

Preparação Exame Final - MAS

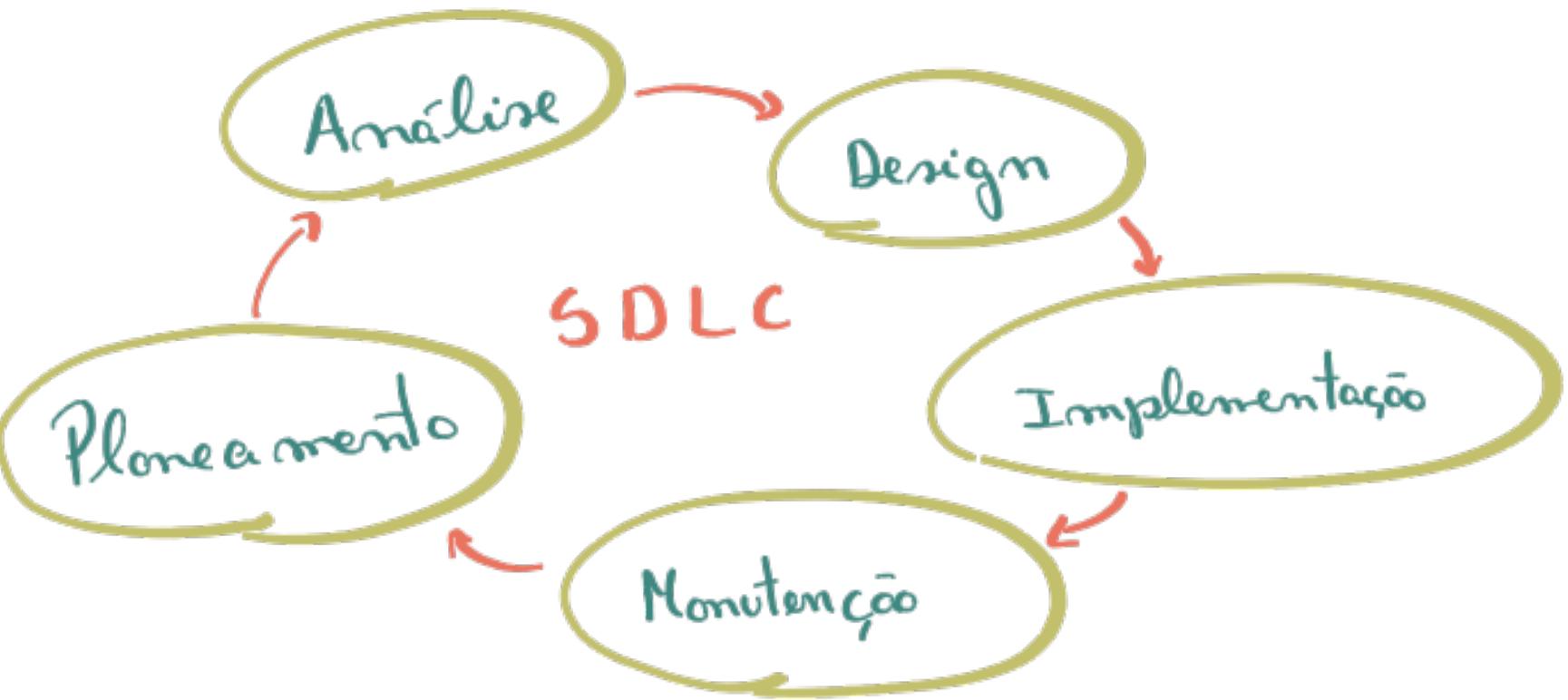
Índice

O que é que o SDLC inclui?	3
O trabalho do Analista na equipa de desenvolvimento	3
O Unified Process/OpenUP	4
Principais características dos métodos Ágeis	8
O papel da modelação (visual)	10
Modelos de Análise	13
Práticas de engenharia de requisitos	13
A modelação do contexto do problema: modelo do domínio/negócio	14
Modelação funcional com casos de utilização	15
Modelação estrutural	18
Modelação de comportamento	20
Modelos no desenho e implementação	21
Vistas de arquitetura	21
Classes e desenho de métodos (perspectiva do programador)	22
Práticas selecionadas na construção do software	24
Garantia de qualidade	24
Abordagens complementares	29
Histórias e métodos ágeis	29
A metodologia SCRUM	31

Introdução

O SDLC é concretizado usando um processo de software sistemático.
Um processo de software é um guia para as atividades, ações e tarefas que são necessárias para construir software de qualidade.

→ Início



O que é que o SDLC inclui?

O trabalho do Analista na equipa de desenvolvimento

● Explique o que é o ciclo de vida de desenvolvimento de sistemas (SDLC):

SDLC (System Development Life Cycle) é o processo que visa compreender de que forma um Sistema de Informação (SI) pode suportar requisitos de negócio, através do design, construção e entrega do sistema aos seus utilizadores. O conceito de SDLC sustenta muitos tipos de metodologias de desenvolvimento de software.

● Descreva as principais atividades (etapas-chave) dentro de cada uma das quatro fases do SDLC:

- Quatro fases fundamentais: Planeamento, Análise, Design e Implementação (PADI).
- Estas fases podem ser abordadas de forma diferente conforme as necessidades do negócio.
- Cada fase é composta por uma série de passos, baseados em técnicas que produzem entregáveis → documentos específicos (relatórios) e ficheiros que facilitam a compreensão do projeto.

Planeamento: → "Início"

- Iniciação: Qual é o valor de negócio do sistema? O projeto deve avançar?

Gestor do Projeto

Management: Criação de um plano de trabalho, preparar a equipa para controlar e direcionar o projeto através do SDLC.

Responde:

- Porquê da necessidade da construção de um SI
- Como a equipa vai construir

- Valor do negócio
- O projeto deve avançar?
- Plano de trabalho

Análise: → "Funcionalidades do sistema"

- Quem vai utilizar o sistema? O que é que o sistema vai fazer? Onde e quando vai ser utilizado?

P1. T2019

O ciclo de desenvolvimento de sistemas (SDLC) engloba quatro etapas genéricas. Qual o trabalho típico da fase de Análise?

- Analizar sistemas existentes, recolher requisitos para o novo sistema em articulação com os promotores, desenvolver o conceito/proposta para a solução.
- Analisar os novos processos de trabalho, definir a arquitetura do novo sistema, selecionar tecnologias para a implementação da solução.
- Analisar o valor que o sistema pode trazer à organização, determinar a viabilidade do projeto, estabelecer um plano de projeto inicial.
- Analizar os casos de utilização, extrair histórias de utilização, criar modelos UML para as classes principais do sistema.
- Identificar tendências emergentes na evolução da tecnologia, propor oportunidades de desmaterialização de processos, caracterizar os casos de utilização alvo.

P12. ✓ T2018

O modelo do domínio é construído na Análise, para mapear os conceitos do universo de discurso.

- ✗ As classes do modelo do domínio são as mesmas classes do código e por isso existe uma continuidade.
- ✗ As classes do modelo do domínio captam a estrutura da informação; terão utilidade posterior para desenhar a base de dados, mas não o código.
- c) Os conceitos identificados no modelo do domínio serão candidatos naturais a constituir classes, na fase de desenho.
- d) A estrutura dos dados de um problema é muito volátil e, por isso, o modelo do domínio é provisório e vai ser alterado.
- e) O modelo do domínio só é construído para projetos que requerem a utilização de bases de dados.

- Estudo do domínio/área e análise dos sistemas existentes
- Levantamento de requisitos Ligações com stakeholders
- Proposta de sistema que responde às necessidades identificadas

Design: → "Modelação dos dados; Dentro do programa; Seleção da estrutura"

- Como vai o sistema operar em termos de hardware, software, infraestruturas de rede?

Design da UI (User Interface), bases de dados, etc. Planeamento da arquitetura e logística de todo o SI.

- Estratégia do desenvolvimento (interno ou contratualizado)
- Concepção da arquitetura do sistema
- Concepção do modelo de dados
- Desenho das entidades de software (programos)
- Seleção de frameworks (padrões de projetos bem estabelecidos)

Implementação:

→ Construção do sistema.

"Sistema é de facto construído"

• Desenvolver código

• Realizar testes

• Integrar módulos e frameworks

• Desenvolver os interfaces de utilizador

→ "Colocar em produção"

Plano de suporte
(facilita futuras modificações)

- Implementação de sistemas
- Instalação e transições
- Plano de suporte

● Descreva o papel e as responsabilidades do Analista no SDLC:

O Analista do Sistema é quem analisa a situação do negócio, identifica oportunidades de evolução e desenha o sistema de informação que implementa essas oportunidades.

O seu principal objetivo é criar valor para a organização.

↳ Projetar um sistema de informação

↳ não é criar um sistema "topo de goma"

"Analisa a situação do negócio"

↳ Define os requisitos do software

● Distinguir os competências de "análise de sistemas" dos de "programação de sistemas", em engenharia de software. Relacionar com conceitos de "soft skills" e "hard skills"

- A "análise de sistemas" e a "programação de sistemas" são dois campos da engenharia de software, que embora relacionados, são distintos. As competências de análise de sistemas incluem a capacidade de entender os requisitos do cliente, logo tendem a ser ligadas mais a "soft skills", pois envolvem a capacidade de entender os requisitos do cliente, comunicar-se eficazmente com o cliente e a equipa de desenvolvimento, e resolver problemas.
- As competências de programação de sistemas, por outro lado, tendem a ser mais ligadas a "hard skills", pois envolvem o conhecimento e a habilidade de usar linguagens de programação específicas para criar o software.

Soft skills

≠

Hard skills

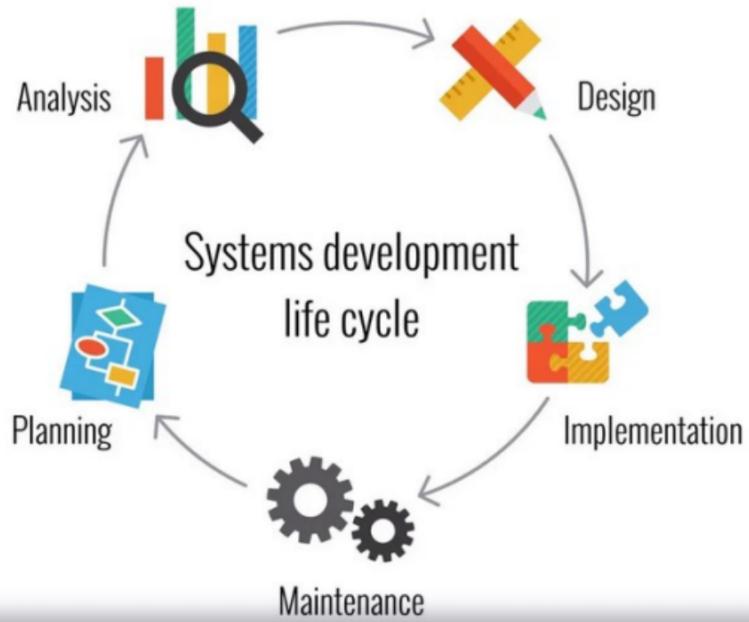
→ habilidades comportamentais
→ interação e comportamento

↓
Analista

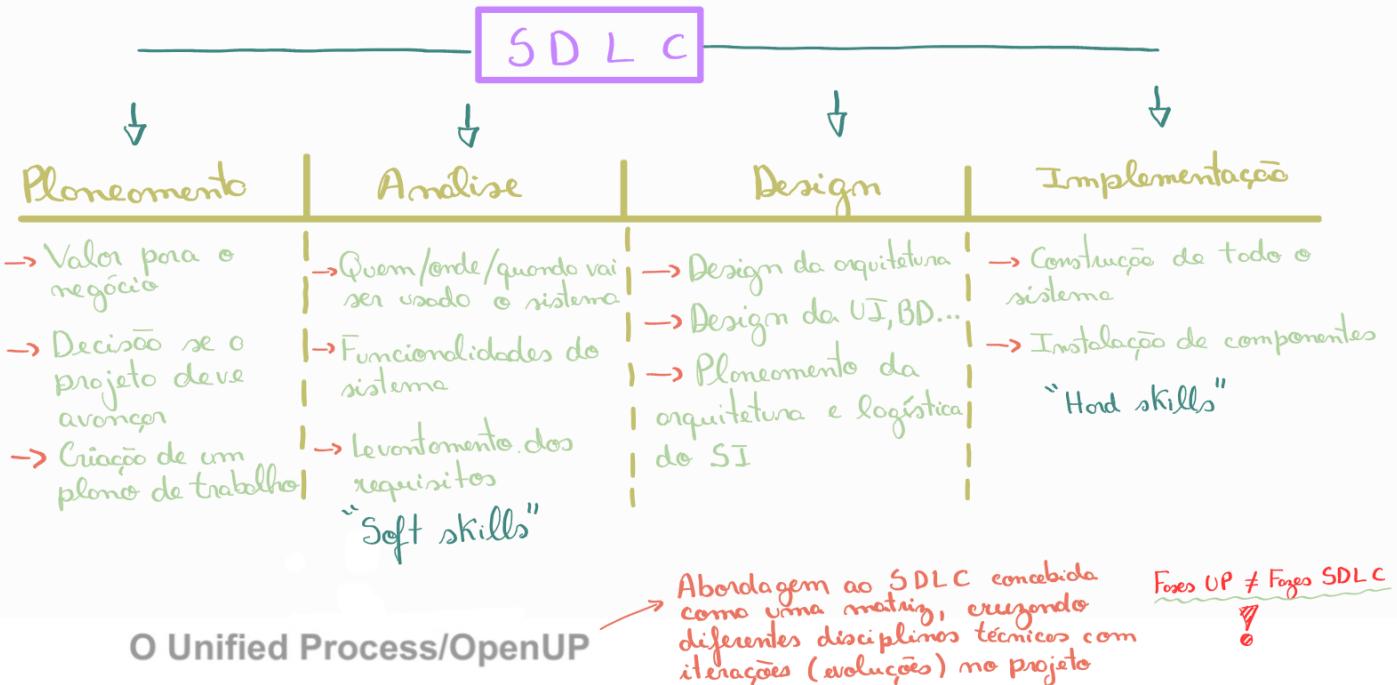
→ conhecimento e a habilidade de usar ferramentas, técnicas e tecnologia

↓
Implementação

Resumo - SDLC



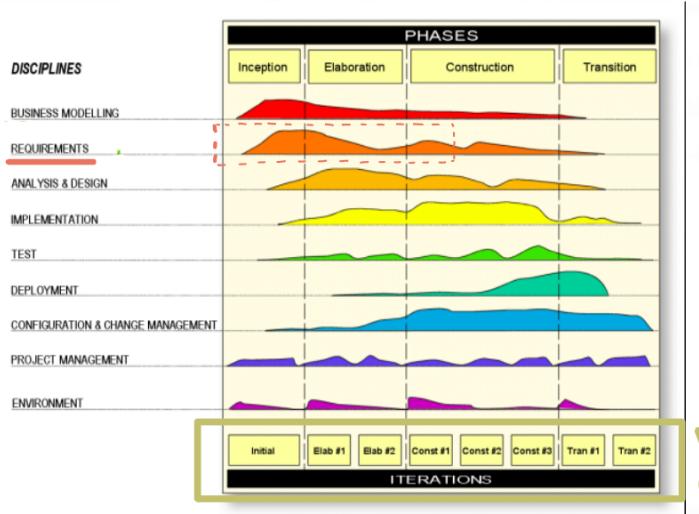
• Processo de determinação de como um sistema de informação (SI) pode suprir as necessidades dos empresários, projetando um sistema, construindo-o e entregando-o aos utilizadores



O Unified Process/OpenUP

Para a construção de um SI com recurso ao SDLC, é necessário um método de trabalho testado, aprovado, que funcione e que implemente o SDLC → OpenUP

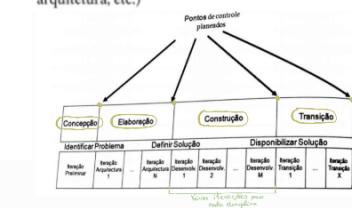
• Descrever a estrutura do OpenUP (fases e iterações):



P2. T2019
O método Unified Process (UP) prevê quatro fases principais no desenvolvimento do projeto, divididas em iterações. Qual das seguintes afirmações NÃO caracteriza o método UP?

- O método UP preconiza o desenvolvimento iterativo e incremental e adota, por isso, uma das ideias principais dos métodos ágeis.
- As fases do UP são de tamanho variável entre si, mas as iterações são regulares.
- É normal que haja lugar ao desenvolvimento de código em iterações anteriores à fase de Construção.
- A verificação dos incrementos, com recurso a testes, pode ser feita em iterações iniciais da Construção.
- Cada iteração foca uma disciplina técnica específica (e.g.: planejamento, levantamento de requisitos, definição da arquitetura, etc.)

P2 (T2019)
Como se pode observar na figura, cada iteração não foca uma disciplina técnica específica. Pelo contrário, são executadas múltiplas iterações para cada disciplina!



(CECT)

"Começo", "Origem"

Inception (Conceção):

Há uma concordância na visão do projeto e seus objetivos? Deve o projeto avançar?

→ Apenas uma pequena iteração

→ Entregável: documento de Visão e caso de negócio (Não requisitos detalhados)

→ Desenvolvimento de requisitos de mais alto nível → O que é preciso de um modo mais geral?
↳ Levantar casos de utilização!

→ Redução do risco → Identificação de requisitos chave

→ Perceber que os requisitos vão inevitavelmente mudar

→ Lidar com essa mudança → Processo iterativo

→ Produção de protótipos conceptuais sempre que necessário.

Ponto de referência

P8. ✓ T2018 *ver milestones sublinhadas a verde

O método Unified Process prevê quatro fases principais no desenvolvimento do projeto. Cada qual tem um grande objetivo a atingir (milestone) para se poder avançar, que são, por ordem:

a) 1/ Decisão de avançar ou parar o projeto; 2/ Arquitetura técnica definida e validada; 3/ Funcionalidades da primeira versão do produto implementadas; 4/ Solução instalada e aceite pelo cliente.

b) 1/ Preparação do documento de Visão; 2/ implementação exploratória da arquitetura técnica; 3/ Produto implementado; 4/ Testes no cliente.

c) 1/ Definição dos casos de utilização; 2/ implementação do protótipo exploratório; 3/ definição da arquitetura; 4/ Implementação do produto concluída.

d) 1/ Plano para o projeto definido; 2/ Análise de requisitos terminada; 3/ Desenho do código completo; 4/ Implementação da solução concluída.

e) Nenhuma das hipóteses anteriores está correta.



Milestone :

→ Neste ponto, deve-se examinar o custo/benefício do projeto e decidir prosseguir com o projeto ou cancelá-lo.

→ "Base do sistema"

Elaboration (Elaboração):

Há uma concordância na arquitetura a usar para desenvolver o SI? O valor produzido até então e o risco que resta é aceitável?

→ Várias iterações (mínimo duas);

→ Atenuar o risco através da produção de valor (código testado) e fornecendo uma base estável para o grande esforço de desenvolvimento da próxima fase.

→ Produção e validação de uma arquitetura executável;

→ Implementação de alguns componentes chave;

→ Identificar dependências com sistemas externos e proceder à integração.

→ Alguns códigos implementados (~10%) → (Não design detalhado)

→ Arquitetura conduzida pelos Use Cases!

Criar uma base
que sustente a
construção?

Can's maiores
30% dos Can's
determinam 80% da
arquitetura

↳ Levantados na Inception

Milestone: → Neste ponto, deve-se acordar os detalhes do sistema e a escolha da arquitetura. Só se continua se a arquitetura for validada.

Construção(Construction): → "Elaboração"

O sistema está perto o suficiente da entrega? A equipa já está na fase de passar a uma finalização que assegura a entrega bem sucedida do sistema?

→ Várias iterações.

→ Construir, desenhar, implementar e testar em todos os cenários possíveis → Incremento a incremento → Guiado pela arquitetura.

Divide-se em iterações para que os stakeholders acompanhem o desenvolvimento

→ Demonstrações frequentes do avanço do projeto.

→ Construção diária através de um processo de construção automatizado.

Milestone:

→ Neste ponto, o SI está pronto para ser entregue à equipa de transição. Todos os funcionalidades foram desenvolvidas e todos os testes alfa (se houver) foram concluídos.

→ Além do software, um manual de utilização foi desenvolvido, e há uma descrição do atual lançamento.

→ O produto está pronto para o teste beta

Transição(Transition): → "Estabilização e entrega"

O SI está pronto para entrega?

→ Estabilização e entrega

→ Bug-fix releases

→ Documentação produzida e organizada

Milestone:

→ Aprovação do cliente após rever e aceitar os entregáveis do projeto

- Identificar as principais atividades de modelação/desenvolvimento associados a cada fase:

Conceção(Inception):

Atividades	Modelação
<ul style="list-style-type: none">• Elaborar modelo de requisitos de alto nível.• Identificar interações com entidades externas.• Casos de utilização levantados (os de maior risco podem ser detalhados).• Planeamento das fases subsequentes e pontos de decisão.	<ul style="list-style-type: none">• Visão geral do problema• Modelo de Casos de Utilização• Glossário inicial• Avaliação de risco inicial• Justificação da viabilidade do projeto• Plano de projeto• Protótipos iniciais (para mitigação de risco) <p><i>Casos chave para o projeto - Palavras-chave</i></p>

Elaboração(Elaboration):

Atividades	Modelação
<ul style="list-style-type: none"> • Detalhar o modelo de casos de utilização • Analisar domínio • Definir arquitetura candidata • Validar arquitetura com implementação 	<ul style="list-style-type: none"> • Modelo de Casos de Utilização (especificação abrangente) • Requisitos (incluindo não funcionais) • Descrição da arquitetura do software • Protótipos (mitigação de risco). • Protótipo executável (validar arquitetura). • Plano de projeto revisto • Medidas para mitigação do risco

- O OpenUP pode ser considerado “método ágil”? → *Abordagem colaborativa e iterativa e o trabalho é dividido em etapas menores e o software é entregue em incrementos frequentes* !

O OpenUP é considerado um método ágil que promove as melhores práticas de desenvolvimento de software:

- Desenvolvimento iterativo
- Colaboração em equipa
- Integrações e testes contínuos
- Entregas frequentes de software funcional
- Adaptação a mudanças, entre outros

- Porque é que o Unified Process é “orientado por casos de utilização, focado na arquitetura, iterativo e incremental”?

Focado na arquitetura → O OpenUP foca-se na arquitetura no sentido de minimizar o

risco e organizar o desenvolvimento. → *(Basear a arquitetura nos casos de utilização nucleares)*

Iterativo e incremental → O OpenUP promove práticas que permitem à equipa ter contínuo feedback dos stakeholders, assim como demonstrar-lhes o valor de cada incremento, com a finalidade de lhes fornecer o máximo valor.

↳ *reduz o risco*

Orientado por casos de utilização → A equipa utiliza casos de utilização para orientar todo o trabalho de desenvolvimento, desde a Inception até à Construção.

P3. T2019

* igual à pergunta P11 do T2018

O método Unified Process é orientado por casos de utilização (CaU), porque neste método:

- a) Os CaU estabelecem uma divisão funcional do sistema que é usada ao longo do desenvolvimento do projeto, na análise, no desenho e nos testes.
- b) A primeira tarefa do SDLC é o levantamento e especificação de CaU, incluindo cenários ~~epicos~~ e alterativos.
- c) A UML e os Diagramas de Casos de Utilização são usados para descrever o projeto.
- d) É recomendada a identificação de personas e a exploração de cenários de uso através de histórias (“user stories”).
- e) Os CaU são usados para delinear os planos de teste.

* Abordagem Waterfall:

Funciona bem onde os requisitos são estáveis e bem compreendidos

Vantagens:

- Simples e fácil de entender e usar
- Fácil de planejar e gerenciar
- Prazos bem definidos, sem alterações
- As fases são processadas e concluídas uma de cada vez

Desvantagens:

- Dificuldade de acmodar mudanças
- Modelo "pobre" para projetos longos e contínuos
- Funcionamento de software só é produzido todo de uma vez
- Não lida com requisitos incertos e em risco

→ Porque pode falhar?

- Projetos reais raramente seguem o fluxo sequencial que o modelo propõe!
- É frequentemente difícil para o cliente explicar todos os requisitos

O cliente tem de ter paciência
Uma versão em funcionamento do programa só estará disponível num fase avançada
do ciclo de vida do projeto

Principais características dos métodos Ágeis:

- Identificar características distintivas dos processos sequenciais, como a abordagem waterfall.

Propõe um fluxo sequencial

Sequencial e linear → Avança-se para a fase seguinte só quando a atual estiver concluída.

→ A partir do momento que uma fase é concluída, não se volta atrás.

→ Não há flexibilidade → Não há espaço para alterações ou erros, é tudo feito conforme planeado inicialmente.

- Identificar as práticas distintivas dos métodos ágeis (o que há de novo no modelo de processo, comparando com a abordagem "tradicional"?).

Os métodos ágeis (OpenUP, SCRUM) surgiram como uma "solução" para as desvantagens da metodologia waterfall. → Pouca flexibilidade

→ Em vez de um processo de design sequencial, a metodologia Agile segue uma abordagem incremental e iterativa.

→ O desenvolvimento iterativo foca a entrega de valor orientada por ciclos curtos.

→ Cada iteração produz algum resultado executável, ao contrário da metodologia em cascata, onde só no fim do processo começam a ser produzidos resultados.

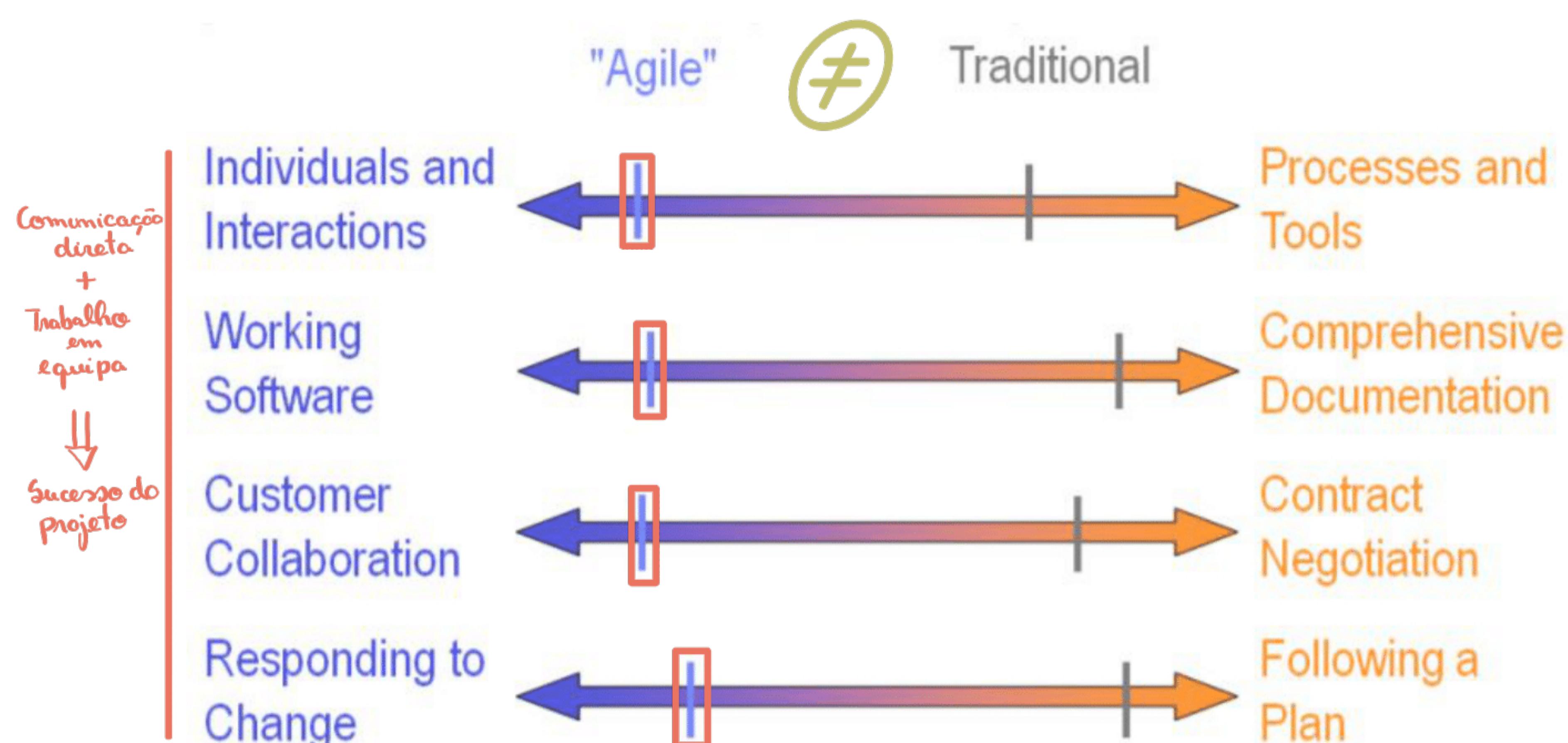
→ Os desenvolvedores começam com um projeto simples a trabalhar em pequenos módulos. O trabalho nesses módulos é feito em sprints semanais ou mensais e, no final de cada sprint, as prioridades do projeto são avaliadas e são executados testes.

→ Ciclos curtos e entrega de valor frequente, integração em contínuo, desenvolvimento orientado por testes (TDD). → Test-Driven Development

→ O objetivo dos métodos ágeis é dar resposta rápida à alteração de condições.

→ Entregas frequentes, integração em contínuo → redução de risco

→ reduz o risco



- Discuta o argumento que “A abordagem em cascata tende a mascarar os riscos reais de um projeto até que seja tarde demais para fazer algo significativo sobre eles.”

A mudança é inevitável ao longo do desenvolvimento de software. À medida que se avança num projeto seguindo o método waterfall, a contínua fixação às condições iniciais tende a mascarar os riscos, até que se chega a uma fase onde a equipa reconhece que não foram corrigidos erros em fases anteriores, não sendo possível voltar atrás para corrigi-los.

- Identifique vantagens de estruturar um projeto em iterações, produzindo incrementos com frequência

- Custo reduzido para corrigir um erro
- Apesar de cada iteração só realizados testes e aprovações por parte do cliente, reduzindo o risco e ajuda a decidir, se necessário, mudanças no projeto
- Permite que os stakeholders estejam por dentro do projeto

todas as iterações devem conter algo parcialmente executável

P4. T2018
Na metodologia SCRUM, há lugar à gestão explícita da pilha de trabalho (<i>backlog</i>). Neste contexto, que propriedades são distintivas do <i>backlog</i> ?
<input checked="" type="checkbox"/> Os itens de trabalho devem ter uma granularidade fina, que não excede um dia de trabalho de uma pessoa.
<input checked="" type="checkbox"/> Todos os itens de trabalho representam funcionalidades do produto.
<input checked="" type="checkbox"/> Cada item é anotado com a velocidade que a equipa precisa de demonstrar na sua implementação.
<input checked="" type="checkbox"/> Cada item de trabalho corresponde a um caso de utilização.
<input checked="" type="checkbox"/> e) A pilha está ordenada, das funcionalidades mais prioritárias (topo) para as menos prioritárias no projeto.

- Caracterizar os princípios da gestão do backlog em projetos ágeis.

Backlog → Work Items List → “Lista de afazeres priorizada”

- Os itens a trabalhar estão ordenados por prioridade (topo → prioridade mais elevada);
- Os itens de prioridade mais alta têm de estar bem definidos, os de mais baixa prioridade podem ainda ser vagos;
- Cada iteração implementa os itens de mais alta prioridade;
- Novos itens podem ser adicionados a qualquer altura a qualquer ponto da lista;
- Itens podem ser eliminados a qualquer momento;
- Itens a trabalhar podem ter a sua prioridade alterada a qualquer momento;

Característica de métodos Ágeis

Entidades da gestão de um projeto à moda da "Scrum"

Backlog:
→ Ordenado por prioridades
→ Features valem pontos
Iterações:
→ Tempo pré-definido
Iteração atual:
→ Atribuição e trabalho

P3. T2018
Os métodos ágeis de desenvolvimento de software preconizam a adoção de práticas que equilibrem a disciplina e a colaboração na equipa, tais como: <i>Não responde</i>
a) Integração dos incrementos no final das iterações, envolver o cliente em todos os testes, entrega frequente de atualizações.

* ver também páginas 30 e 31

P7. T2018
Uma das principais razões para se utilizar métodos ágeis em desenvolvimento, em detrimento dos métodos sequenciais, é a diminuição do risco do projeto. Que prática é decisiva para a mitigação do risco:
<input checked="" type="checkbox"/> a) A ordem dos itens no <i>backlog</i> pode ser alterada, com pouca "cerimónia".
<input checked="" type="checkbox"/> b) Os projetos são mais pequenos e não se gasta tempo a desenvolver documentação do projeto.
<input checked="" type="checkbox"/> c) A entrega frequente de valor e a integração em contínuo diminuem os problemas decorrentes do desenvolvimento de módulos independentes, em paralelo.
<input checked="" type="checkbox"/> d) O teste do software é deslocado da equipa de desenvolvimento para o Cliente.
<input checked="" type="checkbox"/> e) O projeto está dividido em segmentos, chamados iterações, que implementam as camadas de arquitetura sucessivamente.

P7. T2019
Uma das principais razões para se utilizar métodos ágeis em engenharia de software, em detrimento dos métodos sequenciais, é a diminuição do risco do projeto. Que prática é decisiva para a mitigação do risco:
<input checked="" type="checkbox"/> a) O projeto está dividido em iterações, o que assegura uma integração frequente dos incrementos parciais, e menor probabilidade de desvio face aos requisitos.
<input checked="" type="checkbox"/> b) A gestão do risco é baseada em <i>feedback</i> frequente: a comunicação entre equipa de desenvolvimento e os stakeholders (Cliente e/o <i>product owner</i>) é requerida em vários eventos das iterações.
<input checked="" type="checkbox"/> c) Os projetos são mais pequenos e não se gasta tanto tempo em tarefas de coordenação e documentação.
<input checked="" type="checkbox"/> d) A ordem dos itens na pilha do <i>backlog</i> é imutável, tornando o projeto mais previsível.
<input checked="" type="checkbox"/> e) O teste do software é sobretudo feito pelo Cliente, melhorando as condições de aceitação do produto.

- Dado um “princípio” (do Agile Manifesto), explique-o por palavras próprias, concentrando-se na sua novidade (com relação às abordagens “clássicas”) e impacto / benefício.

Manifesto para o Desenvolvimento

Ágil de Software

→ Começámos a valorizar: “Soft skills”

- Indivíduos e interações mais do que processos e ferramentas

→ Prestar mais atenção às pessoas envolvidas no projeto e à forma como trabalham juntas, em vez de seguir processos rígidos e usar ferramentas específicas. Isto pode levar a um resultado final mais eficiente e de alta qualidade, além de criar um ambiente de trabalho mais positivo

- Software funcional mais do que documentação abrangente

→ Concentrar em entregar software que funcione e atender às necessidades do cliente em vez de produzir uma grande quantidade de documentação detalhada. Isto pode levar a um resultado final mais útil e de maior valor para o cliente, além de manter o projeto mais ágil e adaptável.

- Colaboração com o cliente mais do que negociação contratual

→ Trabalhar em colaboração com o cliente durante todo o processo de desenvolvimento para garantir que o software atenda às suas necessidades, em vez de negociar contratos detalhados no início do projeto. Isto pode levar a um resultado final mais útil e de maior valor para o cliente, além de garantir que o software atenda às suas necessidades

- Responder à mudança mais do que seguir um plano

→ Independentemente das mudanças que possam ocorrer, a equipa de desenvolvimento deve estar disposta a adaptar o projeto e o processo de desenvolvimento conforme os necessidades do cliente mudam, nem sempre é fácil para o cliente explicar o que queria no começo

- Apresentar situações em que, de facto, um método sequencial pode ser o adequado.

- Quando o projeto é pequeno e simples, com requisitos bem definidos e não sujeitos a alterações significativas
- Quando a pouca incerteza ou risco envolvido no projeto
- O prazo é apertado e não há margem para a flexibilidade e adaptabilidade do processo de desenvolvimento

- Em geral, os métodos ágeis são mais adequados para projetos de software, pois eles são mais flexíveis e permitem uma adaptação mais rápida às mudanças de requisitos

O papel da modelação (visual)

- **Justifique o uso de modelos na engenharia de sistemas**

→ Os modelos ajudam a gerir a complexidade

G. Booch apresenta 4 razões para usar modelos:

→ Ajudar a visualizar um sistema, como se pretende que venha a ser;

→ Especificar a estrutura e o comportamento do sistema (antes de implementar);

→ Serve como referência / orientação para a construção (“planta”);

→ Documentar as decisões (de desenho) que foram feitas.

P4. T2019

A utilização de modelos em engenharia de software pode contribuir positivamente para o desenvolvimento dos produtos. Identifique um benefício que NÃO decorre do uso de modelos.

- a) Promover uma comunicação mais clara e sucinta dentro da equipa (técnica), recorrendo a uma linguagem visual partilhada;
- b) Manter o desenho (planeamento da solução) e a implementação (construção) mais coerentes; o modelo orienta a construção.
- c) Antecipar problemas de integração entre módulos, e dúvidas quanto aos requisitos dos produtos de software, mesmo antes de iniciar as atividades de implementação;
- d) Construir uma visão partilhada do progresso do projeto e prioridades de cada iteração/Sprint.
- e) Facilitar a geração automática de código (gerar a solução a partir do modelo).



- **Descreva a diferença entre modelos funcionais, modelos estáticos e modelos de comportamento.**

Structural Modeling

Modelos estáticos → Representam as partes estáticas do sistema, tais como classes, objetos, interfaces e as relações entre todos. UML: Diagrama de Classes, Deployment Diagram, Package Diagrams.

Os modelos estáticos definem a estrutura do sistema e as suas partes de diferentes níveis de abstração e implementação.

Structure diagrams show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other.

Modelos funcionais → Representam todos os funcionalidades desempenhadas pelo sistema e como essa é dividida em componentes. Eles descrevem o que o sistema deve fazer mas não como o faz.

Modelos de comportamento → Descrevem a interação no sistema. Representa interações entre os diagrama estruturais. Mostra a natureza dinâmica do sistema.

Behavioral model describes the interaction in the system. It represents the interaction among the structural diagrams. Behavioral modeling shows the dynamic nature of the system.

UML:

- Diagrama de sequência
- Diagrama de atividades
- Diagrama de casos de uso

P5. T2019

* igual à P10 do T2018

Num projeto de modelação (e.g.: do VisualParadigm), foi criada a entidade A (por hipótese, um Ator). Como é que uma dada entidade de modelação se relaciona com os vários diagramas?

- X Um elemento do modelo pertence a um diagrama (1:1); quando se apaga o diagrama do projeto, as entidades nele visualizadas são apagadas do projeto.
- X Um elemento do modelo pode ser visualizado em vários diagramas (1:N), desde que sejam do mesmo tipo (e.g.: o Actor só pode ser representado no diagramas de casos de utilização).
- C Um elemento do modelo pode ser visualizado em vários diagramas (1:N), que podem ser de tipos distintos.
- d) A UML prevê que um elemento de modelação possa ter diferentes representações visuais (stereotypes), mas isso é distrativo e não deve ser usado.
- e) A representação de um elemento do modelo em diagramas distintos, mesmo que permitida pela ferramenta, é desaconselhada na UML. X → a UML desaconselha a redundância!

Útil para mostrar diferentes aspectos ou níveis de detalhe do elemento em questão.

Exemplo: O Ator pode ser representado tanto no Diag. de classes como no Diag. de casos de uso

a UML desaconselha a redundância!

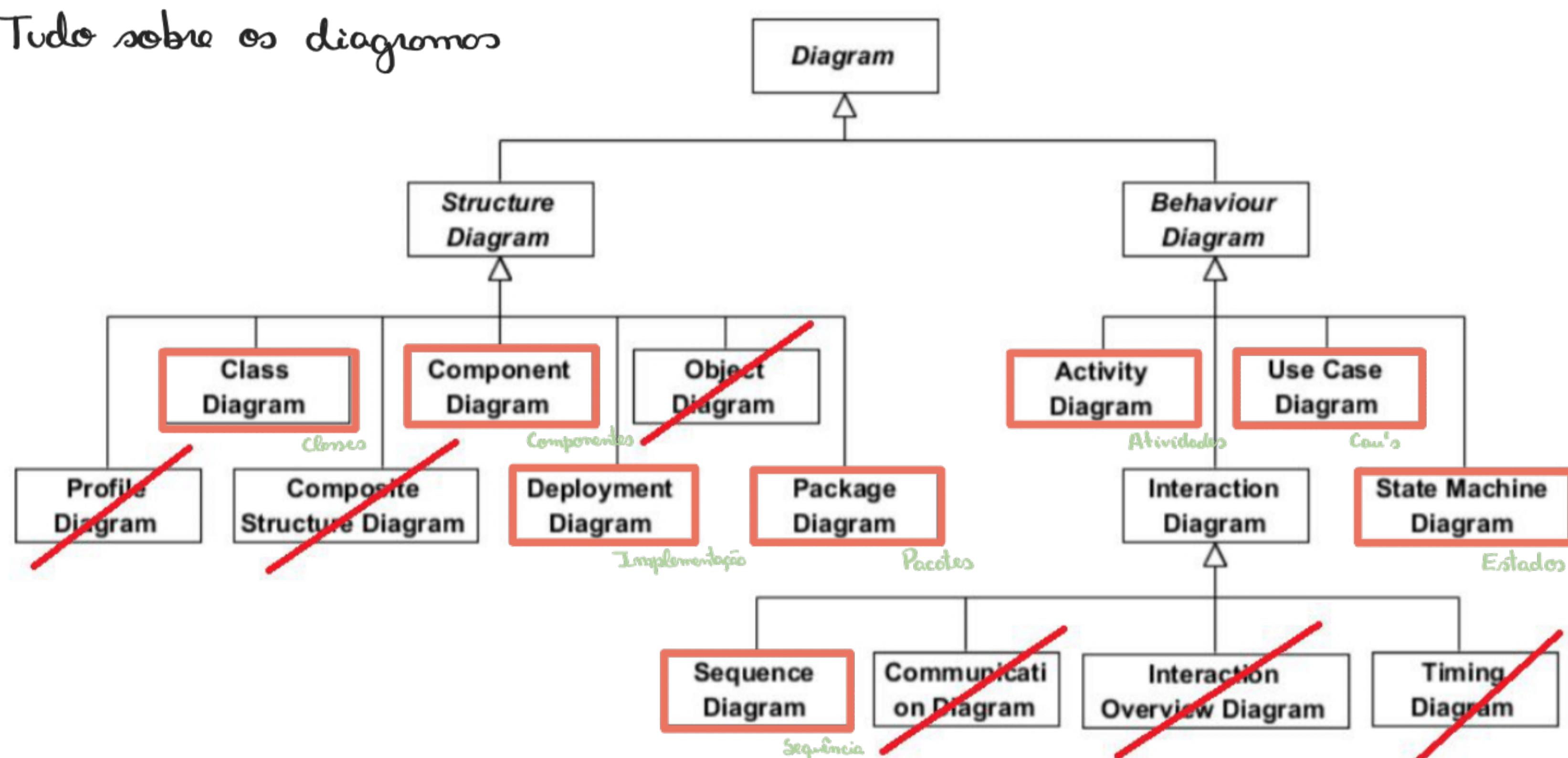
- Enumerar as vantagens dos modelos visuais

- Promover a comunicação mais clara e sucinta;
- Manter o desenho (planeamento) e a implementação (construção) coerentes;
- Mostrar ou esconder diferentes níveis de detalhe, conforme apropriado;
- Pode suportar, em parte, processos de construção automática (gerar a solução a partir do modelo).

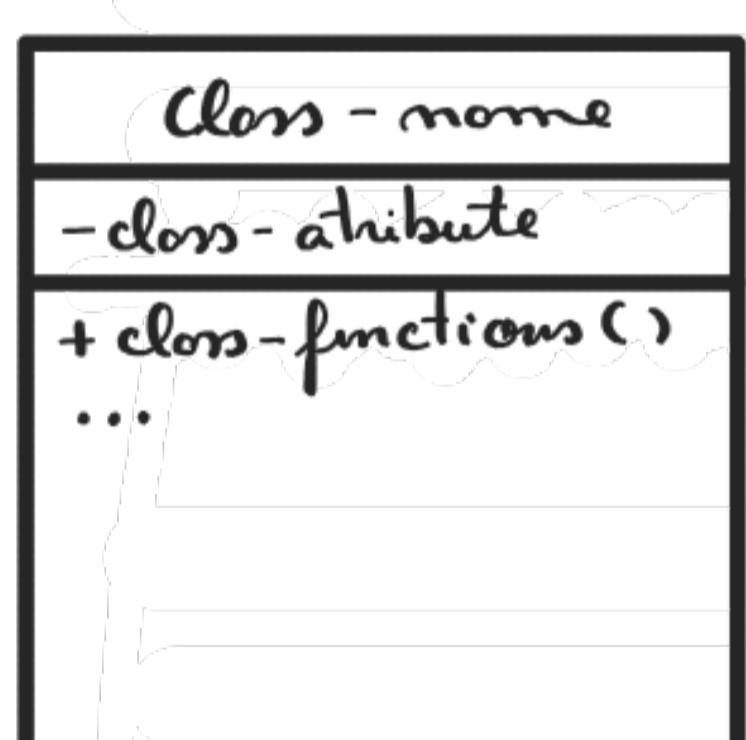
- Comunicação
- Coerência
- Níveis de detalhe
- Construção automática

- Explicar a organização da UML (classificação dos diagramas)

- Tudo sobre os diagramos



→ Cóis estruturais

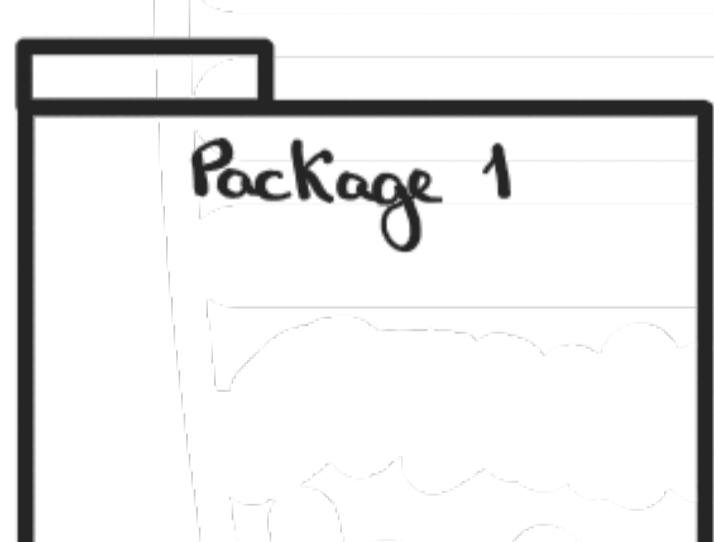


• Classe: usada para representar vários objetos, definir as propriedades e operações de um objeto



• Objeto: entidade que é usada para descrever comportamento e as funções de um sistema. A classe e o objeto têm a mesma notação.

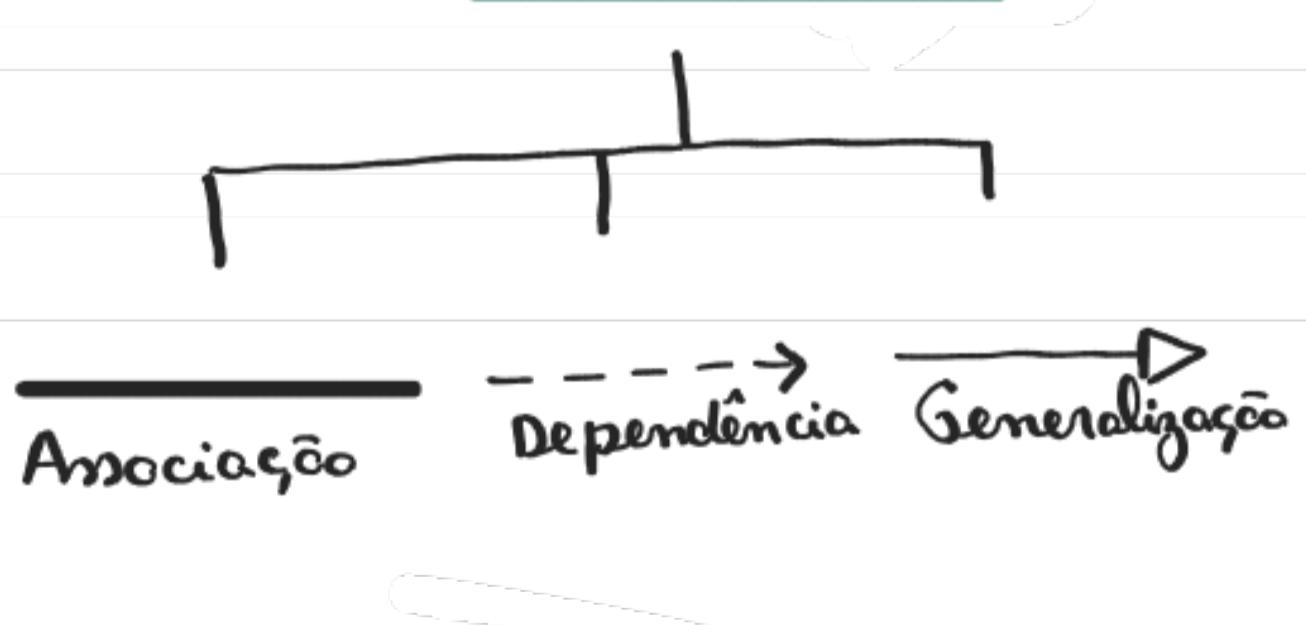
→ Agrupar cóis



→ Anotar cóis

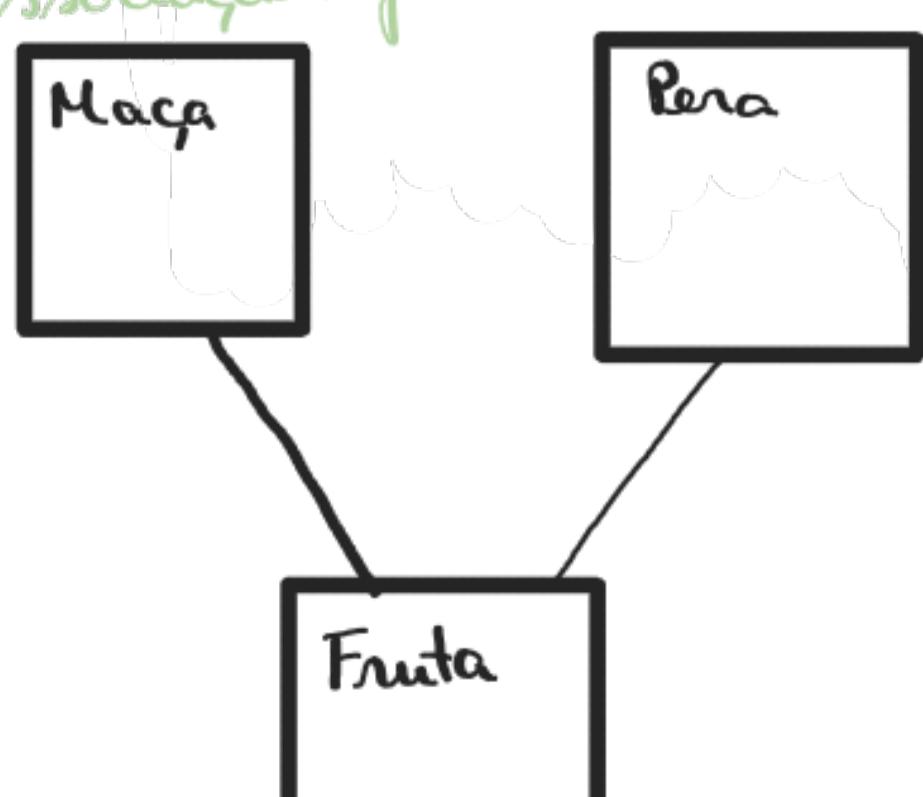


→ Tipos de relações

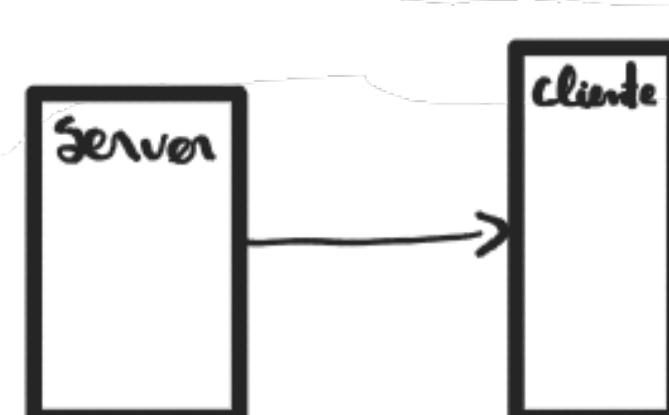


→ Associação

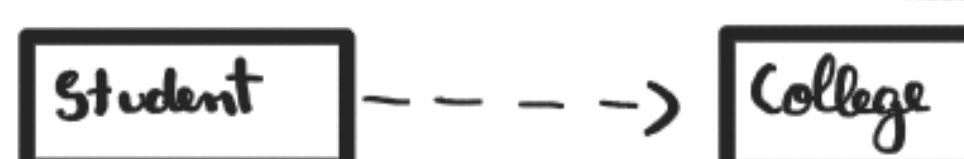
• Associação reflexiva



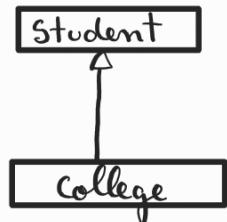
• Associação direcionada



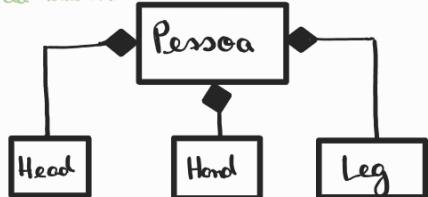
→ Dependência



→ Generalização



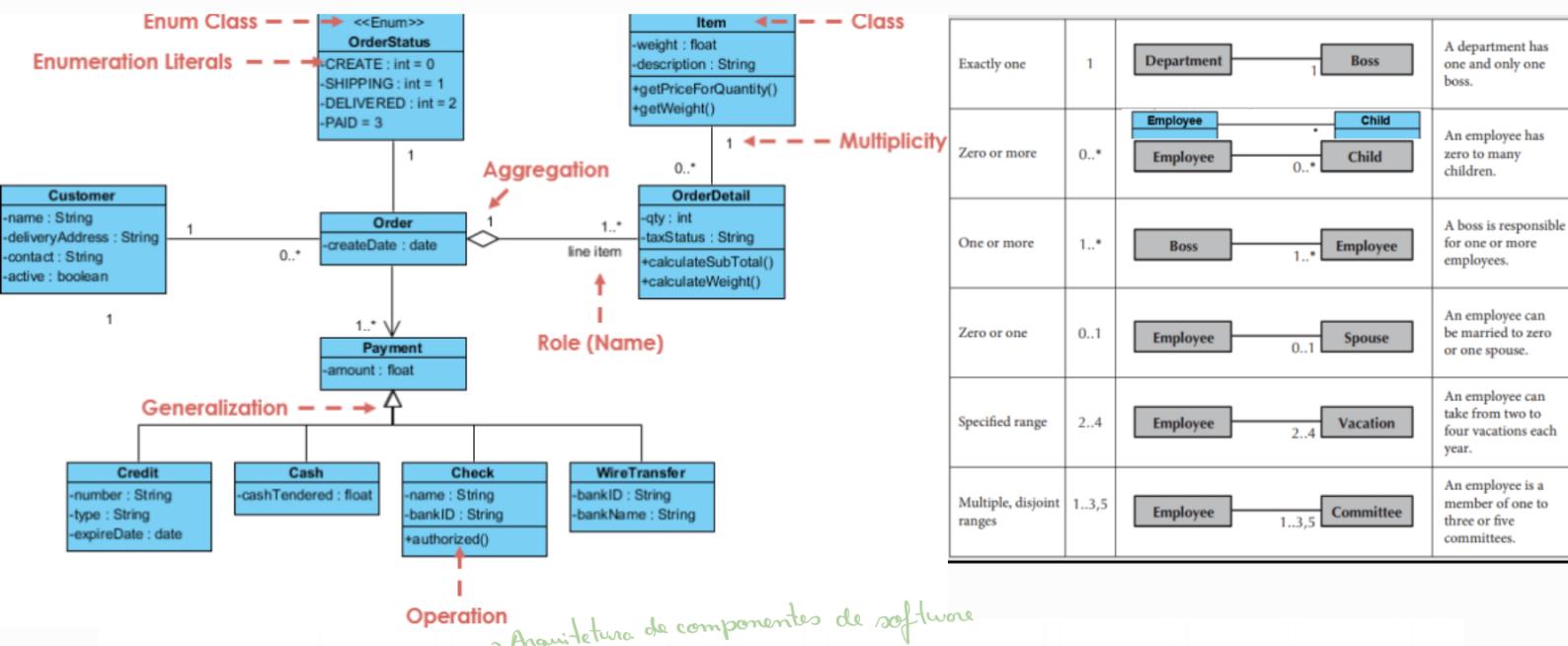
→ Composição



→ Agregação

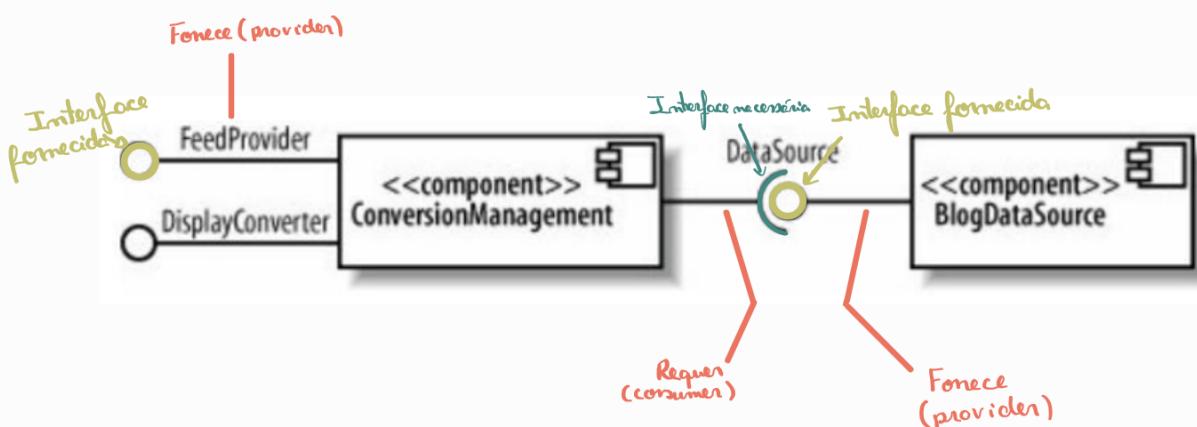


Class Diagram → é um tipo de diagrama de estrutura estática que descreve a estrutura de um sistema, mostrando as classes do sistema, seus atributos, operações (ou métodos) e as relações entre objetos.

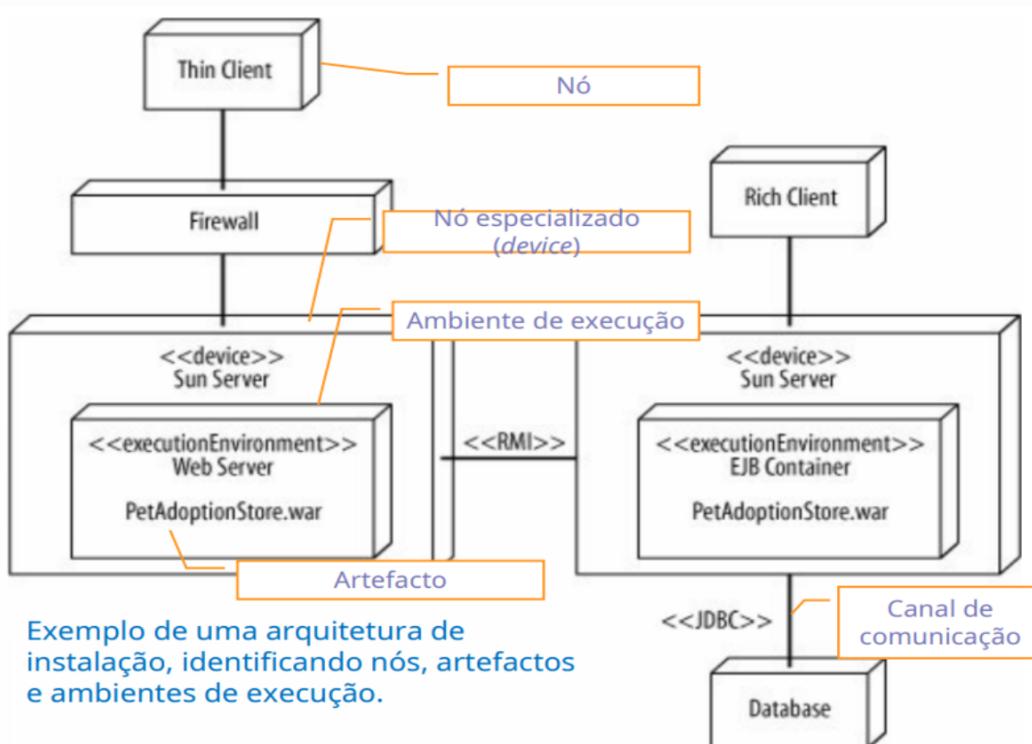


Component Diagram → Os diagramas de componentes UML são usados na modelagem dos aspectos físicos de sistemas orientados a objetos que são usados para visualizar, especificar e documentar sistemas baseados em componentes e também para construir sistemas executáveis por meio de engenharia direta e reversa. Os diagramas de componentes são essencialmente diagramas de classes que focam os componentes de um sistema que geralmente são usados para modelar a visualização da implementação estática de um sistema.

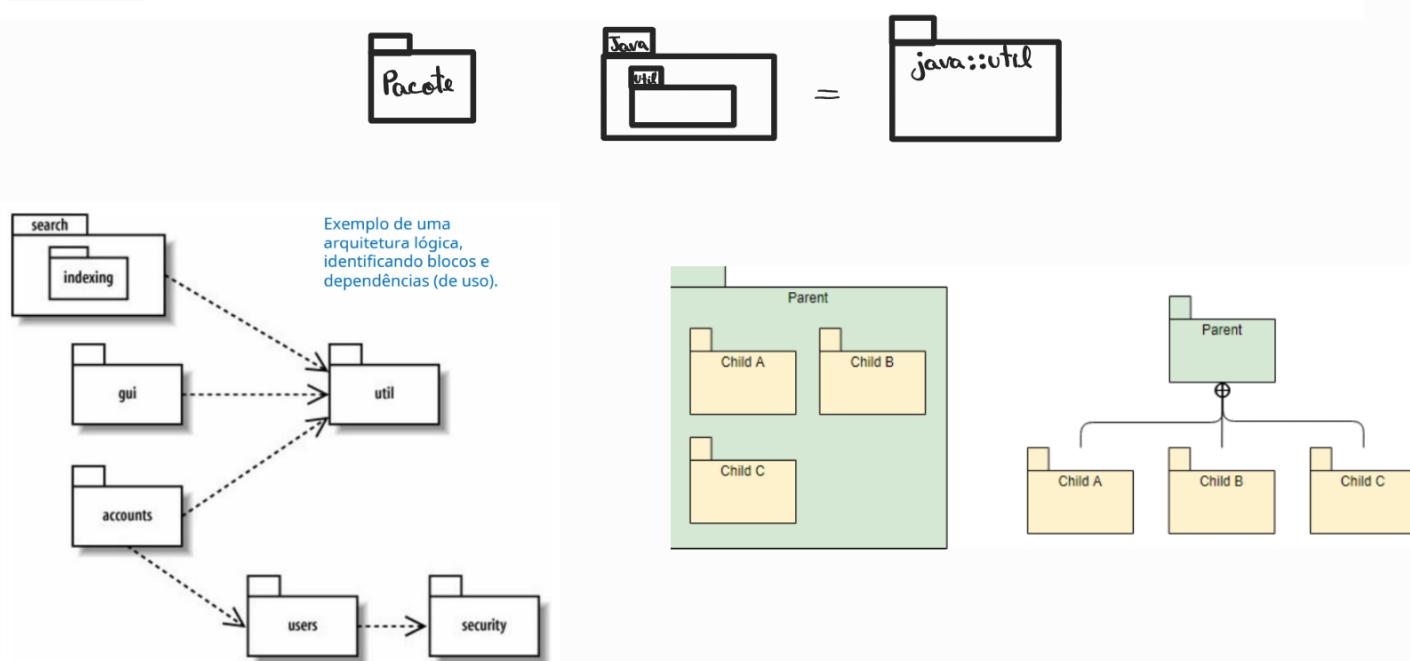
Interações entre componentes



Deployment Diagram → Um diagrama de implementação da UML é um diagrama que mostra a configuração dos nós de processamento de tempo de execução e os componentes que residem neles. Os diagramas de implementação são um tipo de diagrama de estrutura usado na modelagem dos aspectos físicos de um sistema orientado a objetos. Eles costumam ser usados para modelar a visualização de implantação estática de um sistema (topologia do hardware).



Package Diagram → O diagrama de pacotes, um tipo de diagrama estrutural, mostra o arranjo e a organização dos elementos do modelo em projetos de média e grande escala. O diagrama de pacotes pode mostrar a estrutura e as dependências entre subsistemas ou módulos, mostrando diferentes visões de um sistema, por exemplo, como um aplicativo de várias camadas (também conhecido como multcamadas) - modelo de aplicativo com várias camadas.

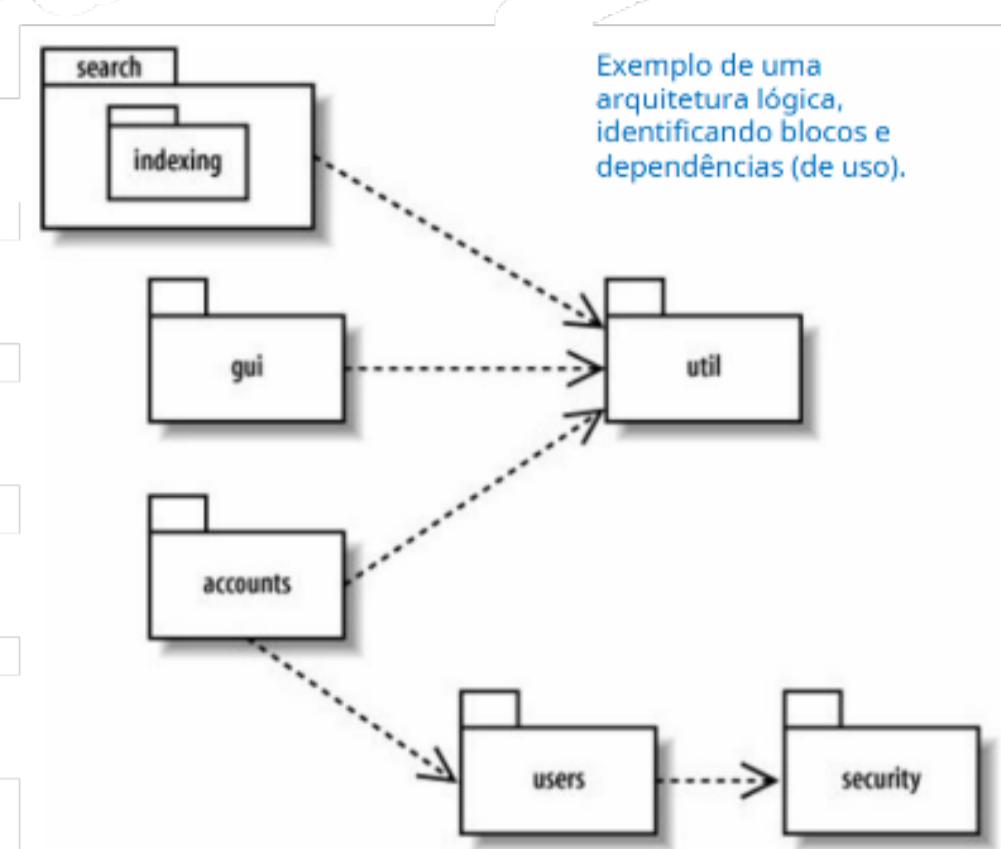


! Os últimos 3 diagramas são muito importantes na arquitetura

① Arquitetura lógica do Software

- Organização geral dos blocos de software independentemente da implementação

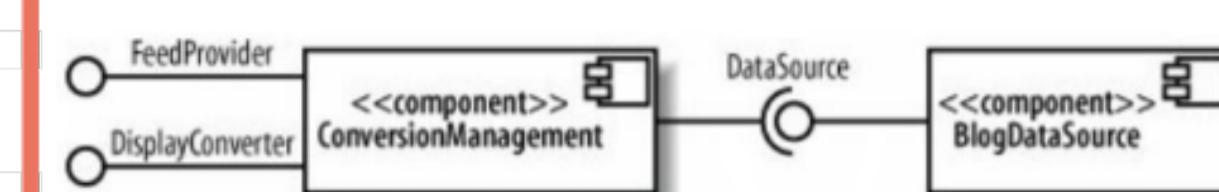
Diagrama de pacotes



② Arquitetura de componentes de SW.

- Peças construídos com uma tecnologia concreta
- Construção modular ("existem já feitas")

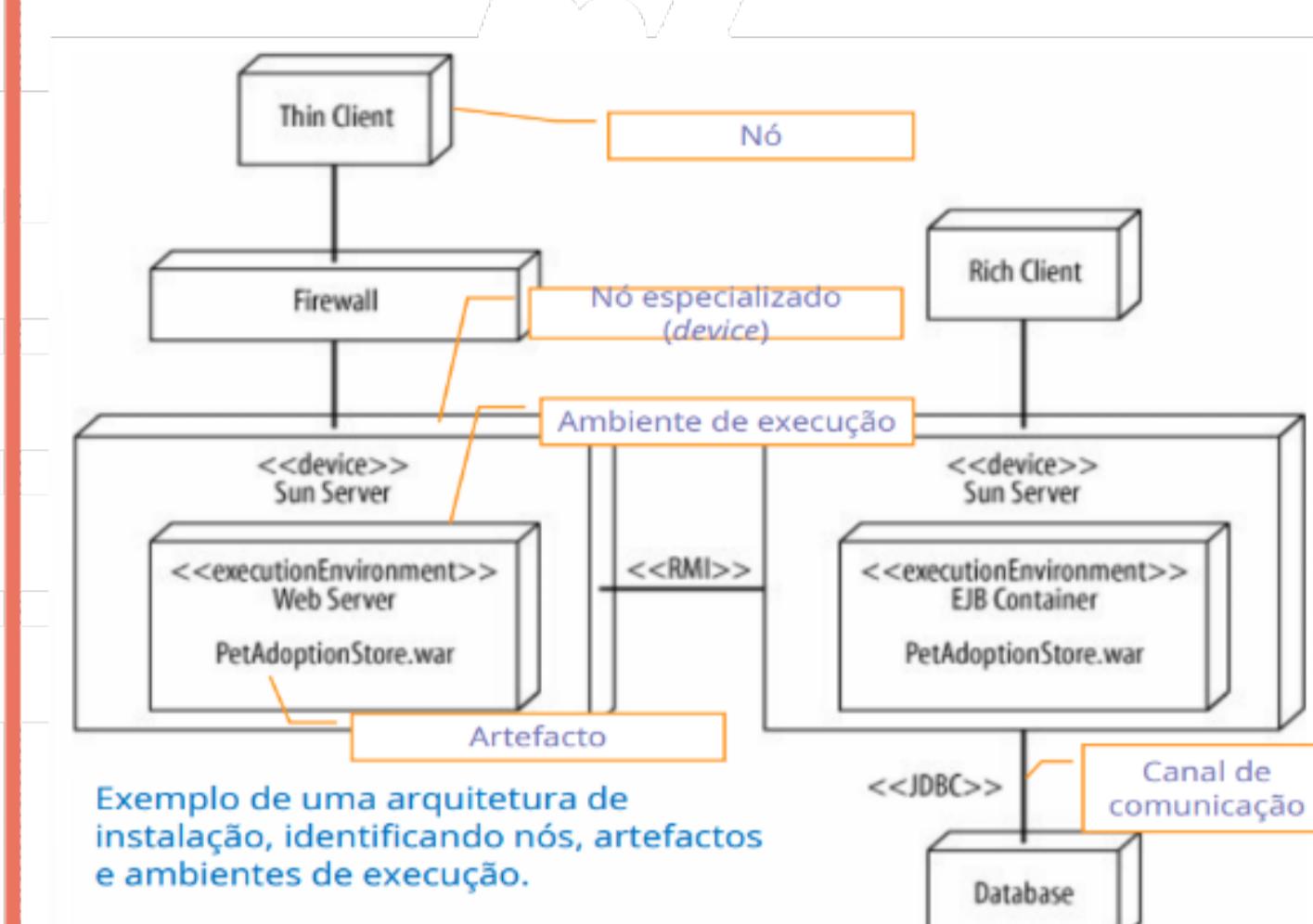
Diagrama de componentes



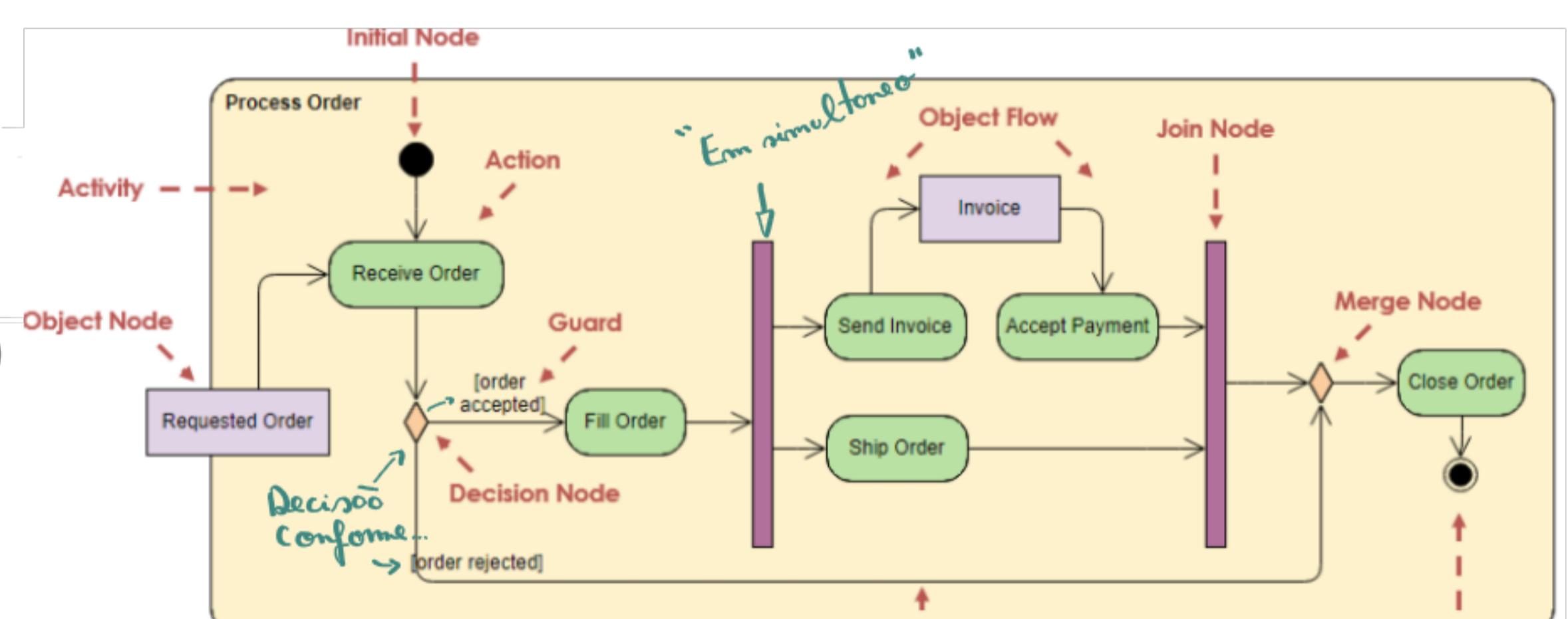
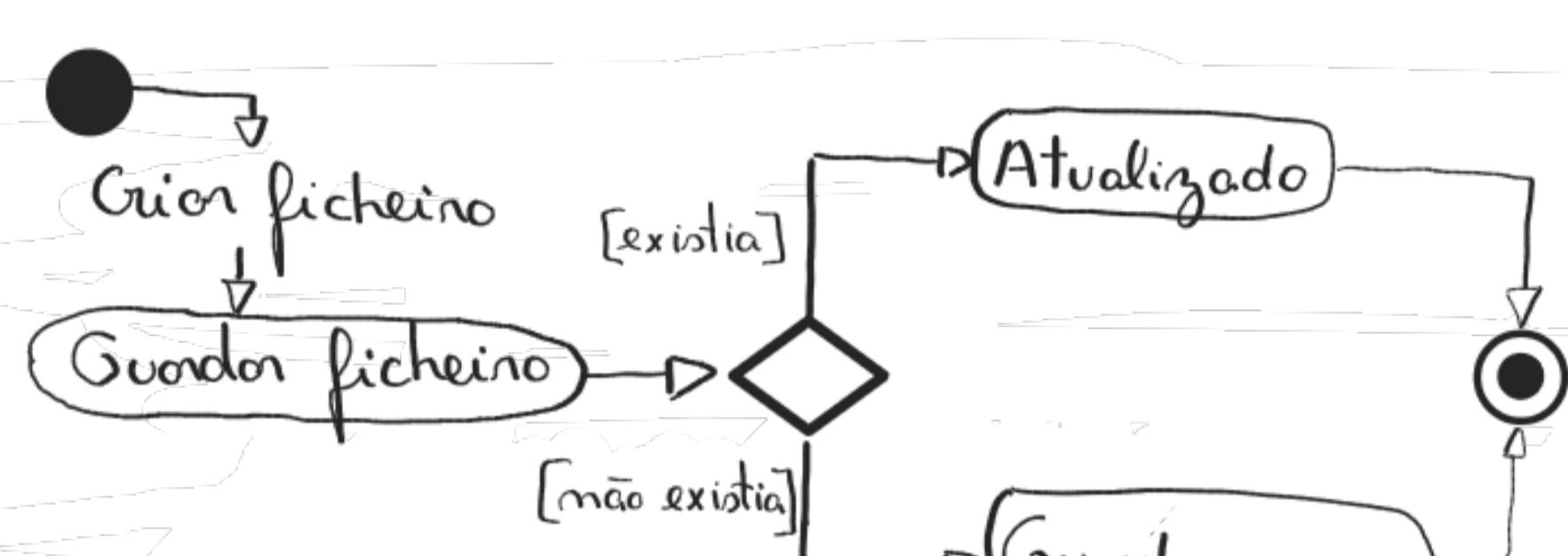
③ Arquitetura de instalação

- Visão dos equipamentos e configuração de produção (conectividade, distribuição...)

Diagrama de implementação



Activity Diagram → O diagrama de atividades é outro diagrama comportamental importante na UML para descrever aspectos dinâmicos do sistema. O diagrama de atividades é essencialmente uma versão avançada do fluxograma que modela o fluxo de uma atividade para outra.



Quando?

- Modelar fluxos do sistema
- Descrever algoritmos
- Descrever sequência de interações entre atores e sistema

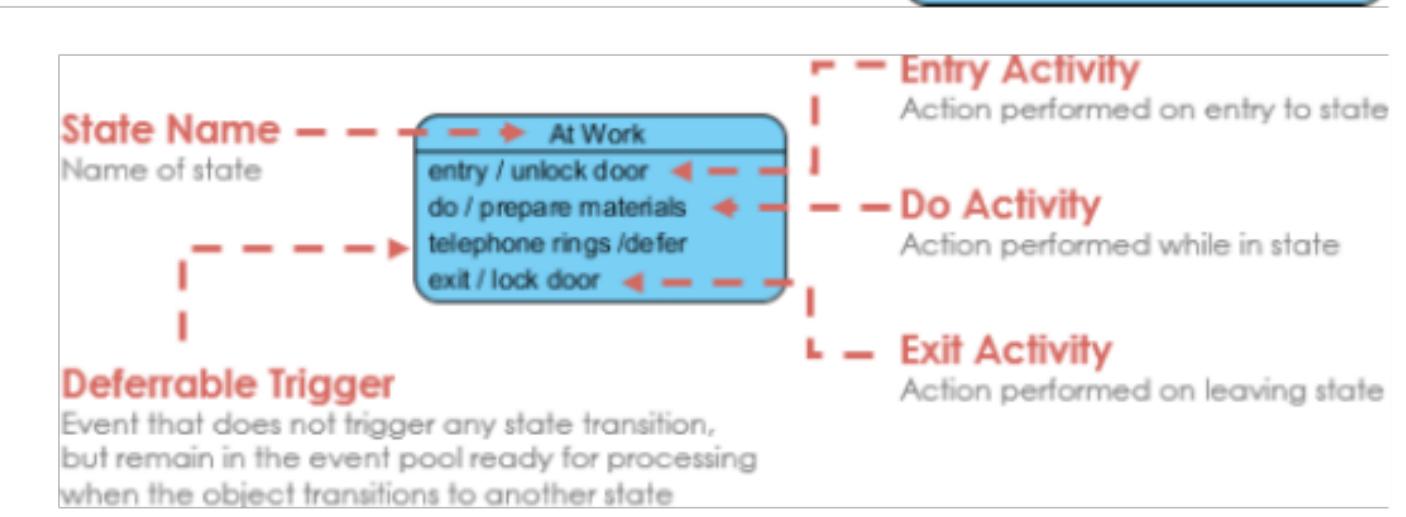
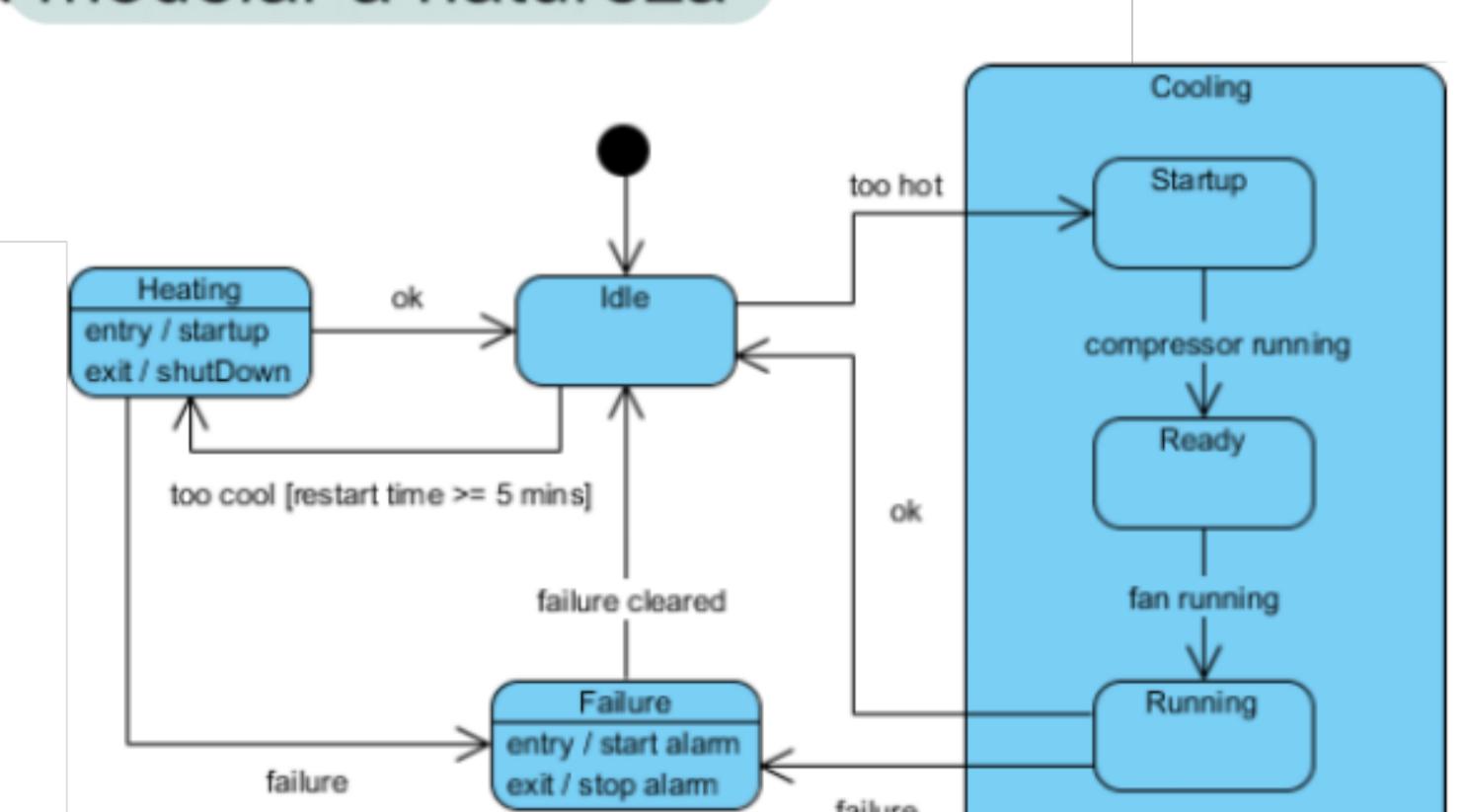
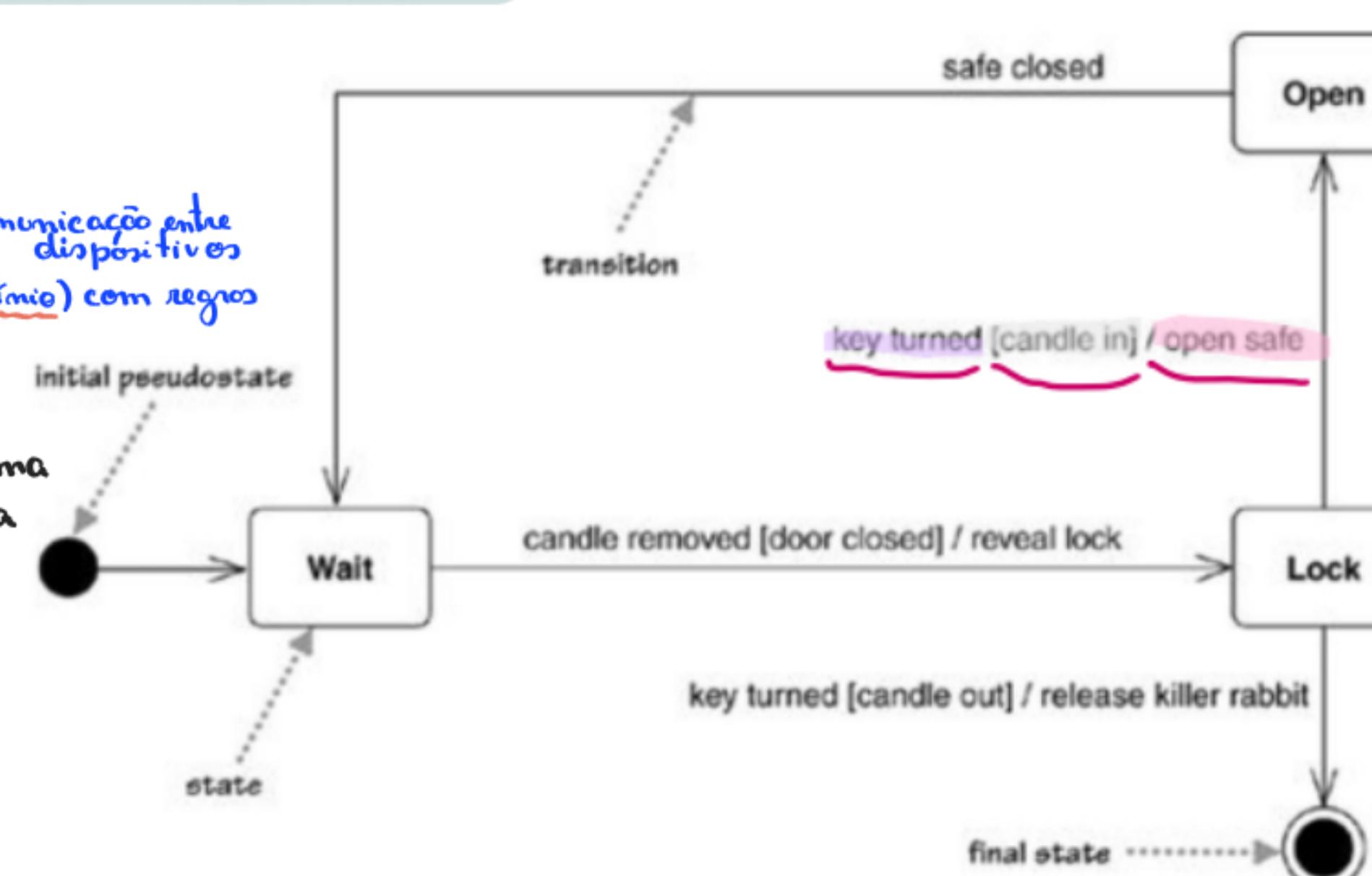
→ "Apenas quando a situação exige um evolução de comportamento interessante"

State Machine Diagram → Diagramas de Máquina de Estado UML mostram os diferentes estados de uma entidade. Os diagramas de máquina de estado também podem mostrar como uma entidade responde a vários eventos mudando de um estado para outro. O diagrama de máquina de estado é um diagrama UML usado para modelar a natureza dinâmica de um sistema.

Exemplos:

- ATM
- Protocolo de comunicação entre dispositivos
- Objetos (do domínio) com regras relevantes

Eg: estados de uma encomenda



Use Case Diagram → Um diagrama de caso de uso da UML é a forma principal de

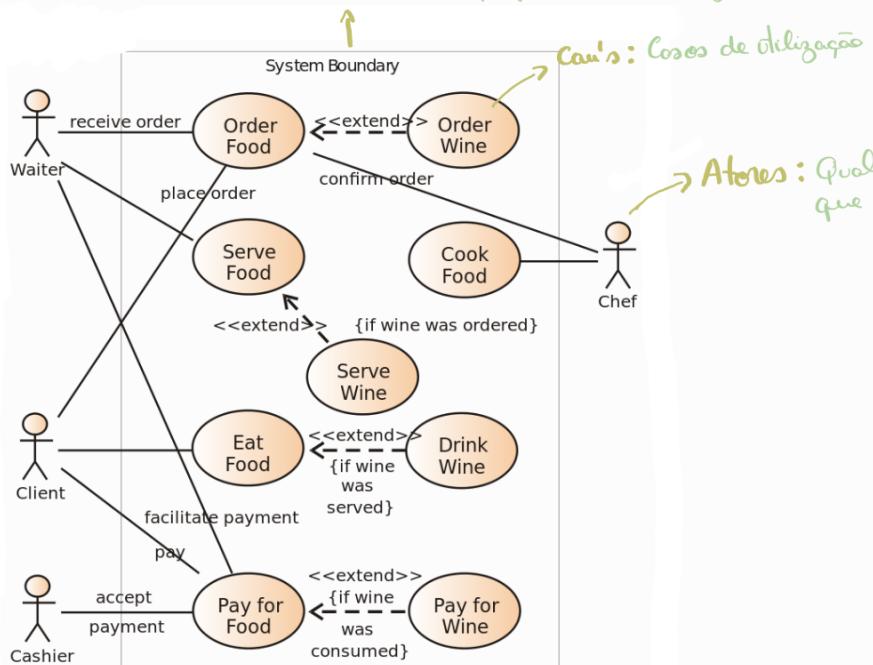
identificar requisitos de sistema / software para um novo programa de software em desenvolvimento. Casos de uso especificam o comportamento esperado (o que), e não o método exato de fazer isso acontecer (como). Os casos de uso, uma vez especificados, podem ser designados como representação textual e visual (como UML). Um

conceito-chave da modelagem de casos de uso é que ela nos ajuda a projetar um sistema a partir da perspectiva do utilizador final. É uma técnica eficaz para comunicar o comportamento do sistema nos termos do usuário, especificando todo o comportamento do sistema visível externamente.

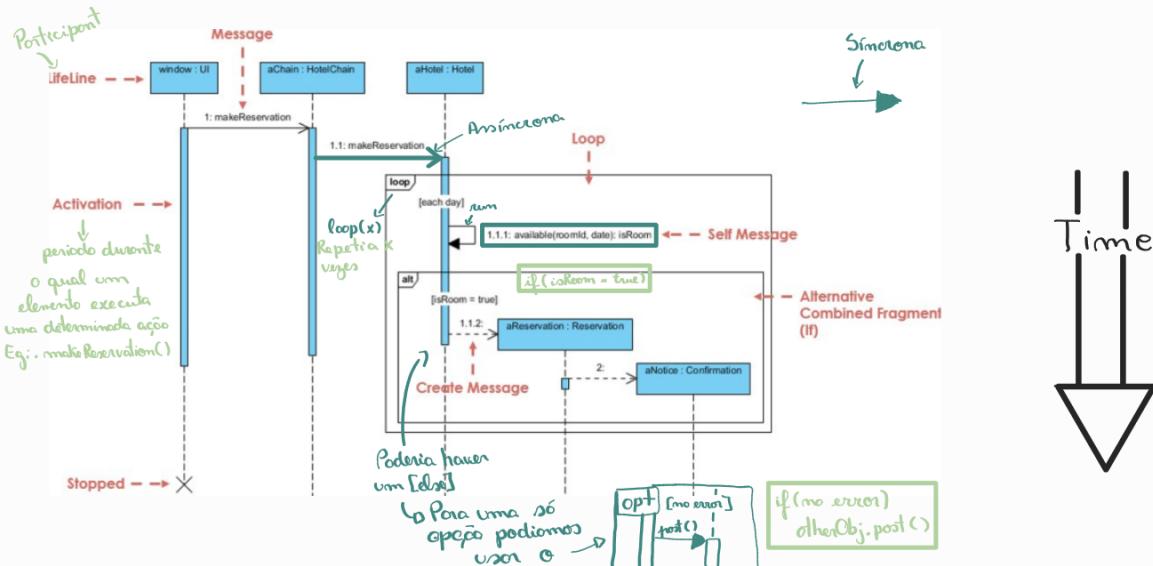


"Fazer login de usuário"
"Fazer uma compra"
"Enviar uma mensagem"

Cenário: Situação particular de utilização do sistema



Sequencial diagram → Um diagrama de sequências da UML mostra como é que os objetos (que seguem uma classe) colaboram com outros, para realizar uma atividade. Também aqui, essa atividade pode ser do nível do domínio ou do nível do código. Na etapa de análise este diagrama explica a colaboração observada para realizar um processo do domínio (os objetos são tipicamente entidades de alto nível (papeis de pessoas ou subsistemas)), já na perspectiva de implementação ajuda a explicar a colaboração entre objetos de código para realizar uma operação (da implementação) e os objetos são instâncias em uma linguagem (Eg. java)



Regras de negócio ("business rule")

→ Política, orientação, norma ou regulamento que define ou restrição algum dos aspectos do negócio.

Ex: - Os clientes devem ser maiores de idade (≥ 18 anos)
- Há uma taxa de administração, a pagar na inscrição, no valor de 50% da mensalidade

⚠️ Não é um requisito de software em si porque têm uma existência para além dos limites de qualquer apl. de software, mas é a origem de vários tipos de requisitos de software.

Não dependem do software, mas têm de ser tidos em consideração na implementação do software?

Modelos de Análise

Lista Funps de requisitos não funcionais:

- Funcionalidade - capacidades
- Usabilidade - estética geral (fatores humanos)
- Confiabilidade - gravidade/capacidade dos falhos
- Performance - velocidade/tempo de resposta/eficiência
- Possibilidade de Suporte - adaptação e manutenção (compatibilidade)

F
U
R
P
S
unctional
ability
eability
erformance
upportability

Práticas de engenharia de requisitos

Saber distinguir entre os tipos de requisitos!

• Distinguir entre requisitos funcionais e não funcionais

Coisas/tarefas que o sistema é capaz de fazer

Funcional:

Uma declaração do que o sistema deve fazer ou uma característica que deve ter

Por norma só levantados na "Elaboração" do OpenUP



- relativo a um processo ou tratamento de dados;
- Captam o comportamento pretendido do sistema. Serviços, funções ou tarefas que o sistema deve realizar. Pode ser captado nos CaU. Pode ser detalhado com diagramas de comportamento: atividades, sequência, etc.

Quão bem deve o sistema fazer a operação

Não-funcional:

- relativo a uma qualidade de sistema.
- Restrições globais num sistema de software E.g.: robustez, portabilidade,... Também designados como atributos de qualidade. Por regra, não afetam apenas um módulo/CaU.



• Distinguir entre abordagens centradas em cenários (utilização) e abordagens centradas no produto para a elicitação de requisitos

Cenários(Utilização) - Uma perspetiva mais abrangente daquilo que é pretendido por quem vai utilizar o sistema e aquilo que pretendem dele.

necessidades dos atores e nos interações com o sistema
A partir de CaUs

Produto - Uma perspetiva mais próxima dos stakeholders para perceber o que os próprios querem do produto final, criando também uma relação de confiança entre os developers e o cliente.

Entrevistas, reuniões, workshops...

processo colaborativo e analítico que inclui atividades para recolher, descobrir, extraer e definir requisitos

=> Focados no negócio + Focados no utilizador

+ Funcionais
+
Não Funcionais

• Justifique que "a elicitação de requisitos é mais que a recolha de requisitos".

O coração do desenvolvimento de requisitos é a elicitação, o processo de identificação das necessidades e restrições das várias partes interessadas para um sistema de software.



Elicitação não é o mesmo que "Reunir requisitos". Também não é uma simples questão de transcrever exatamente o que os utilizadores dizem. Elicitação é um processo colaborativo e analítico que inclui atividades para colecionar, descobrir, extraer e definir requisitos. A elicitação é usada para descobrir negócios, utilizadores, requisitos funcionais e não funcionais, juntamente com outros tipos de informação. A elicitação de requisitos é talvez o aspecto mais desafiador, crítico, propenso a erros e intensivo em comunicação do desenvolvimento de software.

Em resumo:

É preciso fazer perguntas, ouvir o que os stakeholders têm a dizer e analisar os requisitos para garantir que são coerentes, completos e realistas

S	→ Específico
M	→ Mensurável (clara e específica)
A	→ Atingível
R	→ Relevante
T	→ Rotativo no tempo ↳ possível medir o progresso

• Identifique requisitos bem e mal formulados (aplicando os critérios S.M.A.R.T.)

Requisitos de qualidade simplistas, como "O sistema deve ser fácil de usar" ou "O sistema deve estar disponível 24x7" não são úteis. O primeiro é muito subjetivo e vago; o último raramente é realista ou necessário. Nem é mensurável. Tais requisitos fornecem pouca orientação aos desenvolvedores. Assim, o passo final é criar requisitos específicos e verificáveis a partir das informações que foram obtidas em relação a cada atributo de qualidade. Ao escrever requisitos de qualidade, tenha em mente o útil mnemônico SMART:

Idealmente falando, cada objetivo corporativo, departamento e seção deve ser:

- *Specific* – target a specific area for improvement.
- *Measurable* – quantify or at least suggest an indicator of progress.
- *Assignable* – specify who will do it.
- *Realistic* – state what results can realistically be achieved, given available resources.
- *Time-related* – specify when the result(s) can be achieved.

Por outros palavras:

- Um requisito é uma declaração que expressa necessidade e as limitações e condições que lhe são associadas
- A expressão deve incluir um sujeito, um verbo e um complemento

Eg.: O sistema de Faturação [sujeito] deve exibir os fatos pendentes do cliente [acção] por ordem crescente [complemento] em que os fatos devem ser pagos

A modelação do contexto do problema: modelo do domínio/negócio

Caracterizar os conceitos do domínio de aplicação :

- Desenhe um diagrama de classes simples para capturar os conceitos de um domínio de problema. *Saber ler um exerto e retirar os classes!*
- Apresente duas estratégias para descobrir sistematicamente os conceitos candidatos para incluir no modelo de domínio.
 - Procurar numa lista de situações comuns → categorias de classes
 - Explorar documentos /relatórios existentes na área do problema
 - Análise de nomes → explorar descrições do problema à procura dos substantivos.

TP05

Coisas que
tenho de
Saber

- Identificar construções específicas (associadas à implementação) que podem poluir o modelo de domínio (na etapa de análise).

Caracterizar os processos do negócio/organizacionais:

- Leia e desenhe diagramas de atividades para descrever os fluxos de trabalho da organização / negócios.
- Identifique o uso adequado de ações, fluxo de controle, fluxo de objetos, eventos e partições com relação a uma determinada descrição de um processo.
- Relacione os “conceitos da área do negócio” (classes no modelo de domínio) com fluxos de objetos nos modelos de atividade.

→ "ciclo de vida dos objetos"
→ O que é que o Sistema deve fazer?
→ [do ponto de vista do observador externo;
caixa fechada]

Modelação funcional com casos de utilização

- Descrever o processo usado para identificar casos de utilização.
 1. Identificar a fronteira do sistema
 2. Identificar os atores que, de alguma forma, interagem com o sistema
 3. Para cada ator, identificar os objetivos/motivações para usar o sistema
 4. Definir os CaU que satisfazem os objetivos dos atores
- Descrever os elementos essenciais de uma especificação de caso de uso.
 - Um identificador único e um nome sucinto que indica o objetivo do utilizador;
 - Uma breve descrição textual que descreve o propósito do caso de uso;
 - Uma condição de disparo que inicia a execução do caso de uso;

- Zero ou mais **pré-condições** que devem ser satisfeitas antes que o caso de uso possa começar;
- Uma ou mais **pós-condições** que descrevem o estado do sistema após o caso de uso ser concluído com êxito;
- Uma lista numerada de etapas que mostra a **sequência de interações** entre o ator e o sistema - um diálogo - que leva das condições prévias às pós-condições.

fluxo típico / fluxo alternativo

- **Explicar o uso complementar de diagramas de casos de utilização, diagramas de atividades e narrativas de casos de utilização**
 - O **modelo de casos de utilização** fornece o contexto para a descoberta, partilhada e compreensão dos requisitos do sistema.
 - Um modelo de caso de utilização é um modelo de todas as maneiras úteis de usar um sistema. Isso permite apreender rapidamente o âmbito do sistema – o que é incluído e o que não é – e dar à equipa uma visão global de que o sistema vai fazer. *↪ "big-picture"*
 - A **visão gerada pelos modelos de CaU** é conseguida sem nos perdemos nos detalhes dos requisitos ou a parte interna do sistema.
 - O propósito de uma **narrativa de CaU** é contar a história como o sistema e os seus atores trabalham em conjunto para atingir um determinado objetivo.

- **Explicar o sentido da expressão “desenvolvimento orientado por casos de utilização”**
 - O desenvolvimento orientado por casos de utilização (Use Case Driven Development) é a prática que descreve como capturar requisitos com uma combinação de casos de uso e requisitos de todo o sistema e, em seguida, orientar o desenvolvimento e os testes a partir desses casos de uso. *→ Assim, atendemos às necessidades dos usuários de maneira clara e precisa a partir da análise dos Casos no início do processo de desenvolvimento*

Resumo

- **Explicar os seis “Princípios para a adoção de casos de utilização” propostos por Ivar Jacobson (com relação ao “Use Cases 2.0”)**

Os casos de utilização devem:

Foco nos usuários: ser criados com base nas necessidades e expectativas dos usuários, e não nas especificações técnicas do sistema.

Orientação para o negócio: descrever como o sistema é usado para realizar atividades de negócios ou organizacionais.

Visão ampla: considerar todas as formas possíveis de uso do sistema, incluindo as exceções e os casos de uso alternativos.

Análise: ser analisados para identificar os requisitos funcionais e não funcionais do sistema.

Especificação: ser especificados de maneira clara e precisa, incluindo os requisitos funcionais e não funcionais do sistema.

Colaboração: criados em colaboração com os usuários, os gerentes de projeto e outros stakeholders envolvidos no projeto

Os 6 princípios para adoção de Cas's

Princípio 1: Manter simplicidade contando histórias (stories):

Contar histórias é a melhor maneira de comunicar o que um sistema deve fazer e para que toda a equipa trabalhe no sistema concentrando-se nos mesmos objetivos. Os casos de uso capturam os objetivos do sistema. **Para compreender um caso de uso, contamos histórias. As histórias cobrem como alcançar o objetivo e como lidar com os problemas que ocorrem no caminho.** Os casos de uso fornecem uma maneira de identificar e capturar todas as histórias diferentes, mas relacionadas, de forma simples e comprehensível. Isso permite que os requisitos do sistema sejam facilmente capturados, partilhados e compreendidos.

Princípio 2: Entender a grande figura (big picture):

Se o sistema em desenvolvimento é grande ou pequeno, seja um sistema de software, um sistema de hardware ou sistema de negócios, entender a big picture é essencial. Sem uma compreensão do sistema como um todo, será impossível tomar as decisões certas sobre o que incluir no sistema, o que deixar de fora, o custo e que benefício isso proporcionará.

Um diagrama de casos de uso é uma maneira simples de apresentar visão geral dos requisitos de um sistema.



Princípio 3: Foco no valor

Ao tentar compreender como um sistema será utilizado, é sempre importante haver foco no valor que ele proporcionará aos seus utilizadores e partes interessadas (stakeholders). O valor é gerado apenas se o sistema é realmente utilizado, por isso deverá haver uma maior preocupação em como o sistema será utilizado do que em listas longas de funcionalidades ou características que este oferecerá. **Os casos de uso fornecem esse foco, concentrando-se em como sistema será usado para atingir uma meta específica para um determinado utilizador.**

Princípio 4: Construa o sistema em fatias

A maioria dos sistemas exige muito trabalho antes de ser utilizado e pronto para uso operacional. Têm muitos requisitos, sendo que a maioria depende da implementação de outros requisitos, até que possam ser cumpridos e que possa ser entregue valor. É sempre

um erro tentar construir um sistema deste género de uma só vez. O sistema deve ser construído em fatias, cada uma delas com valor para os usuários. A receita é simples. Primeiro, identificar a coisa mais útil que o sistema tem de fazer e focar nisso. Depois, pegar nessa coisa e separar em “fatias” mais pequenas e simples. Decidir sobre os casos de teste que representam a aceitação dessas fatias. Escolher a fatia mais central que percorre todo o conceito de ponta a ponta, ou o mais próximo possível disso. Estudá-la em equipa e começar a construí-la. É esta a abordagem adotada pelo Use-Case 2.0, em que os casos de uso são separados em “fatias” de modo a fornecerem itens de trabalho de tamanho adequado, e onde o próprio sistema evolui fatia a fatia.

Princípio 5: Entregar o sistema em incrementos

A maioria dos sistemas de software evolui através de várias gerações. Não são produzidos de uma só vez; São construídos como uma série de lançamentos, cada um construído sobre o anterior. Cada incremento fornece uma versão demonstrável ou utilizável do sistema. É assim que todos os sistemas devem ser produzidos.

Princípio 6: Adaptar-se para atender às necessidades da equipa

Infelizmente, não existe uma solução única para todos os desafios no desenvolvimento de software; equipas diferentes e situações diferentes exigem estilos diferentes e diferentes níveis de detalhe. Independentemente das práticas selecionadas, é necessária a certeza de que estas são adaptáveis o suficiente para atender às necessidades permanentes da equipa. Cabe à equipa decidir se precisam ou não ir além dos essenciais, acrescentando detalhes de uma forma natural à medida que se deparam com os problemas que o essencial não resolve.

• Compreender a relação entre requisitos e casos de utilização

→ Os CaU contam histórias que mostram os requisitos funcionais em contexto, num formato que legível e compreensível para o utilizador final.

• Identificar as disciplinas e atividades relacionadas aos requisitos no OpenUP

→ Concepção

- Levantamento de requisitos
- Elicitação de requisitos (criação de CaU's)
- Análise de requisitos
- Priorização dos requisitos

→ Elaboração

- Refinamento dos requisitos
- Revisão e aprovação dos requisitos

→ Construção

- Verificação de requisitos

→ Transição

- Validação dos requisitos
- Aceitação dos requisitos



• É importante lembrar que o processo de levantamento, elicitação e análise de requisitos é um processo iterativo e contínuo que ocorre em todos os fases do OpenUP. Isto significa que os requisitos podem ser revisados e ajustados a qualquer momento, conforme o projeto avança e novas informações são coletadas.

Modelação estrutural

→ Quais são os "pontos"/"coisas" que constituem e o relacionamento estrutural entre eles

- Justifique o uso de modelos estruturais na especificação de sistemas

Modelos estruturais

⇒ Estrutura de um sistema
Como os componentes são organizados e como eles se relacionam entre si

↓
Diag. de Classes

Diag. de Componentes

Diag. de Pacotes

Diag. de implementação

Ajuda a compreender a arquitetura do sistema

Facilita a comunicação

Facilita a manutenção

Facilita a reutilização

PL-T2018

Considere o requisito a seguir formulado, relativo à operação de uma loja online, no contexto da análise de requisitos de um sistema de software:

R: "O Portal deve estar conforme as disposições do Regulamento Geral de Proteção de Dados (RGPD)".

✗ É um requisito funcional, mas inadequado, porque não é implementável na sua totalidade.

✗ É um exemplo de um atributo de qualidade, relacionado com a segurança do sistema.

✗ É um exemplo de um requisito não funcional, relacionado com a compatibilidade do sistema (*supportability*).

○ Deve ser desagregado para num nível de detalhe adequado, pois inclui vários requisitos funcionais.

✗ Não é um bom exemplo de um requisito, porque não é verificável.

P10. T2019

O sistema VitalsRecorder, para aquisição de sinais vitais de um grupo de participantes numa experiência, define o seguinte requisito: "R: As recolhas de dados em curso nos dispositivos móveis devem continuar mesmo quando ocorra a desconexão pontual entre algum dispositivo e o tablet de controlo".

✗ É um requisito de desempenho, relacionado com o tempo de resposta das unidades móveis.

✗ É um requisito de usabilidade, crítico para estabelecer a facilidade de utilização do sistema.

○ É um requisito de fiabilidade, que foca a integridade e a recuperação das operações, face a falhas da camada física.

✗ É um requisito relacionado com a manutenção do sistema (*maintainability*), visto que se pretende assegurar a continuidade das operações.

✗ É um requisito funcional (adequação à função) e não está relacionado com a lista de atributos do sistema.

23/37

"A técnica de dominar a complexidade é conhecida desde os tempos antigos". Ao projetar um sistema de software complexo, é essencial decompô-lo em partes mais pequenas, cada uma delas podendo ser então refinada de forma independente. Em vez de tentar ~~detalhar~~ ~~compreender~~ o sistema como um todo, precisamos apenas de compreender algumas partes (ao invés de todas partes) de uma só vez. A complexidade inerente do software força a divisão do sistema em partes simples, sendo que é nessa divisão e decomposição que entram os modelos estruturais.

- Explicar a relação entre os diagramas de classe e de objetos.

Os **Diagramas de Classes** mostram em que consistem os objetos do sistema (membros) e o que são capazes de fazer (métodos).

Os **Diagramas de Objetos** mostraram como os objetos do sistema estão interagem uns com os outros a determinado momento, e que valores esses objetos contêm quando o programa está nesse estado.

UML → DIAGRAMAS DE CLASSES



- Rever um modelo de classes quanto a problemas de sintaxe e semânticos, considerando uma descrição do um problema de aplicação.
- Descreva os tipos e funções das diferentes associações no diagrama de classes.
- Identifique o uso adequado da associação, composição e agregação para modelar a relação entre objetos.
- Identifique o uso adequado de classes de associação.

Modelação de comportamento

→ Como é que os partes colaboram/interagem ao longo do tempo? [perspectiva dinâmica]

- Explique o papel da modelagem de comportamento no SDLC

Os diagramas comportamentais UML visualizam, especificam, constroem e documentam os aspectos dinâmicos de um sistema. Os diagramas comportamentais são dos seguintes tipos: diagramas de casos de uso, diagramas de estado e diagramas de atividades. Estes diagramas dividem-se num subtipo, Diagramas de interação → Diagrama de Sequência.

→ Mais específico

UML → DIAGRAMAS DE SEQUÊNCIA E ESTADO

- Entenda as regras e diretrizes de estilo para diagramas de seqüência, comunicação e estado
- Entenda a complementaridade entre diagramas de sequência e comunicação
~~Diagramas de sequência descrevem a comunicação entre os participantes~~
- Analise criticamente os modelos de diagramas de sequência existentes para descrever a cooperação entre dispositivos ou entidades de software.

- P11_T2019
Que relação pode ser estabelecida entre a arquitetura de um sistema de software e a UML?
- A UML recomenda a utilização de arquiteturas baseadas em componentes.
 - A UML permite descrever a vista física de instalação de um sistema, mas não tem diagramas para ajudar a caracterizar a organização lógica (blocos de alto nível e associações).
 - Podemos mostrar as interações entre objetos recorrendo a diagramas de sequência da UML para clarificar as colaborações entre componentes.
 - A arquitetura pode ser comunicada recorrendo a diagramas de UML que suportam as noções de "módulos" e "dependências". Dessa forma, podemos mostrar as partes constituintes de um sistema e as relações estruturais.
 - Não há uma relação entre os dois conceitos; a UML não prevê diagramas de arquitetura.

Modelos no desenho e implementação

Vistas de arquitetura

- Explicar as atividades associadas ao desenvolvimento de arquitetura de software.

Uma arquitetura é o conjunto de decisões significativas em relação à organização de um sistema de software

A arquitetura consiste na seleção dos **elementos estruturais** e suas **interfaces** pelas quais o sistema é composto, juntamente com seu comportamento (especificado na colaboração entre esses elementos), a composição destes elementos estruturais e comportamentais em subsistemas progressivamente maiores e o estilo de arquitetura que orienta esta organização, estes elementos e suas interfaces, as suas colaborações e a sua composição.

Falta coisas!

• Comodos
• Componentes
• Serviços

Exemplos de requisitos com e sem impacto na arquitetura

Regras para investir na definição de um

P6. T2018

A arquitetura trata da tomada das grandes decisões técnicas em relação ao sistema a desenvolver, tendo em conta os atributos de qualidade pretendidos. Um exemplo de um assunto/decisão de arquitetura é:

- Concretizar os cenários de interoperação com sistemas externos e as tecnologias para os implementar.
- Definir estratégias de redundância e distribuição de carga para garantir a disponibilidade do sistema em picos de 1000 sessões simultâneas.
- Isolar a identidade dos utentes numa base de dados independente da base de dados de operações, para aumentar a proteção dos dados pessoais.
- Todas as opções anteriores são corretas.
- Nenhuma das opções anterior é correta.

- Identifique os elementos abstratos de uma arquitetura de software

nodes = class / module / package / layer / sub system / system

Interconnections = communications / control

"Arquitetura é uma Abstração" → omite intencionalmente certas informações sobre os elementos que não são úteis para o raciocínio sobre o sistema

Detalhes privados dos elementos (têm a ver exclusivamente com a implementação interna) não são da arquitetura.

- Identifique as camadas e partições numa arquitetura de software por camadas.

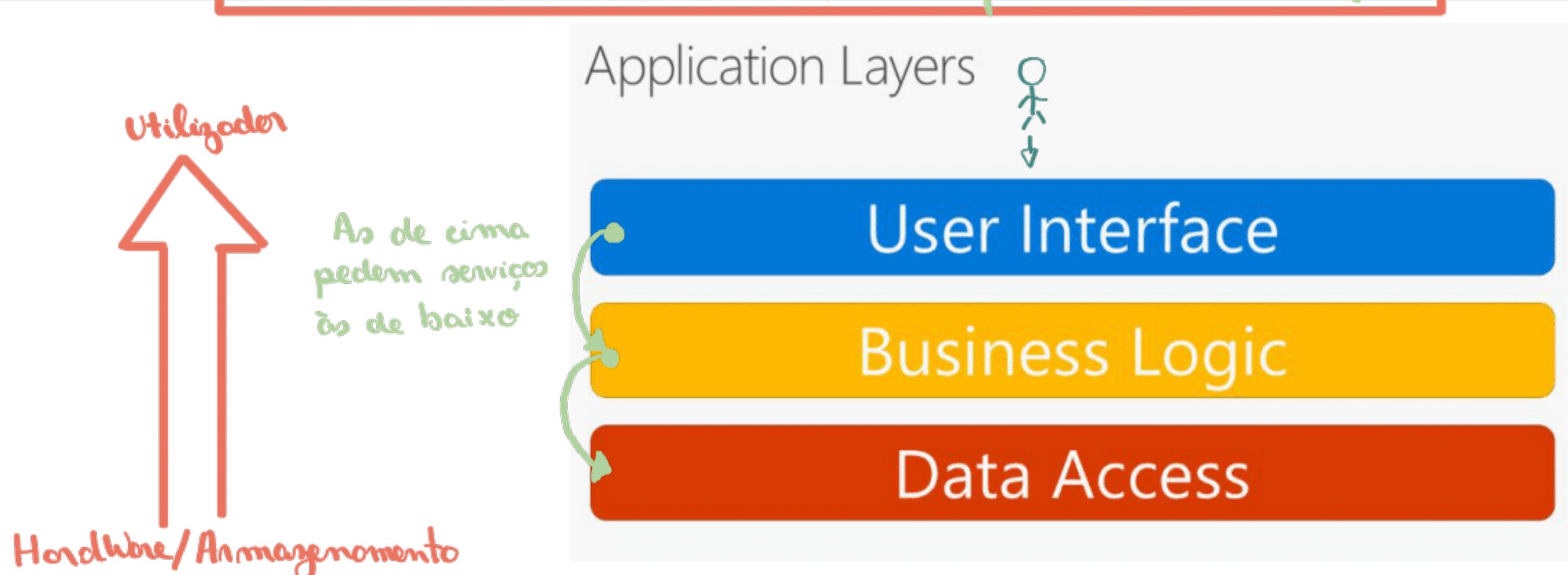
→ Divisão modular da solução de software em camadas/níveis

→ As camadas são sobrepostas

→ Cada camada tem uma especialização

→ Camadas "em cima" pedem serviços às camadas "de baixo"

→ Não se pode saltar camadas. (os componentes de cada camada "falam" com os componentes adjacentes)



(UI) → Interacção com o utilizador

(BL) → Processa solicitação do utilizador e realiza tarefas necessárias (algoritmos)

(DA) → Armazenar e recuperar dados

Usando esta arquitetura, os **utilizadores fazem solicitações** por meio da camada da interface do utilizador (UI), que **interage apenas com a BLL**. A BLL, por sua vez, pode chamar a DAL para solicitações de acesso a dados.

→ **AUMENTA A COESÃO, DIMINUI O ACOPOLAMENTO.**

- Mais fáceis de entender e manter
- Mais fáceis de expandir e modificar, pois não dependem fortemente dos outros módulos ou componentes

Foco em uma única tarefa

↳ Dependência de outros módulos ou componentes

"Arquitetura evolutiva" proposta no OpenUP:

(foco na fase de "elaboração")

Ideia de fundo: a escolha de arquitetura comporta riscos que devem ser controlados cedo, experimentando as capacidades-chave



- Analisar os principais questões que afetam a solução
- Documentar decisões de arquitetura para garantir que foram avaliados e comunicados
- Implementar e testar capacidades-chave como forma de lidar com os desafios da arquitetura
- Evoluir ao longo do tempo, a par com o trabalho de implementação normal

UML → Diagrama de pacotes, Diagrama de componentes

- Rever criticamente um diagrama de pacotes existente para ilustrar uma arquitetura lógica
- Rever criticamente um diagrama de componente existente para descrever as partes tangíveis do software
- Analise criticamente um diagrama de implementação existente para descrever a instalação de um sistema

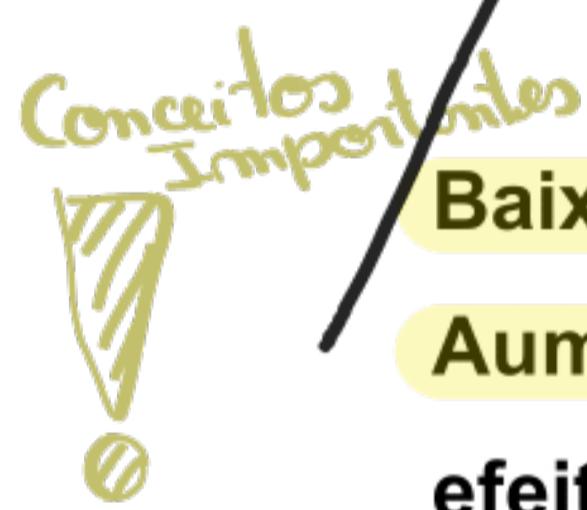
Classes e desenho de métodos (perspectiva do programador)

- Explicar os princípios de baixo acoplamento e alta coesão no desenho por objetos.

Dependências com outras classes
Coupling → Mede a força/intensidade da dependência de uma classe de outras A classe

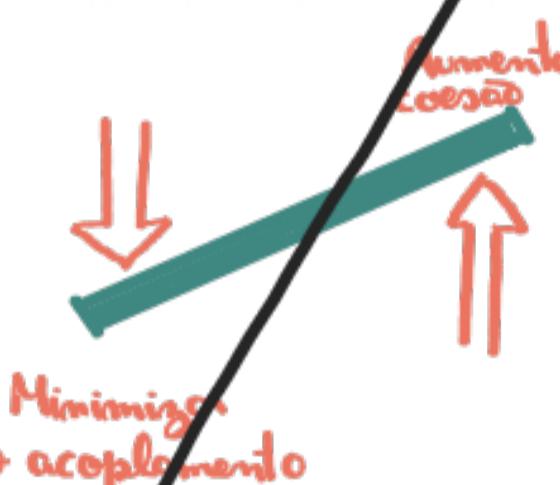
C1 está acoplada com C2 se precisa de C2, direta ou indiretamente. Uma classe que depende de outras 2 tem um "coupling" mais baixo que uma que dependa de 8.

Foco único
Coesão → Mede a força/intensidade do relacionamento dos elementos de uma classe entre si. Todas as operações e dados de uma classe devem estar natural e diretamente relacionados com o conceito que a classe modela. Uma classe deve ter um foco único (vs. responsabilidades desgarradas).



Baixar coupling = Reduzir o impacto da mudança

Aumentar coesão = manter os objetos focados, compreensíveis, gerenciáveis e, como efeito colateral, oferecer suporte a baixo acoplamento



Pouco Importante

- Enumerar e descrever os princípios do GRASP (Larman)

Generic Responsibility Assignment Principles

→ Baixo Coupling

→ Alta Cohesion

→ Information Expert

→ Creator → *Classe ou objeto que cria um objeto deve ser responsável por configura-lo corretamente*

→ Controller

Classe ou objeto que inicia um processo ou ação deve ser responsável por gerenciar o processo da ação até ao fim

Quem deve ser responsável por conhecer algum pedaço de informação?

→ Information expert

→ Atribuir a responsabilidade à informação especialista, ou seja, a classe que tem a informação necessária para cumprir a responsabilidade.

P8. T2019

No desenho de código por objeto, devemos observar princípios que levam a soluções mais fáceis de expandir e manter.

a) Uma classe VideoClip contém métodos para editar o trecho de vídeo que representa. Desde que os seus métodos se foquem apenas em atributos do vídeo, a coesão é mantida.

✗ A elevada coesão de um módulo significa que não usa dados de outros objetos para realizar as suas operações.

✗ As classes de um sistema devem ter um conhecimento abrangente (do desenho) das outras classes; quando um método enviar mensagens apenas para objetos próximos/conhecidos, favorecem a coesão interna.

✗ A colaboração entre objetos, desde que baseada na invocação de métodos públicos, não influencia o acoplamento.

✗ A busca do baixo acoplamento e elevada coesão no desenho por objetos é impossível de atingir, porque são ideias contraditórias.

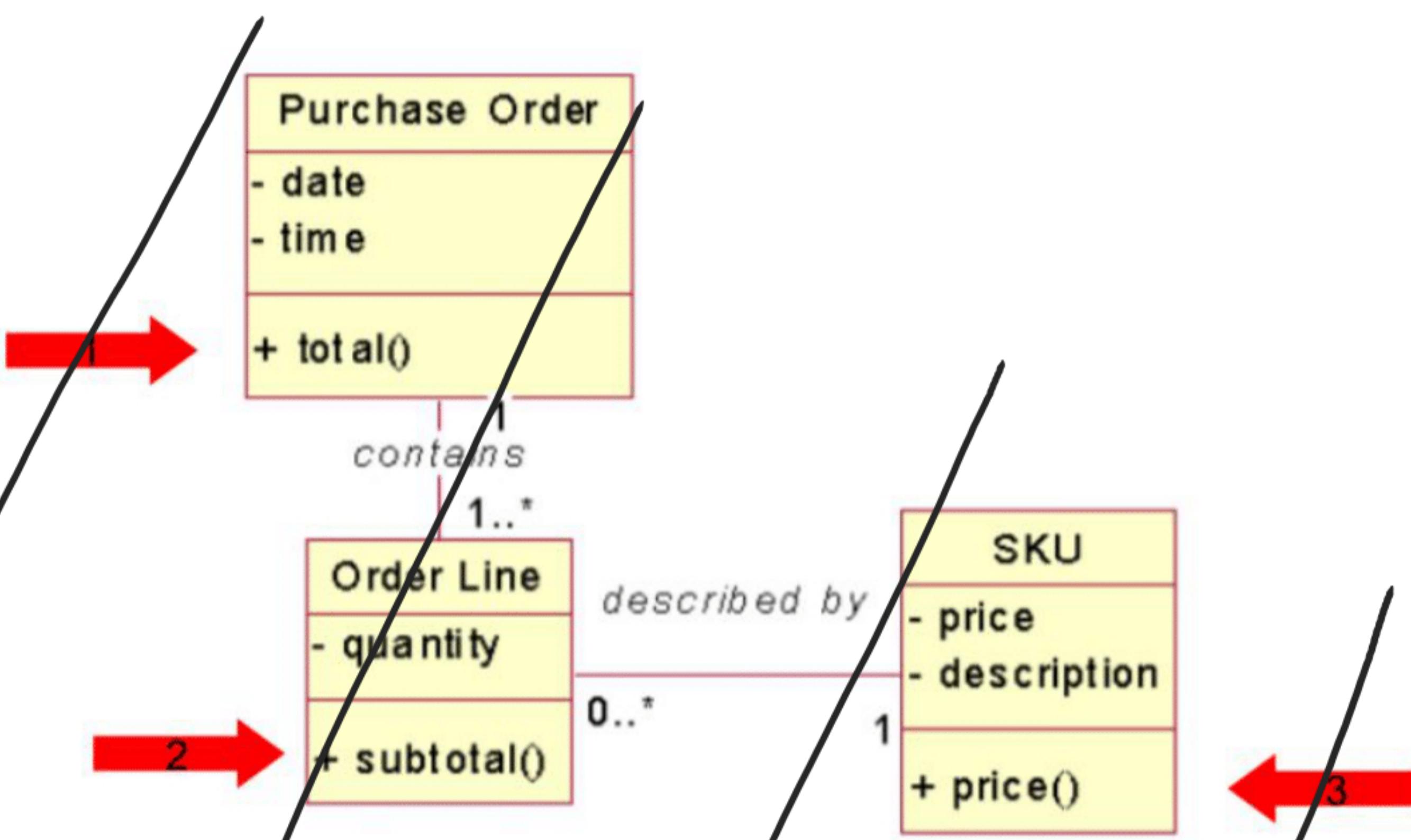


Figure 57 - Behaviours allocated by observing the expert pattern

Quem deve ser responsável por criar uma nova instância de uma classe?

→ **Creator**

→ **Atribuir a uma classe B a responsabilidade de criar um instância de outra classe A,** se um dos seguinte é verdadeiro:

B agrupa A objetos, B contém A objetos, B regista instâncias de objetos A, B usa de perto os objetos A, B tem os dados de inicialização para A.

Quem deve ser responsável por lidar com um evento do sistema?

→ **Controller**

→ **Atribuir responsabilidade para lidar com uma mensagem de evento do sistema** para uma classe que é um:

Controlador de Fachada: Representa o sistema ou organização

Controlador de Função: Representa algo no mundo real que é ativo

Controlador de Caso de Uso: Representa um manipulador artificial de todos os eventos do sistema de um caso de uso.

Código JAVA ↔ DIAGRAMAS DE CLASSES / DIAGRAMAS DE SEQUÊNCIA

- **Explicar as implicações no código da naveabilidade modelada no diagrama de classes.**
- **Construa um diagrama de classes e um diagrama de sequência considerando um código Java.**



Práticas selecionadas na construção do software

Garantia de qualidade

- Identifique as atividades de validação e verificação incluídas no SDLC

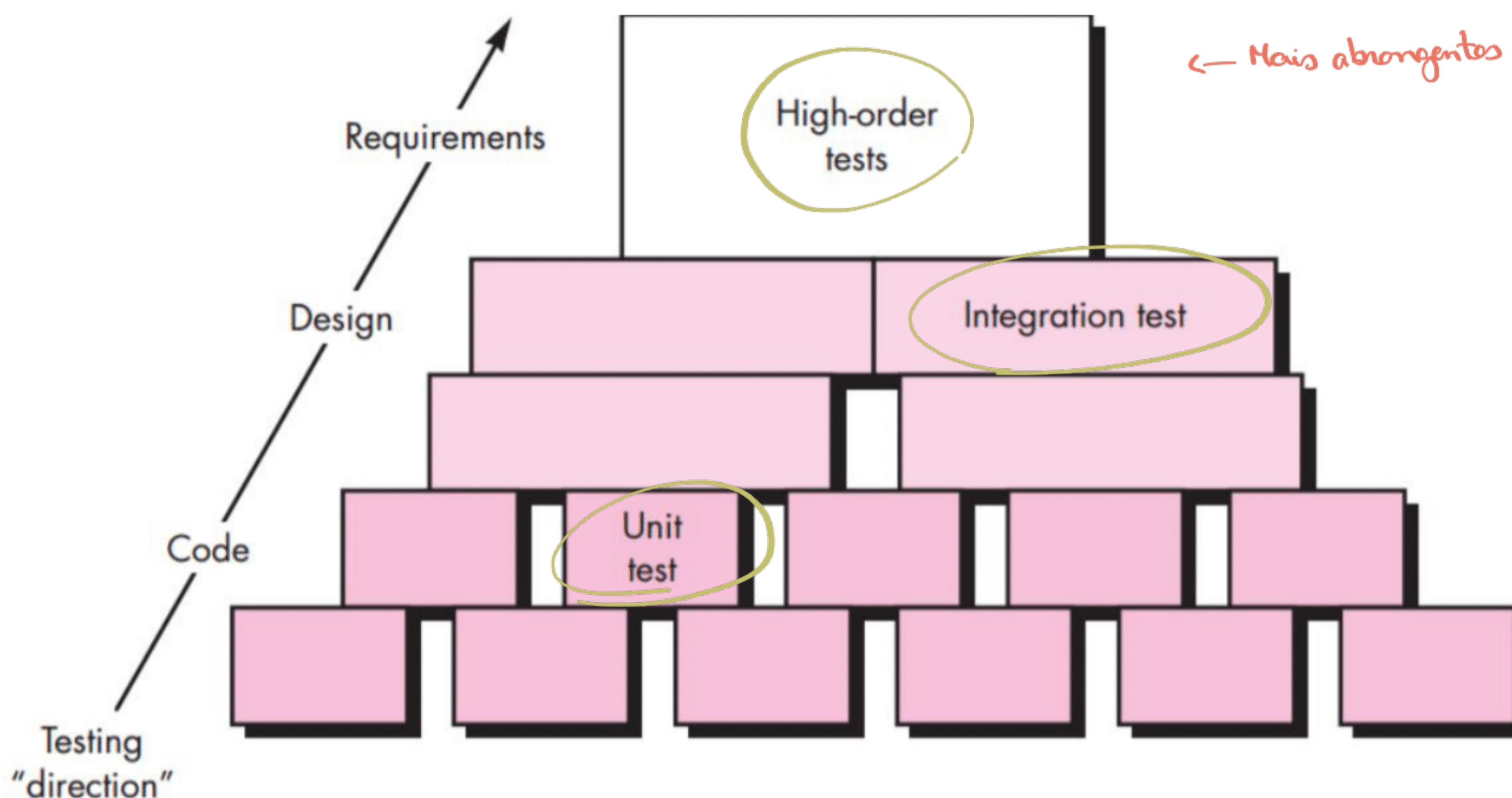
VERIFICAÇÃO: Estamos a fazer o sistema da forma correta?

- Verificar os produtos de trabalho contra as suas especificações
- Verificar a consistência dos módulos
- Verificar através das melhores práticas da indústria...

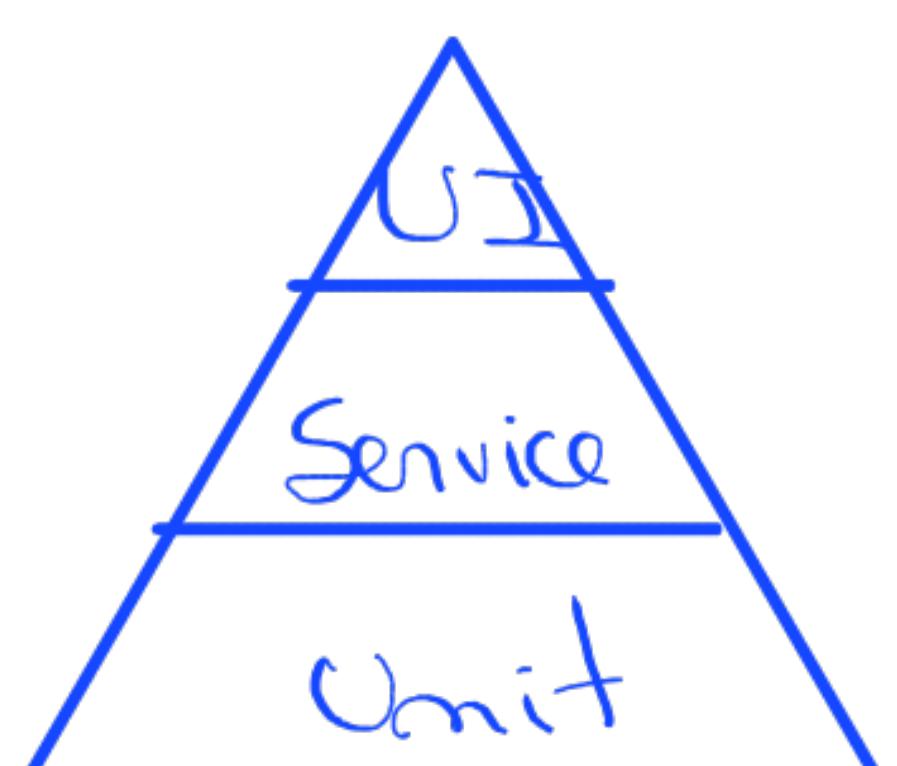
VALIDAÇÃO: Estamos a fazer o sistema correto (que é suposto)?

- Verificar produtos de trabalho através das necessidades e expectativas do utilizador.

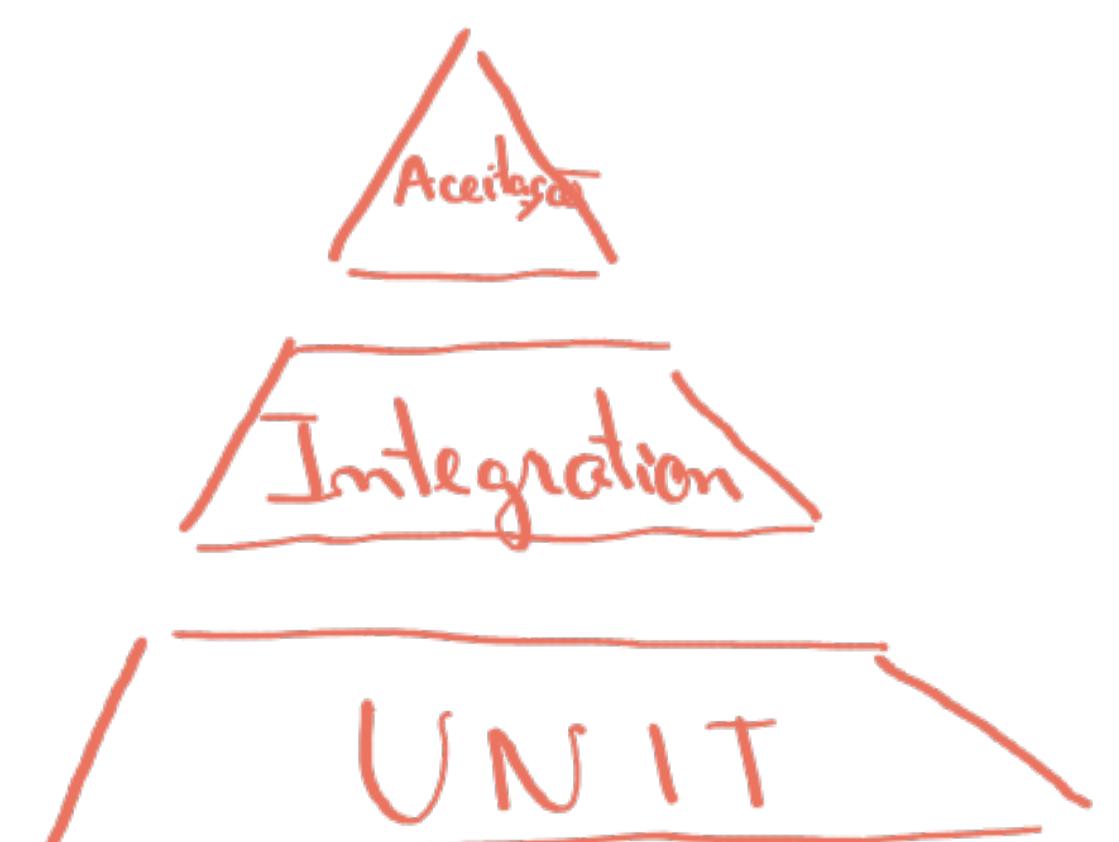
- Descreva quais são as camadas da pirâmide de testes



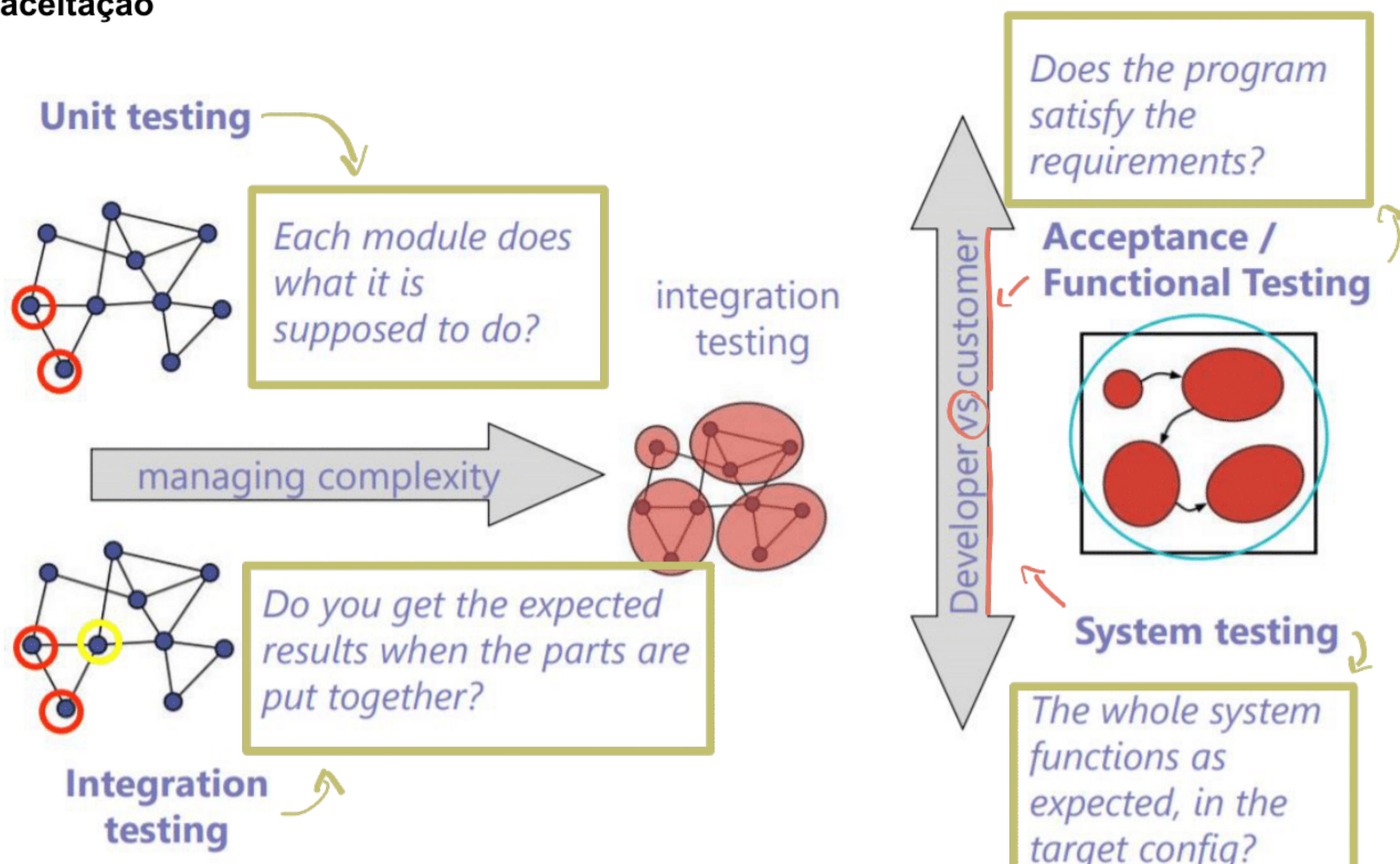
À medida que o projeto avança, são precisos cada vez mais testes, mais específicos e de mais baixo nível (nível de código).



P9. T2019
A “pirâmide dos testes” evidencia que existem diferentes tipos de teste, com granularidades e objetivos diferentes. Qual das seguintes afirmações NÃO é correta:
a) Os testes unitários verificam a implementação de pequenas unidades de código (e.g. classe), de forma isolada.
b) Os testes unitários são preferencialmente escritos por ~~testers~~, e não pelo programador que implementou a unidade sob teste.
c) Os testes de integração verificam o comportamento de um módulo cujo funcionalidade requer a colaboração com um grupo pequeno de outros componentes.
d) Nos testes realizados sobre a camada de UI, procura-se automatizar os cenários descritos nas user stories.
e) Os testes de aceitação verificam se os requisitos estão satisfeitos, na perspetiva do utilizador final.



- Descreva o assunto/objetivo dos testes de unidade, integração, sistema e de aceitação

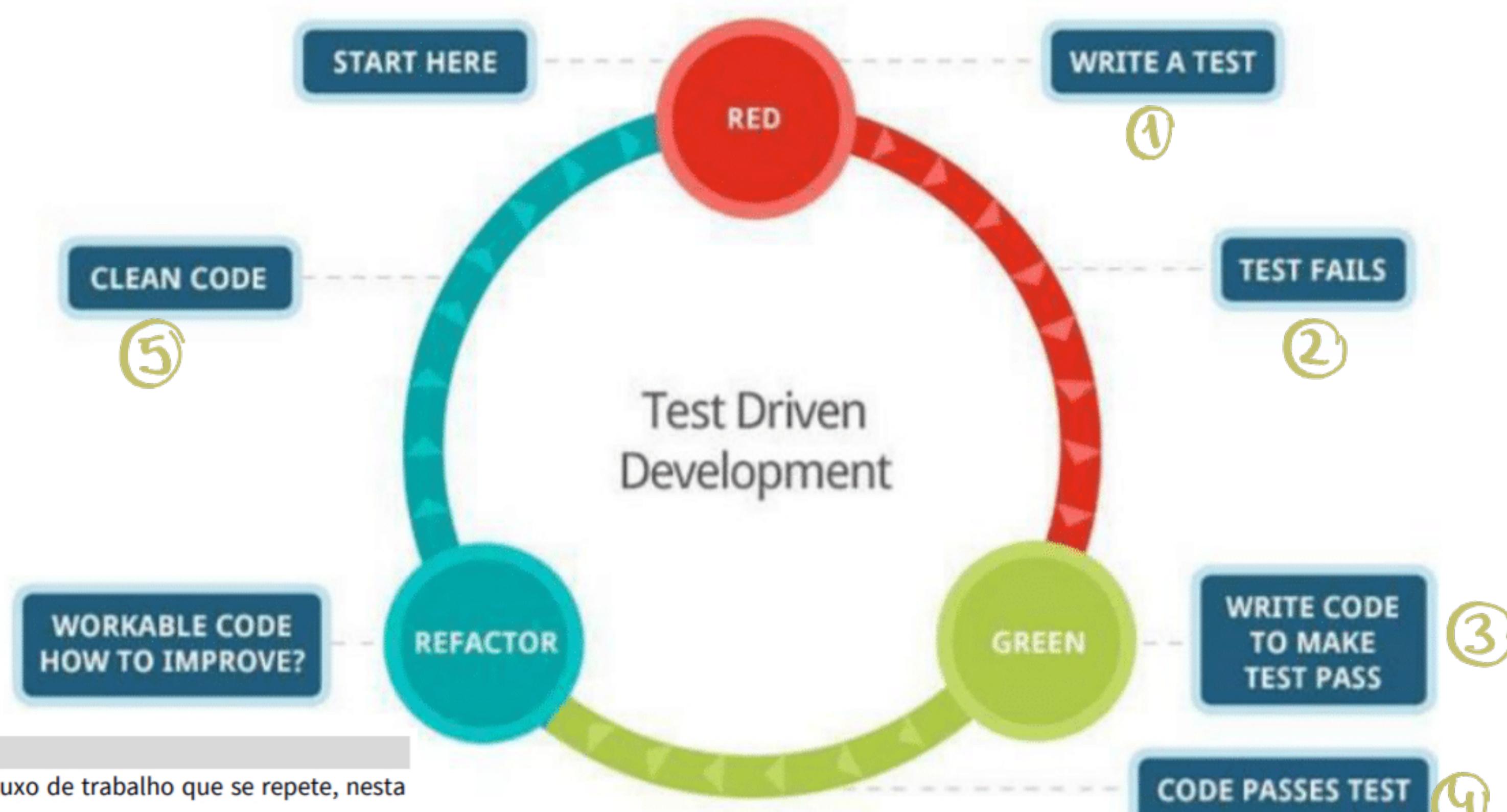


- Explique o ciclo de vida do TDD

Princípios do TDD:

→ Testes pequenos e automáticos.

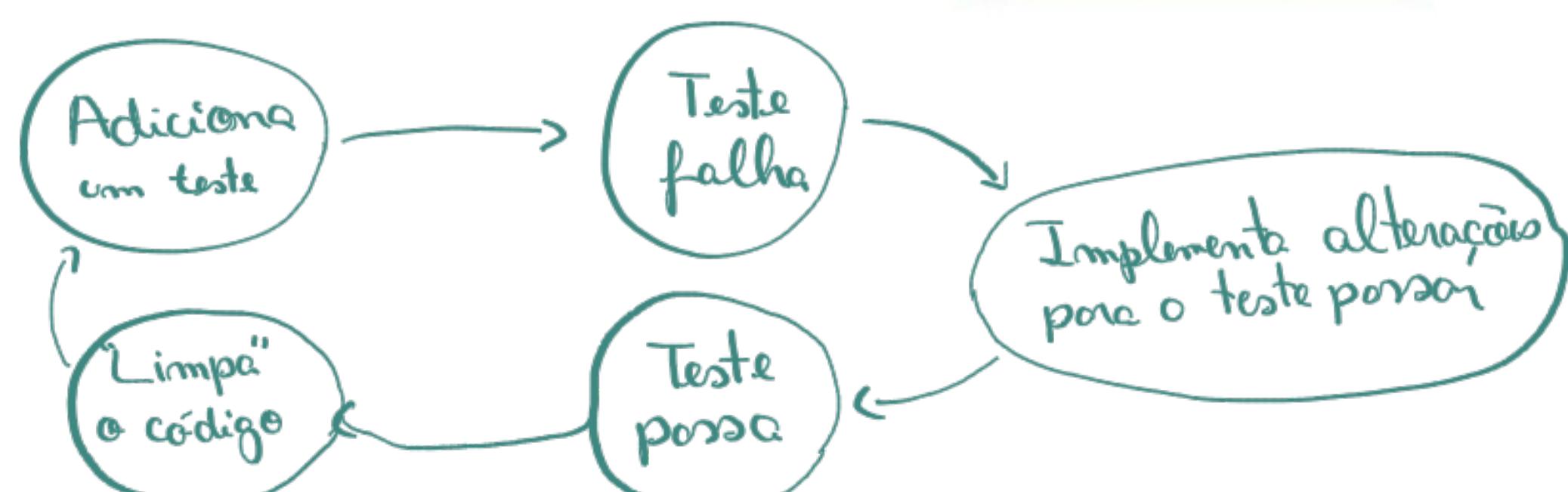
Começou por fazer os testes



P9. ✓ T2018

No cerne do TDD está um fluxo de trabalho que se repete, nesta ordem:

- Adicionar um pequeno teste; executar os testes e verificar que o novo está a falhar; implementar as alterações suficientes para o teste passar; executar os testes e verificar que o novo passa; rever o código para o tornar mais claro.
- Implementar as alterações relativas ao novo incremento; adicionar um teste; executar os testes até se verificar que o novo teste passa.
- Melhorar a clareza do código; implementar um pequeno incremento; escrever um teste para confirmar que o incremento faz o esperado.
- Executar o código; identificar problemas com o incremento implementado; usar ferramenta de debugging para encontrar a origem; resolver os erros.
- Criar testes funcionais a partir das histórias ("user stories"); implementar as funcionalidades necessárias (para os testes passarem); suplementar o projeto com testes unitários.



- Descreva as abordagens “debug-later” e “test-driven”, de acordo com J. Grenning

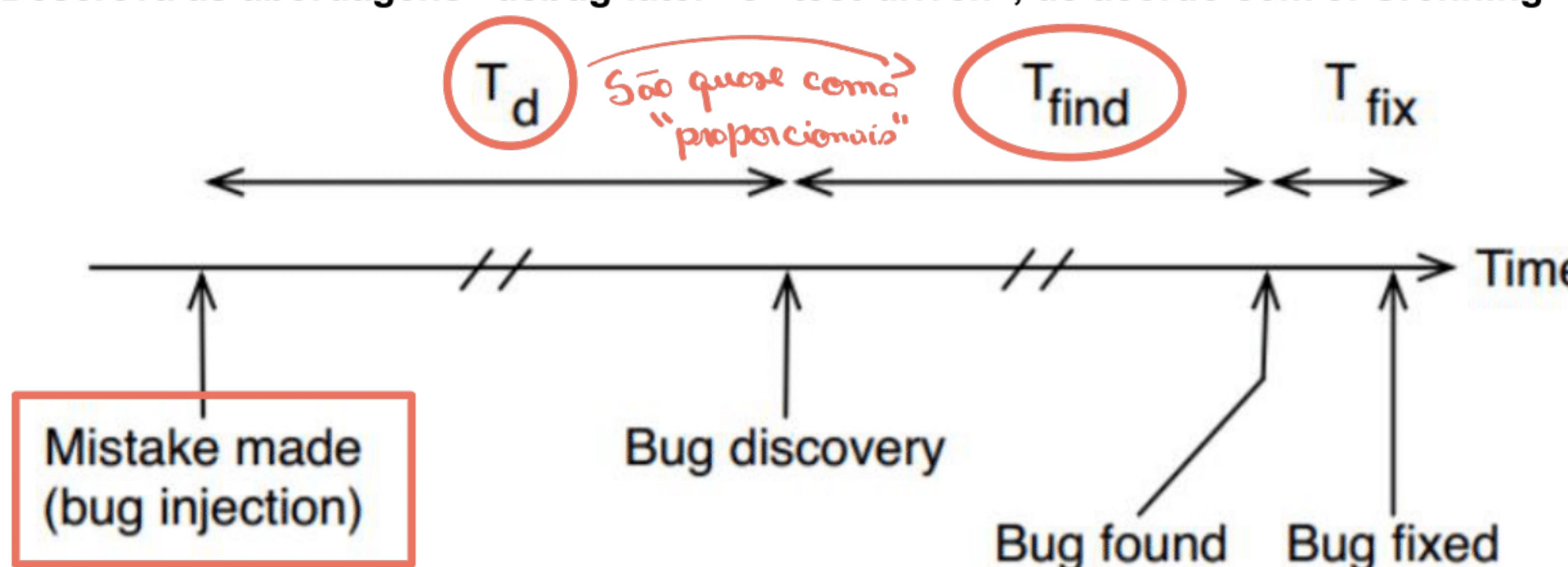


Figure 1.1: Physics of Debug-Later Programming

→ O código é escrito antes dos testes

Quando o tempo para descobrir um erro (T_d) aumenta, o tempo para encontrar a causa raiz desse erro (T_{find}) também aumenta, geralmente de forma dramática. Para alguns bugs, o tempo para corrigir o bug (T_{fix}) não é afetado pelo T_d . Mas se o erro for agravado pelo código construído sobre uma suposição errada, o T_{fix} também poderá aumentar drasticamente.

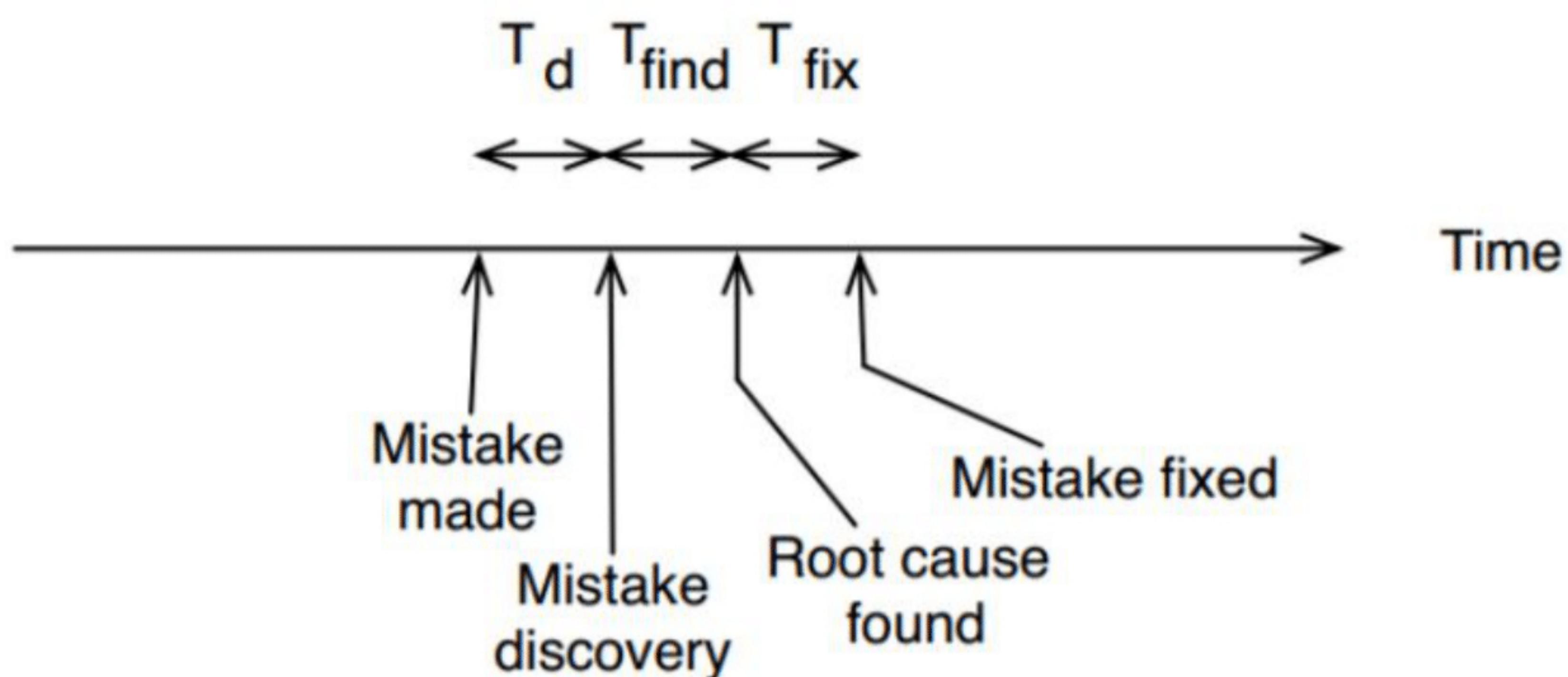


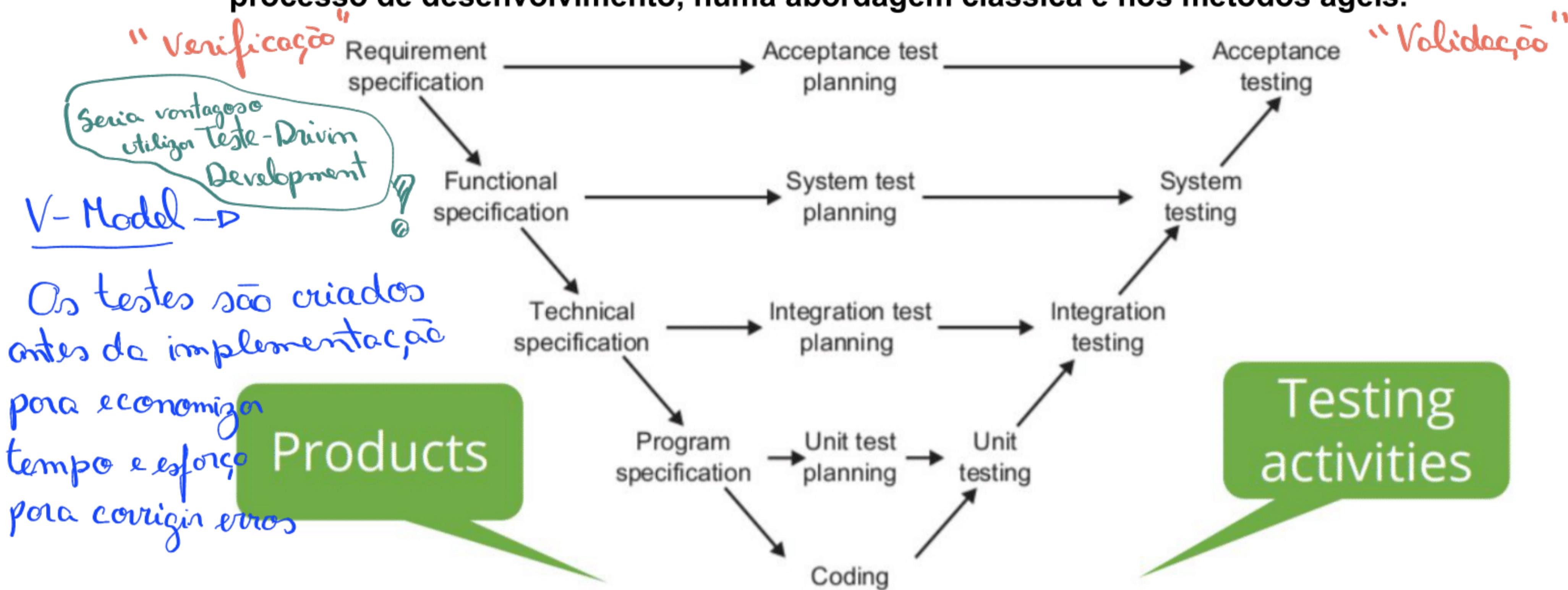
Figure 1.2: Physics of Test-Driven Development

→ os testes são criados antes do código

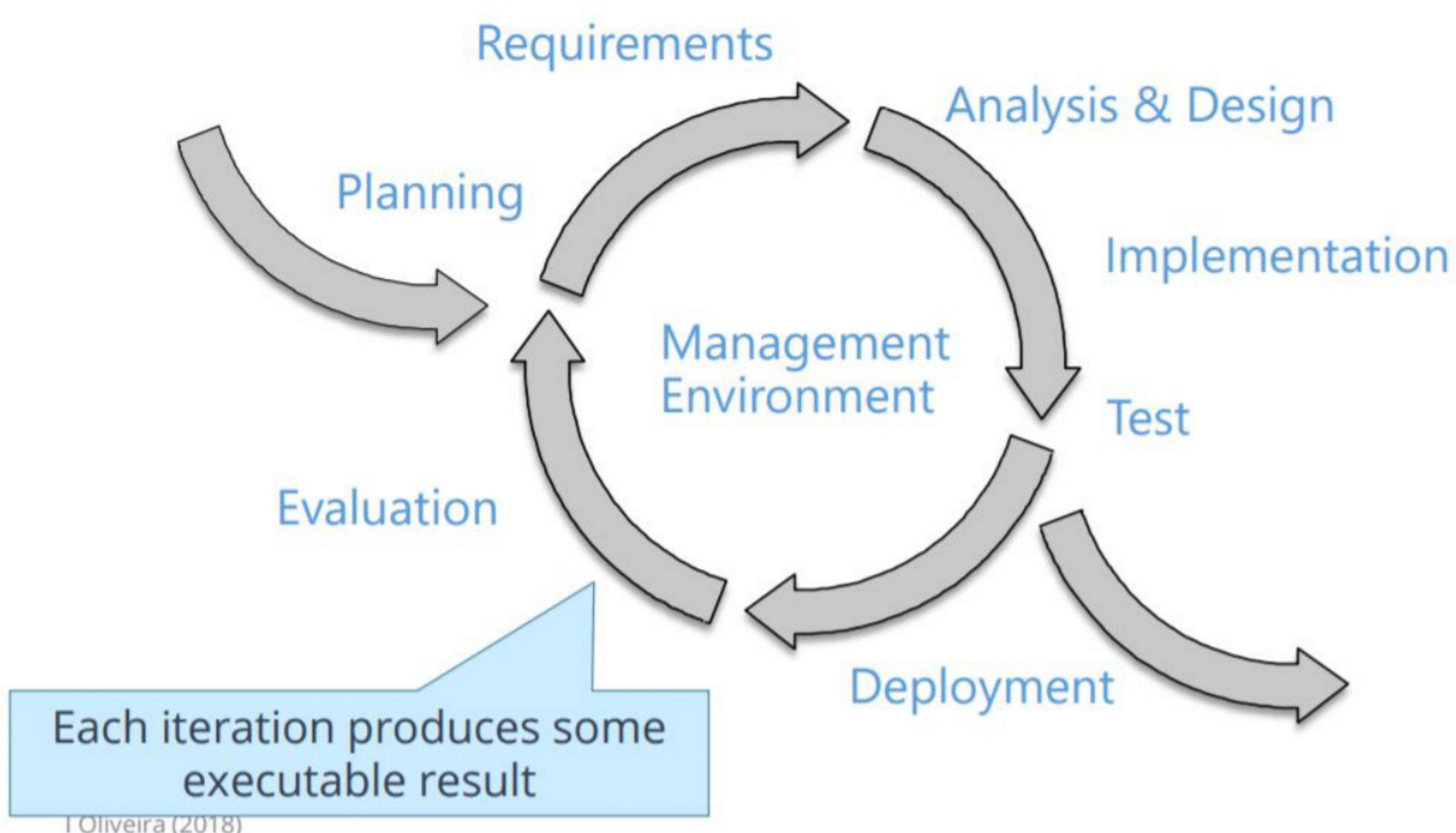
Quando o tempo para descobrir um erro (T_d) se aproxima de zero, o tempo para encontrar a causa raiz do erro (T_{find}) também se aproxima de zero. Porquê? Porque o problema, acabado de introduzir, é muitas vezes óbvio. Quando a causa não é óbvia, o desenvolvedor está a apenas alguns UNDOs longe do estado anterior de passagem de todos os testes. O tempo para corrigir o bug (T_{fix}) é tão baixo quanto possível, já que as coisas só podem

piorar à medida que o tempo embaça a memória do programador, e quanto mais código for construído sobre a base instável.

- Explique como é que as atividades de garantia de qualidade (QA) são inseridas no processo de desenvolvimento, numa abordagem clássica e nos métodos ágeis.



Em relação à abordagem clássica, em cada iteração existe planeamento de testes, mas **os testes só são executados após o código estar completo.**



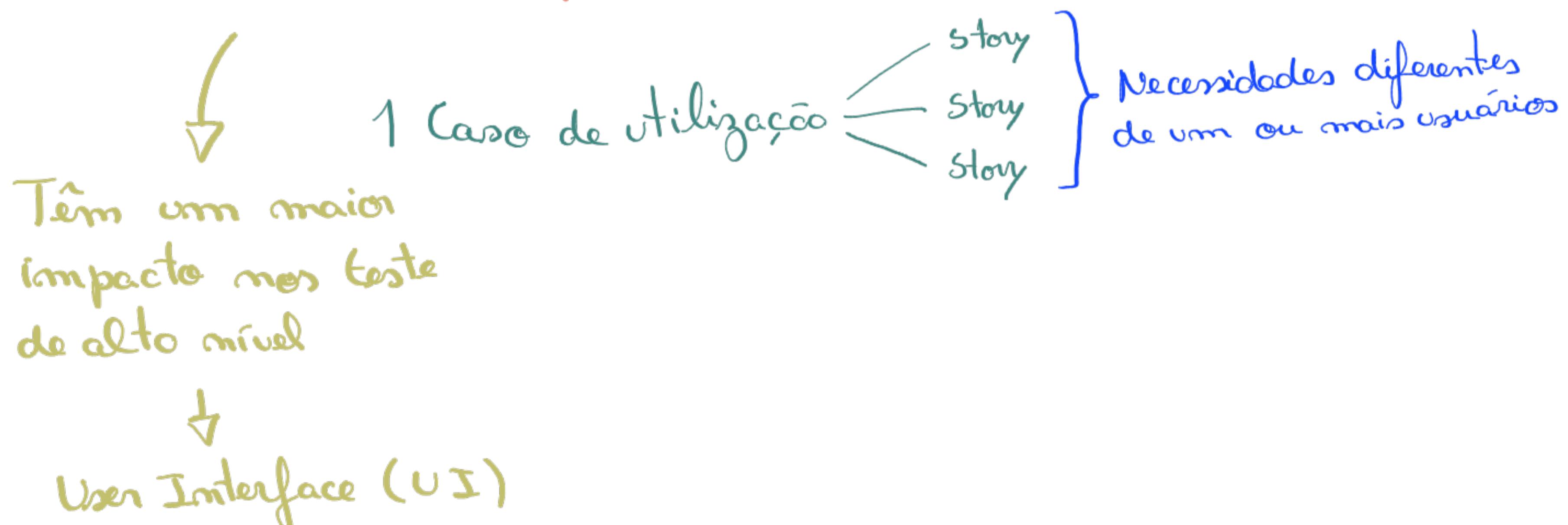
Cada iteração fornece algum software funcional que foi totalmente testado. A equipa de desenvolvimento entrega o conjunto de histórias (stories) em cada iteração e estas são testadas antes da entrega. **A história não está completa até que passe em todos os testes e seja aceita pelo cliente.** Embora as entregas antecipadas possam não fornecer

funcionalidade suficiente para o cliente ter um produto viável, estas produzem software funcional que o cliente pode examinar e verificar se estão satisfeitas e se não houve nenhum mal-entendido.

- O que é o "V-model"? *Imagem em "V" da pergunta anterior*
- Relacione os critérios de aceitação da história (user-story) com o teste Agile.

No contexto do teste Agile:

→ Critérios de aceitação da história (user-story) são utilizados para determinar o que deve ser testado e como deve ser testado



Abordagens Complementares

Gestão do projeto: planeamento e monitorização do processo

- Promotor do projeto: pessoa ou organização que patrocina o projeto de desenvolvimento de um novo sistema. Papel ativo na execução e monitorização (nem todos os stakeholders são promotores)
- Avaliação de viabilidade de um projeto:
 - Viabilidade técnica: Podemos construir isto?
 - Viabilidade económica: Devemos construir isto?
 - Viabilidade organizacional: Se construirmos isso, trará benefícios para a empresa e será usado pelos clientes?

• Work Package breakdown Structure (WBS):

(Utilizada em abordagens sequenciais) → Dividir um projeto em partes menores e mais gerenciáveis
E.g.: Waterfall → Inclui dependências entre diferentes partes do projeto (usualmente com algum nível de hierarquia)

• Kanban board:

- Linguagem visual (1 cartão → 1 tarefa)
- Colunas traduzem fluxo
- Limites para colunas

• Timeboxing:

- Prazo inflexível
- Força a eliminação dos "retoques finais"
- Entrega regular de valor
- Bom para os métodos ágeis

→ Entrega rápida de produtos funcionais
→ Entregar valor regularmente



Histórias e métodos ágeis

- **Defina histórias (user stories) e dê exemplos.**

Uma história (user story) é uma anotação que captura o que um utilizador faz ou precisa de fazer como parte de seu trabalho. Cada história de utilizador consiste numa breve descrição escrita do ponto de vista do utilizador, com linguagem natural.

? Especifica-se para um utilizador (1º pessoa)

As a <type of user>, I want <some goal> so that <some reason>.

Simular a preparação para um certo tipo de pessoas

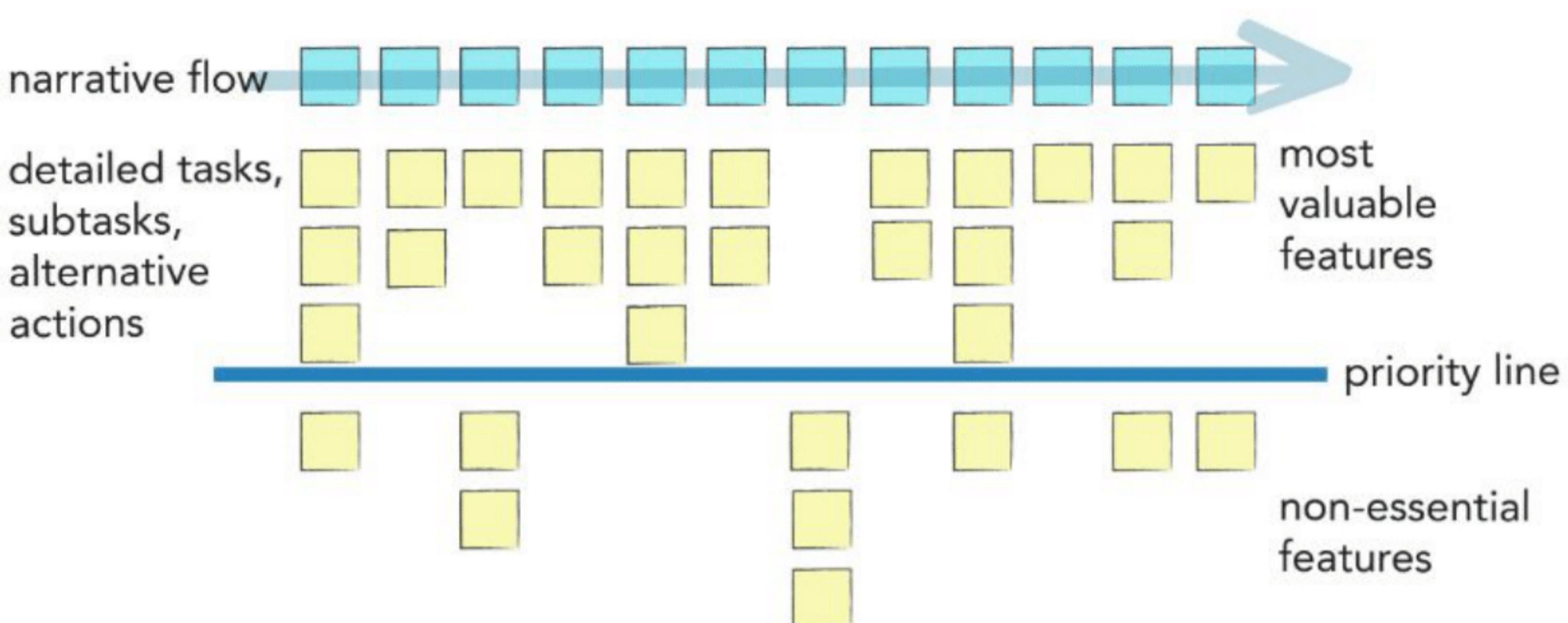
→ Mais detalhados que os Casos

Ao contrário da enumeração tradicional de requisitos, as stories concentram-se nas necessidades do utilizador, em vez do que o sistema deve proporcionar. Isto deixa espaço para uma discussão mais aprofundada das soluções e o resultado de um sistema que pode realmente encaixar-se no fluxo de trabalho comercial dos clientes, resolvendo os seus problemas operacionais e, o mais importante, fornecendo valor à organização.

- Explique a metáfora do “post-it” (para planeamento e seguimento) comum em projetos ágeis.

- Distribuir o trabalho
- Rotatividade para os requisitos
- “post-it” por iteração

User Story Map



- **Identifique os elementos-chave de uma “persona”.**

→ Uma persona define um utilizador hipotético de um sistema, um exemplo do tipo de pessoa que interage com ele. A ideia é que, de modo a desenvolver software eficaz, este tem de ser projetado para uma pessoa específica.

→ As personas representam pessoas fictícias baseadas no seu conhecimento de utilizadores reais.

→ Ajuda a equipa a compreender o destinatário do sistema.

- Compare histórias e casos de utilização em relação a pontos comuns e diferenças.

→ As user stories são centradas no resultado e no benefício do que se está a descrever, enquanto que os Casos de Uso podem ser mais detalhados e descrever como o sistema age.

User Stories versus Casos de Uso - similaridades

→ As User Stories contêm a função do utilizador, o objetivo e os critérios de aceitação.

→ Os Casos de Uso contêm elementos equivalentes: um agente, fluxo de eventos e pós-condições, respectivamente (um modelo detalhado de Caso de Uso pode conter outros elementos).

Quem realiza?

User Stories versus Casos de Uso - diferenças

Os detalhes de uma User Story podem não ser tão documentados como um caso de uso. As User Stories deixam de fora muitos detalhes importantes deliberadamente. As User Stories são destinadas a provocar conversas ao fazerem-se perguntas durante reuniões de Scrum → Pequenos incrementos para obter feedback com mais frequência, em vez de ter uma especificação de requisitos antecipada mais detalhada, como em Casos de Uso.

- Compare “Persona” com Ator com respeito a semelhanças e diferenças.

As personas são utilizadas em User Stories e geralmente são mais robustas que os atores. Uma persona é projetada para ajudar a equipa a compreender o destinatário do sistema a desenvolver. O detalhe permite que a equipa “entre na cabeça dos utilizadores fictícios” à medida que as histórias e funcionalidades do utilizador são desenvolvidas. Os atores são utilizados principalmente em casos de uso, que são usados como uma ferramenta para desenvolver requisitos. Os casos de uso também costumam ser usados para validar projetos e como ferramenta para conduzir atividades de teste. Nesses cenários, o foco é como o trabalho será realizado e em que ordem, em vez de por que e por quais necessidades estão a ser atendidas.

Os atores incluem muito menos detalhes do que uma persona e normalmente são identificados num nível significativamente maior de abstração. Com base no nível mais elevado de abstração de atores, muitas personas podem ser resumidas em apenas um ator. Ambos os atores e personas têm valor, no entanto, se estiverem a ser utilizadas user stories, os atores não fornecem uma compreensão profunda o suficiente das necessidades e motivações dos utilizadores e clientes do sistema. Como alternativa, ao usar técnicas como casos de uso, o desenvolvimento de perfis de utilizadores fictícios repletos de histórias secundárias, necessidades, motivações e imagens é um exagero.

*Para um dig.
de Cau's
chega determinar
os Atores*

- O que é a pontuação de uma história e como é que é determinada?

A pontuação corresponde dum forma informal a dificuldade e velocidade que a equipa acha que levará a implementar tal funcionalidade. *Não tem haver com prioridade?*

- Descreva o conceito de velocidade da equipa (como usado no PivotalTracker e SCRUM).

No desenvolvimento ágil, usamos o princípio do "clima de ontem" para prever como as futuras iterações devem ser planeadas. Em outras palavras, o clima de hoje provavelmente será o mesmo de ontem. Quando os pontos da história (story points) são mapeados por dificuldade ou complexidade, em vez de horas ou dias, é mais provável que uma equipa faça o mesmo número de pontos da história na iteração atual do que nas últimas iterações.

O Tracker foi projetado para planear automaticamente as iterações do projeto com base na quantidade média de story points concluídas ao longo de um número predeterminado de iterações. Chamamos a essa velocidade velocidade média e, no Tracker, a velocidade é a

força dominante por detrás de cada projeto. O tracker determina o que será planeado para futuras iterações e está constantemente em alteração com base na sua taxa real de conclusão.

- **Discutir se os casos de utilização e as histórias são abordagens redundantes ou complementares (quando seguir cada uma das abordagens? Em que condições? ...)**
JÁ FOI RESPONDIDA EM QUESTÕES ANTERIORES

A metodologia SCRUM

- **Explique o objetivo da “Daily Scrum meeting”**

A Daily Scrum meeting não é usada como solução de problemas ou reunião de resolução de problemas. Os problemas levantados são colocados off-line e geralmente tratados por uma sub-equipa imediatamente após a reunião. Durante o scrum diário, cada membro da equipa responde às três perguntas a seguintes:

- O que foi feito ontem?
- O que vai ser feito hoje?
- Há algum obstáculo?

Concentrando-se no que cada pessoa realizou ontem e realizará hoje, a equipa obtém uma excelente compreensão do trabalho realizado e do trabalho que ainda resta. A reunião diária do scrum não é uma reunião em que os membros da equipa se comprometem uns com os outros.

- **Relacione os conceitos de sprint e iteração e discuta a sua duração esperada.**
→ Todas as sprints são iterações, mas nem todas as iterações são sprints. Uma iteração é um termo comum no SDLC. Sprint é um termo específico do Scrum.
→ A duração esperada é duas semanas (de todas as sprints) de acordo com o manual mas pode haver variação conforme as necessidades da equipa.
- **Explique a método de pontuação das histórias (e critérios aplicados)**
Como os story points representam o esforço para desenvolver uma história, a estimativa de uma equipa deve incluir tudo o que puder afetar o esforço. Isto pode incluir:
 - A quantidade de trabalho a fazer
 - A complexidade do trabalho
 - Qualquer risco ou incerteza em fazer o trabalho
- **Relacione as práticas previstas no SCRUM e os princípios do “Agile Manifest”: em que medida estão alinhados?**
Em todas as medidas! SCRUM é Agile.

