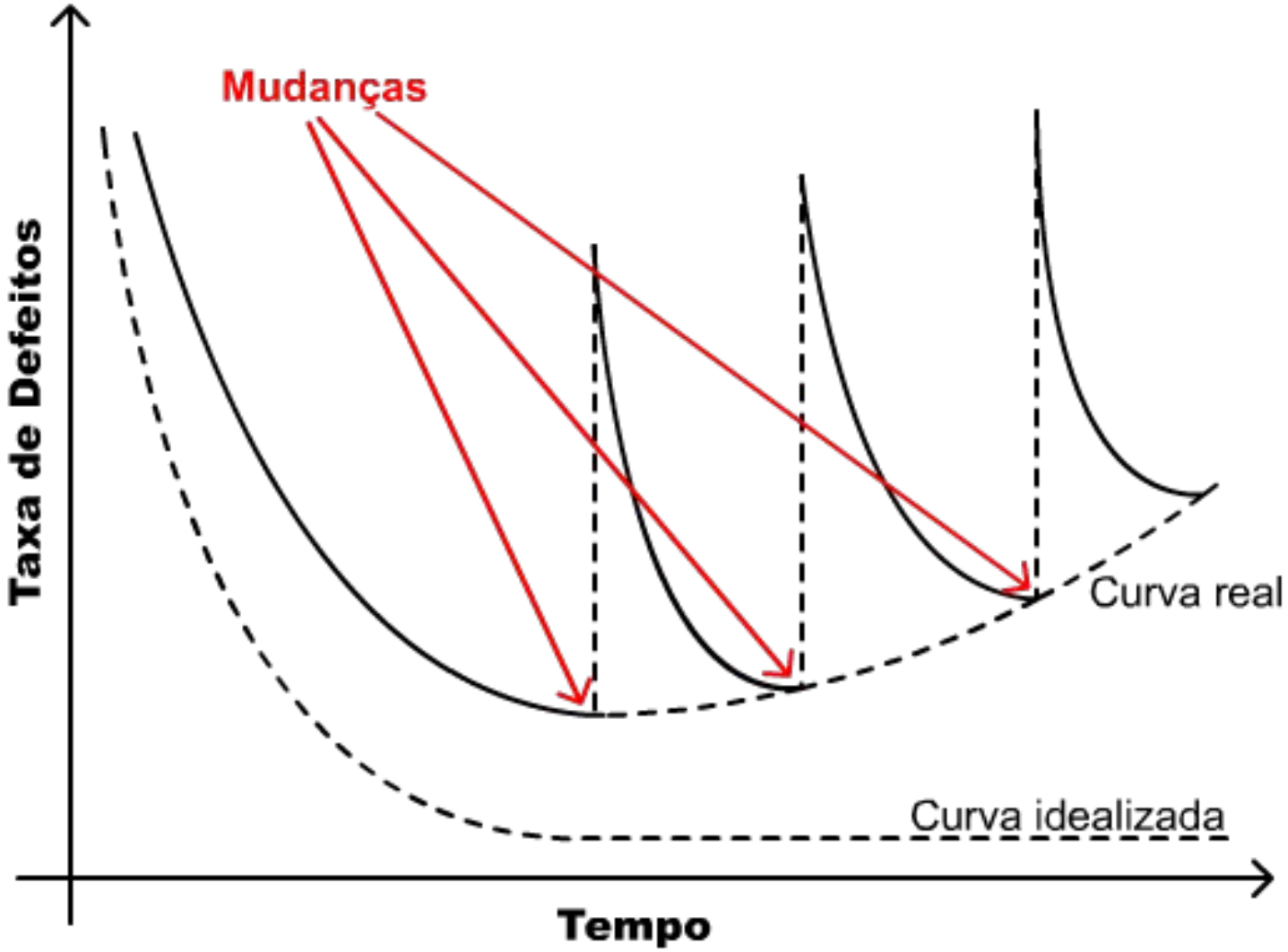
Características do software



Características do software



Características do software



Evolução do software (1950-1965)

- Foco no *hardware* (mudanças contínuas no mesmo). *Hardware* de propósito geral;
- *Software* era uma “arte secundária”, logo havia poucos métodos sistemáticos para o mesmo. *Software* específico para cada aplicação;
- Falta de documentação.

Evolução do software (1965-1975)

- Sistemas multiusuários e multitarefas;
- Sistemas de tempo real;
- 1ª geração de Sistemas Gerenciadores de Bancos de Dados (SGBD);
- Produto de software – software houses;
- Manutenção quase impossível.

... CRISE DO SOFTWARE!

Evolução do software (1975-hoje)

- Sistemas distribuídos;
- Redes locais e globais;
- Uso generalizado dos microprocessadores – produtos inteligentes;
- *Hardware* de baixo custo;
- Crescimento exponencial do consumo de tecnologias.

Evolução do software (quarta era do software de computador)

- Tecnologias orientadas a objetos;
- Inteligência Artificial usada na prática;
- Computação paralela.

Crise do software

- Refere-se a um conjunto de problemas encontrados no desenvolvimento de software;

1. As estimativas de prazo e de custo frequentemente são imprecisas;

   Não dedicamos tempo para coletar dados sobre o processo de desenvolvimento de software. Sem nenhuma indicação sólida da produtividade, não podemos avaliar com precisão a eficácia de novas ferramentas, métodos ou padrões.

Crise do software

2. A produtividade das pessoas da área de software não tem acompanhado a demanda de seus serviços;

   Os projetos de desenvolvimento de software normalmente são efetuados apenas com um vago indício das exigências do cliente.

Crise do software

3. A qualidade de software às vezes é menos que adequada;

   Só recentemente começam a surgir conceitos quantitativos sólidos de garantia de qualidade de software.

Crise do software

4. O software existente é muito difícil de manter;

   Facilidade de manutenção não foi enfatizada como um critério importante, assim tal tarefa torna-se muito dispendiosa.

Crise do software

- ↑ Estimativas de prazo e de custo
- ↓ Produtividade das pessoas
- ↓ Qualidade de software
- ↑ Software difícil de manter

Causas dos problemas associados à Crise do Software

1. O próprio caráter do software
   - O software é um sistema lógico e não físico. Consequentemente o sucesso é medido pela qualidade de **uma única entidade** e não pela qualidade de muitas entidades manufaturadas;

Causas dos problemas associados à Crise do Software

2. Falhas das pessoas responsáveis pelo desenvolvimento
   - Gerentes sem experiência em projetos de software;
   - Desenvolvedores têm recebido pouco treinamento formal em novas técnicas para desenvolvimento;
   - Resistência a mudanças;

Causas dos problemas associados à Crise do Software

3. Mitos relativos ao software.
   - Mitos do gerenciamento;
   - Mitos dos clientes;
   - Mitos dos desenvolvedores.

Mitos relativos ao software

- Mitos do gerenciamento:
  - Já temos um livro cheio de padrões e procedimentos para desenvolver software. Ele não supre meu pessoal com tudo que eles precisam saber?
  - Meu pessoal tem ferramentas de desenvolvimento de software de última geração; afinal lhes compramos os mais novos computadores;
  - Se o cronograma atrasar, poderemos acrescentar mais programadores e ficarmos em dia;
  - Se eu decidir terceirizar o projeto de software, posso simplesmente relaxar e deixar essa empresa realizá-lo.

Mitos relativos ao software

- Mitos dos clientes:
  - Uma definição geral dos objetivos é suficiente para começar a escrever os programas – podemos preencher detalhes posteriormente;
  - Os requisitos de software mudam continuamente, mas as mudanças podem ser facilmente assimiladas, pois o software é flexível.

Mitos relativos ao software

- Mitos dos desenvolvedores:
  - Uma vez feito um programa e colocado em uso, nosso trabalho está terminado;
  - Até que o programa entre em funcionamento, não há maneira de avaliar a qualidade;
  - O único produto passível de entrega é o programa em funcionamento;
  - A Engenharia de Software nos fará criar documentação volumosa e desnecessária e irá nos atrasar.

Custo das mudanças

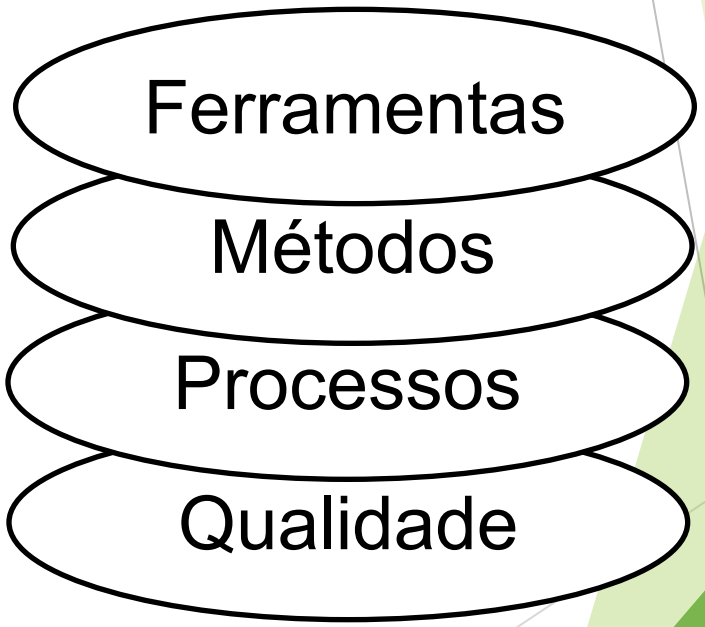
Fase	Custo da mudança
Definição	1x
Desenvolvimento	1.5 – 6x
Manutenção	60-100x

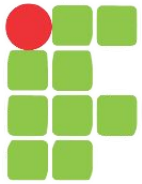
Definição de Engenharia de Software

- Segundo Fritz Bauer (*apud* PRESSMAN, 2011, p. 39):
  - “É o estabelecimento e o emprego de sólidos princípios de engenharia de modo a obter software de maneira econômica, que seja confiável e funcione de forma eficiente em máquinas reais.“
- Segundo a IEEE (*apud* PRESSMAN, 2009, p. 39):
  - "A aplicação de uma abordagem sistemática, disciplinada e quantificável no desenvolvimento, na operação e na manutenção de software; isto é, a aplicação de engenharia ao software."

Camadas da Engenharia de Software

- Qualidade é camada-base da ES e pode ser obtida por meio da satisfação do custo, cronograma e escopo de projetos estabelecidos;
- A fim de alcançá-la, o engenheiro de software emprega processos, métodos e ferramentas;
- Assim, ferramentas auxiliam, mas não substituem as demais camadas!





INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SERGIPE

Exercícios

Introdução à Engenharia de Software

Parte 01

Complete

1. Um dos problemas encontrados na crise do software são as estimativas de _____ e de _____ frequentemente imprecisas.

Complete

2. Um dos mitos do _____ é que se o cronograma do projeto atrasar, basta acrescentar mais programadores para o mesmo ficar em dia novamente.

Complete

3. Ao projetar e construir softwares, o engenheiro de software deve-se atentar ao fato de que o mesmo é composto por _____ , _____ e _____ e entregar algo que satisfaça esses três itens.

Verdadeiro ou Falso

4. ( ) Corrigir um erro ou efetuar uma mudança na fase de manutenção é muito mais barato e rápido do que na fase de definição.

Verdadeiro ou Falso

5. ( ) Por mais longo ou complexo que seja um projeto de software, uma vez iniciado, seus requisitos são imutáveis, de forma que o cliente não pode requerer mudanças ao mesmo.

Complete

6. Segundo o IEEE, _____ é “a aplicação de uma abordagem sistemática, disciplinada e quantificável no desenvolvimento, na operação e na manutenção de software; isto é, a aplicação de engenharia ao software”.

Verdadeiro ou Falso

7. ( ) Uma vez que o software tenha entrado em funcionamento, todo o trabalho da equipe de desenvolvimento estará concluído e não há mais nada a ser feito.

Verdadeiro ou Falso

8. ( ) Durante o desenvolvimento de um software, enquanto o mesmo não esteja concluído e pronto para ser usado, não há meios de avaliar a qualidade.

Complete

9. Do ponto de vista da Engenharia de Software, um projeto de software é considerado um sucesso se o mesmo for concluído dentro do _____ , do _____ e do _____ planejados.

Complete

10. _____ é a camada-base da Engenharia de Software e pode ser alcançada mediante o emprego de _____ , _____ e _____ , que são as camadas seguintes.

Responda

11. Por que se diz que um “software não se desgasta, mas se deteriora”?

Responda

12. Quais as camadas da Engenharia de Software?

SAIBA MAIS!

Categorias de software

- **Software de sistema** – responsáveis por oferecer os recursos básicos disponíveis no computador (gerenciamento de arquivos, gerenciamento de memória, etc.) ao usuário final;

SAIBA MAIS!

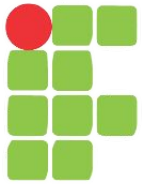
Categorias de software (cont.)

- **Software de aplicação** - são softwares desenvolvidos para cumprimento de tarefas específicas. Podem ser subdivididos em:
  - **Softwares para desktop** – são softwares para notebooks ou computadores com finalidades bastante específicas;
  - **Softwares para web** – tratam-se de todos os aplicativos cujo acesso é feito por meio da Internet, via algum navegador;
  - **Softwares para dispositivos móveis** – comumente chamados *apps*, podem ser executados em smartphones e/ou tablets.

SAIBA MAIS!

Categorias de software (cont.)

- **Software embarcado** - são softwares empregados dentro de máquinas que não são computadores de uso geral e possuem, geralmente, um fim muito específico.



INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SERGIPE

Modelos de Processo

Parte 02

Sumário

- Processo de software
- *Framework* de um processo genérico
- Principais modelos de processo
  - Modelos Cascata e V
  - Modelo Prototipação

Introdução

- Segundo Howard Baetjer Jr. (*apud* PRESSMAN, 2011, p. 53):

  Pelo fato de software, como todo capital, ser conhecimento incorporado, e pelo fato de esse conhecimento ser, inicialmente, disperso, tácito, latente e em considerável medida, incompleto, o desenvolvimento de software é um processo de aprendizado social. [...] Trata-se de um processo iterativo no qual a própria ferramenta em evolução serve como meio de comunicação, com cada nova rodada do diálogo extraindo mais conhecimento útil das pessoas envolvidas.

Processo de Software

- **Processo** é um conjunto de atividades, ações e tarefas realizadas na criação de algum produto de trabalho;
- Uma **atividade** esforça-se para atingir um objetivo amplo (por exemplo, modelagem do sistema);
- Uma **ação** (por exemplo, projeto de arquitetura) envolve um conjunto de tarefas que resultam num artefato de software fundamental (por exemplo, um modelo de projeto de arquitetura);
- Uma **tarefa** se concentra em um objetivo pequeno, porém, bem definido (por exemplo, elaborar diagrama de classes) e produz um resultado tangível.

Framework de um processo genérico

- Uma metodologia (*framework*) identifica um pequeno número de **atividades metodológicas** aplicáveis a todos os projetos de software;
- Engloba também um conjunto de **atividades de apoio** (*umbrella activities*) aplicáveis em todo o processo de software.

Framework de um processo genérico

ATIVIDADES METODOLÓGICAS	ATIVIDADES DE APOIO

► Comunicação;
► Planejamento;
► Modelagem;
► Construção;
  ► Geração de código;
  ► Teste;
► Emprego.
	► Controle e acompanhamento do projeto;
	► Gerenciamento dos riscos;
	► Garantia da qualidade do software;
	► Revisões técnicas;
	► Medições;
	► Gerenciamento da configuração do software;
	► Gerenciamento da reusabilidade;
	► Preparo e produção dos artefatos de software.

Modelos de Processo

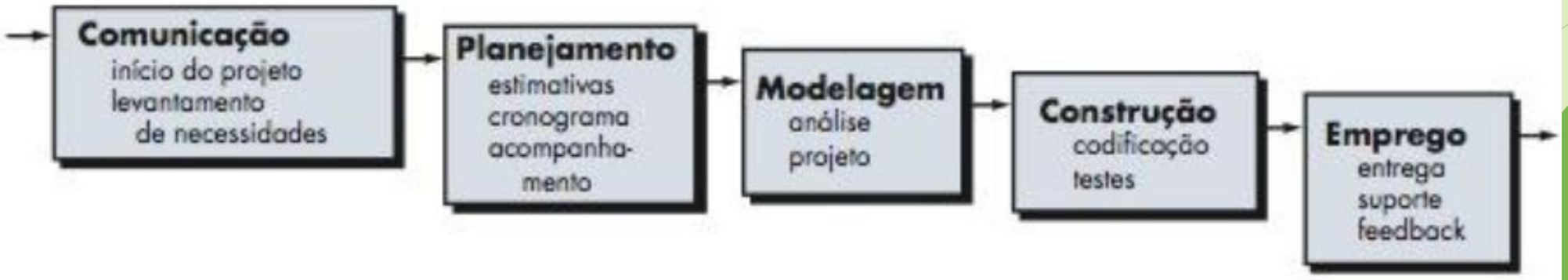
- Modelos de Processos Prescritivos
  - Modelos Sequenciais: Cascata e o Modelo V
  - Modelos Incrementais: Modelo Incremental, Desenvolvimento Rápido de Aplicações (RAD) e Processo Unificado
  - Modelos Evolucionários – Prototipação e Modelo Espiral
  - Modelos Concorrentes
- Modelos de Processos Ágeis
  - Programação Extrema (XP)
  - Scrum
  - Desenvolvimento Dirigido a Testes
- Modelos de Processos Especializados
  - Desenvolvimento Baseado em Componentes
  - Modelo de Métodos Formais
  - Desenvolvimento de Software Orientado a Aspectos
  - Desenvolvimento Dirigido por Modelos

Modelos de Processo Prescritivo

- Promovem uma abordagem ordenada e estruturada da Engenharia de Software;
- São prescritivos porque prescrevem um conjunto de elementos de processo – atividades metodológicas, tarefas, produtos de trabalho etc. – e um fluxo de processo (fluxo de trabalho).

Modelo Cascata

- Também conhecido como Ciclo de Vida Clássico;
- Fluxo de processo é linear:
  - Útil quando os requisitos de um problema são bem compreendidos – o trabalho flui da **comunicação** ao **emprego** de forma bastante linear.

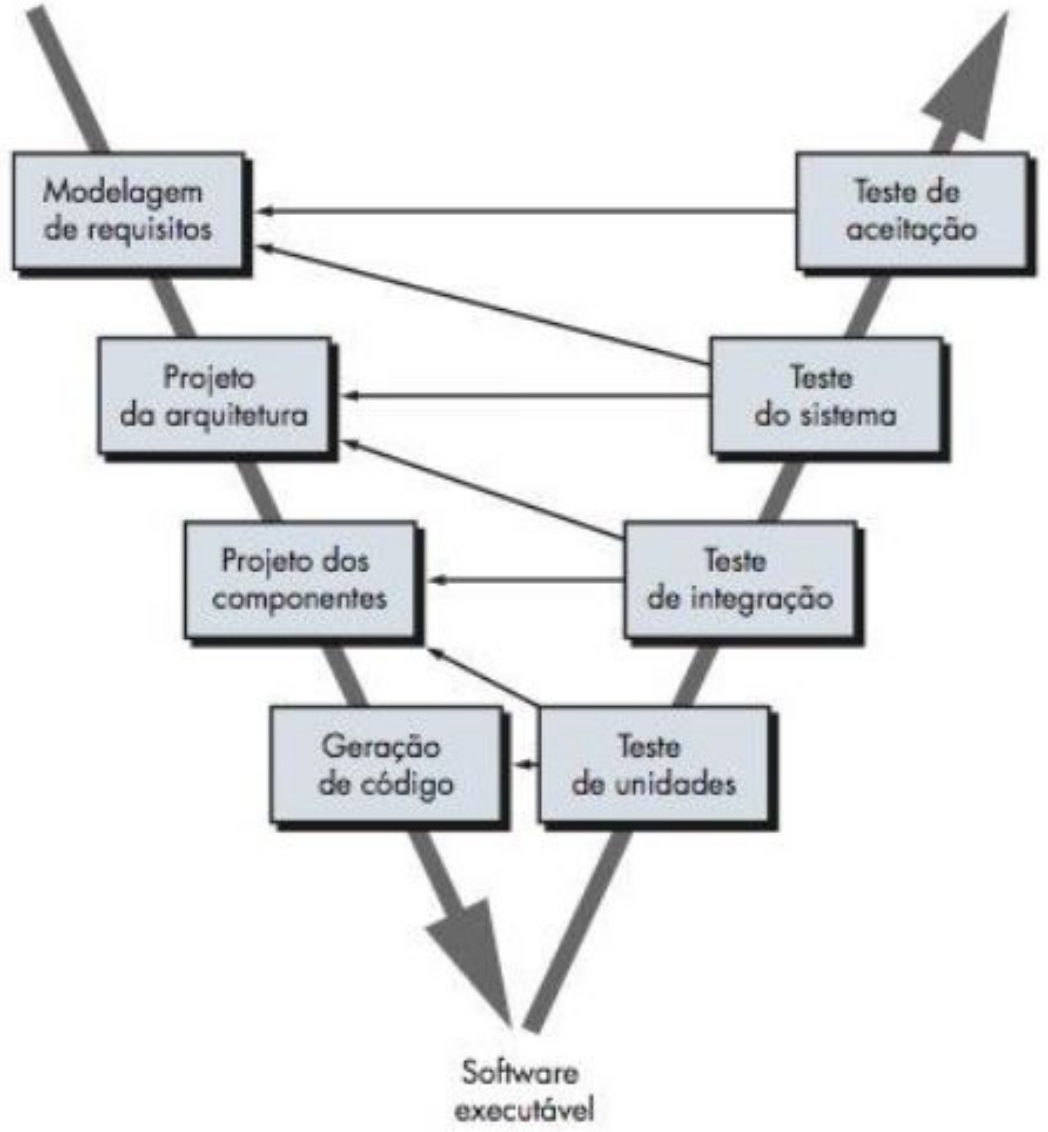


Modelo Cascata

- Problemas do modelo cascata (PRESSMAN, 2011, p. 61):
  - *Projetos reais raramente seguem o fluxo sequencial que o modelo propõe. Embora o modelo linear possa conter iterações, ele o faz indiretamente, podendo provocar confusão à medida que a equipe de projeto prossegue;*
  - *Frequentemente, é difícil para o cliente estabelecer explicitamente todas as necessidades no início do projeto, algo essencial no modelo cascata;*
  - *Uma versão operacional do(s) programa(s) não estará disponível antes de estarmos próximo do final do projeto. Um erro grave, se não detectado até o programa operacional ser revisto, pode ser desastroso.*

Modelo V

- É uma variação do modelo cascata;
- Relaciona ações de garantia de qualidade e as ações associadas à comunicação, modelagem e construção;
- Fornece uma forma de visualizar como verificação e validação serão aplicadas:
  - À medida que a equipe de software “desce” em direção ao lado esquerdo do V, os requisitos básicos do problema são refinados;
  - Uma vez que o código tenha sido gerado, a equipe “se desloca para cima” no lado direito do V, realizando uma série de testes que validam cada um dos modelos criados.



Modelo V

- Problemas do modelo V:
  - O cliente ainda só recebe a primeira versão do software no final do ciclo;
  - Apresenta menor risco que o modelo cascata, porém erros identificados próximos do final do ciclo ainda podem ter impacto significativo.

Modelos Evolucionários

- Softwares podem evoluir ao longo do tempo – conforme desenvolvimento avança ou necessidades do negócio e do produto mudam – assim, detalhes de extensões do produto ou do sistema ainda devem ser definidos;
- Tais modelos de processo permitem desenvolver um produto que evoluirá ao longo do tempo;
- Trata-se de um modelo iterativo;
- Tipos de modelo evolucionário:
  - Prototipação;
  - Modelo espiral.

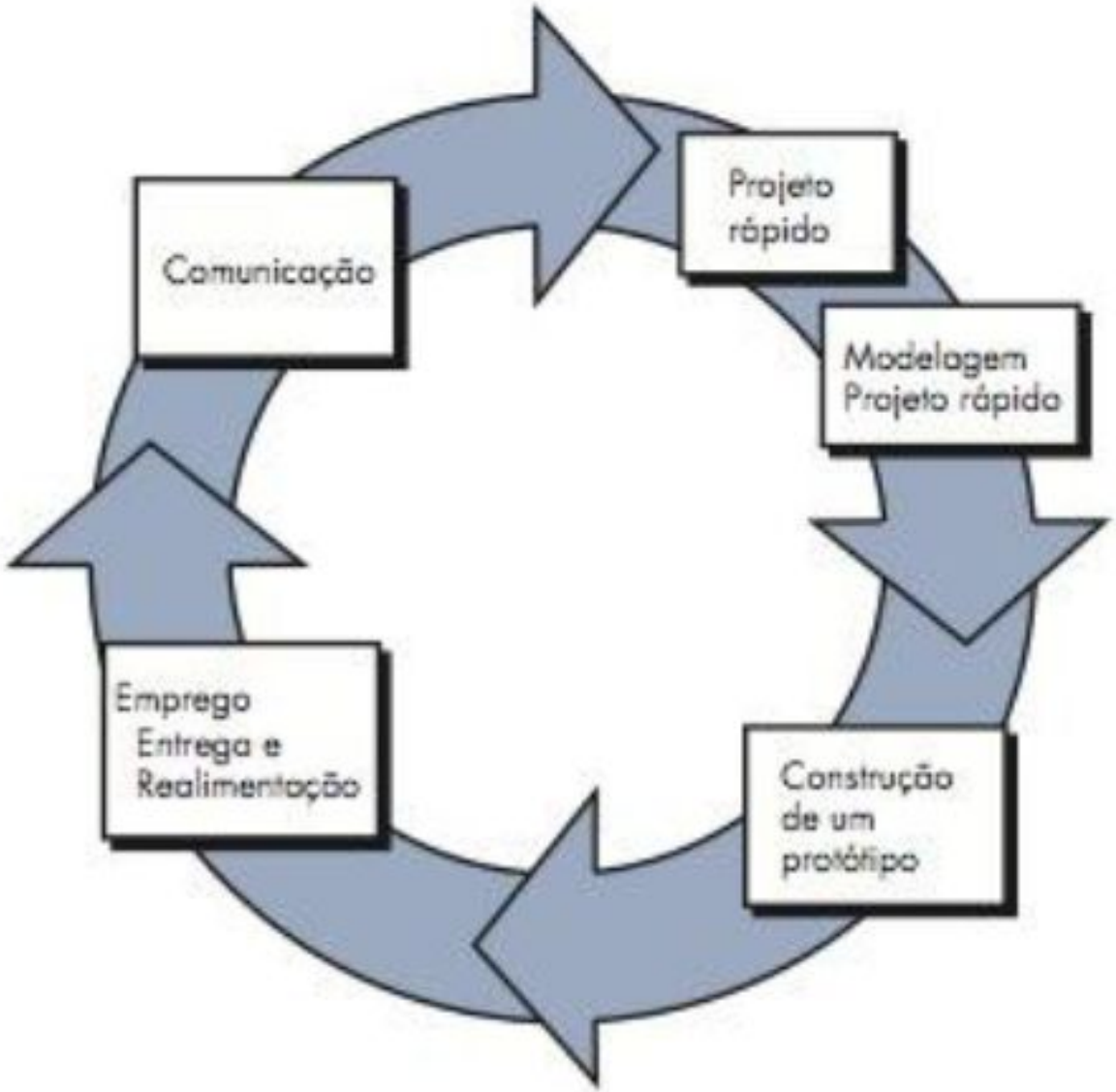
Prototipação

- Possíveis cenários:
  - Cliente define objetivos gerais para o software, mas não detalha os requisitos para funções e recursos;
  - Desenvolvedor encontra-se inseguro quanto à eficiência de um algoritmo, forma que ocorre a interação homem-máquina ou detalhes de uma funcionalidade.

Prototipação

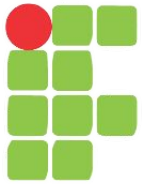
- Prototipação ou prototipagem permite a criação de protótipos (versões parciais descartáveis de um software) que serão apresentados e validados junto ao cliente ou usuário;
- Protótipos descartáveis – são descartados após a validação;
- Protótipos evolutivos – não são descartados e, aos poucos, evoluem até a versão final do software.

Prototipação



Prototipação

- Problemas:
  - Cliente vê a versão (protótipo) em funcionamento e exige alguns acertos para colocar logo em uso;
  - Codificação utilizada para apresentar o protótipo pode ser usada na versão definitiva, mesmo não sendo a mais apropriada;
  - Protótipo pode ser visto como perda de tempo para o cliente.



INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SERGIPE

Exercícios

Modelos de Processo

Parte 02

Complete

1. _____ é um conjunto de atividades, ações e tarefas realizadas na criação de algum produto de trabalho.

Complete

2. As atividades de um *framework* para um processo genérico podem ser divididas em dois grandes grupos: _____ e _____ .

Complete

3. A atividade de construção pode ser subdividida em duas importantes componentes: _____ e _____ .

Complete

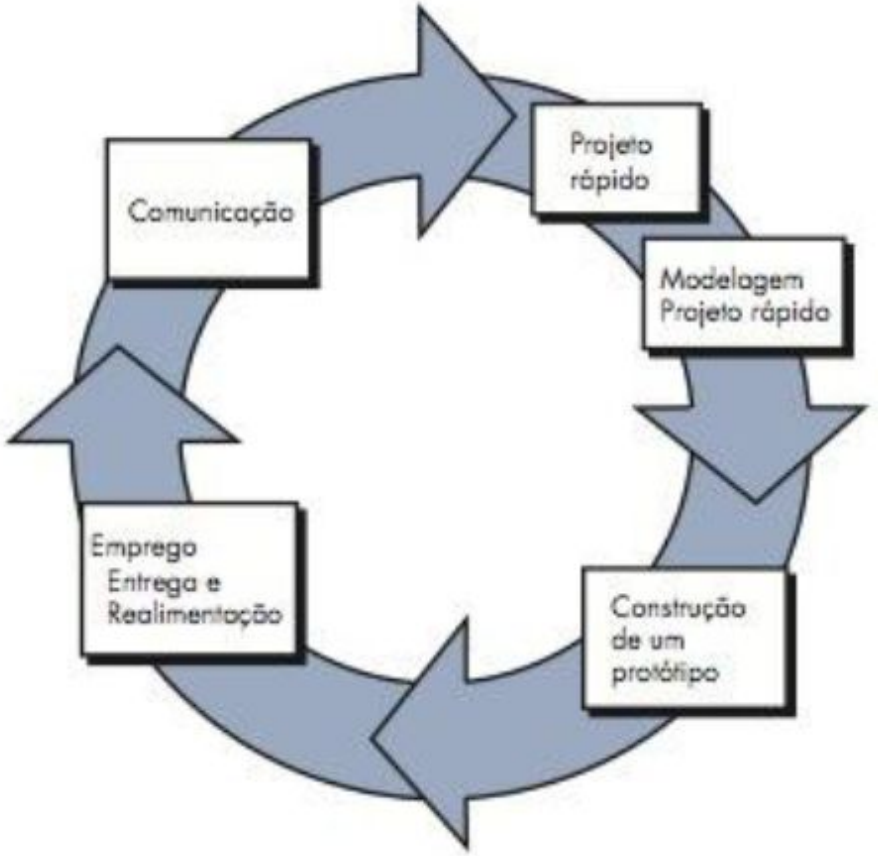
4. Gerenciamento dos riscos, gerenciamento da reusabilidade e garantia da qualidade de softwares são consideradas atividades de _____ .

Responda

5. Defina as atividades metodológicas de um *framework* de processo genérico.

Complete

6. A figura abaixo ilustra o modelo _____ .



Complete

7. Na Duarte Company, todo software é desenvolvido em fases bem sequenciais: inicia-se com todas as tarefas relacionadas à comunicação, passando-se depois para o planejamento, modelagem, codificação e, por fim, a entrega do mesmo para o cliente.

Percebe-se a adoção do modelo _____.

Complete

8. Um problema da _____ é que o _____ pode ser visto como perda de tempo para o cliente.

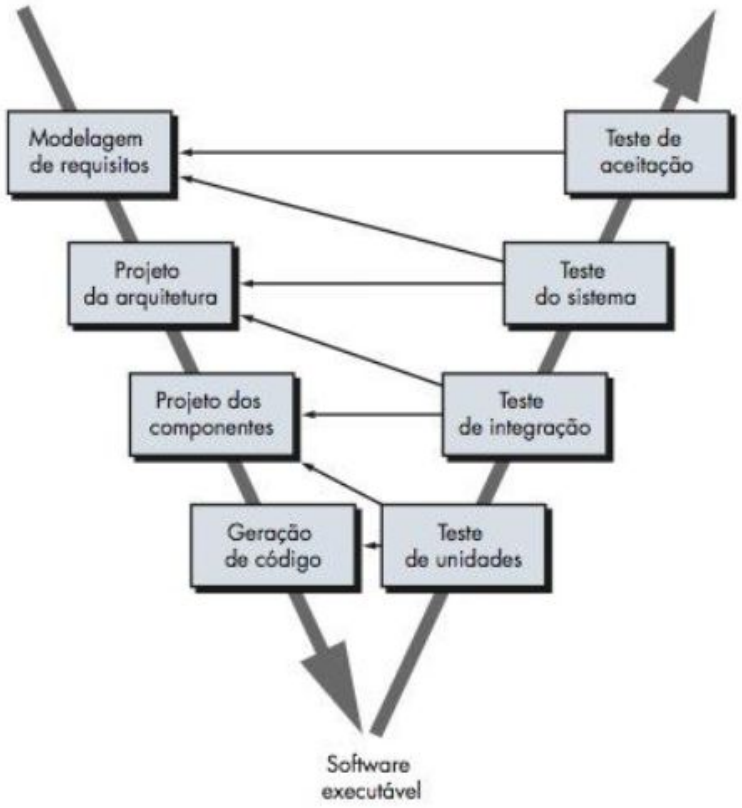
Complete

9. Laerton e sua equipe de desenvolvimento iniciou um projeto de software focando a criação de protótipos descartáveis como forma de acelerar o desenvolvimento e garantir o cumprimento das necessidades do cliente.

Em outras palavras, eles optaram pelo modelo da _____ .

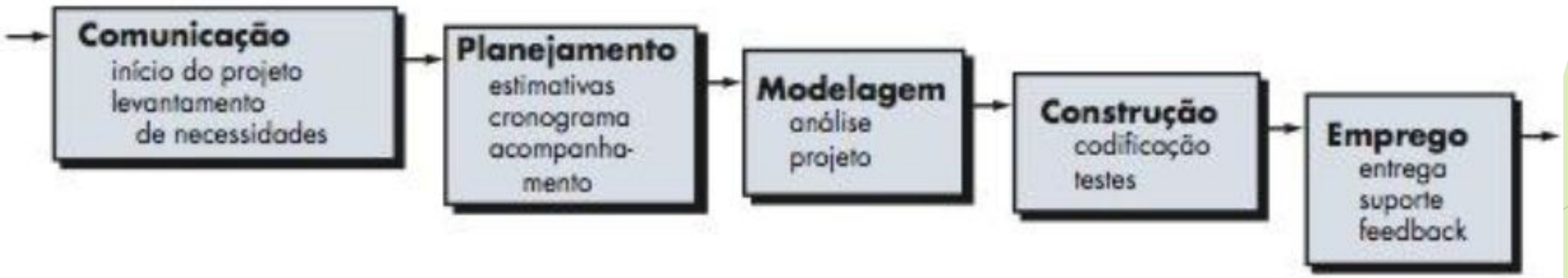
Complete

10. A figura abaixo ilustra o modelo _____ .



Complete

11. A figura abaixo ilustra o modelo _____ .



Responda

12. O que distingue o modelo cascata do modelo V?

Responda

13. Por que o modelo cascata não é amplamente adotado?

SAIBA MAIS!

Framework de um processo genérico

- As cinco atividades metodológicas:
  1. **Comunicação** – lida com as tarefas relacionadas à comunicação inicial e/ou continuada com os diversos interessados do projeto com o intuito de melhor conhecer o problema a ser resolvido bem como validar a solução proposta;

SAIBA MAIS!

Framework de um processo genérico

- As cinco atividades metodológicas:
  2. **Planejamento** – engloba as tarefas relacionadas à elaboração de cronograma, orçamento e plano de desenvolvimento do projeto de software;

SAIBA MAIS!

Framework de um processo genérico

- As cinco atividades metodológicas:
  3. **Modelagem** – é a atividade composta por tarefas referentes à análise e especificação dos requisitos e do projeto por meio de modelos, isto é, representações simplificadas da estrutura ou comportamento do software a ser desenvolvido;

SAIBA MAIS!

Framework de um processo genérico

- As cinco atividades metodológicas:
  4. **Construção** – lida diretamente com as tarefas referentes ao desenvolvimento do software em si, subdivididas aqui em tarefas referentes à geração de código (codificação) e testes;

SAIBA MAIS!

Framework de um processo genérico

- As cinco atividades metodológicas:
  5. **Emprego** – trata da implantação, treinamento e entrega do software.

SAIBA MAIS!

Modelo Espiral

- Acopla a natureza iterativa da prototipação com os aspectos sistemáticos e controlados do modelo cascata;
- Modelos ou protótipos podem ser construídos nas primeiras iterações, mas posteriormente são produzidas versões cada vez mais completas do sistema.

SAIBA MAIS!

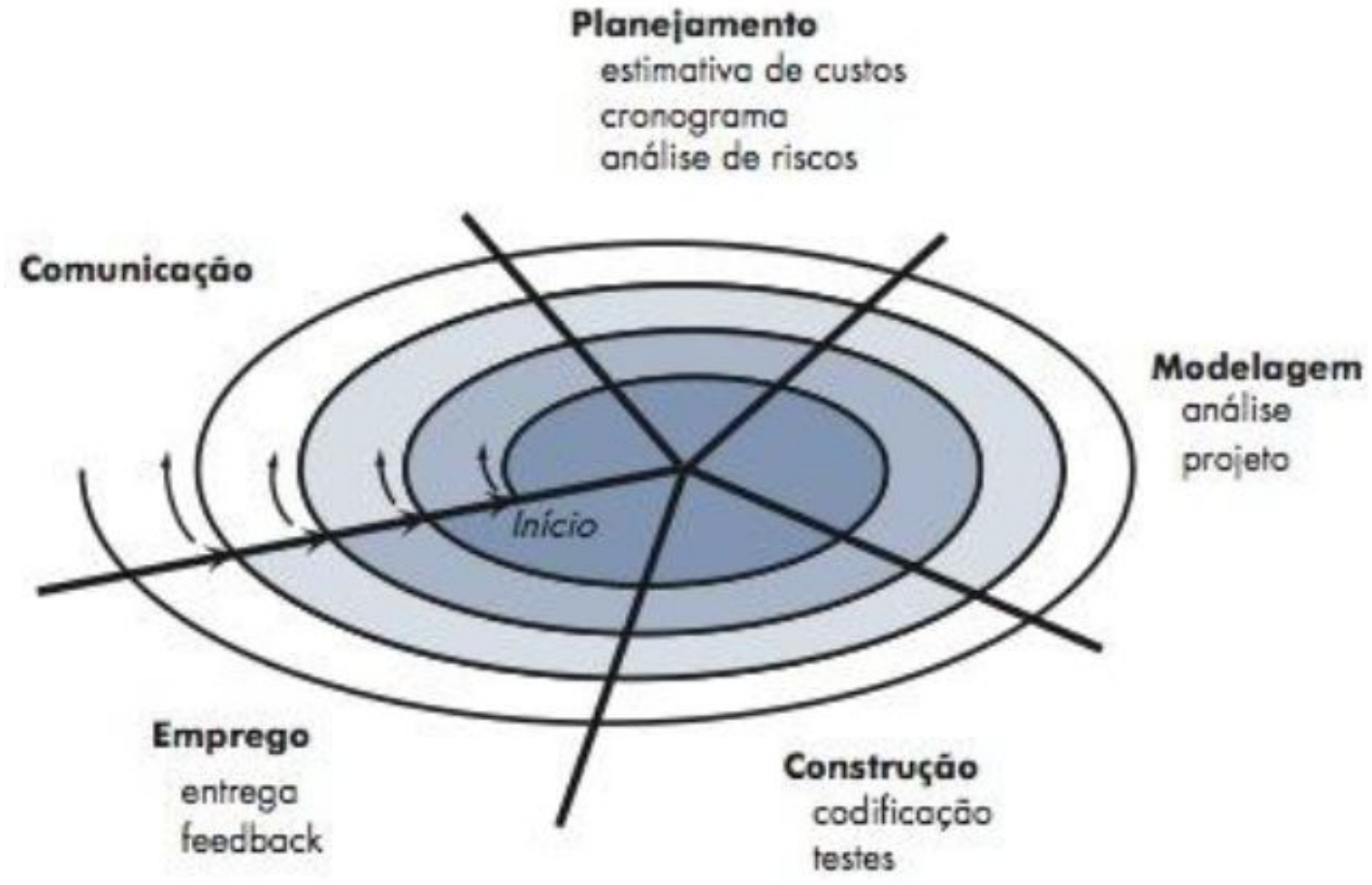
Modelo Espiral

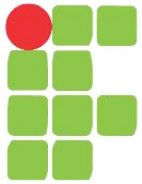
- Segundo Boehm (*apud* PRESSMAN, 2011, p. 65):

  O modelo espiral [...] possui duas características principais que o distinguem. A primeira consiste em uma abordagem cíclica voltada para ampliar, incrementalmente, o grau de definição e a implementação de um sistema, enquanto diminui o grau de risco do mesmo. A segunda característica consiste em uma série de pontos âncora de controle para assegurar o comprometimento de interessados quanto à busca de soluções de sistema que sejam mutuamente satisfatórias e praticáveis.

SAIBA MAIS!

Modelo Espiral





INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SERGIPE

Desenvolvimento Ágil

Parte 03

Sumário

- Deficiência dos modelos de processo prescritivos
- Origem do Desenvolvimento Ágil
- Filosofia do Desenvolvimento Ágil
- Características do processo ágil
- Fatores humanos no Desenvolvimento Ágil
- Programação Extrema – XP
- Outros modelos de processos ágeis

Deficiência dos modelos de processo prescritivos

- Requerem muita disciplina para o sucesso em seu uso;
- Entretanto, engenheiros de software podem não ter toda a disciplina necessária para tal;
- Além disso, vivemos em um mundo repleto de mudanças:
  - No software em desenvolvimento;
  - Na equipe;
  - Nas tecnologias utilizadas.
- E modelos prescritivos apresentam maior dificuldade em responder a tais mudanças.

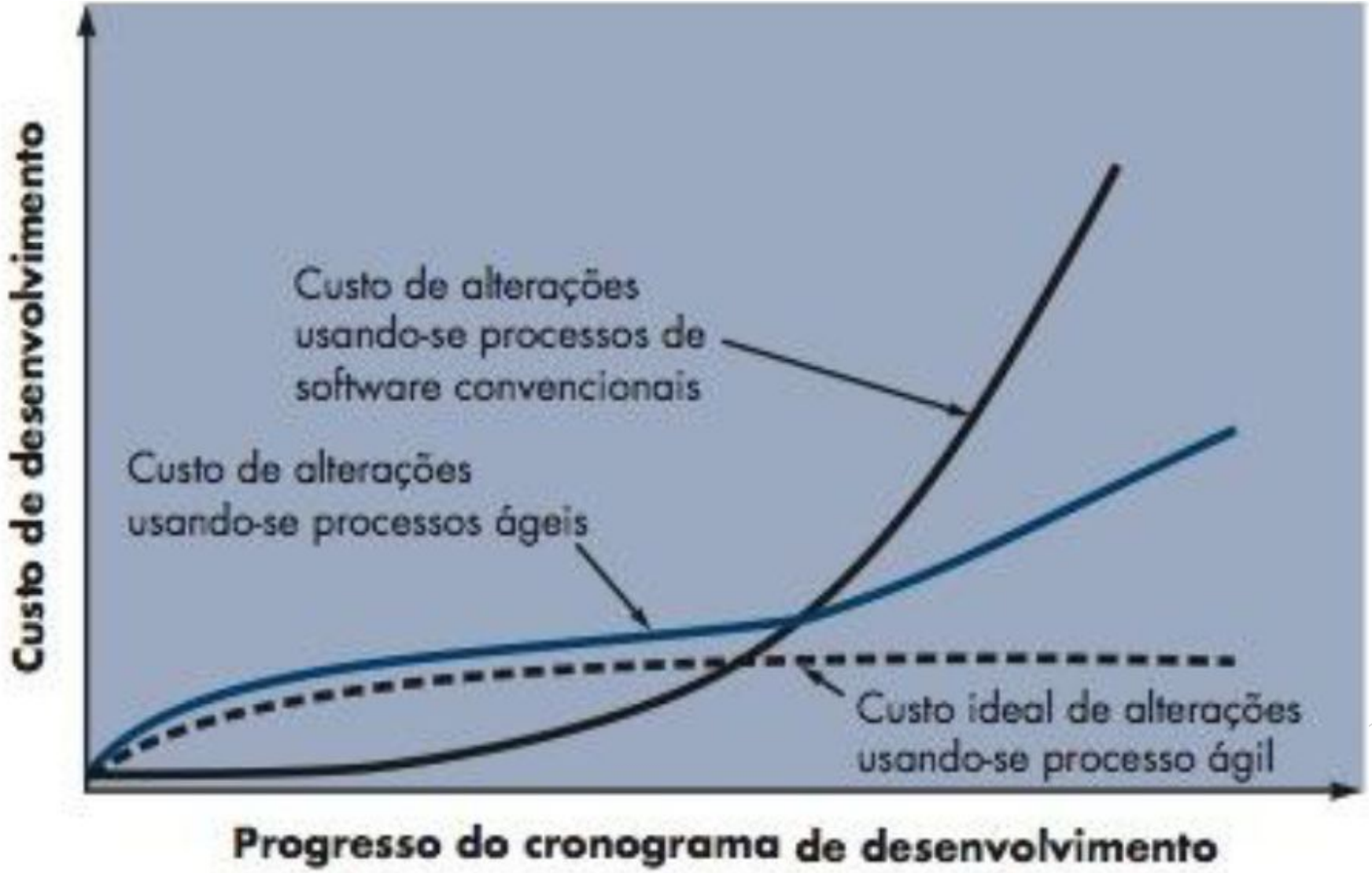
Origem do Desenvolvimento Ágil

- Manifesto para o Desenvolvimento Ágil de software (2001), que valoriza:
  - Indivíduos e interações acima de processos e ferramentas;
  - Software operacional acima de documentação completa;
  - Colaboração dos clientes acima de negociação contratual;
  - Respostas a mudanças acima de seguir um plano.
- Embora haja valor no que está relacionado à direita, prioriza-se o que é exposto à esquerda!
- Por ágil, deve-se entender “capaz de responder de forma rápida e apropriada a mudanças”!

Filosofia do Desenvolvimento Ágil

- Satisfação do cliente por meio de entregas incrementais;
- Equipes de projetos pequenas e bem motivadas;
- Simplicidade no desenvolvimento reduzindo a quantidade de artefatos de Engenharia de Software produzidos;
- Comunicação contínua entre todos os envolvidos;
- Emprega as mesmas atividades metodológicas no processo, porém com foco na entrega.

Custo da mudança em função do tempo



Características de um processo ágil

- **Adaptável** – equipe de desenvolvimento pode adaptá-lo segundo suas necessidades;
- **Incremental** – entrega de versões incrementais do software, importante para validação junto ao cliente;
- **Iterativo** – desenvolvimento se dá por meio de iterações, assim as primeiras iterações não precisam lidar com toda a complexidade do software;
- **Focado na comunicação** – maior foco na comunicação contínua, em vez de ter foco na execução segundo um contrato ou plano rígido.

Programação Extrema – XP (Extreme Programming)

- Uma abordagem para desenvolvimento ágil;
- “Ajusta-se bem a equipes pequenas e médias em desenvolvimento de software com requisitos vagos e em constante mudança. Para isso, adota a estratégia de constante acompanhamento e realização de vários pequenos ajustes durante o desenvolvimento de software.” (Wikipédia, 2019)

Valores da XP

1. **Comunicação**, por meio de:
   - Comunicação estreita e informal entre clientes e desenvolvedores;
   - Metáforas (histórias sobre como o sistema funciona) para comunicar conceitos importantes;
   - *Feedback* contínuo;
   - Evitar excesso de documentação na comunicação.

Valores da XP

2. **Simplicidade**, por meio de:
   - Foco apenas nas necessidades imediatas, ignorando as futuras;
   - Se necessitar de melhorias, o projeto pode ser refatorado (retrabalhado).

Valores da XP

3. **Feedback**, por meio:
   - Do próprio software implementado, a partir de resultados de testes;
   - Do cliente, a partir de testes de aceitação considerando as histórias de usuário ou casos de uso;
   - De outros membros da equipe, referente ao impacto nos custos e no cronograma que certas mudanças podem causar.

Valores da XP

4. **Coragem / disciplina**, a fim de “projetar para hoje”, uma vez que necessidades futuras podem sofrer alterações drásticas em qualquer momento, levando a retrabalho;

5. **Respeito**, conforme as versões incrementais do software são entregues com sucesso.

Processo XP

- Adota principalmente a Orientação a Objetos;
- Envolve quatro atividades metodológicas:
  - Planejamento;
  - Projeto;
  - Codificação;
  - Testes.

Processo XP

1. **Planejamento**
   - Inicia com a elicitação (levantamento) de requisitos, a fim de compreender o ambiente de negócios em que o *software* executará e funcionalidades esperadas;
   - Conduz o desenvolvimento de *user stories* (histórias de usuários) descrevendo cada funcionalidade requisitada;
   - Para cada *user story*, o cliente atribui um valor (prioridade) e a equipe de desenvolvimento, um custo (tempo para implementar aquela funcionalidade);
     - Caso uma história demore mais que três semanas, o cliente deve dividi-la em histórias menores.

User story (História de usuário)

- Um dos artefatos primários no desenvolvimento em Scrum ou XP;
- É uma definição de alto nível de um requisito, contendo apenas informação suficiente para que desenvolvedores possam definir uma estimativa do esforço (tempo) para implementá-la;
- É escrita em cartão de 8 cm x 13 cm.

User story (História de usuário)

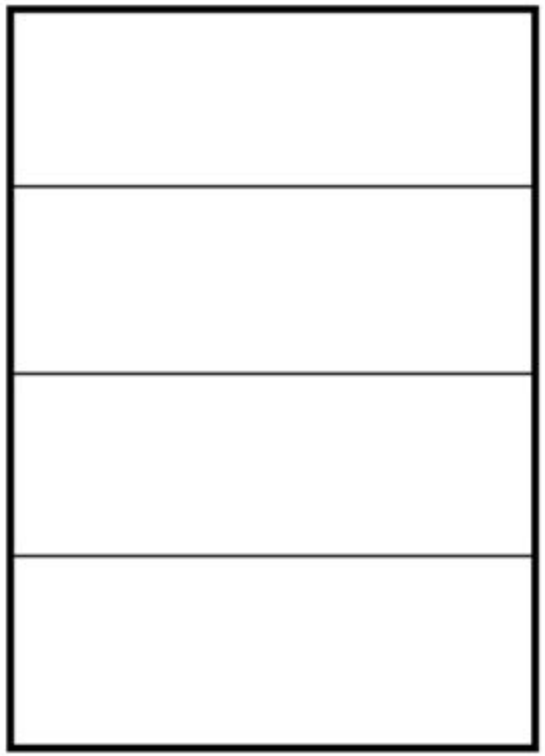
- Geralmente assume o formato:

  Como um(a) <persona>, [quando <evento>] eu quero <ação> [tal que <resultado esperado>].

- Exemplos:
  - Como um administrador, eu quero cadastrar novos funcionários;
  - Como um leitor, quando eu atrasar a devolução de um livro, eu quero receber um e-mail notificando isso.

Vamos praticar!

Agora, elabore 4 histórias de usuário para um sistema de biblioteca!



Exemplos:

- ✔ Como um administrador, eu quero cadastrar novos funcionários;
- ✔ Como um leitor, quando eu atrasar a devolução de um livro, eu quero receber um e-mail notificando isso.

Processo XP

2. **Projeto**
   - Segue o princípio KISS (*keep it simple, stupid*);
   - Oferece um guia para implementação das *user stories* na medida em que são escritas;
   - Encoraja-se o uso de cartões CRC (classe-responsabilidade-colaborador);
   - Desenvolvimento de soluções pontuais:
     - Caso um difícil problema de projeto (a partir de uma *user story*) seja encontrado, recomenda-se implementar e avaliar um protótipo.

Cartão CRC

- O modelo CRC é uma coleção de cartões (8 cm x 13 cm) divididos em três seções (Classe, Responsabilidades e Colaboradores);
  - Uma **classe** é uma coleção de objetos similares (ex: coleção de livros, usuários, empréstimos etc.);
  - Uma **responsabilidade** é algo que uma classe sabe ou faz;
  - Um **colaborador** é outra classe com quem nossa classe interage a fim de cumprir suas responsabilidades.

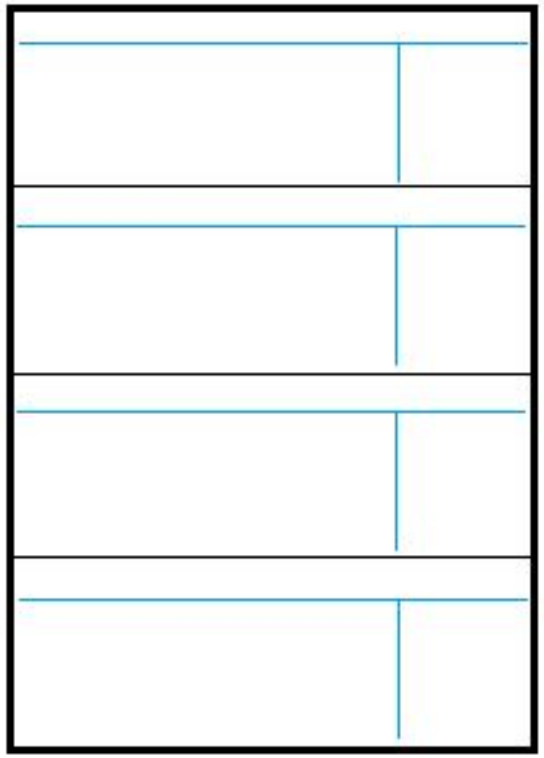
Cartão CRC

- Exemplo de Cartão CRC:

Classe Leitor	
Responsabilidades Faz empréstimo Sabe nome Sabe endereço Sabe telefone Sabe histórico de empréstimos	**Colaboradores** Empréstimo

Vamos praticar!

Agora, elabore 4 cartões CRC para um sistema de biblioteca!



Classe Consumidor	
Responsabilidades Faz pedidos Sabe nome Sabe endereço Sabe telefone Sabe histórico de pedidos	**Colaboradores** Pedido

Processo XP

3. **Codificação**
   - Inicia com o desenvolvimento de uma série de testes de unidades para cada história que será implementada no incremento atual;
   - Após a implementação (seguindo o princípio KISS), o código é testado em unidade;
   - Programação em pares (duplas). Assim, o código é revisto na medida em que é criado;
   - Integração contínua do código produzido por todas as duplas.

Processo XP

4. **Testes**
   - Testes de unidade são definidos antes da codificação;
   - Foco na automatização dos testes de unidade;
   - Encorajamento de testes de regressão;
   - Testes de aceitação (testes de cliente) são obtidos das histórias de usuário e possuem como foco as características do sistema que o cliente pode revisar.

Exemplo de teste de unidade automatizado

```
public class MathExt {
        public static void factorial(int x) {
         if (x < 0) {
                    return 0;
               } else if (x == 0) {
                    return 1;
               } else {
                    return x * factorial(x-1);
               }
        }
}
```

Classe a ser testada

```
public class Tester {
        public static void main(String[] args) {
               if (MathExt.factorial(-1) != 0) {
                      Logger.log("MathExt.factorial(-1) failed!");
               }
               if (MathExt.factorial(0) != 1) {
                      Logger.log("MathExt.factorial(0) failed!");
               }
               if (MathExt.factorial(5) != 120) {
                      Logger.log("MathExt.factorial(5) failed!");
               }
                Logger.log("Test is finished.");
        }
}
```

Classe testadora

Processo XP


projeto simples
cartões CRC
soluções pontuais
protótipos
valores das histórias
de usuários
critérios de teste de aceitação
plano de iteração
projeto
planejamento
codificação
refabricação
teste
programação em dupla
Versão
incremento de software
velocidade de projeto registrado
(computada)
teste de unidades
integração contínua
teste de aceitação

Críticas à abordagem XP

- **Volatilidade de requisitos**. Como o cliente é um membro ativo na XP, alterações podem ser solicitadas informalmente, levando a possíveis mudanças de escopo;

Críticas à abordagem XP

- **Necessidades conflitantes de clientes**. Em XP, a equipe de desenvolvimento deve assimilar as necessidades de múltiplos clientes, muitas vezes conflitantes;

Críticas à abordagem XP

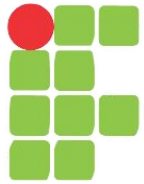
- **Requisitos elicitados informalmente**. As histórias de usuários são a única manifestação explícita dos requisitos, não havendo assim um modelo mais formal para evitar omissões e inconsistências;

Críticas à abordagem XP

- **Falta de projeto formal.** Sistemas complexos podem requerer maior elaboração do projeto para garantir qualidade e facilidade de manutenção do software, mas a XP não enfatiza a necessidade do projeto da arquitetura.

Outros modelos de processos ágeis

1. Scrum;
2. Desenvolvimento de Software Adaptativo;
3. Método de Desenvolvimento de Sistemas Dinâmicos;
4. Crystal;
5. Desenvolvimento Dirigido a Funcionalidades;
6. Desenvolvimento de Software Enxuto;
7. Modelagem Ágil;
8. Processo Unificado Ágil.



INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SERGIPE

Exercícios

Desenvolvimento Ágil

Parte 03

Complete

1. Por ágil, deve-se entender “ser capaz de _____ de forma rápida e apropriada a _____ ” .

Múltipla Escolha

2. Marque com um X as opções que fazem parte da filosofia do desenvolvimento ágil:

a ( ) Obediência à execução sequencial das atividades metodológicas;

b ( ) Satisfação do cliente por meio de entregas incrementais;

c ( ) Equipes de projetos pequenas e bem motivadas;

d ( ) Eliminação de toda a documentação;

e ( ) Comunicação contínua entre todos os envolvidos.

Complete

3. Modelos de processo _____ requerem muita disciplina, entretanto engenheiros de software podem não ter toda a disciplina necessária.

Já os modelos de processo _____ não requerem tanta disciplina e visam responder apropriadamente às mudanças.

Complete

4. No desenvolvimento ágil, devido à importância da entrega incremental do software, é comum o emprego de um fluxo de processo _____ .

Verdadeiro ou Falso

5. ( ) O processo XP envolve quatro atividades: planejamento, projeto, codificação e testes.

Múltipla Escolha

6. Marque com um X os itens que contêm valores da abordagem XP:

a) ( ) Comunicação;

b) ( ) Planejamento;

c) ( ) Simplicidade;

d) ( ) Feedback;

e) ( ) Documentação;

f) ( ) Coragem / Disciplina;

g) ( ) Respeito.

Verdadeiro ou Falso

7. ( ) Em XP, desenvolvem-se *user stories* com o intuito de descrever cada funcionalidade requisitada.

Verdadeiro ou Falso

8. ( ) Em XP, inicia-se a atividade de planejamento com o desenvolvimento de uma série de testes de unidade para cada história de usuário.

Verdadeiro ou Falso

9. ( ) Não se recomenda a programação em pares no processo XP, pois alocar dois programadores para implementar uma mesma funcionalidade seria um grande desperdício.

Complete

10. Em XP, caso um difícil problema de projeto seja encontrado, recomenda-se implementar e validar um _____ .

Responda

11. Em nossas aulas, comentamos que “vivemos em um mundo repleto de mudanças”. Em se tratando de desenvolvimento de software:

a) Quais são essas possíveis mudanças?

b) Quais as consequências delas para o desenvolvimento de um software?

c) O que pode ser feito para minimizar o impacto de tais mudanças?

Responda

12. Explique com suas palavras o que o Manifesto Ágil quer dizer quando afirma que deve se priorizar:

a) Indivíduos e interações acima de processos e ferramentas?

b) Software operacional acima de documentação completa?

c) Colaboração dos clientes acima de negociação contratual?

d) Respostas a mudanças acima de seguir um plano?

Responda

13. Explique o que significa cada uma das seguintes características do desenvolvimento ágil:

a) Adaptável;

b) Incremental;

c) Iterativo;

d) Focado na comunicação.

Responda

14. Descreva agilidade (no contexto de desenvolvimento de software) com suas palavras;

Responda

15. Por que o processo iterativo facilita o gerenciamento de mudanças?

Responda

16. O que é refatoração?

Responda

17. Qual a vantagem da programação em duplas?

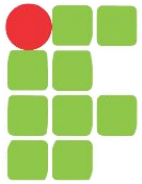
Escreva

18. Escolha um dos modelos de processo ágil citados (mas não descritos), pesquise na web ou em livros e escreva a respeito.

SAIBA MAIS!

Fatores humanos no Desenvolvimento Ágil

- O processo deve adequar-se às necessidades da equipe de desenvolvimento e não o contrário!
- Membros de uma equipe ágil devem ter:
  - Competência (talento inato + habilidades relacionadas a desenvolvimento de software + conhecimento generalizado do processo escolhido);
  - Foco comum (entregar um incremento de software funcionando ao cliente, no prazo definido);
  - Habilidades de colaboração, tomada de decisão, solução de problemas e auto-organização;
  - Confiança mútua e respeito.



INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SERGIPE

Técnicas de Elicitação de Requisitos

Parte 04

Sumário

- Um “caso real”
- Definição de elicitação de requisitos
- Tipos de requisitos
- Dilema dos requisitos
- Dificuldades na elicitação de requisitos
- Elicitação, Análise e Negociação
- Técnicas de elicitação de requisitos

Um “caso real”

(RAMOS, 2013)

- **Cliente:** O Sistema que queremos deve fazer isto, isto... e nesse caso, isto;
- **Analista:** Sim, sim estou anotando;
- **Cliente:** Conversei com os funcionários (futuros usuários) e basicamente este é o Sistema que precisamos.

Um “caso real”

… Mais tarde…

- **Analista:** Conversei com o nosso cliente sobre o Sistema;
- **Chefe:** Ótimo, comece a especificar os requisitos imediatamente!

Um “caso real”

… Quatro meses depois…

- **Analista:** senhor cliente, após o emprego das mais modernas técnicas de especificação, produzimos este documento que descreve minuciosamente o sistema;
- **Cliente:** Ótimo! Bom! Hum… é um documento com 300 páginas e todos estes gráficos, tabelas. Enfim, vamos analisá-lo e voltamos a falar.

Um “caso real”

… Depois de um mês e meio…

- **Cliente:** senhor analista, nosso pessoal analisou com cuidado o documento. Tivemos muita dificuldade e dúvidas para entendê-lo. Mas o que percebemos é que **não fomos corretamente entendidos**!
- **Analista:** como não? Está tudo aí (fruto do nosso entendimento pessoal). Realmente, **vocês não sabem o que querem**!

Exercício: Análise do “caso real”

- Que tal retornarmos agora e analisarmos cada trecho a fim de identificar os possíveis equívocos cometidos?

Definição de Elicitação de Requisitos

- **Elicitação:** ato de extrair informações para o conhecimento do objeto em questão;
- **Requisitos:** Características requeridas para um determinado fim;
- **Elicitação de Requisitos (de Software):** processo de obtenção de informações acerca das características de um software para o cumprimento de suas tarefas.

No âmbito da engenharia, um requisito consiste na definição documentada de uma propriedade ou comportamento que um produto ou serviço particular deve atender (WIKIPÉDIA, 2015b).

Tipos de requisitos

Em se tratando de software, costumamos dividi-los em (WIKIPÉDIA, 2015c; WIKIPÉDIA, 2015d):

- **Requisitos funcionais** – um requisito funcional define uma função de um sistema de software ou seu componente. Geralmente se referem a tarefas (funcionalidades) que o sistema será capaz de realizar;
- **Requisitos não funcionais** – são requisitos relacionados ao uso da aplicação em termos de desempenho, usabilidade, confiabilidade, segurança, disponibilidade, manutenibilidade e tecnologias envolvidas.

Dilema dos requisitos

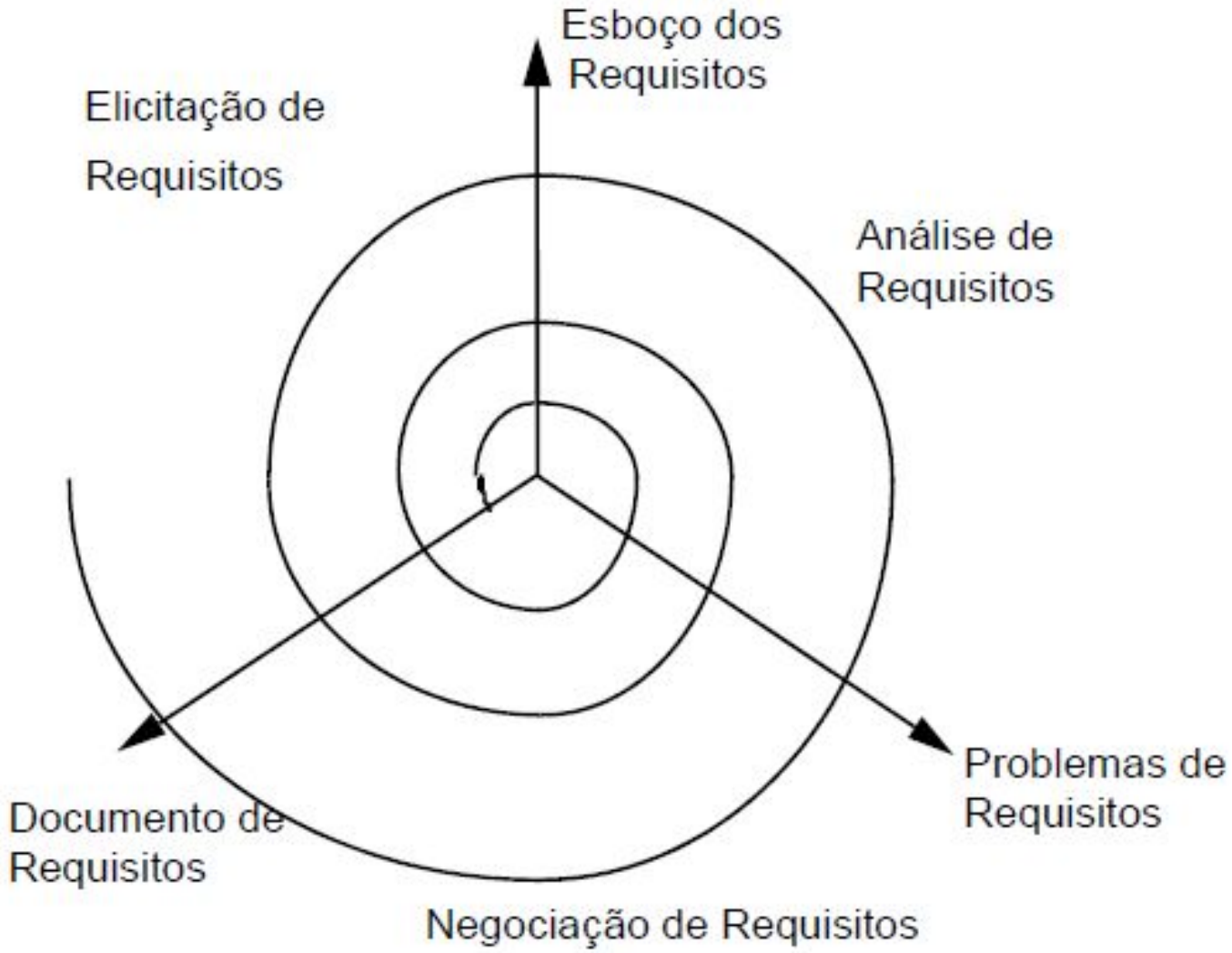
- **Requisitos descritos de forma vaga** estão sujeitos a ambiguidade, incompletude e inconsistência;
  - Técnicas como inspeções rigorosas têm sido usadas para ajudar a lidar com questões de ambiguidade;
  - A análise de requisitos esforça-se pode endereçar estes assuntos.
- Entretanto, **requisitos muito detalhados**:
  - Demoram muito tempo para serem descritos (modelados);
  - Limitam as opções possíveis de implementação;
  - Sua produção (modelagem) fica demasiada cara.

Dificuldades na elicitação de requisitos

- Usuários podem não ter uma ideia precisa do sistema por eles requerido;
- Usuários têm dificuldades para descrever seu conhecimento sobre o domínio do problema;
- Usuários e Analistas têm diferentes pontos de vista do problema (por terem diferentes formações);
- Usuários podem antipatizar-se com o novo sistema e se negarem a participar da elicitação (ou mesmo fornecer informações errôneas).

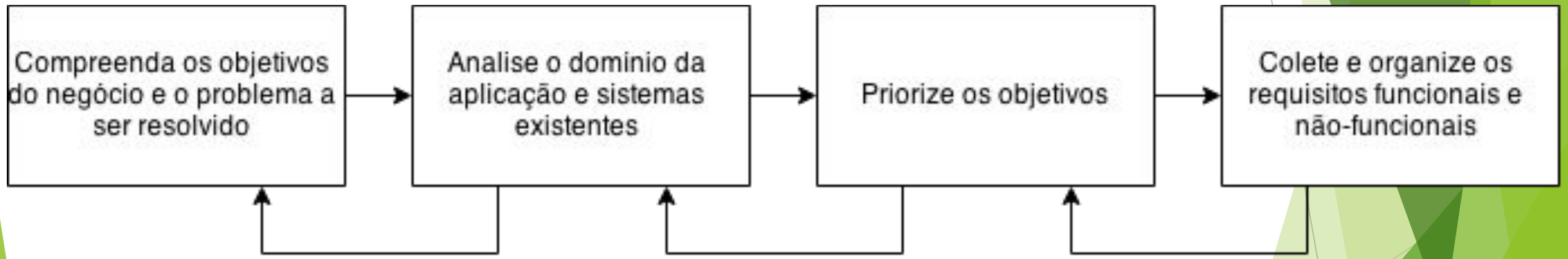
(RAMOS, 2013)

Elicitação, Análise e Negociação



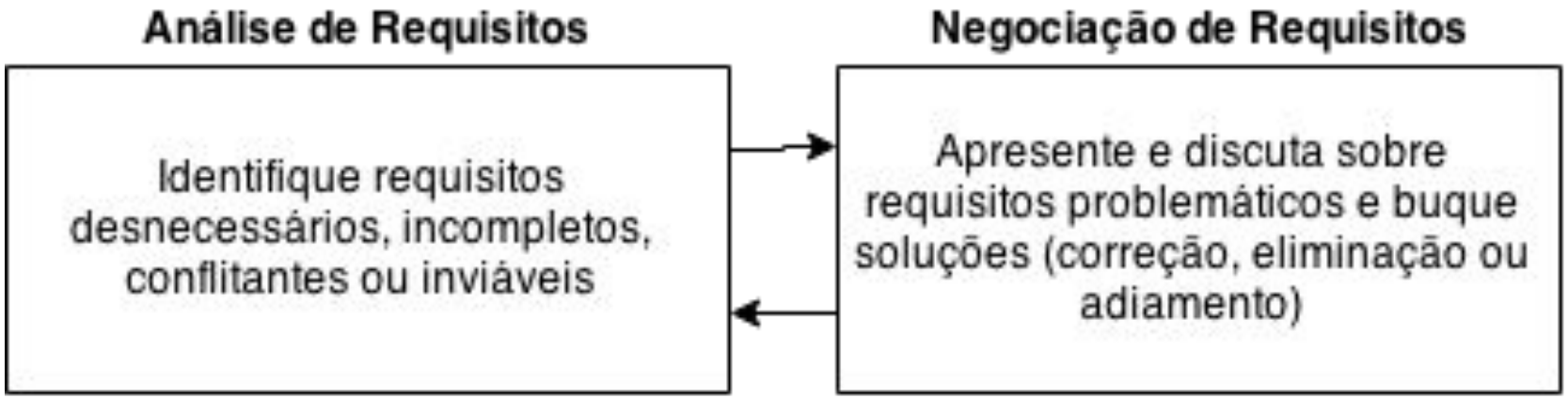
Atividades da Elicitação de Requisitos (RAMOS, 2013)

Elicitação de Requisitos



Processo da Elicitação de Requisitos

Análise e Negociação de Requisitos



Processo da Análise e Negociação de Requisitos

Técnicas de Elicitação de Requisitos

- Reuniões
  - Entrevista
  - Brainstorming
  - Tutorial
- Leitura de documentos
- Análise da tarefa
  - Análise de protocolos
  - Observação e análise sociais
- Cenários
- Reúso de requisitos
- Prototipagem

Entrevista

- O engenheiro (sozinho ou com alguns membros-chave da equipe de desenvolvimento) conversa sobre o sistema com representantes dos diversos tipos de *stakeholders* relacionados ao projeto (individualmente ou em grupo);
  - **Vantagem:** contato direto com clientes e usuários;
  - **Desvantagem:** conhecimento tácito ou diferenças culturais podem dificultar a transmissão de informação relevante;
- Entrevistas podem ser fechadas (buscam-se respostas para um conjunto de perguntas pré-definidas) ou abertas (não há uma agenda pré-definida).

Brainstorming

- Técnica empregada para o desenvolvimento de novos produtos (incluindo *softwares*) e na resolução de problemas;
- Em certos aspectos, assemelha-se a uma entrevista aberta;
- Apresenta três etapas principais:
  - Identificação dos fatos (definição do problema e preparação);
  - Geração da ideia;
  - Identificação da solução.

Brainstorming

- Princípios
  - Atraso do julgamento;
  - Criatividade em quantidade e qualidade.
- Regras
  - Críticas são rejeitadas;
  - Criatividade é bem-vinda;
  - Quantidade é necessária;
  - Combinação e aperfeiçoamento são necessários.

Prototipagem

- Protótipo é uma versão de parte do sistema (executável ou não) desenvolvida para ser validada pelo cliente a fim de identificar possíveis equívocos na elicitação;
- Abordagens:
  - Prototipagem no papel ou de tela;
  - Prototipagem executável;
    - Descartável;
    - Evolucionária.

Prototipagem

- **Vantagens:** permite que o usuário “experimente” o sistema antes que o mesmo esteja pronto, facilita identificar possíveis erros na modelagem dos requisitos e força um estudo detalhado dos requisitos, revelando inconsistências e omissões;
- **Desvantagens:** custo de treinamento para o uso de ferramentas de prototipagem, custo de desenvolvimento (a depender do tipo de protótipo) e alguns requisitos críticos podem não ser possíveis de prototipar.

Atividade em grupo

- Cada grupo deve elicitar os requisitos para o seu sistema junto ao cliente.

SAIBA MAIS!

Tutorial

- Trata-se de uma apresentação de um conjunto de informações acerca do problema ou das necessidades do software coordenada pelo próprio cliente ou usuário, em um formato similar a uma aula.

SAIBA MAIS!

Leitura de documentos

- Pode-se optar pela leitura de documentos pertinentes à execução da tarefa em questão, de formulários a manuais;
- **Vantagens:** facilidade de acesso, volume de informações e acesso ao vocabulário da aplicação;
- **Desvantagens:** dispersão das informações e volume de trabalho.

SAIBA MAIS!

Análise de protocolos

- Análise do trabalho realizado por uma pessoa por meio da comunicação verbal com a mesma, visando compreender todos os passos intrínsecos em sua execução;
- **Vantagem:** possibilidade de elicitar fatos não facilmente observáveis;
- **Desvantagem:** pode haver discrepância entre o que é falado e o que é realmente feito.

SAIBA MAIS!

Observação e análise sociais

- É a observação de pessoas realizando a tarefa em si no ambiente de trabalho, com o intuito de coletar informações, compreendê-las e modelá-las;
- **Vantagem:** oferece uma visão mais completa e realista de como a tarefa é realizada;
- **Desvantagens:** pode consumir muito tempo e apresenta pouca sistematização em sua execução.

SAIBA MAIS!

Cenários

- Cenários são estórias que explicam como um sistema poderá ser usado. Eles devem incluir:
  - Descrição do estado do sistema antes de começar o cenário;
  - Fluxo normal de eventos do cenário e possíveis exceções;
  - Descrição do estado do sistema ao final do cenário.
- **Vantagens:** expõem interações possíveis do sistema e revela as facilidades que o sistema pode precisar;
- **Desvantagens:** detalhes relevantes podem ser ignorados e dificuldade na elaboração ou validação dos cenários pelos clientes/usuários.

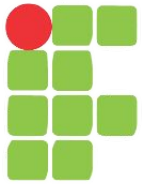
(RAMOS, 2013)

SAIBA MAIS!

Reúso de requisitos

- Reúso envolve considerar requisitos que foram desenvolvidos para um sistema e usá-los em sistemas diferentes, economizando tempo e esforço (requisitos reutilizados já foram analisados e validados);
- Atualmente o reuso de requisitos é um processo informal. Contudo, um reuso mais sistemático economizaria muito esforço;
- **Vantagens:** produtividade e qualidade (componentes já validados);
- **Desvantagens:** dificuldade de se promover reutilização sem modificação.

(RAMOS, 2013)



INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SERGIPE

Introdução à UML

Sumário

- O que é UML?
- O que é um diagrama?
- Diagrama de casos de uso
- Descrição de caso de uso
- Diagrama de atividades
- Diagrama de classes
- Diagrama de sequência

O que é UML?

- UML (*Unified Modeling Language* – Linguagem de Modelagem Unificada) é uma linguagem que integra elementos visuais e textuais com o objetivo de descrever/documentar um projeto de software;
- A compreensão do “vocabulário” da UML permite especificar ou compreender especificações de um projeto de software bem como explicar o mesmo para outros interessados;

O que é UML?

- Desenvolvida por Grady Booch, Jim Rumbaugh e Ivar Jacobson na década de 1990, adota diversos conceitos de orientação a objetos;
- É mantida pela *Object Management Group* (OMG);
- Atualmente na versão 2.5.1, fornece 14 tipos de diagramas para modelagem de software.

O que é um diagrama?

- Trata-se de uma representação visual, estruturada e simplificada de um conceito, ideia, produto etc.
  - Geralmente contendo entidades, relações e anotações.
- Há inúmeros tipos de diagramas presentes na literatura (e não somente os diagramas da UML!);
  - Diagrama Entidade-Relacionamento.

Diagramas da UML

- Diagramas Estruturais:
  - Diagrama de Classes;
  - ~~Diagrama de Objetos;~~
  - Diagrama de Pacotes;
  - Diagrama de Estrutura Composta;
  - Diagrama de Componentes;
  - Diagrama de Implantação;
  - Diagrama de Perfil.
- Diagramas Comportamentais:
  - Diagrama de Casos de Uso;*
  - Diagrama de Atividades;
  - Diagrama de Máquina de Estados;
  - Diagramas de Interação:
    - Diagrama de Sequência;
    - Diagrama de Comunicação;
    - Diagrama de Tempo;
    - Diagrama de Visão Geral da Interação.

Diagrama de casos de uso

- Representa o sistema, os atores que interagem com o sistema e cada um dos casos de uso que os mesmos podem disparar.

Diagrama de casos de uso

- Um diagrama de casos de uso pode representar cada componente da seguinte forma:
  - Sistema: grande retângulo contendo os casos de uso;

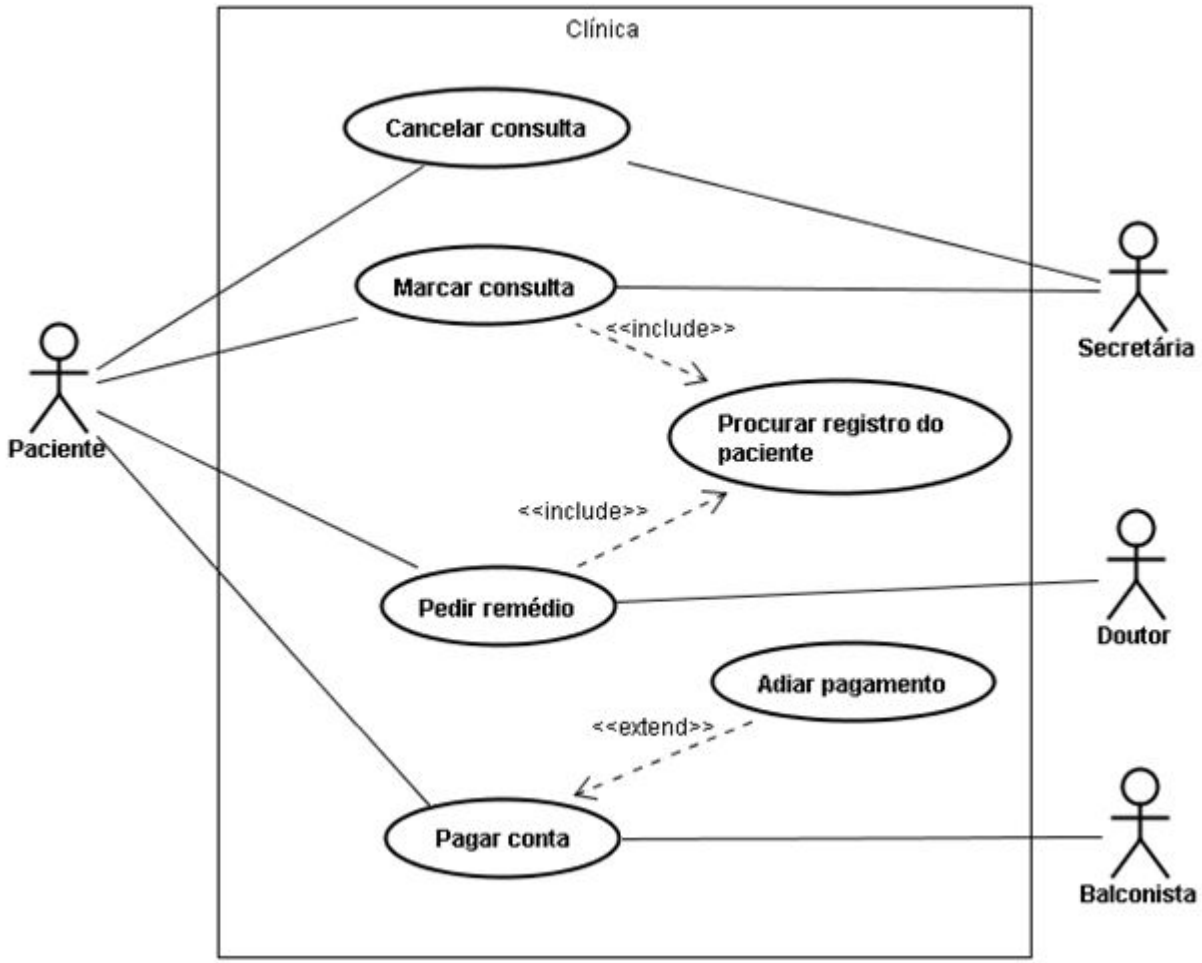


Diagrama de casos de uso para uma clínica

Diagrama de casos de uso

- Um diagrama de casos de uso pode representar cada componente da seguinte forma: (cont.)
  - Atores: bonecos, tanto para atores humanos quanto para dispositivos ou sistemas externos (alguns autores recomendam adoção de um retângulo pequeno para representar atores não-humanos);

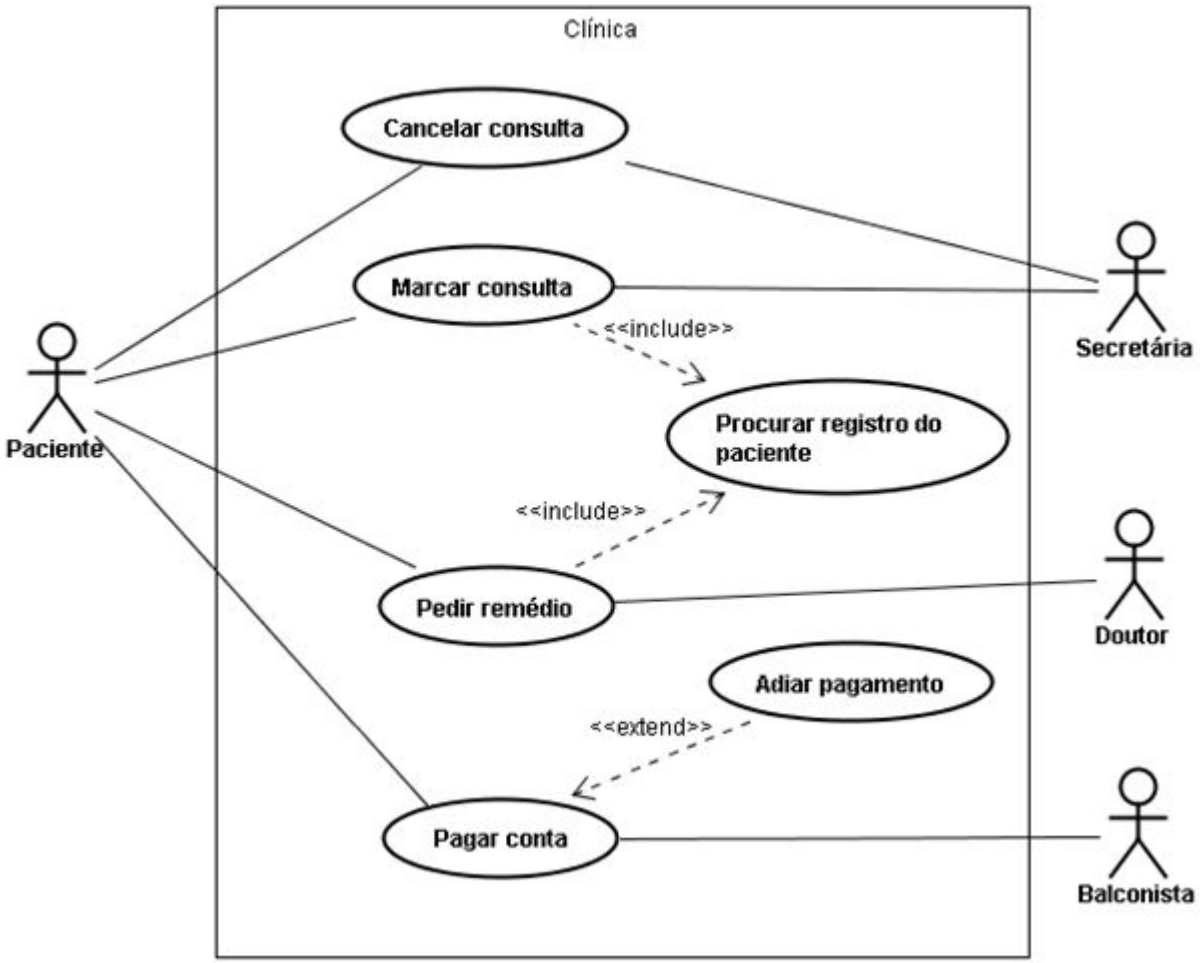


Diagrama de casos de uso para uma clínica

Diagrama de casos de uso

- Um diagrama de casos de uso pode representar cada componente da seguinte forma: (cont.)
  - Casos de uso: elipses contendo dentro somente o nome do caso;

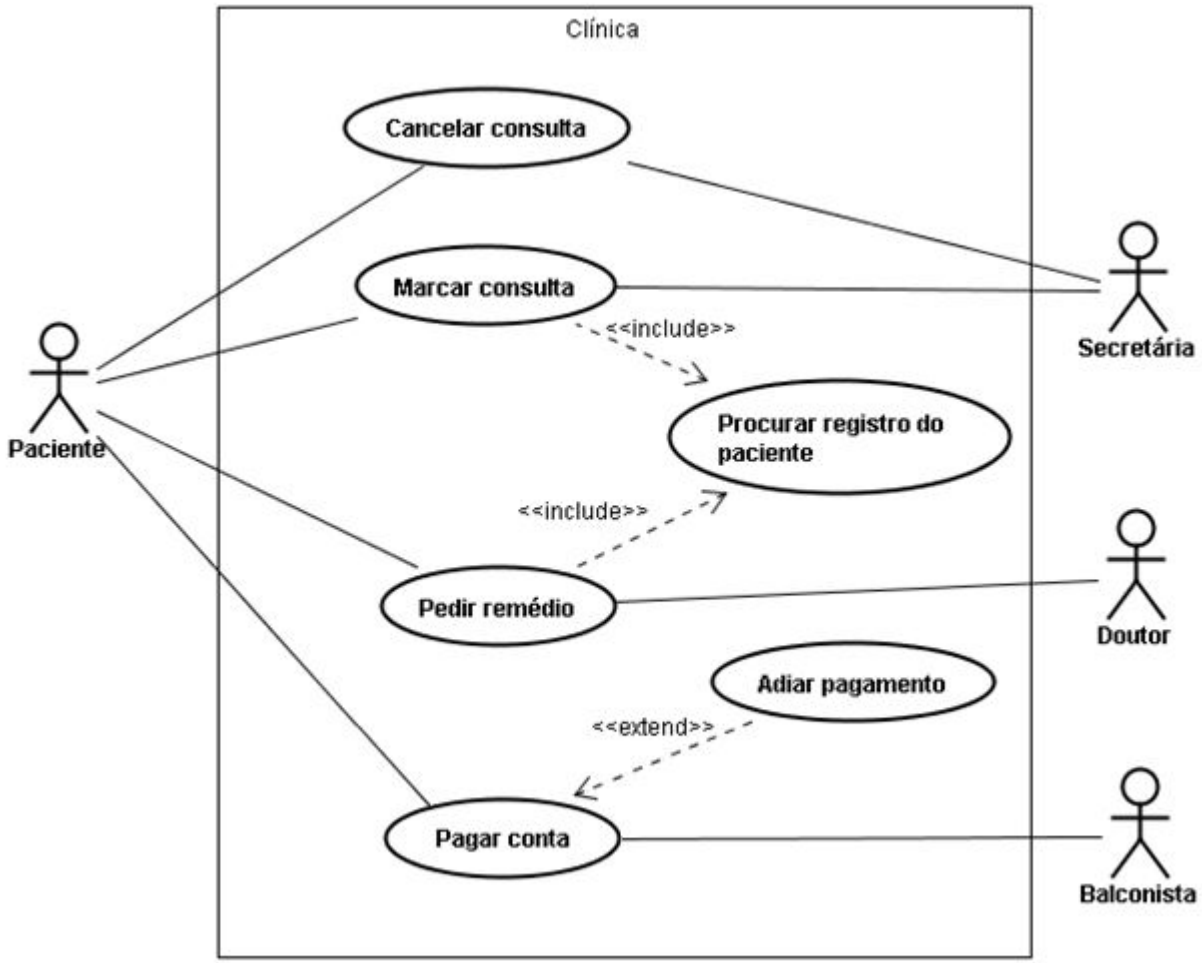


Diagrama de casos de uso para uma clínica

Diagrama de casos de uso

- Um diagrama de casos de uso pode representar cada componente da seguinte forma: (cont.)
  - Quem-faz-o-quê: setas ou simplesmente linhas retas;

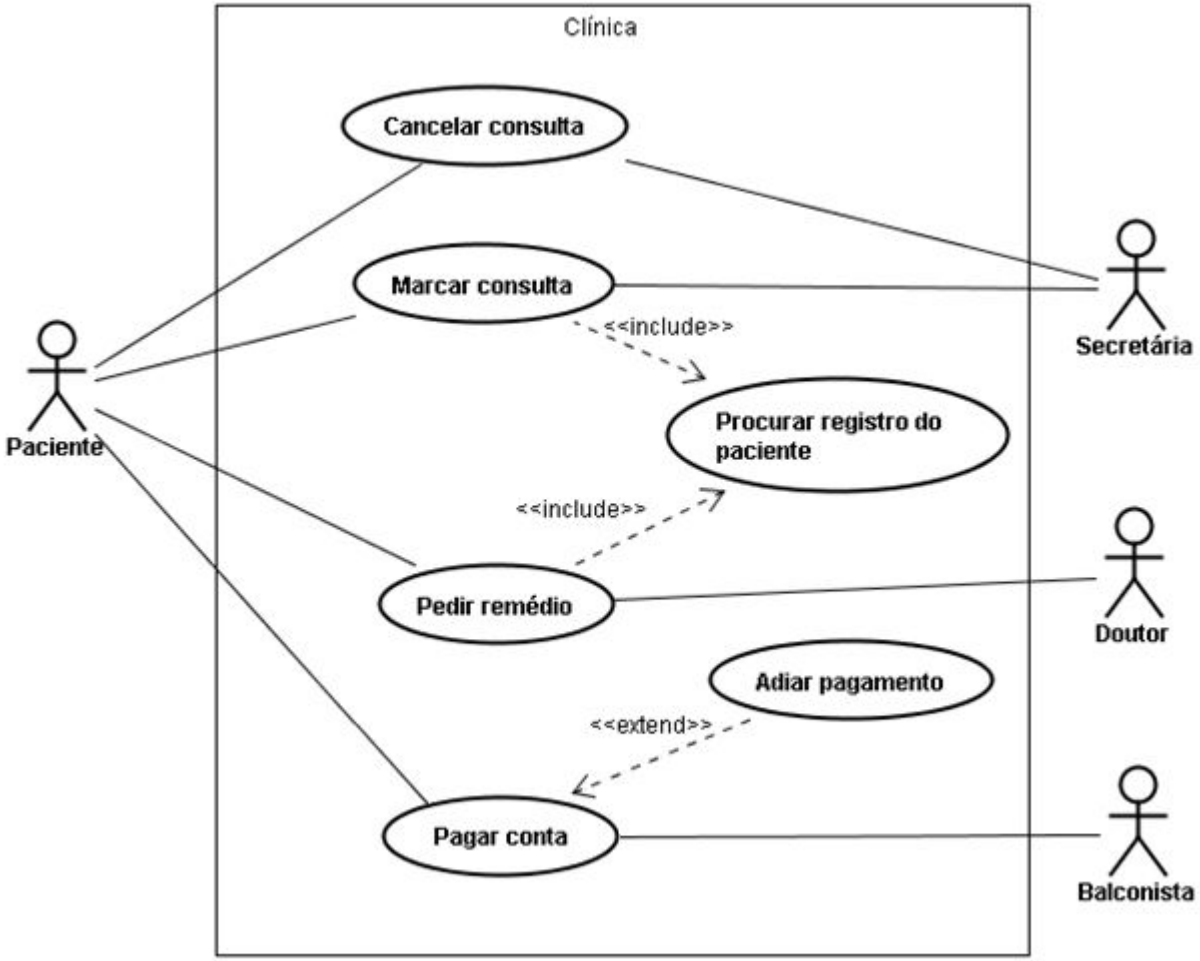


Diagrama de casos de uso para uma clínica

Diagrama de casos de uso

- Um diagrama de casos de uso pode representar cada componente da seguinte forma: (cont.)
  - Relações entre casos de uso: setas pontilhadas, contendo uma notação descrevendo o tipo de relação (inclui, estende etc.).

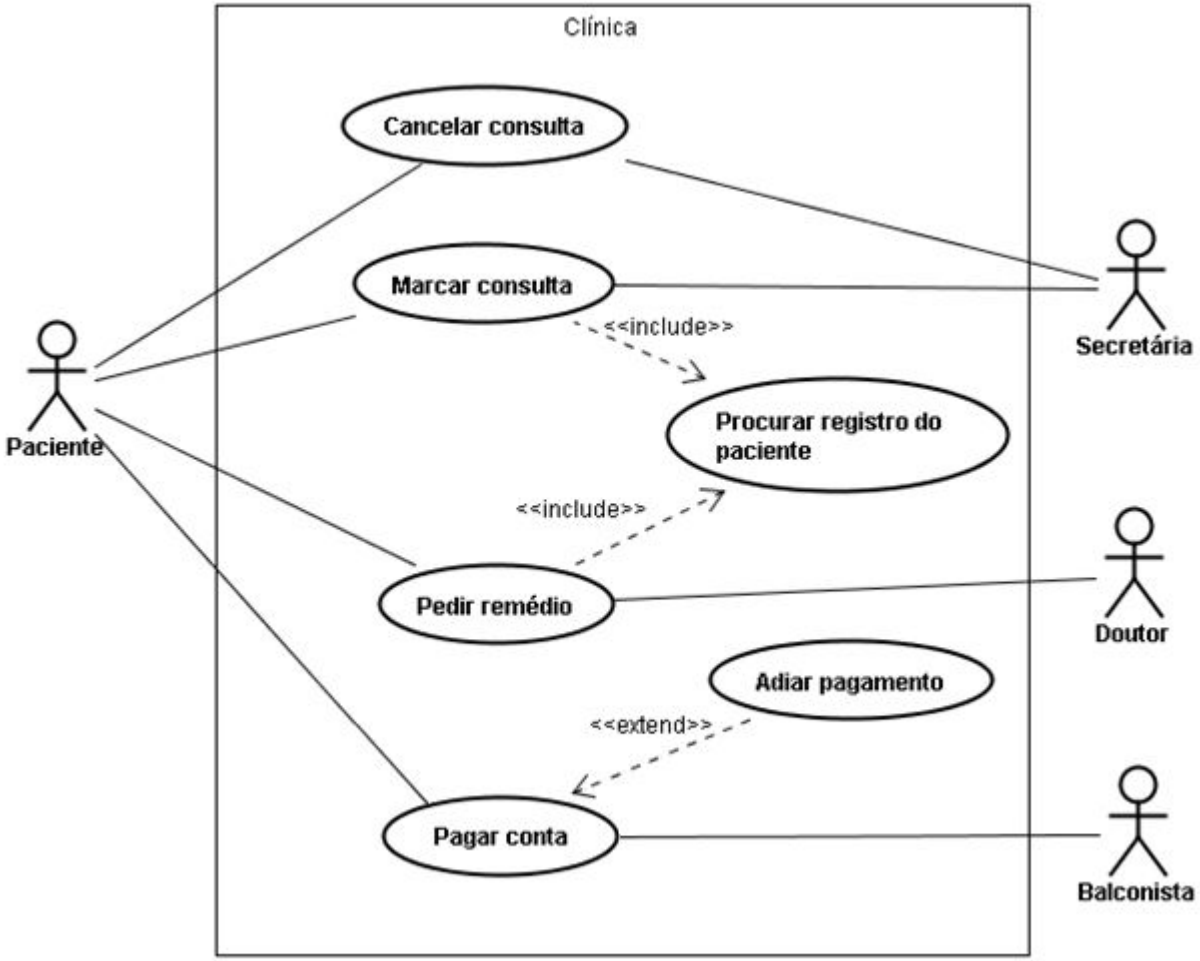


Diagrama de casos de uso para uma clínica

Atividade em grupo

- Cada grupo deve modelar o diagrama de casos de uso para o seu sistema.

Descrição de caso de uso

- A descrição de um caso de uso detalha as condições necessárias para a execução de um caso de uso bem como cada passo executado pelo mesmo;
- Apesar de não ser padronizada pela UML, é importante que o engenheiro de software saiba como descrever um caso de uso e interpretar as informações de uma descrição.

Descrição de caso de uso

- As seguintes perguntas devem ser respondidas por um caso de uso:
  - Quais atores estão envolvidos?
  - O que um ator deve fazer para executá-lo?
  - Que passos serão executados pelo ator/sistema?
  - Que exceções podem ocorrer durante a realização dos passos supracitados e como o sistema deve comportar-se diante de cada uma delas?

Exemplo de descrição de caso de uso

Caso de uso	Efetuar Empréstimo
Ator primário	Bibliotecário
Disparador	Ator clica na opção “Efetuar Empréstimo”.
Cenário	1. Sistema exibe formulário para empréstimo com campos para informar ID do leitor e dos livros e os botões “Efetuar” e “Cancelar”; 2. Bibliotecário informa ID do leitor; 3. Sistema exibe dados do leitor; 4. Bibliotecário informa ID de cada livro; 5. Sistema adiciona à lista dados de cada livro; 6. Bibliotecário clica em “Efetuar”; 7. Sistema valida os dados, registra o empréstimo no Banco de Dados e imprime extrato.
Exceções	3a. ID do leitor inválida: Sistema exibe mensagem “ID do leitor inválida” e retorna para o campo ID do leitor; 3b. Leitor com empréstimo atrasado: Sistema exibe mensagem “Leitor com empréstimo pendente”; 3c. Leitor já atingiu máximo de empréstimos: Sistema exibe mensagem “Leitor já atingiu máximo de empréstimos simultâneos”; 5a. ID do livro inválida: Sistema exibe mensagem “ID do livro inválido” e retorna para o campo ID do livro.

Que outras exceções podem ser incluídas?

Atividade em grupo

- Cada grupo deve elaborar a descrição para três casos de uso de seu sistema.

Diagrama de atividades

- “O diagrama de atividades mostra o comportamento dinâmico de um sistema ou parte de um sistema através do fluxo de controle entre ações que o sistema executa” (PRESSMAN, 2011, p. 737);

Diagrama de atividades

- Componentes:
  - Nó inicial: círculo preto;
  - Nó final: círculo preto envolvido por uma circunferência preta;
  - Nó ação: retângulo arredondado;
  - Nó decisão: losângulo;
  - Início (*fork*) ou fim (junção/*join*) de atividades concorrentes: uma barra horizontal preta;
  - Fluxo: seta com linha sólida.

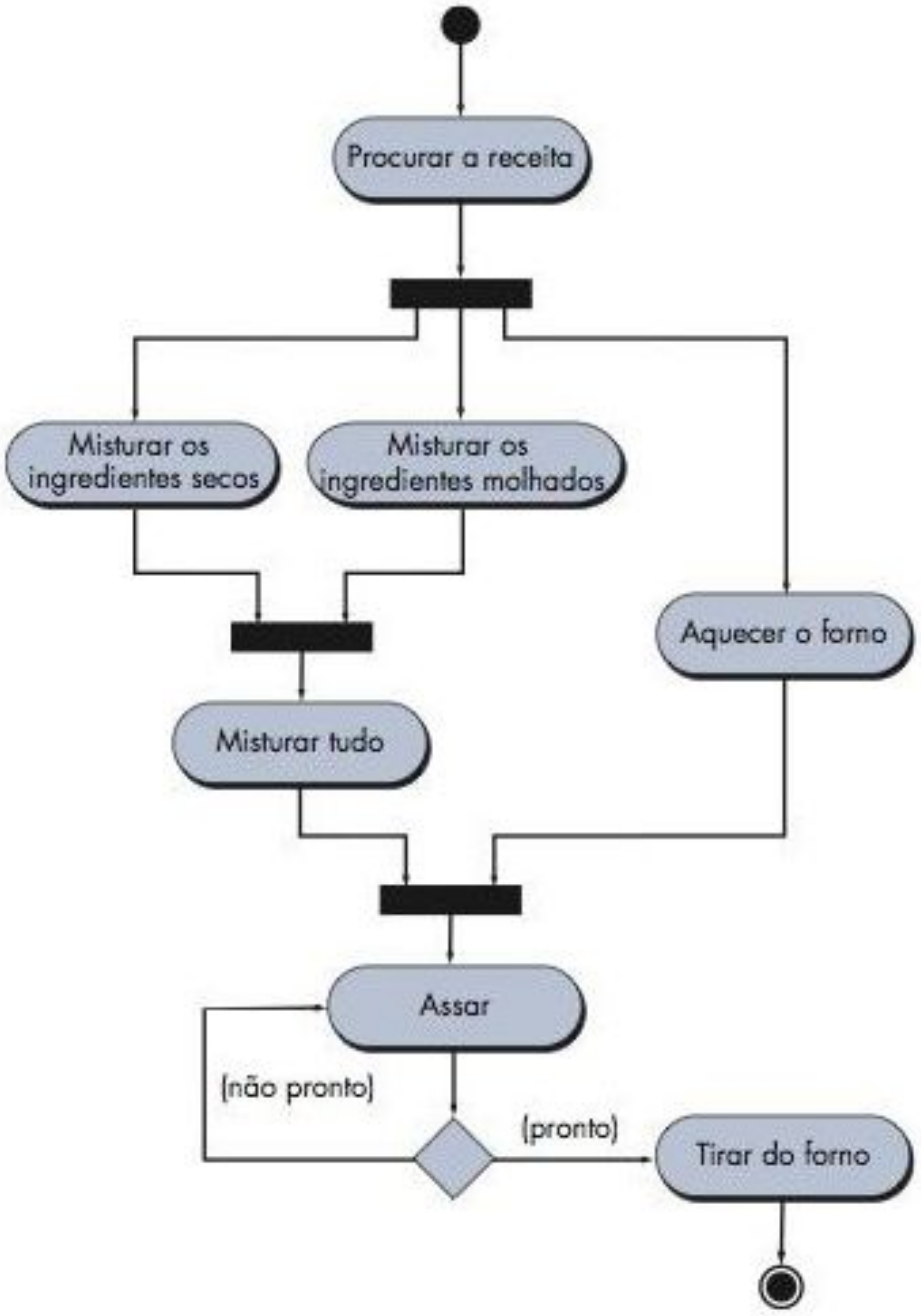


Diagrama de atividades para cozinhar

Diagrama de atividades com raias

- Quando há mais de um participante na execução das ações em um diagrama de atividades, pode ser confuso determinar quem é responsável pelo que;

- Nesses casos, podem-se adotar raias (*swimlanes*) no diagrama de atividades, onde cada raia pertence a um participante (geralmente um ator ou o sistema) e todas as ações presentes nela são executadas pelo seu mesmo.

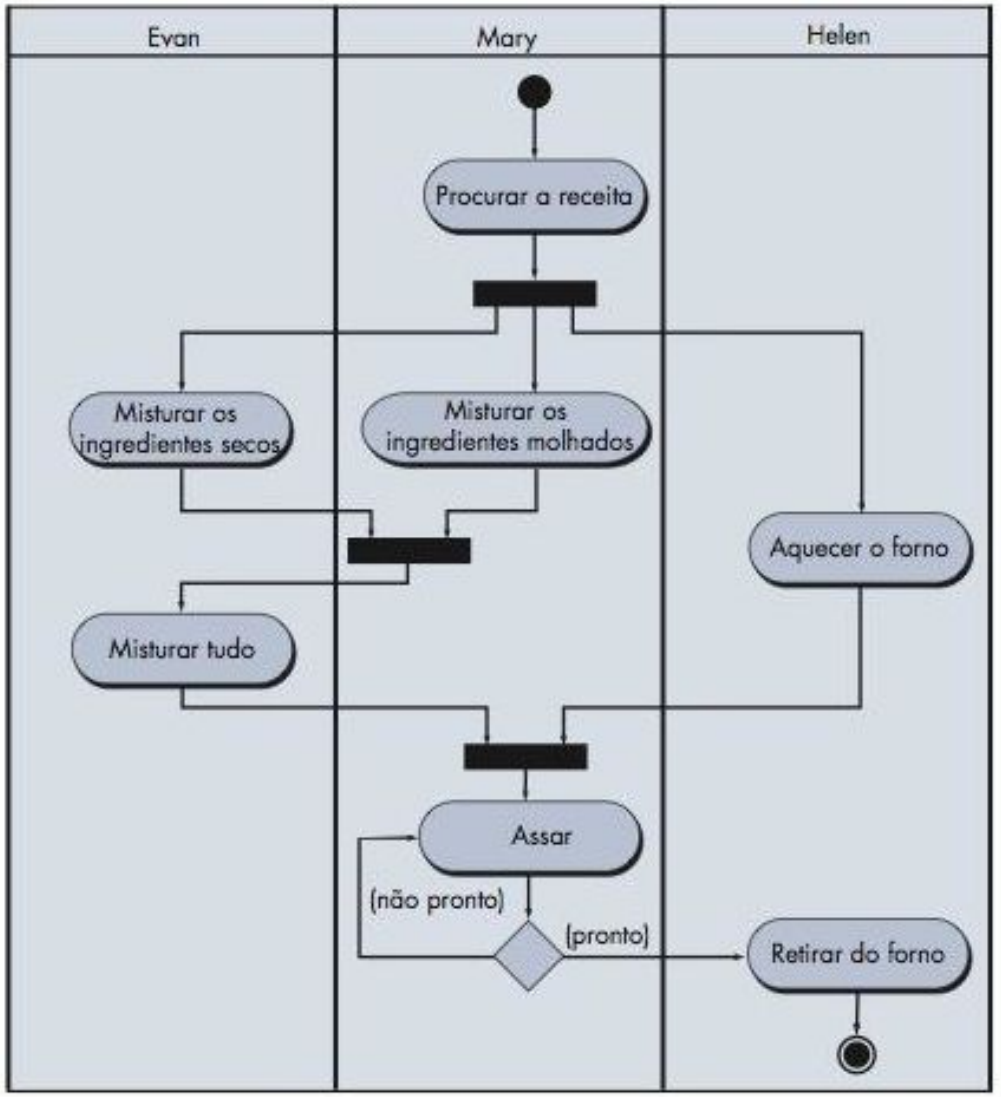


Diagrama de atividades para cozinhar (com raias)

Exemplo de diagrama de atividades para um *software*

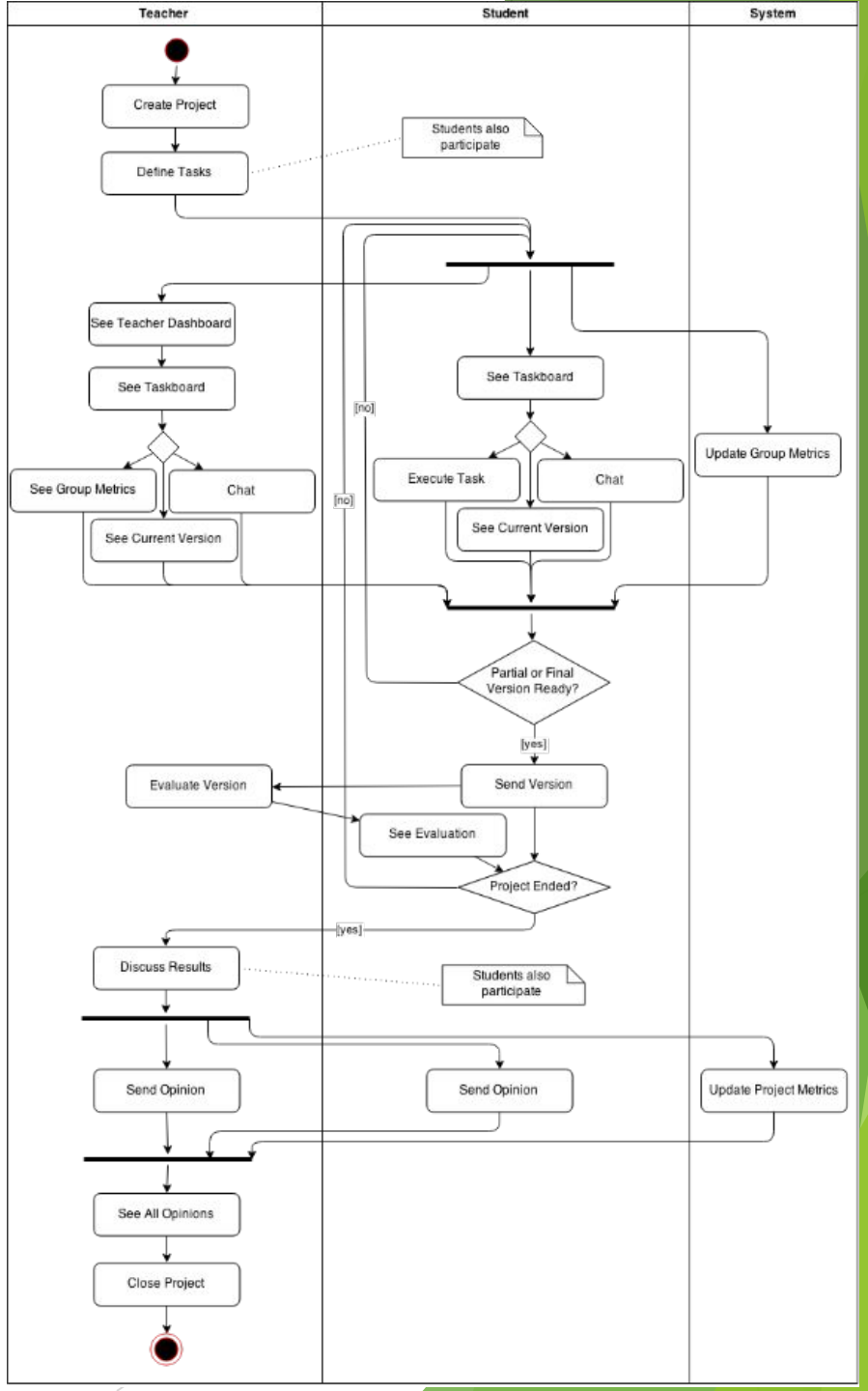


Diagrama de atividades para criação e execução de um projeto educativo (SANTOS, 2014)

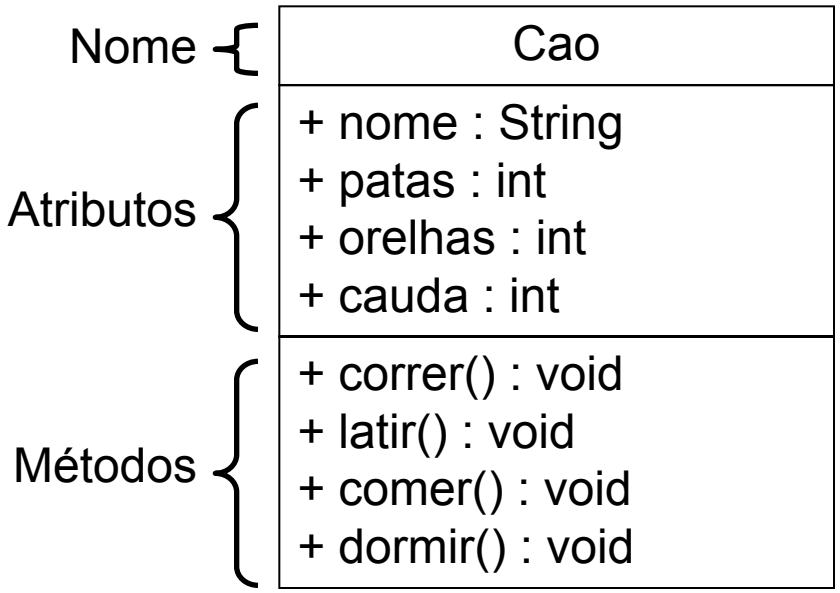
Atividade em grupo

- Cada grupo de desenhar o diagrama de atividades para dois casos de uso de seu sistema.

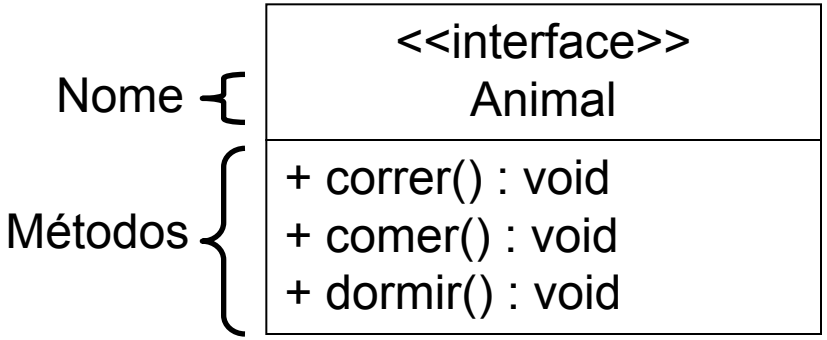
Diagrama de classes

- Permite modelar classes (e seus atributos e métodos) e suas relações com outras classes em um sistema;
- Fornece uma visão estática (estrutural) de um sistema, mas não fornece uma visão dinâmica (comportamental) do sistema;
  - Por exemplo, não explica como a comunicação é feita entre os objetos.
- Principais elementos são classes, interfaces e suas associações.

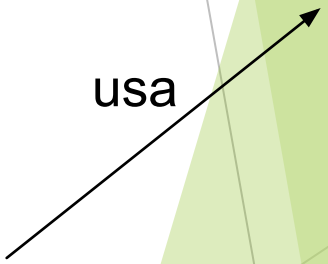
Diagrama de classes



Classe



Interface



Associação

Orientação a Objetos

- No mundo real, pessoas, animais e coisas (objetos) podem ser agrupados segundo suas similaridades (classes), possuem características próprias (atributos) e podem executar determinadas ações (métodos).

Cães

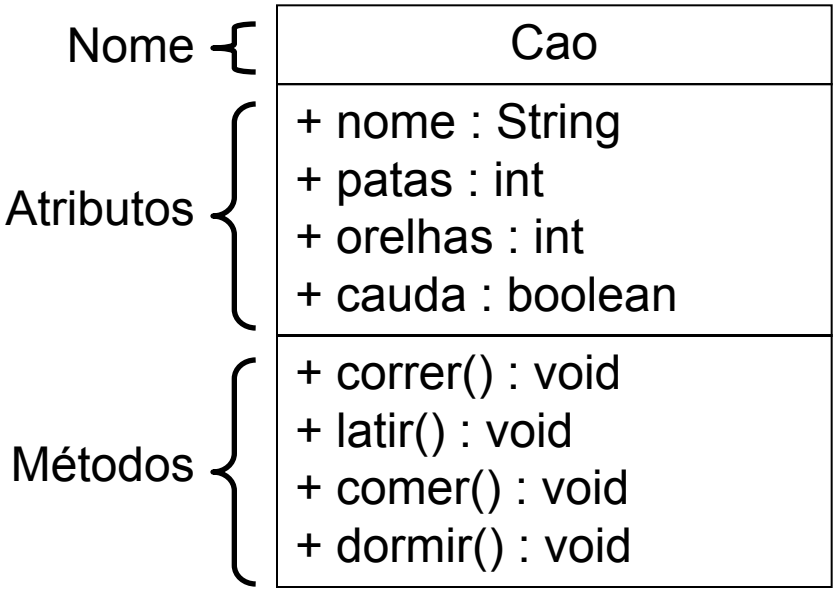
Rex, Johny, Lily, Boris, Hulk, Tico

Todos os cães possuem (características):
- nome;
- patas;
- orelhas;
- cauda.

Todos os cães podem (ações):
- correr;
- latir;
- comer;
- dormir.

Orientação a Objetos

- Em desenvolvimento de software orientado a objetos, podemos representar as informações e partes do sistema por meio de classes...



Classe Cao em UML

```
public class Cao {

   public String nome;
   public int patas;
   public int orelhas;
   public boolean cauda;

   public void correr() {
      ...
   }
   ...
}
```

Classe Cao em Java

Orientação a Objetos

- ... que serão instanciadas (ou não!) pelo programa!
  - Instanciar uma classe quer dizer criar um objeto (instância) da mesma.

```
public class Application {
   public Cao cao1;
   public Cao cao2;

   public static void main(String[] args) {
      cao1 = new Cao();
      cao1.nome = “Rex”;
      cao2 = new Cao();
      cao2.nome = “Johny”;
   }
   ...
}
```

Programa em Java instanciando objetos da classe Cao

Exercício

- Identifique as classes presentes em uma partida de futebol e desenhe-as como classes UML.

Um pouco de Análise e Projeto Orientado a Objetos

- A modelagem das classes concentra-se primeiro sobre classes extraídas diretamente do enunciado do problema;
  - Tipicamente representam itens que devem ser armazenados em base de dados;
- O projeto refina e amplia o conjunto de classes de entidade, desenvolvendo ou refinando:
  - Classes controladoras;
  - Classes de *front-end*.

(PRESSMAN, 2011, p. 746)

Estudo de Caso - Sistema de Biblioteca

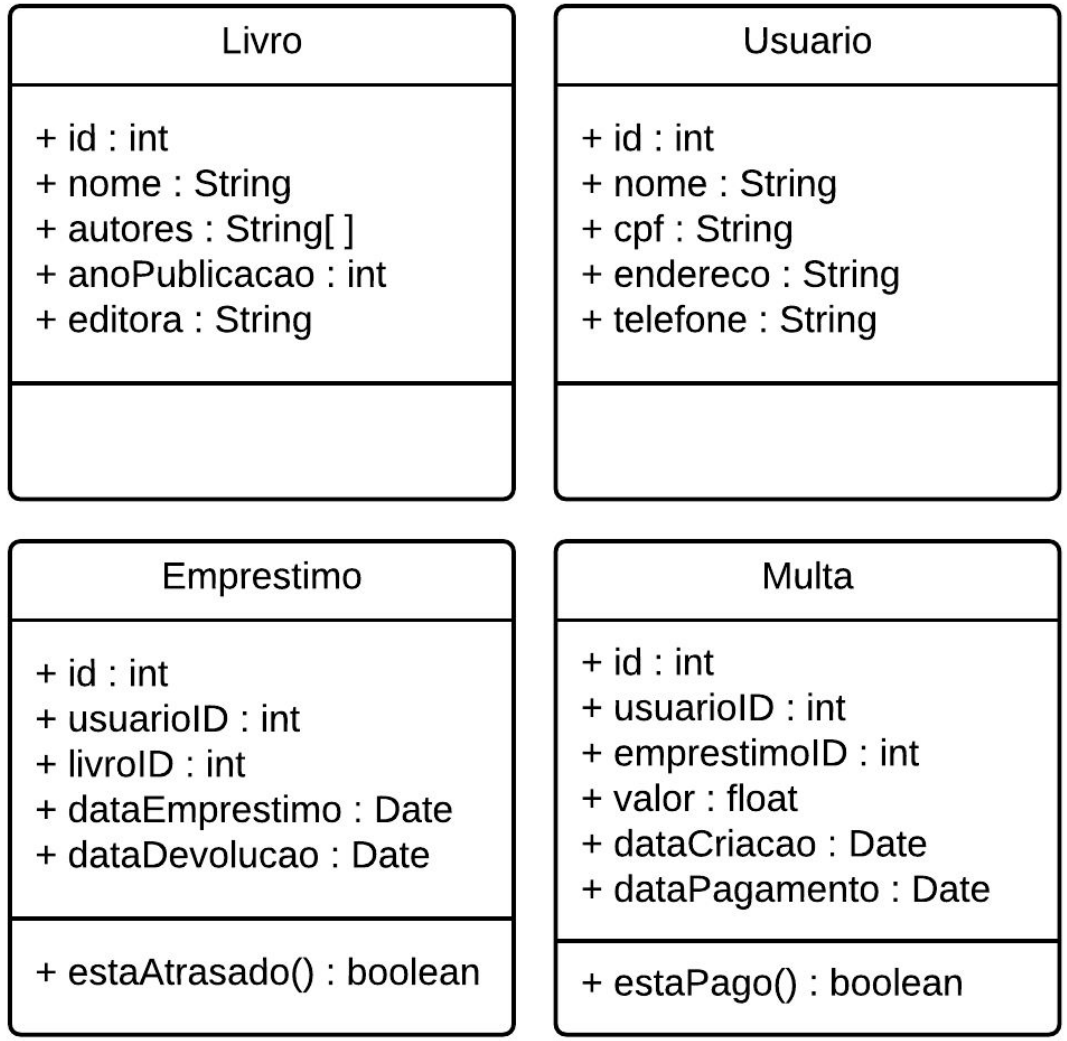
- Um sistema para controlar entrada e saída de livros emprestados bem como o pagamento de multas.

Estudo de Caso - Sistema de Biblioteca

- O sistema deve gerenciar:
  - Livros;
  - Usuários;
  - Empréstimos;
  - Multas.

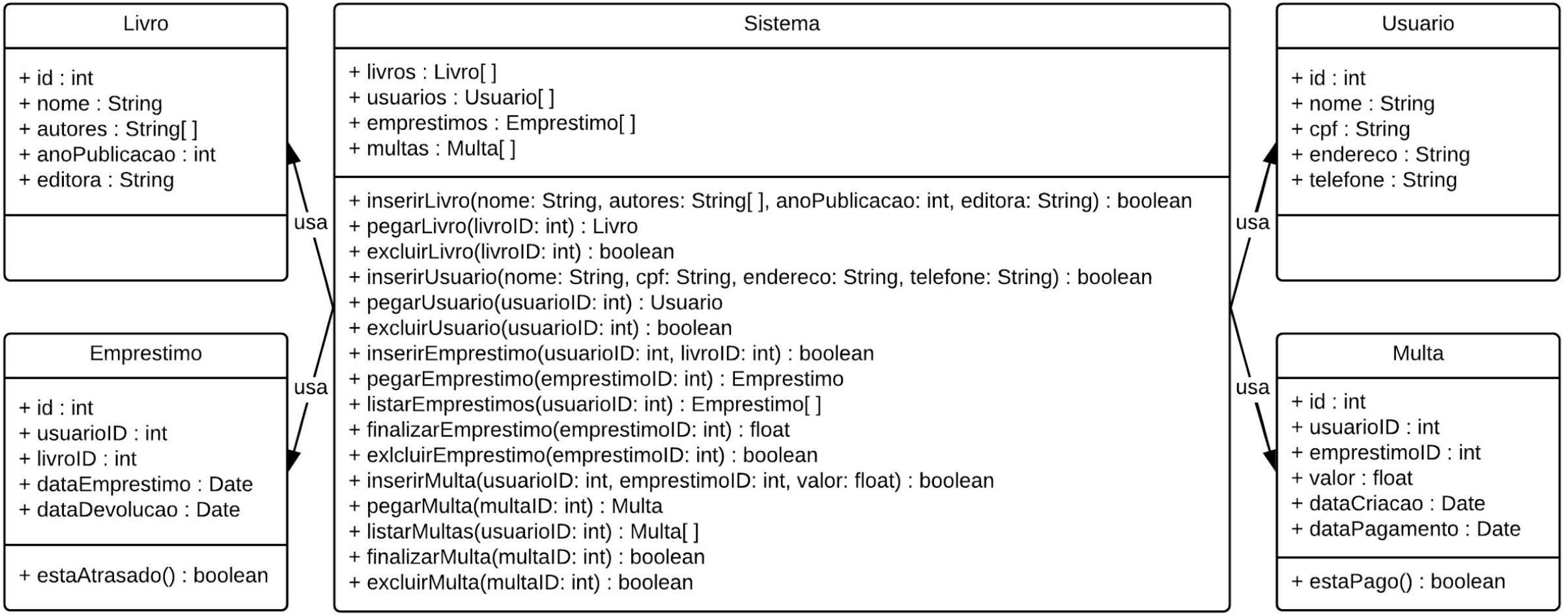
Estudo de Caso - Sistema de Biblioteca

- O sistema deve gerenciar:
  - Livros;
  - Usuários;
  - Empréstimos;
  - Multas.



Representações das classes em UML

Estudo de Caso - Sistema de Biblioteca



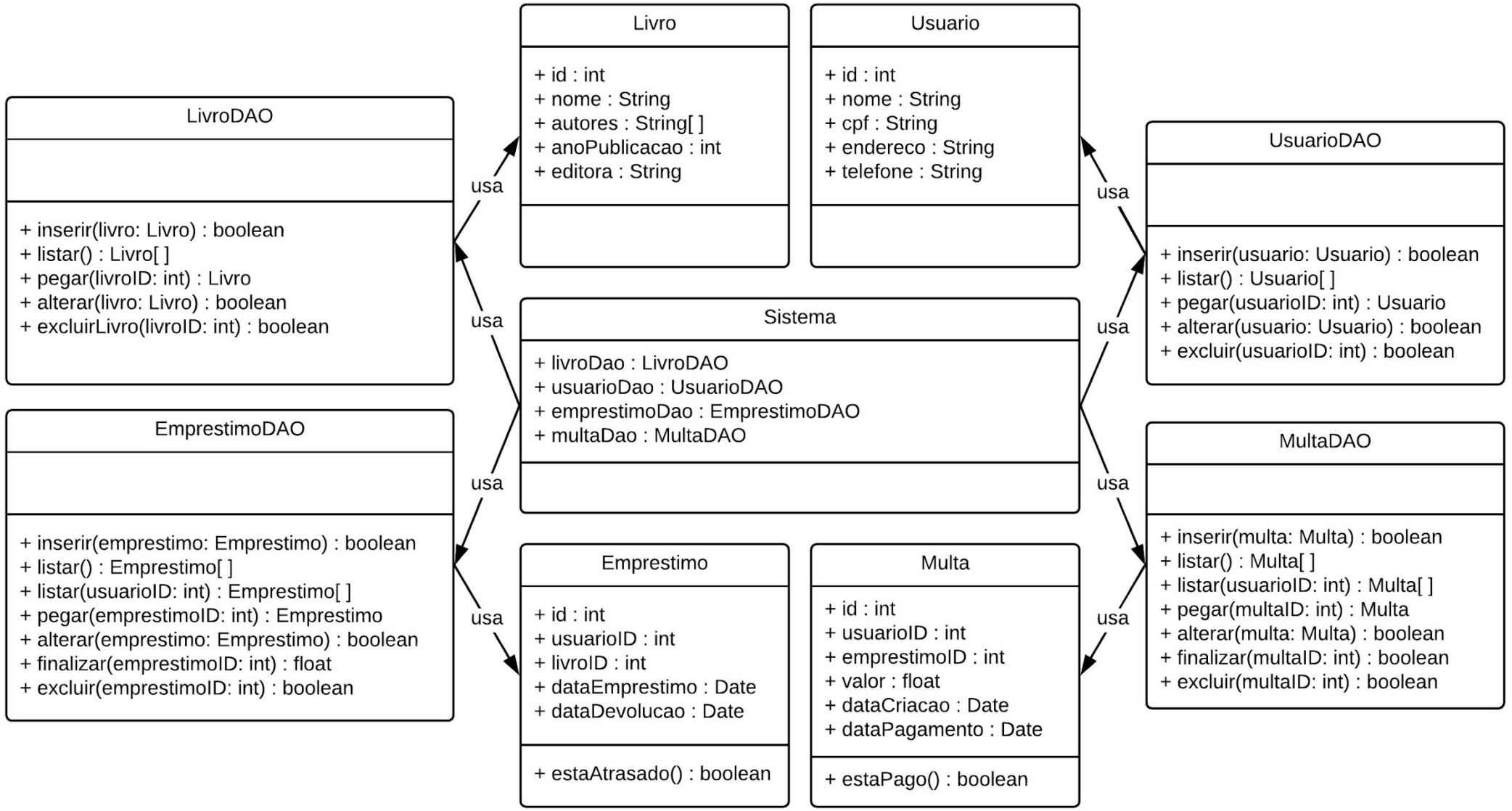
1ª Representação em classes do Sistema de Biblioteca

Estudo de Caso - Sistema de Biblioteca

```
public class Sistema {

   public ArrayList<Livro> livros = new ArrayList<Livro>();
   public ArrayList<Usuario> usuarios = new ArrayList<Usuario>();
   public ArrayList<Emprestimo> emprestimos = new ArrayList<Emprestimo();
   public ArrayList<Multa> multas = new ArrayList<Multa>();

   public static void main(String[] args) {
      ...
   }

   public boolean adicionarLivro(String nome, String[] autores, int anoPublicacao, String editora) {
      Livro obj = new Livro();
      obj.nome = nome;
      obj.autores = autores;
      obj.anoPublicacao = anoPublicacao;
      obj.editora = editora;
      livros.add( obj );
      return true;
   }
   ...
}
```

Esboço em Java da classe Sistema da 1ª Representação

Estudo de Caso - Sistema de Biblioteca



2ª Representação em classes do Sistema de Biblioteca

Estudo de Caso - Sistema de Biblioteca

```
public class Sistema {

   public LivroDAO livroDao = new LivroDAO();
   public UsuarioDAO usuarioDao = new UsuarioDAO();
   public EmprestimoDAO emprestimoDao = new EmprestimoDAO();
   public MultaDAO multaDao = new MultaDAO();

   public static void main(String[ ] args) {
      ...
      Livro livro = new Livro("Engenharia de Software", {"Carlos Andrade, "Thiago Moura"}, 2015, "Editora IFS");
      livroDao.inserir( livro );
      ...
      Usuario leitor = usuarioDao.pegar(57);
      ...
   }


}
```

Esboço em Java da classe Sistema da 2ª Representação

Atividade em grupo

- Cada grupo deve elaborar o diagrama de classes para o seu sistema.

SAIBA MAIS!

Diagrama de sequência

- Em contraste com o diagrama de classes, o diagrama de sequência representa a comunicação dinâmica (fluxo de invocação) entre objetos durante a execução de uma tarefa;
  - Apresenta a ordem em que as mensagens são enviadas entre os objetos.

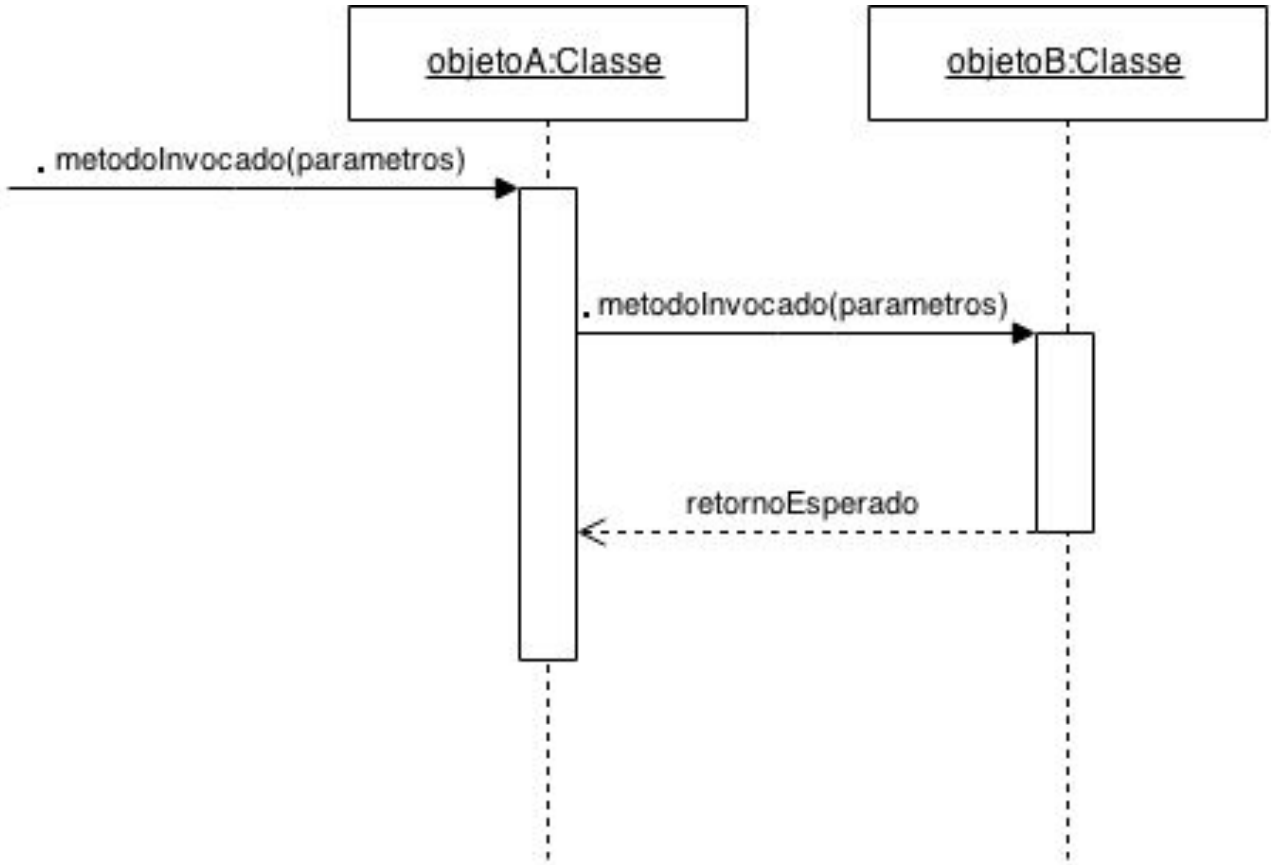
SAIBA MAIS!

Diagrama de sequência

- Cada objeto é representado por uma caixa contendo seu nome (opcional) e o seu tipo e uma linha vertical representando o tempo durante a execução da tarefa;
- Cada mensagem entre objetos é representada por uma seta de linha sólida ou tracejada.

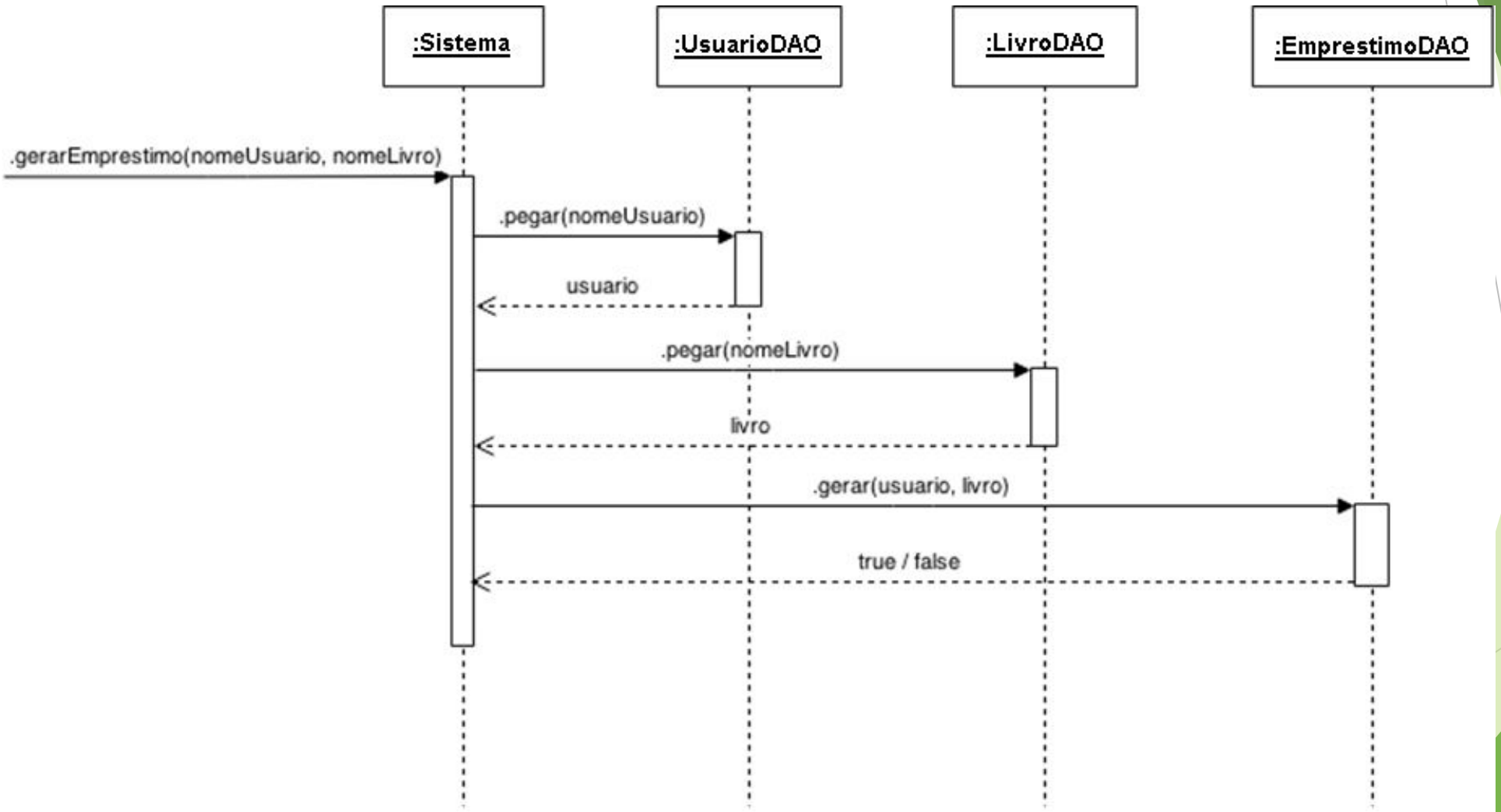
SAIBA MAIS!

Diagrama de sequência



SAIBA MAIS!

Exemplo de diagrama de sequência



Exemplo de diagrama de sequência para a tarefa “Gerar Empréstimo” de um sistema de biblioteca

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SERGIPE

Gestão de Projetos

Parte 06

Sumário

- Introdução à Gestão de Projetos
- Algumas abordagens de Gestão de Projetos
  - *Lean Thinking* (Kanban)
  - Métodos Ágeis (Scrum)

Uma breve introdução

- Originária da década de 1950, a disciplina de gestão de projetos visa à “aplicação de conhecimentos, habilidades, ferramentas e técnicas às atividades de um projeto para encontrar os requisitos do projeto” (PMI, 2013, p.5), entendendo-se projeto como um empreendimento com o objetivo de produzir um produto ou serviço único e que apresenta um início e um fim (VERZUH, 2008).

Uma breve introdução

- Na Engenharia de Software o emprego da gestão de projetos apresenta resultados favoráveis, como uma forma de gerenciar diversos aspectos dos projetos;

- E conforme projetos se complexificaram várias abordagens de gestão de projetos foram concebidas.

Abordagens de gestão de projetos

- *Os primeiros métodos de gestão de projetos modernos foram elaborados para lidar com esses enormes projetos. Seus nomes – técnica de avaliação e revisão do programa (PERT) e método do caminho crítico (CPM) – ainda são bem conhecidos hoje. (VERZUH, 2008, p. 16)*
- Os métodos PERT e CPM são assim precursores e levaram à busca de novas abordagens de gestão, sendo algumas das mais atuais o *Lean Thinking*, os métodos ágeis e as abordagens dirigidas a processos.

Lean Thinking

- Também conhecido como “Pensamento Enxuto”, trata-se de um dos princípios que norteiam o Sistema de Produção Toyota;
- Fundamentado na Teoria das Restrições, segundo a qual toda organização apresenta pelo menos uma restrição (interna ou externa) que, de forma direta ou indireta, limita o seu desempenho (LEACH, 2000). A restrição seria, portanto, todo tipo de desperdício presente no processo.

Lean Thinking

- Categorias de desperdício:
  - Superprodução – produção em massa leva à produção de mais itens do que o necessário;
  - Espera – certas atividades da produção podem levar a desperdício enquanto se aguarda a conclusão de outras atividades antes de serem iniciadas;
  - Transporte – também o tempo e recursos envolvidos no transporte de materiais e pessoas são considerados;
  - Processamento – refere-se a desperdícios devido ao produto ou serviço passar por mais manipulações e processamentos do que o necessário;

Lean Thinking

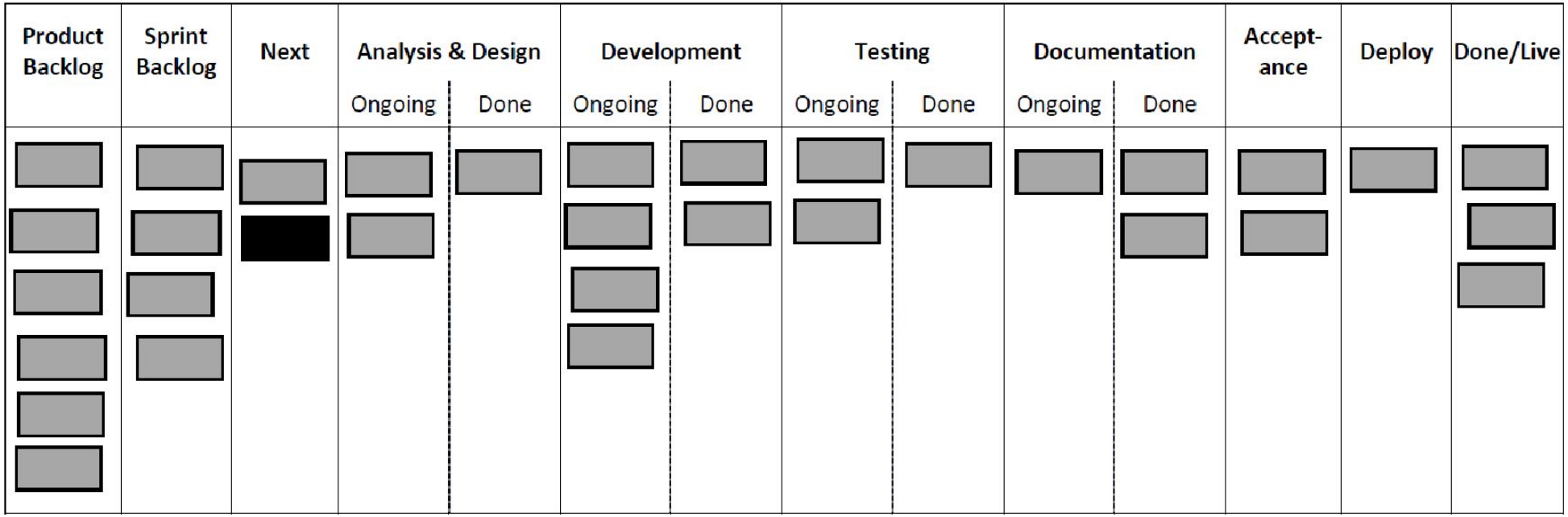
- Categorias de desperdício: (cont.)
  - Estoque – trata-se do desperdício gerado pelo acúmulo de itens em estoque;
  - Movimento – refere-se ao desperdício gerado por todo tipo de movimentação, seja humana, de dados etc.
  - Produtos com defeito – lida com os problemas de qualidade decorrentes de um processo de produção de má qualidade.
- Mironiuk (2012) destaca um oitavo tipo de desperdício: criatividade subutilizada.

Lean Thinking

- Deu origem ao *Lean Project Management*, que não define um novo processo, mas define cinco passos a serem empregados na melhoria do processo de gestão adotado:
  - Identificação do que é valor para o cliente;
  - Mapeamento do fluxo de valor e identificação dos desperdícios;
  - Eliminação dos desperdícios tornando o fluxo do processo contínuo, sem atrasos (*process flow*);
  - Aplicação do *pull system*, isto é, um sistema no qual o consumidor "puxa" os produtos ou serviços;
  - Busca da perfeição.
- Pode-se apontar a ferramenta Kanban como apresentando grande potencial na gestão de projetos de *software*.

Kanban

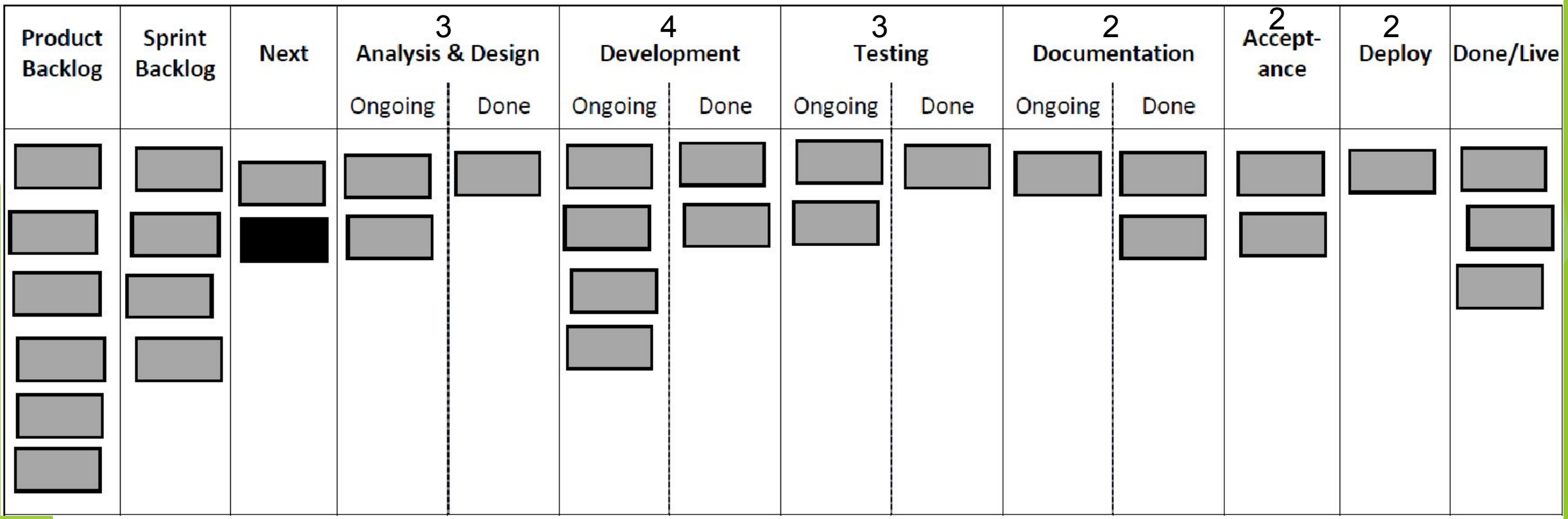
- Palavra de origem japonesa que significa “registro” ou “cartão visual”, lida com cartões para representar tarefas em um quadro que apresenta o fluxo normal das mesmas;
  - Principal artefato: *kanban board*, representação visual do estado em que se encontra cada tarefa de um processo (MOREIRA, 2011; MIRONIUK, 2012).



Exemplo de *kanban board* (MAHNIC, 2014)

Kanban

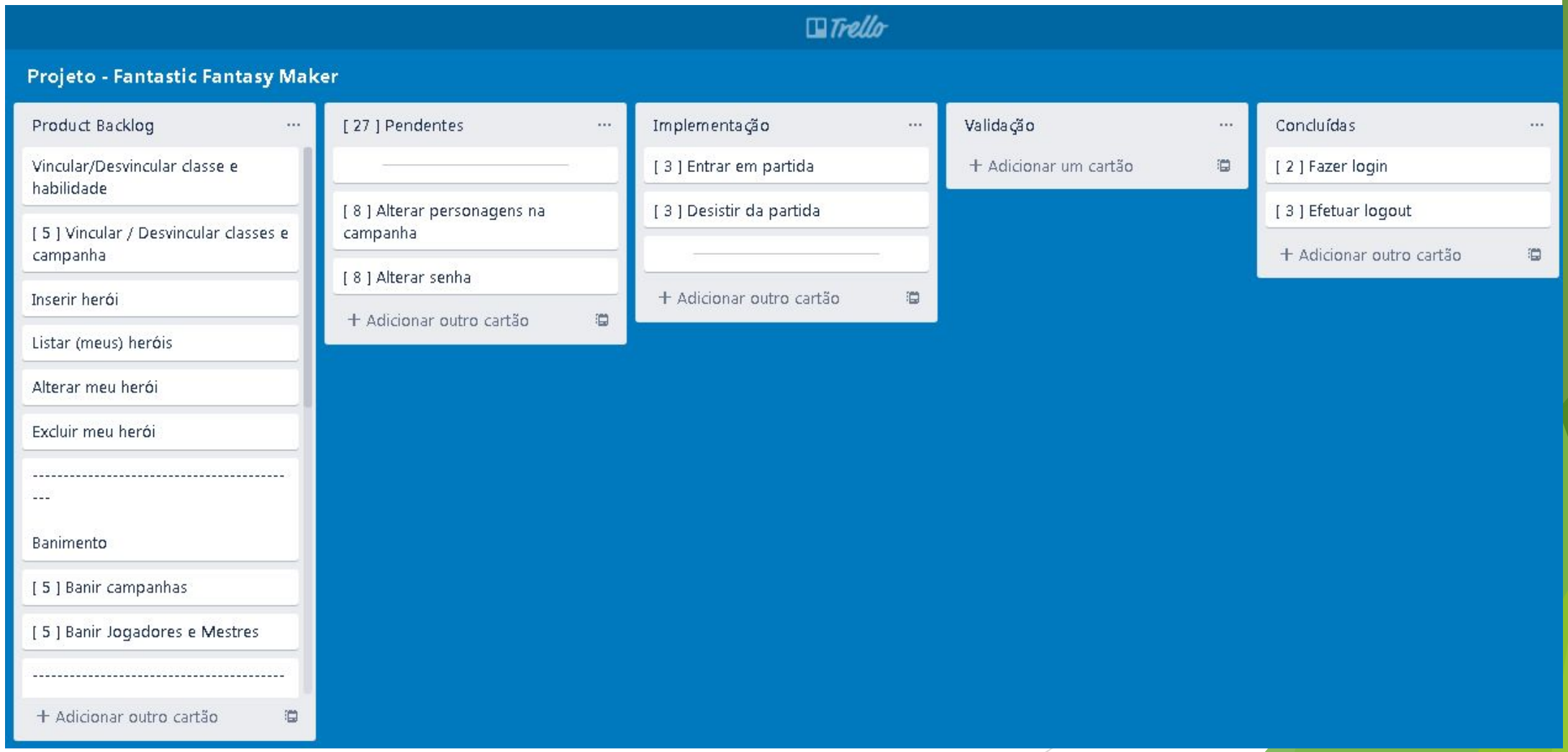
- Facilita a identificação de “gargalos” por meio do controle do *work in progress* (WIP), isto é, quantidade de trabalho em progresso que pode ser mantida em uma dada etapa (representada por meio de uma coluna).



Exemplo de *kanban board* (MAHNIC, 2014)

Kanban

- Atualmente é encontrado principalmente em sua versão eletrônica.



Métodos Ágeis

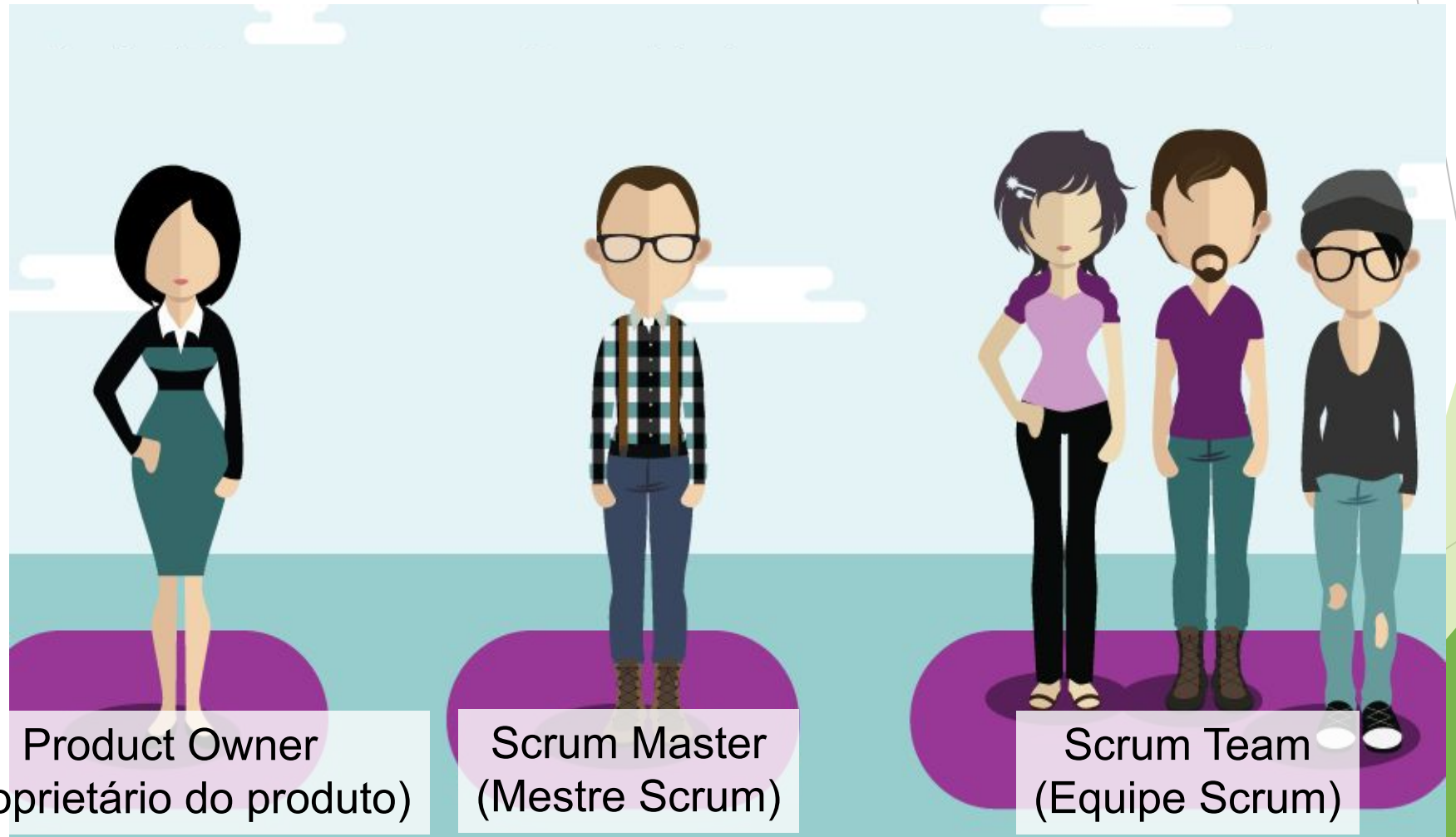
- Concebidos como um conjunto de métodos para desenvolvimento de *software*, deram origem ao conceito de *Agile Project Management* (APM), isto é, Gestão de Projetos Ágil (HIGHSMITH, 2009);
- No desenvolvimento de *software*, diversos métodos ágeis são propostos e explorados: Scrum, Extreme Programming, Crystal, FDD, Dynamic Systems Development Method etc.
- Entretanto, quanto à gestão de projetos ágil, é o *framework* Scrum que vem ganhando destaque.

Scrum

- *Framework* para desenvolvimento de *software* ágil, criado por Ken Schwaber e Jeff Shutterland no início da década de 1990;
  - Papel do gerente de projetos é um pouco diferente do tradicional;
  - E introduz novos artefatos e reuniões para controle do progresso do projeto (SCHWABER, 2004).

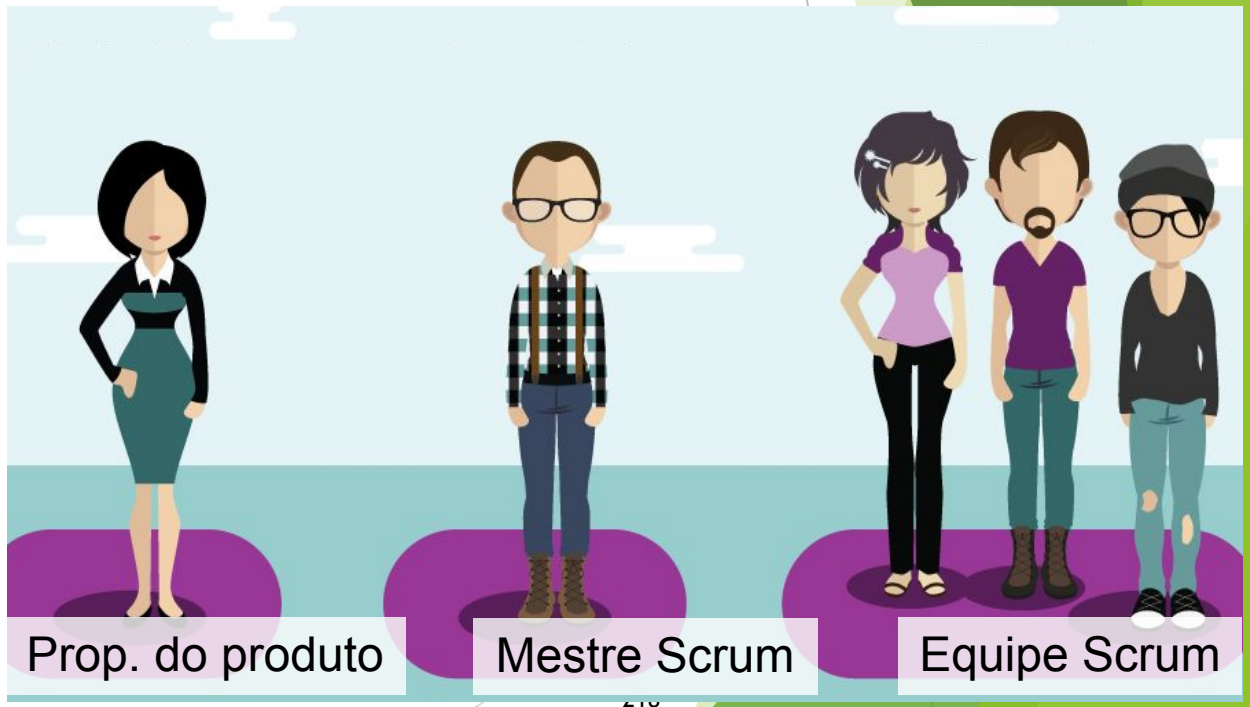
Scrum

- No Scrum, são três os papéis considerados:



Vamos praticar!

Defina o papel de cada um em sua equipe!



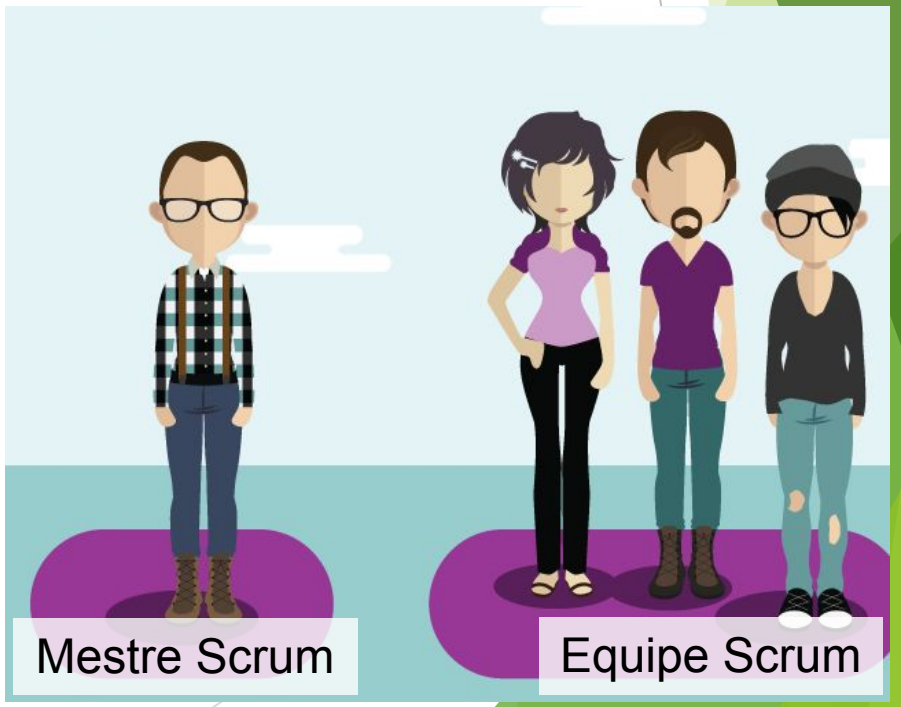
Scrum – Como estimar o esforço para cada tarefa?

- Uma possibilidade é o “Poker do Planejamento” (SUTHERLAND, 2016) – uma espécie de “jogo de cartas” para estimar o tempo/esforço para implementação de cada item do *backlog*;
- Cada membro possui cartas com os números 2, 3, 5, 8 e 13;
- Para cada item, cada membro escolhe uma carta para estimar o tempo da tarefa (de 2 – muito rápida, até 13 – muito demorada). Todos revelam suas cartas ao mesmo tempo;
- Caso dois membros tenham escolhido cartas muito distantes (2 e 8, 2 e 13 ou 3 e 13), eles devem explicar por que estimaram daquela forma e todos “jogam” suas cartas novamente;
- A estimativa para aquele item será a média dos valores.

Vamos praticar!

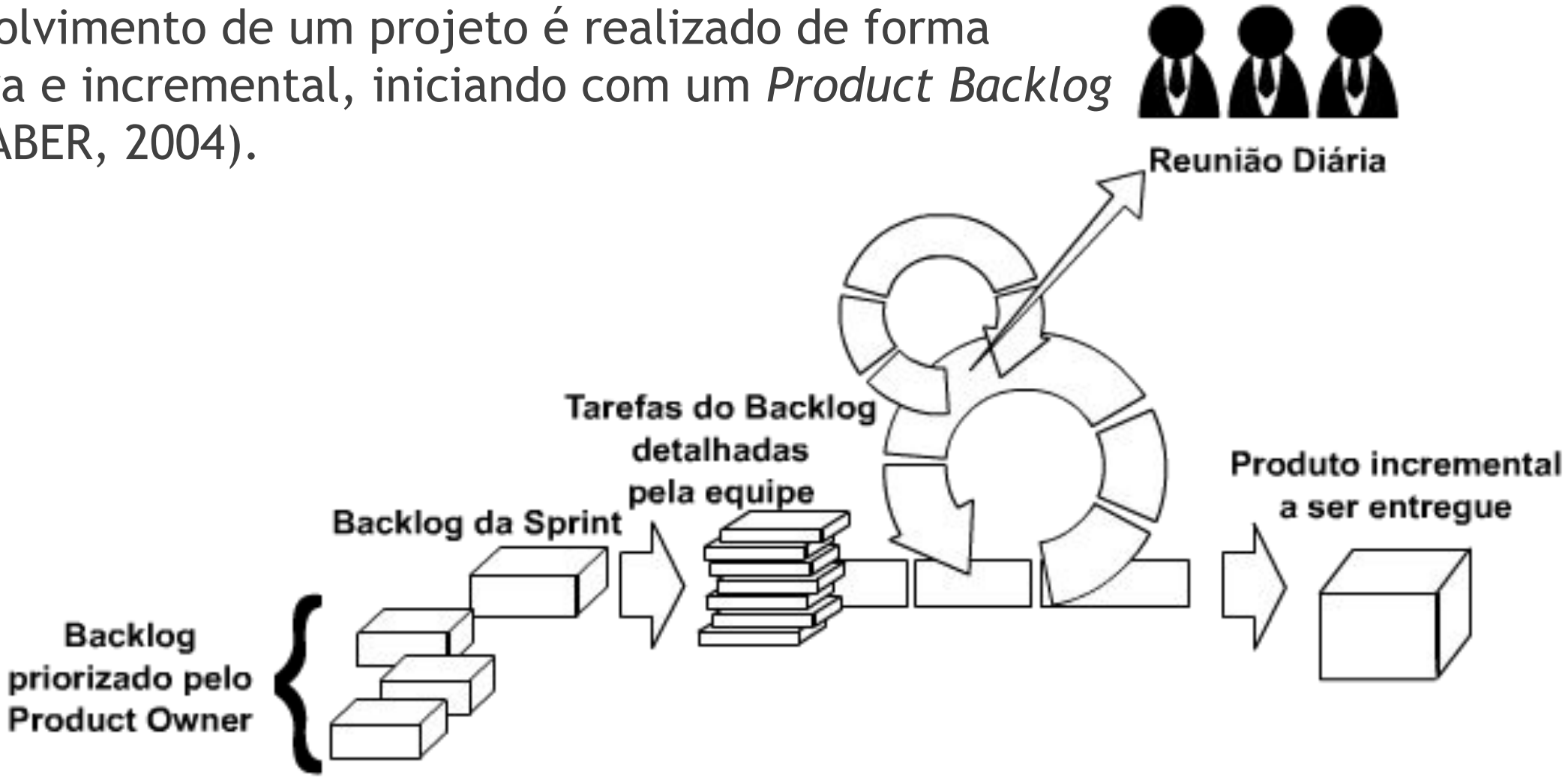
Equipe de desenvolvimento, já temos nosso *backlog* de produto (histórias de usuário), que tal estimarmos o tempo com o “poker do planejamento”?

	3
5	4,5
7,5	13
10	3



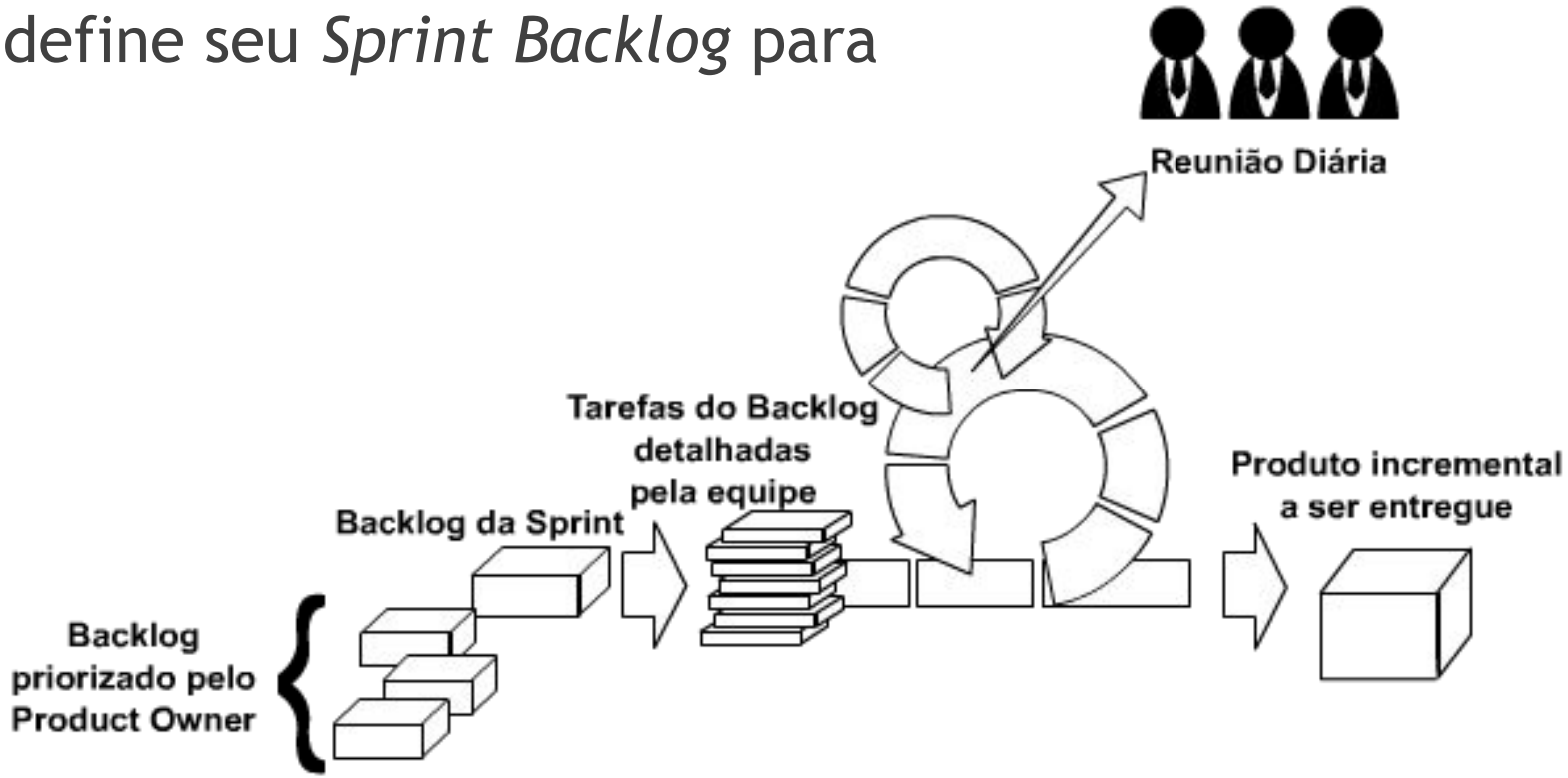
Scrum

- Desenvolvimento de um projeto é realizado de forma iterativa e incremental, iniciando com um *Product Backlog* (SCHWABER, 2004).



Scrum

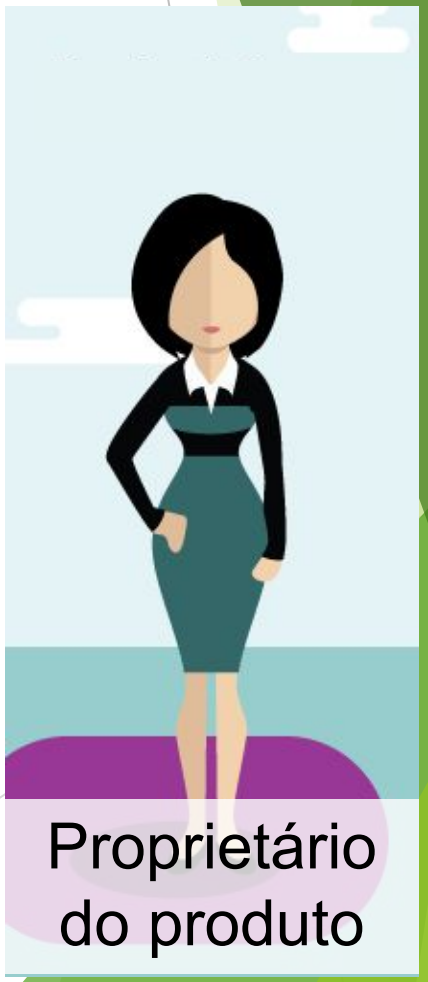
- Cada *sprint* (ciclo) inicia com *Sprint Planning Meeting*, isto é, reunião em que o *Product Owner* determinará, juntamente com a equipe, quais funcionalidades devem ser atendidas naquele ciclo. A equipe então, na segunda parte da mesma reunião, define seu *Sprint Backlog* para aquele ciclo.



Vamos praticar!

Proprietário do produto, priorize as 7 histórias de usuário de seu sistema.

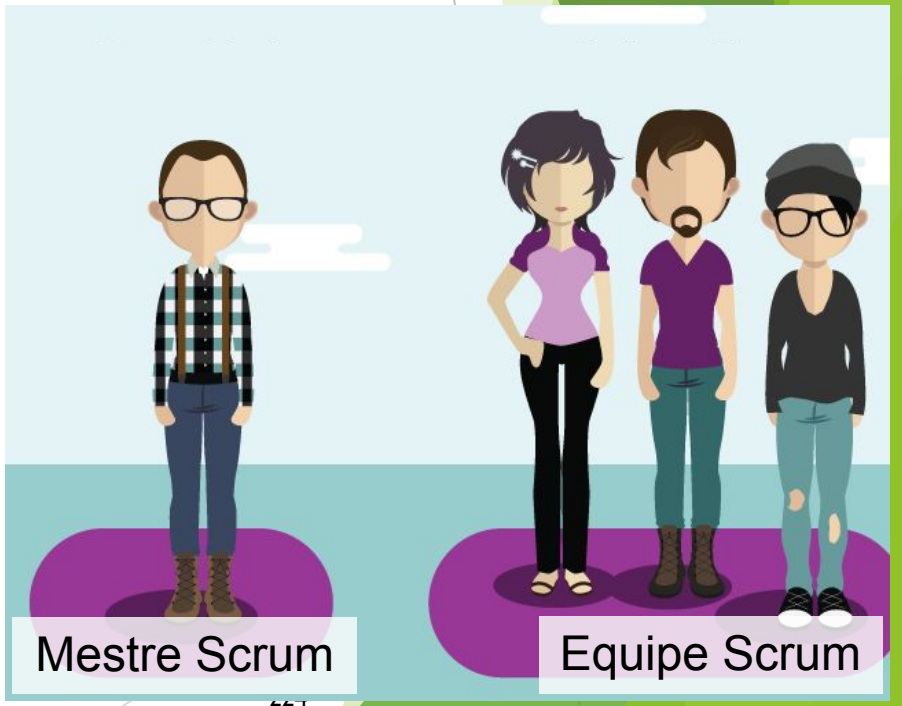
	2
1	5
3	6
7	4



Vamos praticar!

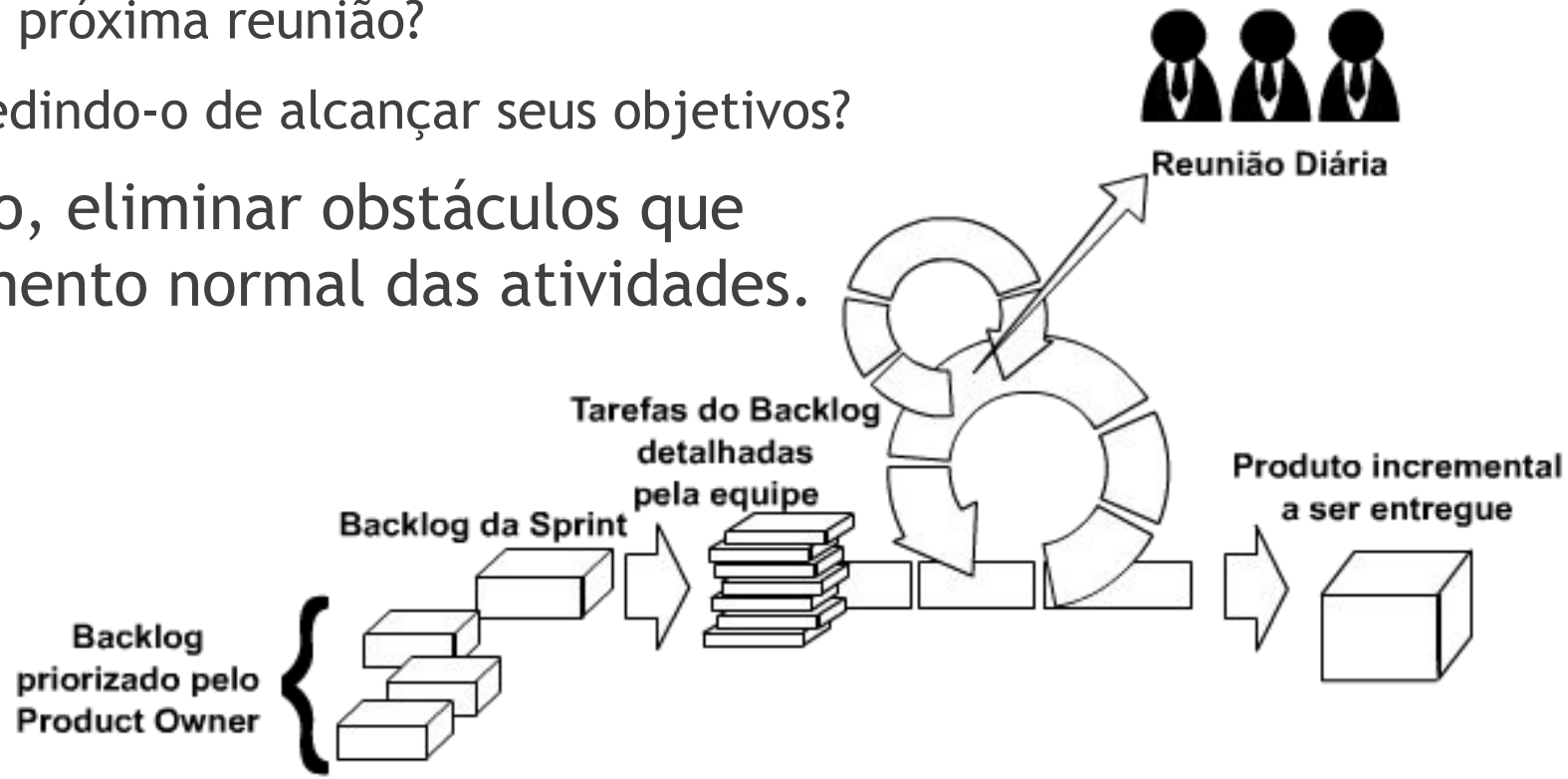
Equipe de desenvolvimento, escolha os itens que irão compor o backlog do ciclo.

	② 3 X
① 5 X	⑤ 4,5
③ 7,5	⑥ 13
⑦ 10	④ 3 X



Scrum

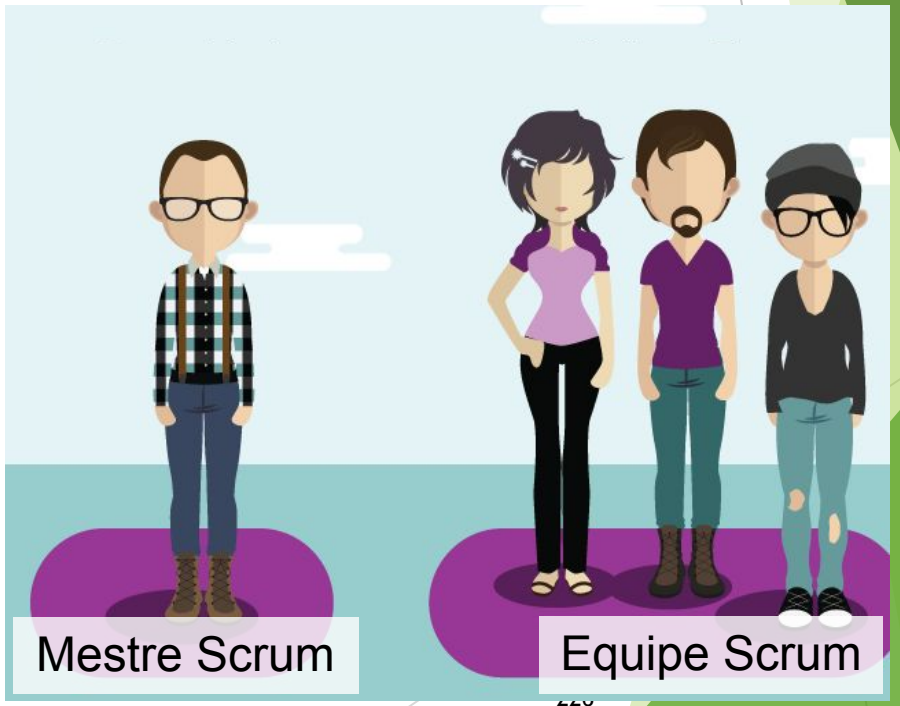
- Durante o desenvolvimento das atividades há reuniões diárias conhecidas como *Daily Scrum Meetings*;
  - O que foi feito desde a última reunião?
  - O que pretende fazer até a próxima reunião?
  - Que empecilhos estão impedindo-o de alcançar seus objetivos?
- *Scrum Master* pode, então, eliminar obstáculos que prejudicam o desenvolvimento normal das atividades.



Vamos praticar!

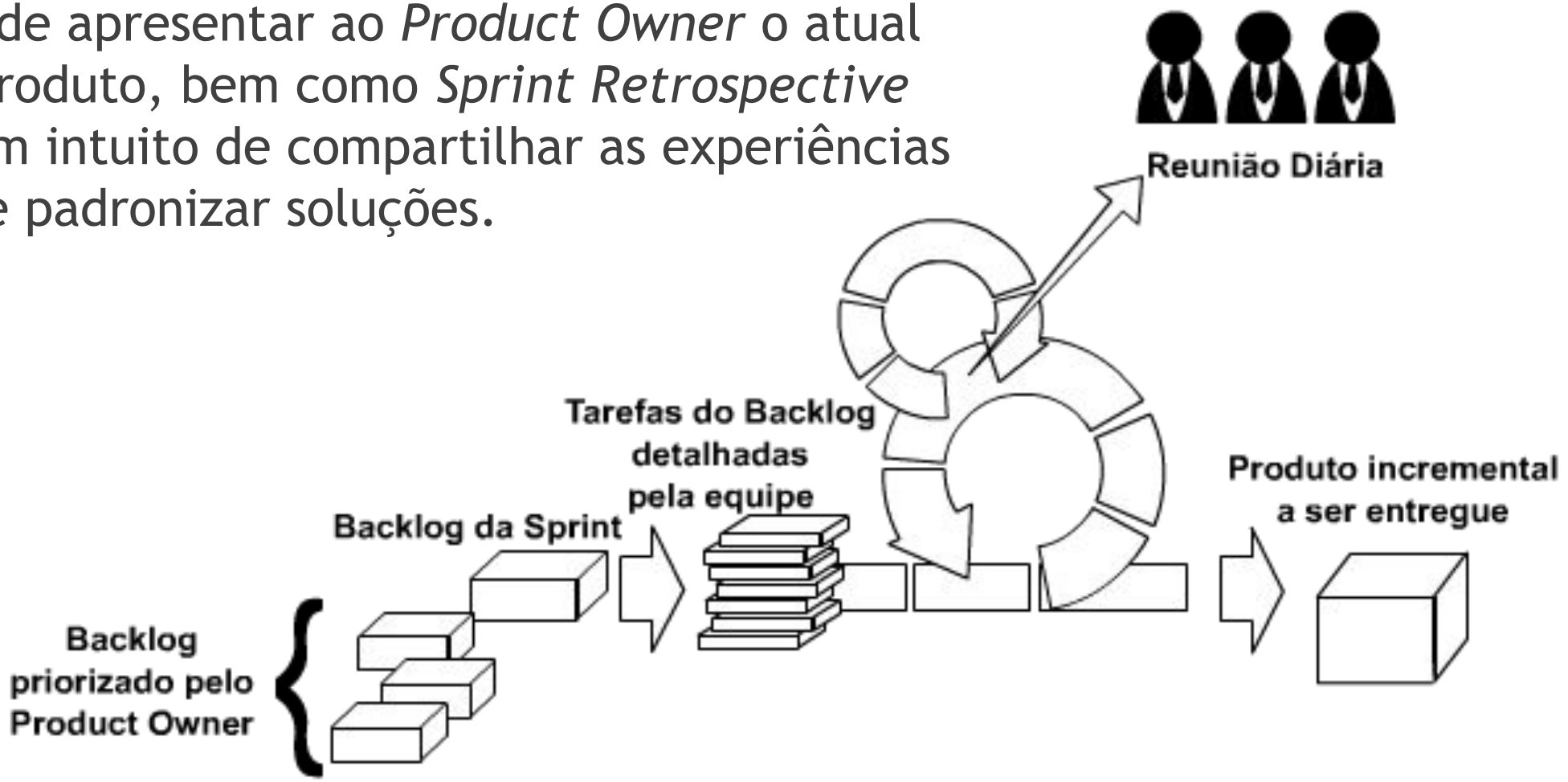
Suponha que o projeto já está andando há alguns ciclos. Mestre Scrum, faça as três perguntas a um dos membros da equipe.

- O que foi feito desde a última reunião?
- O que será feito até a próxima reunião?
- Quais obstáculos estão impedindo o seu avanço?



Scrum

- Ao final de cada *sprint*, realiza-se *Sprint Review Meeting* com intuito de apresentar ao *Product Owner* o atual estágio do produto, bem como *Sprint Retrospective Meeting*, com intuito de compartilhar as experiências aprendidas e padronizar soluções.



SAIBA MAIS!

Abordagens dirigidas a processos

- Diversas abordagens se apoiam na definição clara de processos, atividades e tarefas como forma de orientar a gestão de projetos. Dentre tais abordagens pode ser destacada o PMBOK.

SAIBA MAIS!

PMBOK

- Conjunto de boas práticas em gestão de projetos a partir da visão de profissionais e pesquisadores, reunidas em um guia pelo *Project Management Institute* (PMI), atualmente na 6ª edição (PMI, 2013);
- Foco em torno de 49 processos, reunidos em cinco grupos de processos e dez áreas de conhecimento (PMI, 2013).

SAIBA MAIS!

PMBOK

- Os cinco grupos de processos do PMBOK são:
  - Iniciar (initiating) – processos com o foco em definir um novo projeto ou nova fase de um projeto existente;
  - Planejar (planning) – processos que estabelecem o escopo total bem como desenvolvem o curso de ação necessário para alcançar os objetivos que satisfazem aquele escopo;
  - Executar (executing) – processos cuja execução visa completar o trabalho definido pelo plano de gestão do projeto, envolvendo a coordenação de pessoas e recursos;
  - Monitorar e Controlar (controlling) – processos cuja finalidade é monitorar, mensurar e revisar o progresso e desempenho do projeto;
  - Encerrar (closing) – processos realizados na conclusão de um projeto ou de uma fase do projeto.
- Esses grupos de processos são conhecidos como ciclo IPECC, comum em outras abordagens de gestão dirigidas a processos.

SAIBA MAIS!

PMBOK

- Dez áreas de conhecimento:
  1. Gestão de Integração;
  2. Gestão de Escopo;
  3. Gestão de Cronograma;
  4. Gestão de Custo;
  5. Gestão de Qualidade;
  6. Gestão de Recursos;
  7. Gestão de Comunicação;
  8. Gestão de Riscos;
  9. Gestão de Contratos;
  10. Gestão de *Stakeholders*.

SAIBA MAIS!

PMBOK

Áreas de conhecimento	Grupos de processo				
	Iniciar	Planejar	Executar	Monitorar e Controlar	Encerrar
1. Gestão de integração do projeto	1.1. Desenvolver *project charter*	1.2. Desenvolver plano de gestão do projeto	1.3. Dirigir e gerenciar o trabalho do projeto	1.4. Monitorar e controlar o trabalho do projeto 1.5. Realizar controle de mudança integrado	1.6. Encerrar projeto ou fase
2. Gestão de escopo do projeto		2.1. Planejar gestão de escopo 2.2. Coletar requisitos 2.3. Definir escopo 2.4. Criar *work breakdown sheet* (WBS)		2.5. Validar escopo 2.6. Controlar escopo	

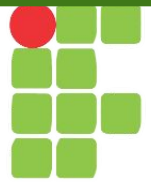
Processos de duas áreas de conhecimento do PMBOK
Fonte: (PMI, 2013, p. 61)

SAIBA MAIS!

Comparação das abordagens de gestão de projetos

Característica	*Lean Thinking*	Métodos Ágeis	Abordagens dirigidas a processos
Foco da abordagem	No valor para o *stakeholder* por meio da eliminação de desperdícios	No valor para o *stakeholder* por meio da resposta rápida à mudança	Nos processos
Instrumento empregado	Mapeamento do fluxo de valor	Desenvolvimento iterativo e incremental do produto	Execução de planos
Tamanho das equipes	Quaisquer equipes	Pequenas equipes	Grandes equipes
Complexidade na aprendizagem	Baixa	Baixa	Alta

Comparação das características das abordagens de gestão de projetos (SANTOS, 2014)



INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SERGIPE

SAIBA MAIS!

Princípios na prática de Engenharia de Software

Parte 07

SAIBA MAIS!

Sumário

- Conhecimento em Engenharia de Software
- Princípios fundamentais
- Princípios das atividades metodológicas
  - Princípios da comunicação
  - Princípios de planejamento
  - Princípios de modelagem
  - Princípios de construção
  - Princípios de disponibilização

SAIBA MAIS!

Conhecimento em Engenharia de Software

- Pode-se dividir o conhecimento acumulado em Engenharia de Software em dois grandes grupos:
  - Conhecimentos orientados a tecnologias específicas – giram em torno do domínio de certas tecnologias por parte do engenheiro de software, apresentando um tempo de vida curto (“meia-vida” de três anos, segundo a IEEE Software);
  - Conhecimentos orientados a princípios – focam em princípios que orientam a prática e constituem uma base de conhecimentos que sofre menos alterações ao longo do tempo.

SAIBA MAIS!

Conhecimento em Engenharia de Software

- Pressman (2011) classifica os princípios que norteiam a prática de Engenharia de Software em duas categorias:
  - Princípios fundamentais;
  - Princípios das atividades metodológicas.

SAIBA MAIS!

Princípios fundamentais

- Oferecem suporte à:
  - Aplicação de um **processo** de software, orientando a equipe de software quanto ao desenvolvimento de atividades de apoio e estruturais;
  - Execução de métodos (**prática**) de Engenharia de Software, estabelecendo artefatos e regras para guiar a análise, projeto, implementação, testes e emprego do software.

SAIBA MAIS!

Princípios fundamentais

1. **Seja ágil:**
   - Não importa se você opta por um modelo de processo prescritivo ou ágil, é sempre possível aplicar os princípios da agilidade (isto é, que visam à pronta resposta em caso de mudanças).
   - E apesar de mudanças serem bem-vindas (mudanças são importantes a fim de que um software sobreviva no longo prazo), são necessários métodos para sua solicitação, avaliação e implementação.

SAIBA MAIS!

Princípios fundamentais

2. **Esteja pronto para adaptações:**
   - Cada projeto de software pode requerer mudanças quanto ao processo de desenvolvimento do mesmo, então não se preocupe caso necessite realizar adaptações ao modelo de processo de software escolhido – na verdade, pode ser um sinal de amadurecimento da equipe!

SAIBA MAIS!

Princípios fundamentais

3. **Estabeleça mecanismos para comunicação e coordenação:**
   - A chave para o sucesso de qualquer processo encontra-se na comunicação entre todos os *stakeholders* (pessoas ou entidades envolvidas: desenvolvedores, gerente, clientes, usuários finais etc.) bem como na coordenação das atividades;

SAIBA MAIS!

Princípios fundamentais

4. **Concentre-se na qualidade em todas as etapas:**
   - No desenvolvimento de um software, não se deve buscar qualidade somente na etapa de testes! Qualidade deve ser visada desde as primeiras atividades de planejamento, estabelecendo métricas e meios de avaliá-la.

SAIBA MAIS!

Princípios fundamentais

5. **Esforce-se por consistência:**
   - Ser consistente na execução das atividades metodológicas bem como na geração dos artefatos permite uma melhor compreensão dos mesmos por terceiros bem como conduz a resultados mais rápidos. Assim, seja na elaboração de modelos, da documentação (técnica ou de usuário), no projeto de interfaces de usuário ou na codificação, busque sempre consistência.

SAIBA MAIS!

Princípios fundamentais

6. **Modularize:**
   - Ao projetar um software, adote a estratégia “dividir para conquistar”, dividindo-o em partes menores (módulos), o que facilitará a sua compreensão, modelagem, implementação e validação.

SAIBA MAIS!

Princípios fundamentais

7. **Foque em facilidade de manutenção do software:**
   - Mesmo que você seja o responsável pela manutenção do software após a conclusão do mesmo, foque em todas as atividades do desenvolvimento na construção de artefatos que podem ser mantidos por qualquer um da melhor forma possível. Isso implica em máxima legibilidade do código, modelos em conformidade, documentação suficiente etc.

SAIBA MAIS!

Princípios fundamentais

8. **Esforce-se por colaboração de todos os interessados:**
   - "O todo é maior do que a soma das partes". Quando todos os membros de uma equipe colaboram uns com os outros na execução de suas tarefas, o resultado final é sempre muito melhor do que aquele em que cada tarefa é executada de forma totalmente individual. A colaboração é a chave para o processo de compartilhamento do conhecimento gerado durante o processo (aprendizagem) e por detectar e eliminar erros (qualidade).

SAIBA MAIS!

Princípios fundamentais

9. **Gerencie as expectativas do cliente quanto ao software:**
   - Muitas vezes, geralmente devido a ruídos na comunicação, erros de interpretação ou “prometer mais do que se pode cumprir”, os clientes podem ter expectativas sobre o software muito além do que a equipe de desenvolvimento espera suprir na próxima entrega. O engenheiro de software deve, então, gerenciar as expectativas dos clientes a fim de que eles compreendam melhor o que podem esperar.

SAIBA MAIS!

Princípios das atividades metodológicas

- São focados em cada uma das atividades metodológicas genéricas;
- São agrupados em:
  - Princípios da comunicação;
  - Princípios do planejamento;
  - Princípios da modelagem;
  - Princípios da construção;
  - Princípios do emprego.

SAIBA MAIS!

Princípios da comunicação

1. **Prepare-se antes de se comunicar:**
   - Antes de uma reunião com clientes, usuários ou outros membros da equipe de desenvolvimento, tome o tempo necessário para preparar-se. Identifique os objetivos da reunião, levante possíveis perguntas e entenda jargões e termos técnicos que fazem parte do negócio em questão. Se você é o responsável pela reunião, prepare antecipadamente a agenda da reunião e envie-a para cada um dos participantes.

SAIBA MAIS!

Princípios da comunicação

2. **Concentre-se mais em ouvir do que em buscar respostas:**
   - Um erro que muitos desenvolvedores cometem em reuniões para elicitação de requisitos é estarem mais preocupados com quais soluções tecnológicas podem resolver o problema em vez de prestarem atenção à descrição do mesmo pelo cliente, gerando ruído na comunicação e, assim, falha na interpretação de requisitos, transformando-se numa “bola de neve”.

SAIBA MAIS!

Princípios da comunicação

3. **Documente as decisões:**
   - Em todas as reuniões, por mais informais que possam parecer, preocupe-se sempre em documentar todas as decisões (e as razões pelas quais foram tomadas). Isso evitará que uma decisão importante do projeto seja esquecida ou distorcida ao longo do mesmo.

SAIBA MAIS!

Princípios da comunicação

4. **Mantenha o foco por meio de uma discussão modular:**
   - Reuniões geralmente visam mais de um tópico a ser tratado e, conforme o número de participantes da mesma cresce, é muito comum perder-se o foco e, assim, não alcançar os resultados esperados. A fim de evitar isso, mantenha a conversação em um formato modular, discutindo todos os aspectos de um tópico antes de passar para o seguinte.

SAIBA MAIS!

Princípios da comunicação

5. **Em uma negociação, todos devem sair ganhando:**
   - A fim de que uma reunião seja considerada bem sucedida, é importante que todos os envolvidos (clientes, usuários finais, desenvolvedores, gerente etc.) saiam com a sensação de que “saíram ganhando” da mesma. Assim sendo, toda vez que surgir um conflito de interesses, uma negociação deve ser realizada com o intuito de beneficiar todos os envolvidos.

SAIBA MAIS!

Princípios do planejamento

1. **Compreenda o escopo do projeto:**
   - Um dos resultados de uma atividade de planejamento bem sucedida é a compreensão objetiva e não ambígua do escopo do projeto. Assim sendo, deve-se buscar compreender as necessidades do negócio, o domínio do problema em questão e os objetivos do projeto.

SAIBA MAIS!

Princípios do planejamento

2. **Faça estimativas com base no que conhece:**
   - Um erro cometido por muitos desenvolvedores de software é realizar estimativas de custo e cronograma para tarefas sem nenhuma base prática. Tais estimativas tornam-se muito vagas e imprecisas, levando muitas vezes a subestimar prazos e orçamentos.

SAIBA MAIS!

Princípios do planejamento

3. **Considere os riscos ao definir o plano:**
   - Todo processo de desenvolvimento de software pode ser alvo de ameaças internas ou externas que podem levar a atrasos de cronograma, aumento dos custos ou dificuldades na implementação de um software que abranja todo o escopo definido. Assim, deve ser levantado o maior número possível de riscos do projeto e apresentado no plano, bem como possíveis estratégias de soluções para os mesmos.

SAIBA MAIS!

Princípios do planejamento

4. **Defina como se pretende garantir a qualidade:**
   - Como já foi afirmado anteriormente, deve-se garantir a qualidade do software em todas as etapas. No planejamento, devem-se elencar métodos, técnicas e artefatos a serem adotados no controle e garantia da qualidade, como a adoção da programação em pares, revisões técnicas, desenvolvimento dirigido a testes etc.

SAIBA MAIS!

Princípios do planejamento

5. **Reconheça que o planejamento é iterativo:**
   - Todo plano sofrerá alterações ao longo do processo de desenvolvimento. Isso é visível principalmente em modelos de processo iterativos, uma vez que a cada nova iteração uma nova atividade de planejamento é iniciada, entretanto isso também ocorre em outros modelos de processo. Assim, é importante que o engenheiro de software esteja pronto para realizar os ajustes necessários ao seu planejamento no decorrer do projeto.

SAIBA MAIS!

Princípios da modelagem

1. **O objetivo principal da equipe de software é construir software, não criar modelos:**
   - Modelos devem ser utilizados como ferramentas para facilitar a construção de software que atenda às necessidades do cliente. Assim sendo, crie somente modelos que possuem um propósito claro no processo de desenvolvimento e agreguem valor ao mesmo. Além disso, tente mantê-los tão simples e fáceis de sofrer alterações quanto for possível e fuja da tentação de “construir modelos perfeitos”.

SAIBA MAIS!

Princípios da modelagem

2. **Não seja muito rígido quanto à sintaxe do modelo. Se esta transmite o conteúdo com sucesso, a representação é secundária:**
   - Mais uma vez, o objetivo principal é a construção do software, não a criação de modelos. Então mesmo se a sintaxe do modelo não está correta, mas se a ideia representada pelo modelo está bem clara, sua missão foi cumprida com sucesso;
   - Obs: Desenvolvimento Dirigido a Modelos (MDD).

SAIBA MAIS!

Princípios da modelagem

3. **Obtenha *feedback* sobre os modelos o quanto antes:**
   - A fim de garantir que os modelos desenvolvidos respondem às expectativas dos clientes, é importante que os mesmos sejam **revisados** por outros membros da equipe de software e **validados** junto aos clientes.

SAIBA MAIS!

Princípios da modelagem

4. **O universo de informações (domínio) de um problema deve ser representado e compreendido:**
   - A modelagem do problema e das informações que giram em torno do mesmo (dados sobre usuários, subsistemas, sistemas externos etc.) facilita a compreensão do problema bem como a busca por possíveis soluções que atendam às necessidades específicas daquele projeto.

SAIBA MAIS!

Princípios da modelagem

5. **Análise deve partir da informação essencial (nível mais alto) para o detalhamento da implementação (nível mais baixo):**
   - Começar de um nível mais alto permite abstrair muitos detalhes a fim de concentrar-se no problema e na solução proposta em si, independentemente de detalhes como plataforma ou ambiente que será empregado, por exemplo. Conforme mais detalhes sobre a solução são apresentados, aspectos relevantes para a implementação da mesma são adicionados aos novos modelos criados.

SAIBA MAIS!

Princípios da modelagem

6. **As funções que o software desempenha devem ser bem definidas:**
   - A partir do domínio do problema e das necessidades do projeto, podem ser modeladas as funções exequíveis pelo software em níveis de maior ou menor abstração como parte da solução. Tais representações das funções podem ser utilizadas para compreender melhor o funcionamento do software a ser implementado.

SAIBA MAIS!

Princípios da modelagem

7. **Considere a arquitetura do sistema a ser construído:**
   - A arquitetura serve como alicerce e estrutura-base para a construção de todo o *software*, assim, a modelagem e tomada de decisões referentes à mesma deve ser o primeiro passo no processo de desenvolvimento. Escolhas ruins quanto à arquitetura do software podem levar a problemas na adequação de estruturas de dados, interfaces e fluxo de controle de programas.

SAIBA MAIS!

Princípios da modelagem

8. **O projeto no nível de componentes deve ser funcionalmente independente:**
   - Seguindo a ideia de modularização do software, cada componente deve ser projetado como um módulo independente dos demais. Tal estratégia de “dividir para conquistar” possui como benefício a redução da complexidade do software, facilidade de reúso dos módulos (componentes) em outros sistemas e possibilita que cada desenvolvedor trabalhe em um módulo distinto, causando pouco impacto sobre as atividades dos demais.

SAIBA MAIS!

Princípios da modelagem

9. **Modelos devem ser de fácil compreensão:**
   - Dois dos objetivos dos modelos são facilitar a comunicação entre todos os envolvidos e orientar a execução das atividades seguintes. Projetos mal modelados ou de difícil compreensão podem levar a falhas na comunicação entre os desenvolvedores, atrasos na execução das tarefas e criação de modelos subsequentes em menor conformidade, levando assim à codificação de um software que não atende adequadamente às necessidades dos usuários.

SAIBA MAIS!

Princípios da construção

1. **Compreenda bem o problema a ser solucionado:**
   - Não inicie a codificação sem compreender bem o problema ou tarefa que está sendo encarado;
   - Pode-se, entretanto, adotar a abordagem “dividir para conquistar” para reduzir o problema a partes menores, compreendê-las individualmente e, então, iniciar a codificação de cada parte.

SAIBA MAIS!

Princípios da construção

2. **Escolha uma linguagem e ambiente de programação adequados às necessidades do *software* e do ambiente onde irá operar:**
   - Após a compreensão do problema, já é possível estudar as opções de linguagens e ferramentas para desenvolvimento disponíveis no mercado que possam atender as necessidades do projeto e escolher aquelas que oferecem melhor suporte para as mesmas.

SAIBA MAIS!

Princípios da construção

3. **Alinhe os testes com os requisitos do cliente:**
   - Testes são ferramentas para verificação (de erros e conformidade entre modelo) e validação (dos requisitos) do software desenvolvido e devem ser elaborados de acordo com as especificações dadas pelo cliente (casos de uso, histórias de usuário etc.).

SAIBA MAIS!

Princípios da construção

4. **Planeje os testes antes da codificação do módulo em questão:**
   - A elaboração dos testes antes de iniciar a codificação evita elaborar casos de testes viciados, focados somente naquilo que foi efetivamente implementado, em vez de ter o foco naquilo que foi pedido;
   - Além disso, o planejamento antecipado dos testes ajuda a compreender melhor o domínio do problema sendo resolvido, contribuindo assim com uma codificação mais eficiente.

SAIBA MAIS!

Princípios da construção

5. **Comece com testes para casos particulares e então generalize:**
   - Planejar testes generalizados são mais complexos e passíveis de não cobrir alguns casos. Seguindo o caminho inverso, isto é, pensando-se nos possíveis domínios de dados de entrada do problema e elaborando casos de testes para grupos de domínios há uma maior chance de **cobrir razoavelmente** a maioria dos casos.

SAIBA MAIS!

Princípios da construção

6. **Considere o uso de programação em pares:**
   - A alocação de um par de programadores em cada tarefa em vez de um único programador permite um maior foco sobre a mesma, o que refletirá em maior qualidade na medida em que mais casos de testes serão elaborados, mais erros serão identificados etc.
   - Tal decisão não é um desperdício de recursos, pois além de aumentar a produtividade e a velocidade de conclusão das tarefas, cada programador fica encarregado de uma parte diferente. Exemplo: um codifica e outro revisa o código.

SAIBA MAIS!

Princípios da construção

7. **Domine a arquitetura do *software* e crie interfaces consistentes com ela:**
   - Uma abordagem de codificação *top-down*, isto é, “de cima para baixo”, começando pela arquitetura para só depois implementar seus componentes, permite buscar maior compatibilidade entre estes e aquela. Entretanto, para que tais componentes possam ser facilmente reusáveis neste e em outros projetos, deve-se atentar à criação de interfaces consistentes.

SAIBA MAIS!

Princípios da construção

8. **Priorize a legibilidade e manutenibilidade do seu código:**
   - Escreva todo o seu código considerando que outra pessoa (ou mesmo empresa) o manterá mais tarde! Quanto melhor for a legibilidade, modularidade e consistência do código, menores serão os custos para realizar correções e incrementos no mesmo.

SAIBA MAIS!

Princípios da construção

9. **Codifique de forma que seja "autodocumentável":**
   - Atualmente, há ferramentas de desenvolvimento capazes de gerar documentação específica a partir de comentários estruturados dentro do próprio código do projeto. Além disso, empregar nomes de variáveis e funções que explicitem o papel de cada uma delas facilita a compreensão do código.
     - Ex: JavaDoc.

SAIBA MAIS!

Princípios da construção

10. **Aplique uma revisão do código quando for apropriado:**
    - A revisão de código é um dos métodos que podem ser empregados no controle e garantia da qualidade do *software*. Nela, um membro da equipe deve revisar o código de outro com o intuito de encontrar possíveis casos não cobertos, problemas de otimização, *bugs* etc.
    - Outro método empregável é a inspeção de código, que pode adotar *checklists* para analisar o mesmo quanto a diversos itens referentes à cobertura do escopo, otimização, desempenho, segurança etc.

SAIBA MAIS!

Princípios da construção

11. **Realize testes de unidade, integração e regressão (e corrija os erros encontrados):**
    - Os testes unitários, elaborados antes do processo de codificação, são empregados com o intuito de verificar e validar o código;
    - Após a integração do novo módulo ao sistema em desenvolvimento, devem ser aplicados também testes de integração para garantir que o módulo funciona adequadamente com todo o restante;
    - Caso um módulo tenha sofrido alterações, testes de regressão também devem ser feitos.

SAIBA MAIS!

Princípios da construção

12. **Refaça a codificação:**
    - A refatoração permite melhorias incrementais no código visando objetivos específicos, geralmente focando melhoria de desempenho e facilidade de manutenção. Tais melhorias podem ser alcançadas aplicando-se um subconjunto dos diversos tipos de refatoração (*refactoring*).

Princípios da construção

SAIBA MAIS!

E lembre-se:

"Um teste bem sucedido é aquele que revela um novo erro!"

SAIBA MAIS!

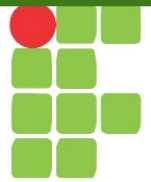
Princípios do emprego

1. **Ofereça documentação e instruções suficientes para o seu uso:**
   - É responsabilidade da equipe de desenvolvimento oferecer todo o suporte necessário para o emprego do software ou de cada incremento do mesmo, não sendo portanto admissível a entrega do mesmo sem as devidas instruções sobre seu uso bem como a documentação necessária para tornar o conhecimento sobre seu uso explícito.

SAIBA MAIS!

Princípios do emprego

2. ***Software* com *bugs* (erros) deve, primeiro, ser corrigido e, depois, entregue:**
   - Por mais apertados que sejam os prazos, entregar um *software* com bugs ou não testado suficientemente não é uma boa opção. Erros disparados no ambiente de produção podem ser vistos pelo cliente como "amadorismo" da equipe de desenvolvimento, além de poderem causar problemas ocultos nos dados armazenados pelo mesmo, levando até mesmo a prejuízos no negócio em questão.



INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SERGIPE

SAIBA MAIS!

Exercícios

Princípios na Prática de Engenharia de Software

Parte 07

SAIBA MAIS!

Enumere

1. Comunicação;
2. Planejamento;
3. Modelagem;
4. Construção.

( ) Prepare-se antes de se comunicar;

( ) Objetivo principal da equipe de software é construir software, não criar modelos;

( ) Realize testes de unidades e corrija os erros encontrados;

( ) Faça estimativas com base no que conhece.

SAIBA MAIS!

Enumere

1. Comunicação;
2. Planejamento;
3. Modelagem;
4. Construção;
5. Emprego.

( ) Compreenda o escopo do projeto;

( ) Obtenha feedback sobre os modelos o quanto antes;

( ) Aplique uma revisão do código quando for apropriado;

( ) Em uma negociação, todos devem sair ganhando;

( ) Ofereça documentação e instruções suficientes para o uso do software.

SAIBA MAIS!

Enumere

1. Comunicação;
2. Planejamento;
3. Modelagem;
4. Construção;
5. Emprego.

( ) Software com bugs deve, primeiro, ser corrigido e, depois, entregue;

( ) O projeto no nível de componentes deve ser funcionalmente independente;

( ) Planeje os testes antes da codificação do módulo em questão;

( ) Considere os riscos ao definir o plano;

( ) Concentre-se mais em ouvir do que em buscar respostas.

SAIBA MAIS!

Responda

Quais os dois grandes grupos de princípios que norteiam a prática da Engenharia de Software?

SAIBA MAIS!

Responda

Tendo em vista os princípios apresentados em sala de aula, descreva como deve ser conduzida(o):

a) Comunicação;

b) Planejamento;

c) Modelagem;

d) Construção;

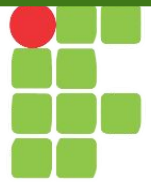
e) Disponibilização.

SAIBA MAIS!

Software Engineering Body Of Knowledge (SWEBOK)

- Documento que reúne os principais conhecimentos em torno da área de Engenharia de Software;
- Pode ser encontrado na *web* em:

https://www.computer.org/education/bodies-of-knowledge/software-engineering



INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SERGIPE

SAIBA MAIS!

Teste de Software

Parte 08

SAIBA MAIS!

Sumário

- Introdução ao Teste de Software
- Níveis de Teste
- Teste Caixa-Branca x Teste Caixa-Preta
- Testes Alfa x Testes Beta
- Testes de Software Orientado a Objetos

SAIBA MAIS!

Uma breve introdução

SAIBA MAIS!

Uma breve introdução

- Teste de software corresponde ao conjunto de ações e tarefas que visa identificar possíveis falhas em um software;
  - Um teste bem sucedido é aquele que encontra erros!
  - Não se pode garantir que um software esteja 100% livre de erros!
- O processo de um teste começa em seu planejamento (tomando como base a elicitação de requisitos, descrições de casos de uso etc.) até a execução dos testes e coleta de dados para análise.

SAIBA MAIS!

Uma breve introdução

- Possíveis causas de uma falha:
  - Especificação errada ou incompleta;
  - Requisitos impossíveis de serem implementados, devido a limitações de hardware ou software;
  - Implementação errada ou incompleta.
- Por meio da **verificação** será analisado se o produto foi feito corretamente, se ele está de acordo com os requisitos especificados. Por meio da **validação** será analisado se foi feito o produto correto, se ele está de acordo com as necessidades e expectativas do cliente.

SAIBA MAIS!

Uma breve introdução

- Teste de software contribui, assim, para o processo de qualidade de software;
- Atributos qualitativos segundo a norma ISO 9126:
  - Funcionalidade;
  - Confiabilidade;
  - Usabilidade;
  - Eficiência;
  - Manutenibilidade;
  - Portabilidade.

SAIBA MAIS!

Conceitos

- **Defeito:** ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta. Por exemplo, uma instrução ou comando incorreto;
- **Erro:** manifestação concreta de um defeito num artefato de software. Diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui um erro;
- **Falha:** comportamento operacional do software diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos erros e alguns erros podem nunca causar uma falha.

Teste de software revela simplesmente falhas em um produto. Após a execução dos testes é necessária a execução de um processo de depuração para a identificação e correção dos defeitos que originaram essa falha!

Conceitos

SAIBA MAIS!

- **Plano de Teste:** Documento que detalha o planejamento dos testes referentes ao software em desenvolvimento. Contém instruções detalhadas sobre como um teste deve ser planejado, executado e seu resultado analisado;
- **Caso de Teste:** Descrição de um teste a ser executado. Cada caso de uso pode envolver um ou mais casos de teste (exemplo: cálculo de um fatorial). É composta por valores de entrada, restrições para sua execução e o resultado ou comportamento esperado;
- **Script de Teste:** Automação da execução de um caso de teste. Pode ser implementada por meio de ferramentas específicas para testes ou “classes testadoras”.

SAIBA MAIS!

Níveis de Teste de Software

- Testes de unidade ou unitários;
- Testes de integração;
- Testes de sistema;
- Testes de aceitação.

SAIBA MAIS!

Testes de Unidade

- Visam explorar a menor unidade do projeto, seja em nível de módulo (classe), em nível de método ou até mesmo de um fragmento de código menor;

SAIBA MAIS!

Testes de Integração

- Visam identificar possíveis falhas na integração dos diferentes módulos componentes de um sistema ou na comunicação entre os mesmos;

SAIBA MAIS!

Testes de Sistema

- Avaliam o software em busca de falhas por meio da utilização do mesmo, como se fosse um usuário final;
- Os testes devem ser executados em ambientes e condições similares às que o usuário empregará o software em seu dia-a-dia.

SAIBA MAIS!

Testes de Aceitação

- São realizados geralmente por um restrito grupo de usuários finais do sistema. Esses simulam operações de rotina do sistema de modo a verificar se seu comportamento está de acordo com o solicitado.

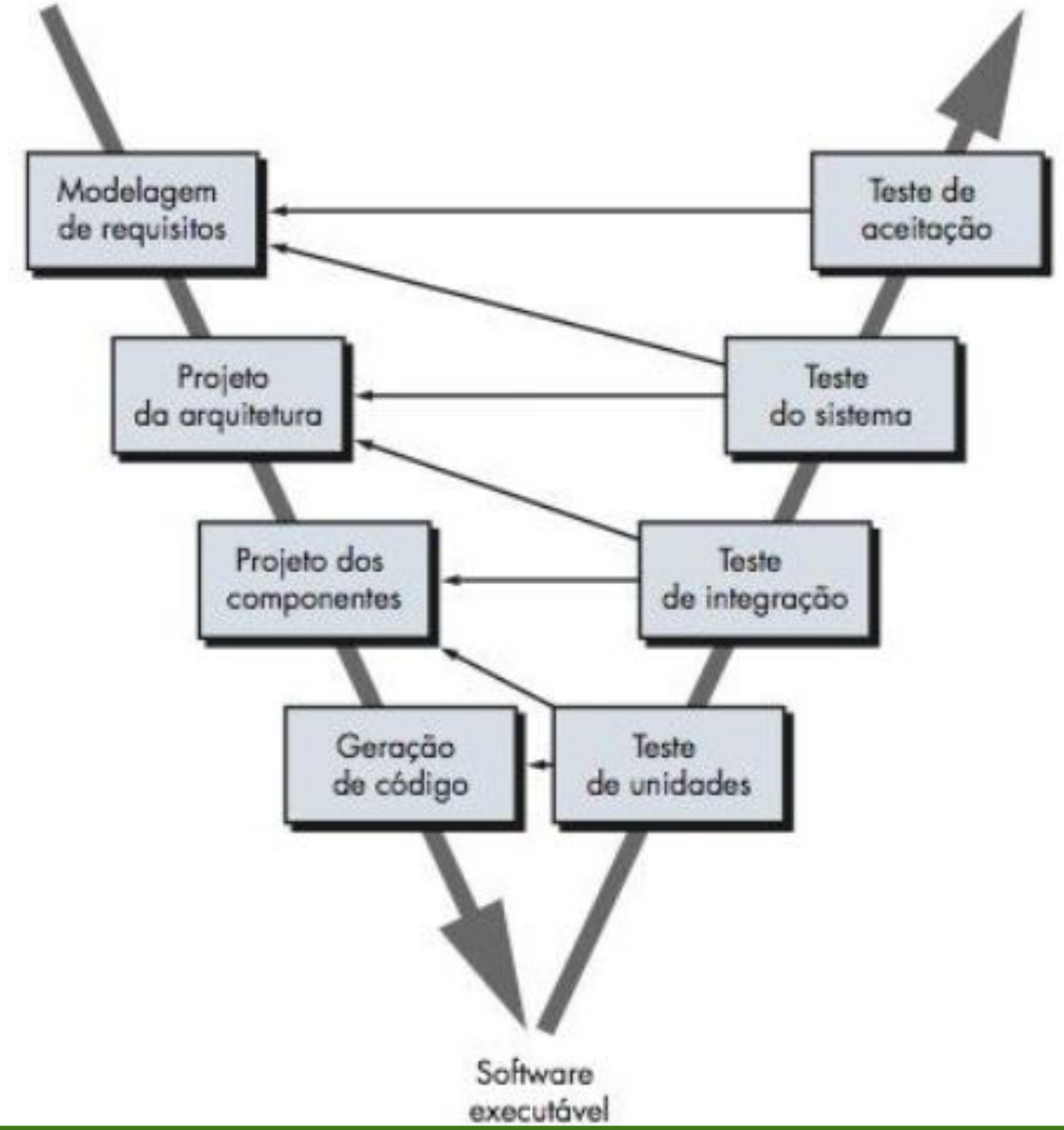
SAIBA MAIS!

Testes de Regressão

- Não correspondem a um nível de teste, mas são importantes para reduzir os “efeitos colaterais” causados por uma inclusão, modificação ou exclusão de funcionalidade, módulo etc.
- Consistem em aplicar, a cada nova versão do software ou a cada ciclo, todos os testes que já foram aplicados nas versões ou ciclos de teste anteriores do sistema;
- Podem ser aplicado em qualquer nível de teste, porém os testes de unidade e os de integração são os mais facilmente automatizados, permitindo assim testes de regressão automatizados.

SAIBA MAIS!

Teste de Software no modelo V



SAIBA MAIS!

Ciclo do Desenvolvimento Dirigido a Testes

1. Adicione um teste

- Cada nova funcionalidade inicia com a criação de um teste. Este teste precisa inevitavelmente falhar porque ele é escrito antes da funcionalidade a ser implementada (se ele falha, então a funcionalidade ou melhoria 'proposta' é óbvia). Para escrever um teste, o desenvolvedor precisa claramente entender as especificações e requisitos da funcionalidade. O desenvolvedor pode fazer isso através de casos de uso ou user stories que cubram os requisitos e exceções condicionais.

SAIBA MAIS!

Ciclo do Desenvolvimento Dirigido a Testes

2. Execute todos os testes e veja se algum deles falha

- Esse passo valida se todos os testes estão funcionando corretamente e se o novo teste não traz nenhum equívoco, sem requerer nenhum código novo. Pode-se considerar que este passo então testa o próprio teste: ele regula a possibilidade de novo teste passar.
- O novo teste deve então falhar pela razão esperada: a funcionalidade não foi desenvolvida. Isto aumenta a confiança (por outro lado não exatamente a garante) que se está testando a coisa certa, e que o teste somente irá passar nos casos intencionados.

SAIBA MAIS!

Ciclo do Desenvolvimento Dirigido a Testes

3. Escrever código

- O próximo passo é escrever código que irá ocasionar ao teste passar. O novo código escrito até esse ponto poderá não ser perfeito e pode, por exemplo, passar no teste de uma forma não elegante. Isso é aceitável porque posteriormente ele será melhorado.
- O importante é que o código escrito deve ser construído somente para passar no teste; nenhuma funcionalidade (muito menos não testada) deve ser predita ou permitida em qualquer ponto.

SAIBA MAIS!

Ciclo do Desenvolvimento Dirigido a Testes

4. Execute os testes automatizados e veja-os executarem com sucesso

- Se todos os testes passam agora, o programador pode ficar confiante de que o código possui todos os requisitos testados. Este é um bom ponto que inicia o passo final do ciclo TDD.

SAIBA MAIS!

Ciclo do Desenvolvimento Dirigido a Testes

5. Refatorar código

- Nesse ponto o código pode ser limpo como necessário. Ao re-executar os testes, o desenvolvedor pode confiar que a refatoração não é um processo danoso a qualquer funcionalidade existente. Um conceito relevante nesse momento é o de remoção de duplicação de código, considerado um importante aspecto ao design de um software. Nesse caso, entretanto, isso aplica remover qualquer duplicação entre código de teste e código de produção — por exemplo magic numbers or strings que nós repetimos nos testes e no código de produção, de forma que faça o teste passar no passo 3.

SAIBA MAIS!

Ciclo do Desenvolvimento Dirigido a Testes

6. Repita tudo

- Iniciando com outro teste, o ciclo é então repetido, empurrando a funcionalidade a frente. O tamanho dos passos deve ser pequeno - tão quanto de 1 a 10 edições de texto entre cada execução de testes. Se novo código não satisfaz rapidamente um novo teste, ou outros testes falham inesperadamente, o programador deve desfazer ou reverter as alterações ao invés do uso de excessiva depuração. A Integração contínua ajuda a prover pontos reversíveis. É importante lembrar que ao usar bibliotecas externas não é interessante gerar incrementos tão pequenos que possam efetivamente testar a biblioteca ,[3] a menos que haja alguma razão para acreditar que a biblioteca tenha defeitos ou não seja suficientemente completada com suas funcionalidades de forma a servir às necessidades do programa em que está sendo escrito.

SAIBA MAIS!

Teste Caixa-Branca

- Um teste caixa-branca (também conhecido como teste estrutural) é aquele que leva em consideração o interno do componente de software;
- Em outras palavras, ele leva em consideração como o componente foi implementado (seu código) em seu planejamento e execução;
- Essa técnica trabalha diretamente sobre o código fonte do componente de software para avaliar aspectos tais como: teste de condição, teste de fluxo de dados, teste de ciclos e teste de caminhos lógicos etc.
- Testes de unidade e testes de integração são, muitas vezes, executados como testes caixa-branca.

SAIBA MAIS!

Teste Caixa-Preta

- Um teste caixa-preta (também conhecido como teste funcional) desconhece o comportamento interno do componente de software, tratando-o como uma “caixa-preta”. Assim, tal teste somente pode considerar os diversos tipos de entrada e os possíveis resultados esperados;
- Pode ser executado em todos os níveis de teste, mas úteis principalmente em testes de sistema e de aceitação.

SAIBA MAIS!

Análise do Valor Limite

- Um grande número de erros tende a ocorrer nos limites do domínio de entrada invés de no “centro”;
- Assim, os critérios de teste devem explora os limites dos valores de cada classe de equivalência para preparar os casos de teste;
- Se uma condição de entrada especifica uma faixa de valores limitada em a e b, casos de teste devem ser projetados com valores a e b e imediatamente acima e abaixo de a e b;
  - Exemplo: Intervalo = {1..10}; Casos de Teste à {1, 10, 0,11}.

SAIBA MAIS!

Análise do Valor Limite

- Por exemplo: "... o cálculo do desconto por dependente é feito da seguinte forma: a entrada é a idade do dependente que deve estar restrita ao intervalo [0..24]. Para dependentes até 12 anos (inclusive) o desconto é de 15%. Entre 13 e 18 (inclusive) o desconto é de 12%. Dos 19 aos 21 (inclusive) o desconto é de 5% e dos 22 aos 24 de 3%..."

- Aplicando o teste de valor limite convencional serão obtidos casos de teste semelhantes a este: {-1,0,12,13,18,19,21,22,24,25}.

SAIBA MAIS!

Testes Alfa

- Diz-se que um software está em estágio alfa quando o mesmo encontra-se em um estágio que, mesmo ainda incompleto, já permite o seu teste em um ambiente controlado;
- Trata-se de um conjunto de testes a serem executados por testadores de software da equipe de desenvolvimento em um ambiente muito similar ao que os usuários finais irão utilizar;
- Pode-se utilizar dessa estratégia para maior eliminação de erros possível antes de o software passar para o próximo estágio (beta).

SAIBA MAIS!

Teste Beta

- Diz-se que um software está em estágio beta quando o mesmo encontra-se em um estágio bastante avançado e em um momento propício para o seu teste fora de ambiente controlado;
- Trata-se de um conjunto de testes a serem executados por futuros usuários do produto final, em ambiente de produção real e para o desenvolvimento das tarefas referentes ao mesmo.

Testes de sistema e testes de aceitação podem se utilizar tanto de testes alfa e beta para maior eliminação de erros possível. O que mudará é a forma como receber e lidar com os resultados dos testes.

SAIBA MAIS!

Testes de Software Orientado a Objetos

- O planejamento de testes projetados e programados segundo a orientação a objetos introduz alguns novos desafios:
  1. Herança:
     - Métodos testados em superclasses precisam ser retestados em subclasses e vice-versa;
     - Mesmo métodos herdados integralmente de uma superclasse devem ser retestados nas subclasses;
     - Herança múltipla introduz ainda mais desafios.

SAIBA MAIS!

Testes de Software Orientado a Objetos

- O planejamento de testes projetados e programados segundo a orientação a objetos introduz alguns novos desafios:
  2. Encapsulamento:
     - Limita a controlabilidade e a observabilidade;
     - Visibilidade dos estados reduzida;
     - Dificulta inicialização dos itens de dados e chamada de métodos.

SAIBA MAIS!

Testes de Software Orientado a Objetos

- O planejamento de testes projetados e programados segundo a orientação a objetos introduz alguns novos desafios:

  3. Polimorfismo:
     - Cada possibilidade de acoplamento de uma mensagem polimórfica é uma computação única;
     - Indecidibilidade ao teste: dificuldade em antecipar possíveis acoplamentos.

SAIBA MAIS!

Outros Tipos de Teste

- **Teste de configuração -** Testa se o software funciona no hardware a ser instalado;
- **Teste de instalação -** Testa se o software instala como planejado em diferentes hardwares e sob diferentes condições como pouco espaço de memória, interrupções de rede, interrupções na instalação, etc.
- **Teste de integridade -** Testa a resistência do software a falhas (robustez);
- **Teste de segurança -** Testa se o sistema e os dados são acessados de maneira segura apenas pelo autor das ações;
- **Teste de volume -** Testa o comportamento do sistema operando com o volume “normal” de dados e transações envolvendo o banco de dados durante um longo período de tempo.

SAIBA MAIS!

Outros Tipos de Teste

- **Teste de performance -** O teste de performance se divide em 3 tipos:
  - Teste de carga - Testa o software sob as condições normais de uso. Ex.: tempo de resposta, número de transações por minuto, usuários simultâneos, etc.
  - Teste de stress - Testa o software sob condições extremas de uso. Grande volume de transações e usuários simultâneos. Picos excessivos de carga em curtos períodos de tempo.
  - Teste de estabilidade - Testa se o sistema se mantém funcionando de maneira satisfatória após um período de uso.
- **Teste de usabilidade -** Teste focado na experiência do usuário, consistência da interface, layout, acesso às funcionalidades, etc.
- **Teste de manutenção -** Testa se a mudança de ambiente não interferiu no funcionamento do sistema.

Referências Bibliográficas

- ABRAHAMSSON, Pekka; SALO, Outi; RONKAINEN, Jussi; WARSTA, Juhani. Agile Software Development Methods – Review and Analysis. ESPOO, 2002. VTT Publications 478, 107 p. Disponível em: http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf . Acessado em 28/02/2014.
- AGILE MANIFESTO. Manifesto for Agile Software Development. s.d. Disponível em: http://agilemanifesto.org . Acessado em 28/02/2014.
- BOSCÀ, Neus. Lean Project Management: Assessment of project risk management processes. Dissertação de Mestrado. KTH, School of Industrial Engineering and Management (ITM). Suécia, 2012. Disponível em: http://kth.diva-portal.org/smash/get/diva2:534029/FULLTEXT01.pdf . Acessado em 27/02/2014.
- COUTINHO, Ítalo de A. Estudo da aderência dos processos de gestão de projetos em empresas de engenharia consultiva de Belo Horizonte. 153f. Dissertação (Mestrado em Administração) – Universidade FUMEC, Belo Horizonte-MG, 2009. Disponível em: http://www.fumec.br/anexos/cursos/mestrado/dissertacoes/completa/italo_azeredo_coutinho.pdf . Acessado em 25/02/2014.
- HIGHSMITH, Jim. Agile Project Management – Creating Innovative Products. Editora Addison-Wesley, ed. 7, 2009.

Referências Bibliográficas

- LEACH, Lawrence P. Critical Chain Project Management. Artech House, 2000.
- OBJECT MANAGEMENT GROUP. UML. 2015. Disponível em: http://www.omg.org/spec/UML/ , acessado em 08/03/2015.
- MACHADO, Marcos; MEDINA, Sérgio. SCRUM – Método Ágil: uma mudança cultural na gestão de projetos de desenvolvimento de software. In: Revista Científica Intr@ciência, v. 1, n. 1, pp. 58-71, 2009. Disponível em: http://www.faculdadedoguaruja.edu.br/revista/downloads/edicao12009/Artigo_5_Prof_Marcos.pdf . Acessado em 09/04/2014.
- MAHNIC, Viljan. Improving Software Development Through Combination of Scrum and Kanban. In: Recent Advances in Computer Engineering, Communications and Information Technology, Espanha, 2014. Disponível em: http://www.wseas.us/e-library/conferences/2014/Tenerife/INFORM/INFORM-40.pdf . Acessado em 27/02/2014.
- MIRONIUK, Kseniia. Lean Office Concept – Implementation in R-Pro Consulting Company. Mikkeli University of Applied Sciences, 2012. Disponível em: https://www.theseus.fi/bitstream/handle/10024/42325/Mironiuk_Kseniia.pdf . Acessado em 27/02/2014.

Referências Bibliográficas

- MOREIRA, Sônia. Aplicação das Ferramentas Lean – Caso de Estudo. Dissertação de Mestrado. Instituto Superior de Engenharia de Lisboa, Portugal, 2011. Disponível em: http://repositorio.ipl.pt/bitstream/10400.21/1167/1/Dissertação.pdf . Acessado em 27/02/2014.
- PRESSMAN, Roger. Engenharia de Software: Uma abordagem profissional. 7ª edição. Editora: McGraw-Hill – Artmed, 2011. ISBN: 9788563308337.
- PROJECT MANAGEMENT INSTITUTE. A Guide to the Project Management Body of Knowledge. Ed. 5, PMI, 2013.
- RAMOS, Ricardo A. Elicitação e Análise de Requisitos. Notas de aula, 2013. Disponível em: http://www.univasf.edu.br/~ricardo.aramos/disciplinas/ES_I_2013_2/ElicitacaoRequisitos.pdf , acessado em 07/03/2015.
- SUTHERLAND, Jeff. Scrum – A arte de fazer o dobro na metade do tempo. Ed. 3. Editora LeYa, 2016.
- SCHWABER, Ken. Agile Project Management with Scrum. Microsoft Press, 2004.
- VERZUH, Eric. The Fast Forward MBA in Project Management. Ed. 3. Editora Wiley, 2008.

Referências Bibliográficas

- WIKIPÉDIA. Brainstorming. 2015a. Disponível em: http://pt.wikipedia.org/wiki/Brainstorming , acessado em 07/03/2015.
- WIKIPÉDIA. Programação Extrema. 2019. Disponível em: https://pt.wikipedia.org/wiki/Programação_extrema, acessado em 23/01/2020.
- WIKIPÉDIA. Requisito. 2015b. Disponível em: http://pt.wikipedia.org/wiki/Requisito , acessado em 07/03/2015.
- WIKIPÉDIA. Requisito Funcional. 2015c. Disponível em: http://pt.wikipedia.org/wiki/Requisito_funcional , acessado em 06/06/2015.
- WIKIPÉDIA. Requisito Não-Funcional. 2015d. Disponível em: http://pt.wikipedia.org/wiki/Requisito_não-funcional , acessado em 06/06/2015.

LEITURA RECOMENDADA

- 201 Princípios de Desenvolvimento de Software.