Preparação Teste 2 - AS

Índice

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:
  → Iniciação: Qual é o valor de negócio do sistema? O projeto deve avançar?
  → Management: Criação de um plano de trabalho, preparar a equipa para controlar e direcionar o projeto através do SDLC.
  Key-steps:
  → value for the business
  → should the project proceed?
  → workplan
  Análise:
  → Quem vai utilizar o sistema? O que é que o sistema vai fazer? Onde e quando vai ser utilizado?
  Key Steps:
  → analysis of existing systems;
  → requirements gathering;
  → solution concept (system proposal).
  Design:
  → Como vai o sistema operar em termos de hardware, software, infra estruturas de rede? Design da UI (User Interface), bases de dados, etc. Planeamento da arquitetura e logística de todo o SI.

  Key steps:

  → system architecture design;
  → data model design;

→ program design;

→ selection of frameworks.

Implementação:

→ Construção do sistema.

Key Steps:

→ System construction (build and Quality Assurance);

→ Installation and transition;

→ Support plan (post-install review and change management);

- **Descreva o papel e as responsabilidades do Analista no SDLC:**

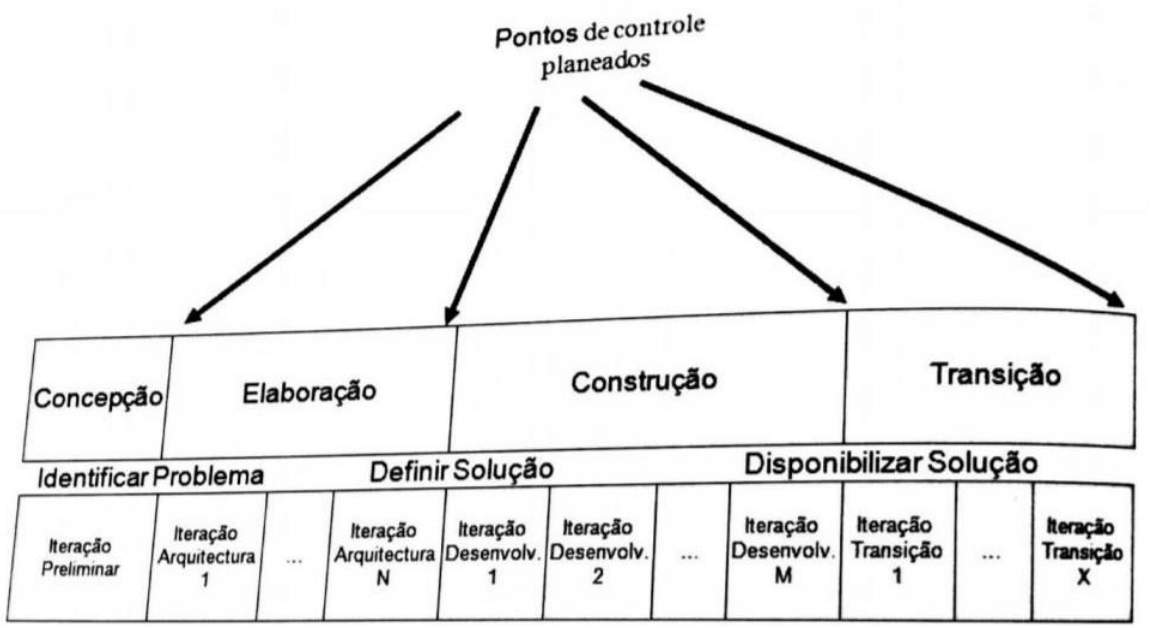
  O Analista do Sistema é quem analisa a situação do negócio, identifica oportunidades de evolução e desenha o sistema de informação que implementa essas oportunidades.

  O seu principal objetivo é criar valor para a organização.

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):**



(CECT)

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

→ Desenvolvimento de requisitos de mais alto nível → O que é preciso de um modo mais geral?

→ 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.

Milestone: At this point, you examine the cost versus benefits of the project, and decide either to proceed with the project or to cancel it.

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 a 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.

→ Algum código implementado (~10%)

→ Arquitetura conduzida pelos Use Cases!

Milestone: Validação da arquitetura.

Construção(Construction):

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.

→ 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. Todas as funcionalidades foram desenvolvidas e todos os testes alfa (se houver) foram concluídos. Além do software, um manual de utilizador foi desenvolvido, e há uma descrição do atual lançamento. O produto está pronto para o teste beta.

Transição(Transition):

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**
• 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.	• **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)

  Elaboração(Elaboration):

Atividades	**Modelação**
• Detalhar o modelo de casos de utilização • Analisar domínio • Definir arquitetura candidata • Validar arquitetura com implementação	• **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"?**

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

  → iterative development

  → team collaboration

  → continuous integration and tests

  → frequent deliveries of working software

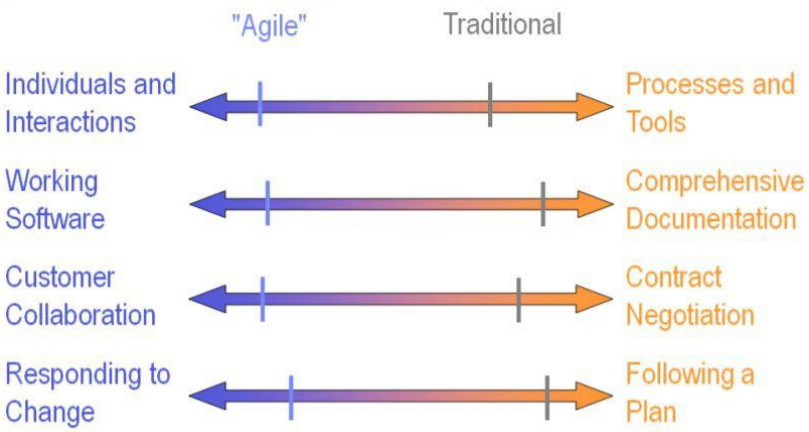
  → adaptation to changes, and so on.

- **Porque é que o Unified Process é "orientado por casos de utilização, focado na arquitetura, iterativo e incremental"?**
  Focado na arquitetura → O OpenUP é foca-se na arquitetura no sentido de minimizar o risco e organizar o desenvolvimento.
  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.
  Orientado por casos de utilização → A equipa utiliza casos de utilização para orientar todo o trabalho de desenvolvimento, desde a Inception até à Construção.

Principais características dos métodos Ágeis

- **Identificar características distintivas dos processos sequenciais, como a abordagem waterfall.**
  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.
  → 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).**
  → O objetivo dos métodos ágeis é **dar resposta rápida à alteração de condições.**
  → **Entregas frequentes, integração em contínuo → redução de 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**
  → Development in short cycles
  → Each one is evaluated and integrated.
  → Each one gives an executable (partial) increment.
  → Feedback from each iteration leads to refinement and adaptation of the next.

- **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;

- **Dado um "princípio" (do Agile Manifest), explique-o por palavras próprias , concentrando-se na sua novidade (com relação às abordagens "clássicas") e impacto / benefício.**
  1. **Indivíduos e interações mais do que processos e ferramentas**
  São as pessoas que respondem às necessidades de negócios e conduzem o processo de desenvolvimento. Se o processo ou as ferramentas impulsionam o desenvolvimento, a

equipa responde menos às mudanças e tem menor probabilidade de atender às necessidades dos clientes.

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.

- **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 → não sei lil pump eskeetit

  Modelos de comportamento → Descrevem a interação no sistema. Representa interações entre os diagramas 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.

- **Enumerar as vantagens dos modelos visuais**

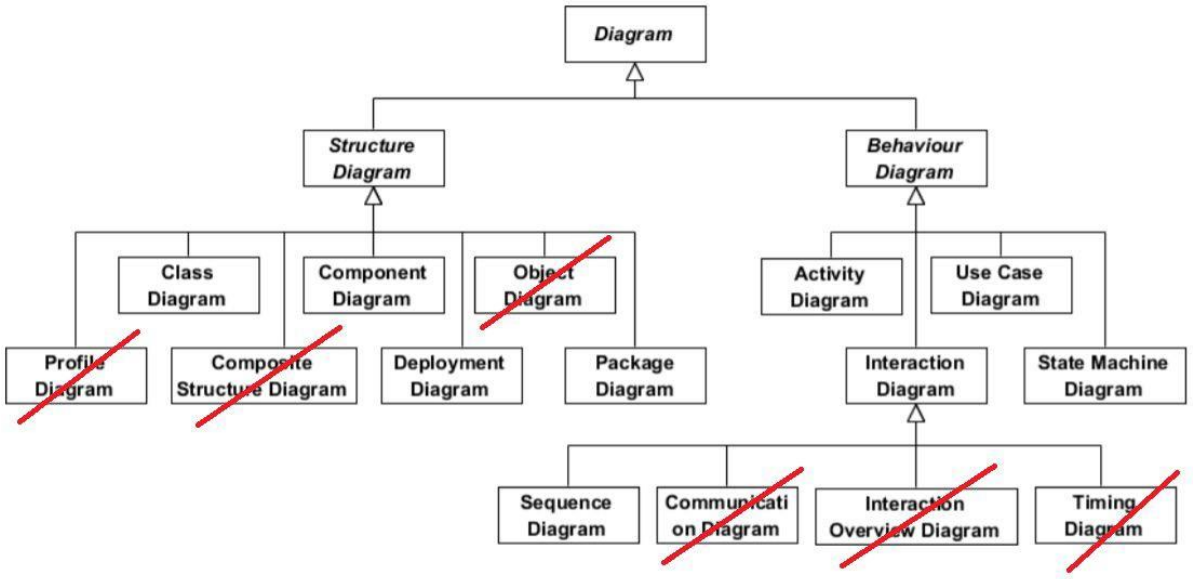
  → Promover a comunicação mais clara e sucinta;

  → 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).

- **Explicar a organização da UML (classificação dos diagramas)**



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.

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 multicamadas) - modelo de aplicativo com várias camadas.

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.

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.

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 perspetiva 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.

Modelos de Análise

Práticas de engenharia de requisitos

- **Distinguir entre requisitos funcionais e não funcionais**
  Funcional:
  → 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.
  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 eliciação de requisitos**

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

- **Justifique que "a eliciação de requisitos é mais que a recolha de requisitos".**
  O coração do desenvolvimento de requisitos é a eliciação, o processo de identificação das necessidades e restrições das várias partes interessadas para um sistema de software. Eliciação não é o mesmo que "Reunir requisitos". Também não é uma simples questão de transcrever exatamente o que os utilizadores dizem. Eliciação é um processo colaborativo e analítico que inclui atividades para colecionar, descobrir, extrair e definir requisitos. A eliciação é usada para descobrir negócios, utilizadores, requisitos funcionais e não funcionais, juntamente com outros tipos de informação. A eliciação de requisitos é talvez o aspectos mais desafiador, crítico, propenso a erros e intensivo em comunicação do desenvolvimento de software.

- **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.

- **Enumerar os principais recursos das ferramentas / ambientes do software Requirements Management.**

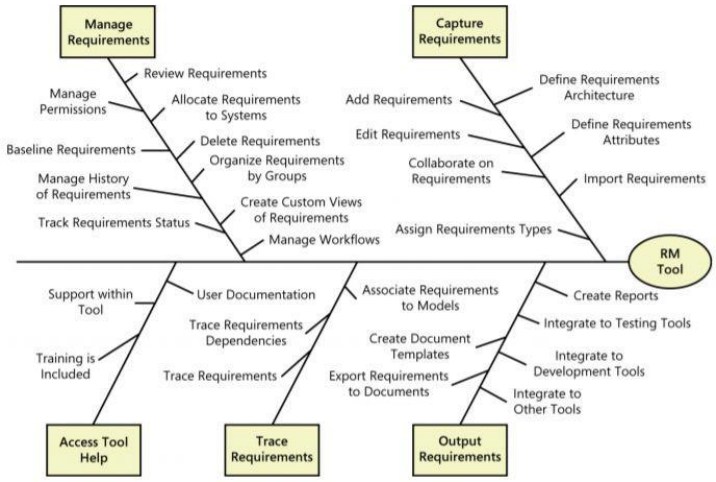


FIGURE 30-1 Common RM tool features.

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.**
- **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.

- **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.**

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.

- **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.

→ A **visão gerada pelos modelos de CaU** é conseguida sem nos perdermos 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.

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

Existem seis princípios básicos no coração de qualquer sucesso de aplicação de casos de uso:

1. Manter a simplicidade contando histórias (stories).
2. Entender a grande figura (big picture).
3. Concentre-se no valor.

4. Construir o sistema em fatias (slices).
5. Entregar o sistema em incrementos.
6. Adaptar-se para atender às necessidades da equipa.

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 compreensível. 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**

Modelos no desenho e implementação

Modelação estrutural

- **Distinguir entre a análise de sistemas baseada numa abordagem algorítmica top-down e baseada nos conceitos do domínio do problema.**

  Decomposição algorítmica

  A maioria de nós foi formalmente treinada no dogma do design estruturado de cima para baixo, e por isso abordamos a decomposição como uma simples questão de decomposição algorítmica, em que cada módulo no sistema denota um passo importante no processo geral.

Decomposição Orientada a Objetos

Objetos fazem coisas, e nós pedimos que executem o que fazem enviando mensagens. Como a nossa decomposição é baseada em objetos e não em algoritmos, chamamos isso de decomposição orientada a objetos.

Decomposição algorítmica *vs* orientada a objetos

Qual é o caminho certo para decompor um sistema complexo - por algoritmos ou por objetos? Na verdade, esta é uma pergunta com rasteira, porque a resposta certa é ambos. As visões são importantes: **a visualização algorítmica destaca a ordenação dos eventos** e **a visão orientada a objetos enfatiza os agentes que provocam a ação ou que são assuntos em que essas operações atuam**.

- **Justifique o uso de modelos estruturais na especificação de sistemas.**
  "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 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

- **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.

UML → DIAGRAMAS DE SEQUÊNCIA E ESTADO

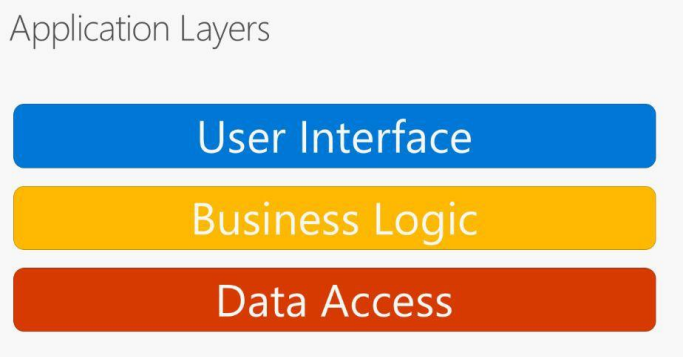
- **Entenda as regras e diretrizes de estilo para diagramas de sequência, comunicação e estado**
- **Entenda a complementaridade entre diagramas de sequência e comunicação Diagramas de sequência de mapa em código orientado a objeto e reverso.**
- **Analise criticamente os modelos de diagramas de sequência existentes para descrever a cooperação entre dispositivos ou entidades de software.**

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.

- **Identifique os elementos abstratos de uma arquitetura de software**
  nodes = class / module / package / layer / sub system / system
  Interconnections = communications / control

- **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.



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.**

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 (perspetiva do programador)

- **Explicar os princípios de baixo acoplamento e alta coesão no desenho por objetos.**
  Coupling → Mede a força/intensidade da dependência de uma classe de outras A classe C1 está acopolada 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.
  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

- **Enumerar e descrever os princípios do GRASP (Larman)**
  Generic Responsibility Assignment Principles
  → Baixo Coupling

→ Alta Cohesion

→ Information Expert

→ Creator

→ Controller

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.

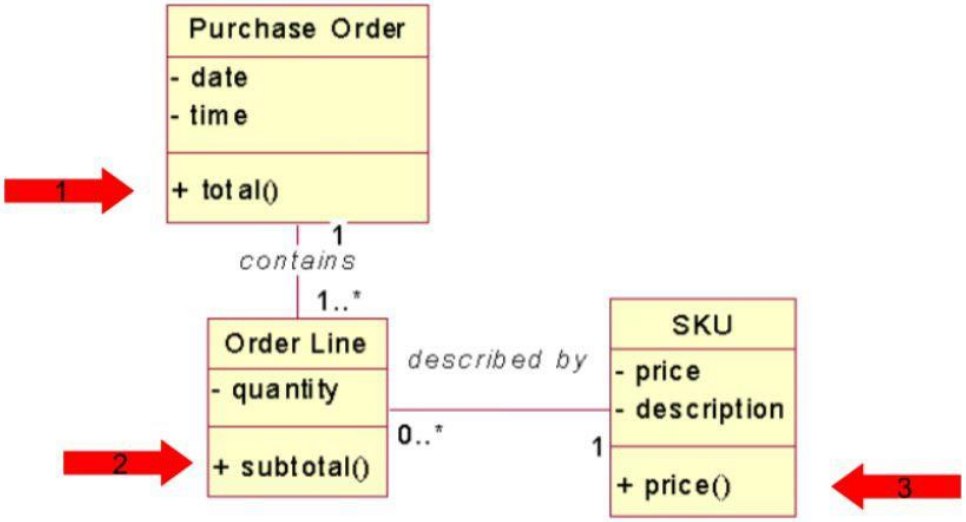


Figure 57 - Behaviours allocated by observing the expert pattern

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

→ **Creator**

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

B agrega 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 navegabilidade 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

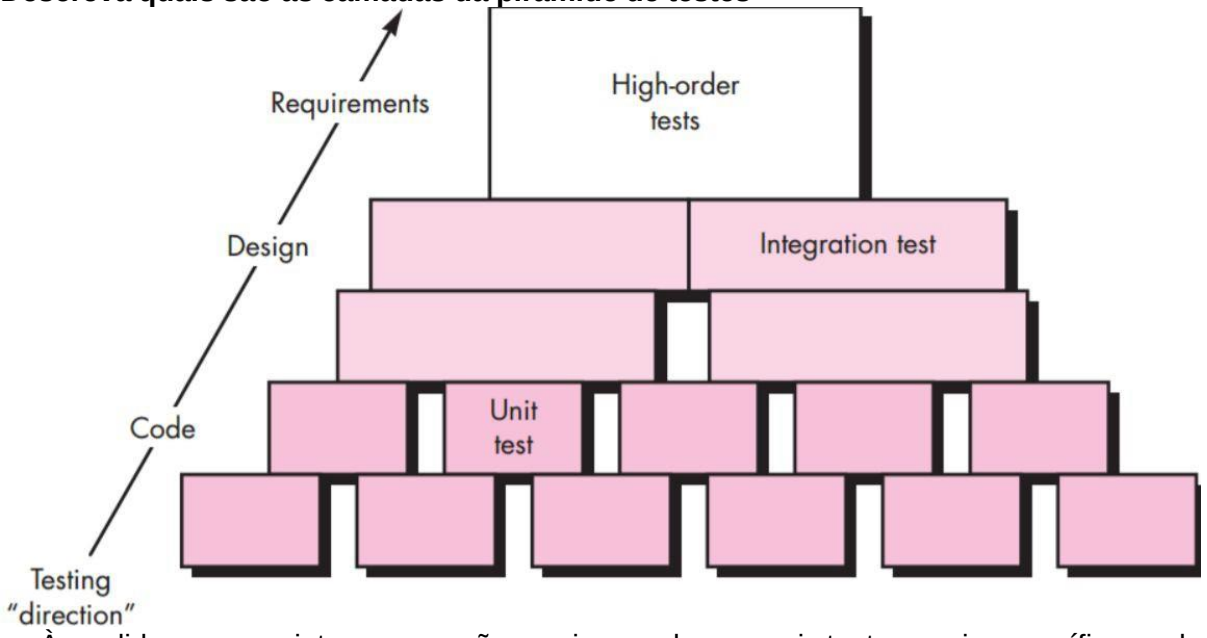
  → Verificar a consistência dos módulos

  → Verificar através das melhores práticas da indústria…

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

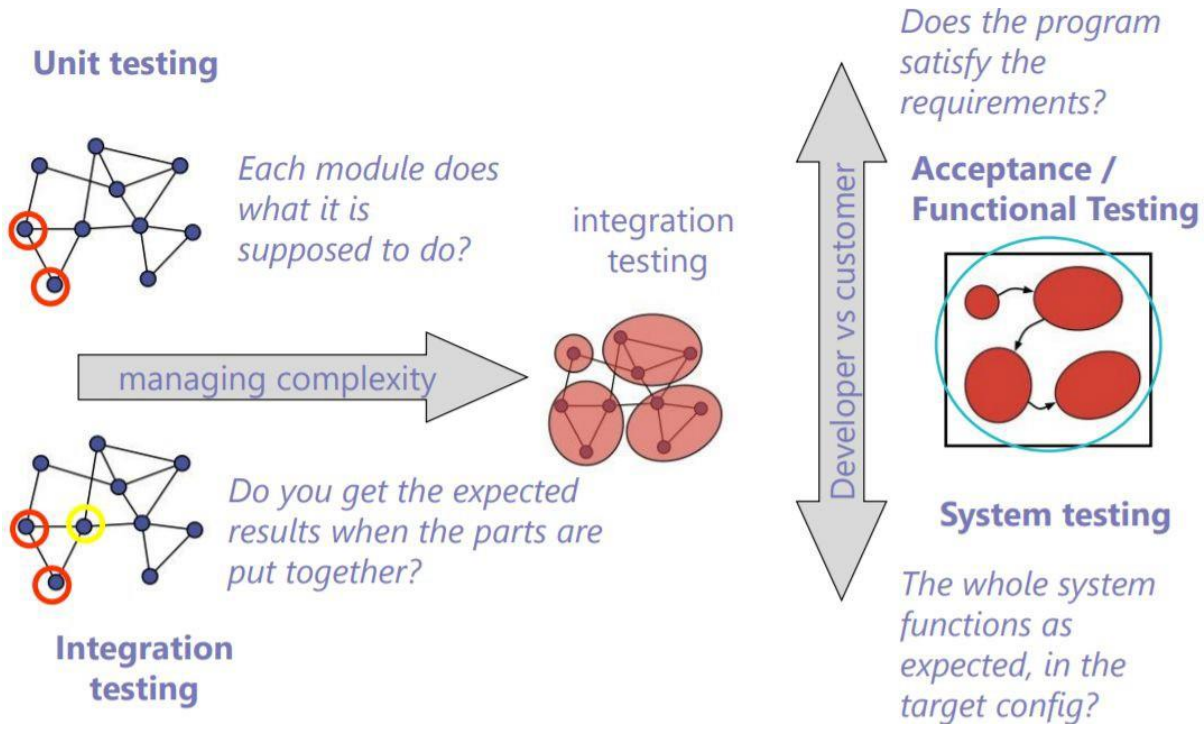
  → Verificar produtos de trabalho através das necessidades e expectativas do utilizador.

- **Descreva quais são as camadas da pirâmide de testes**

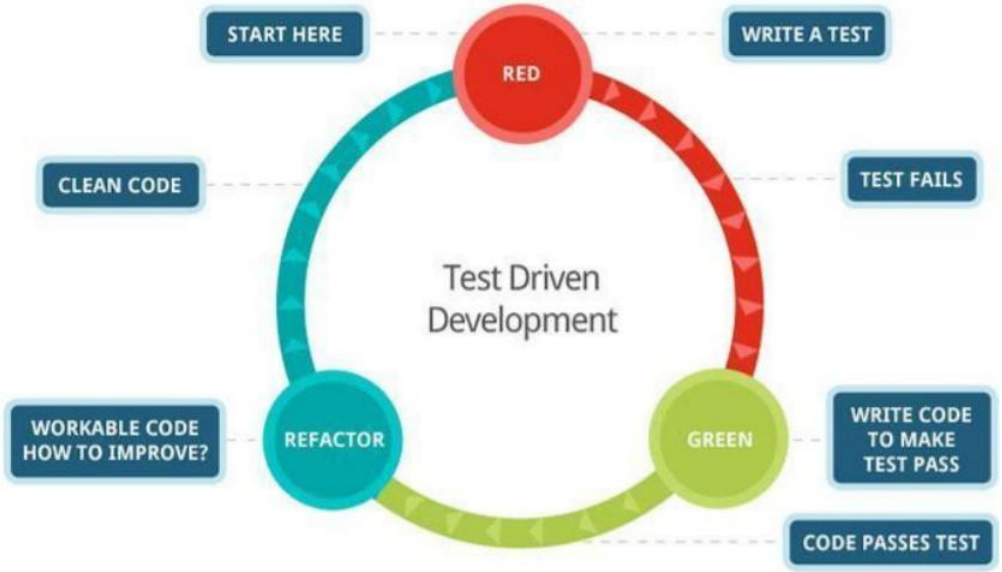


→ À medida que o projeto avança, são precisos cada vez mais testes, mais específicos e de mais baixo nível (nível de código).

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



- **Explique o ciclo de vida do TDD**
  Principios do TDD:
  → Build and then fix it;
  → Testes pequenos e automáticos.



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

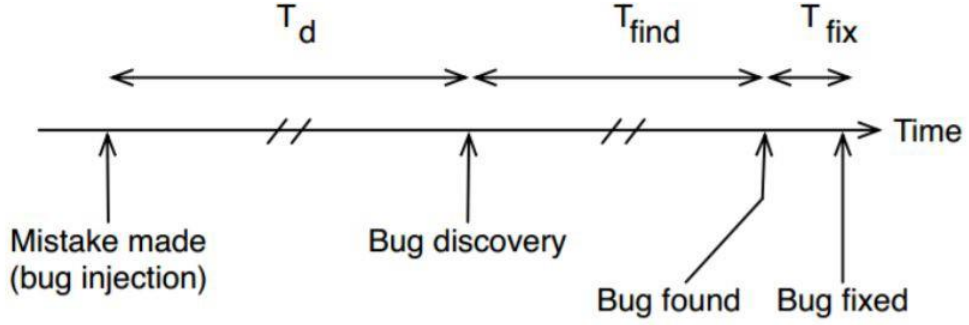


Figure 1.1: Physics of Debug-Later Programming

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

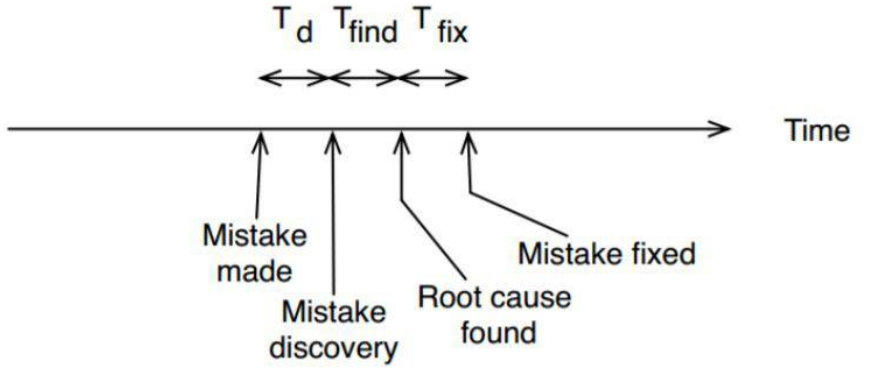
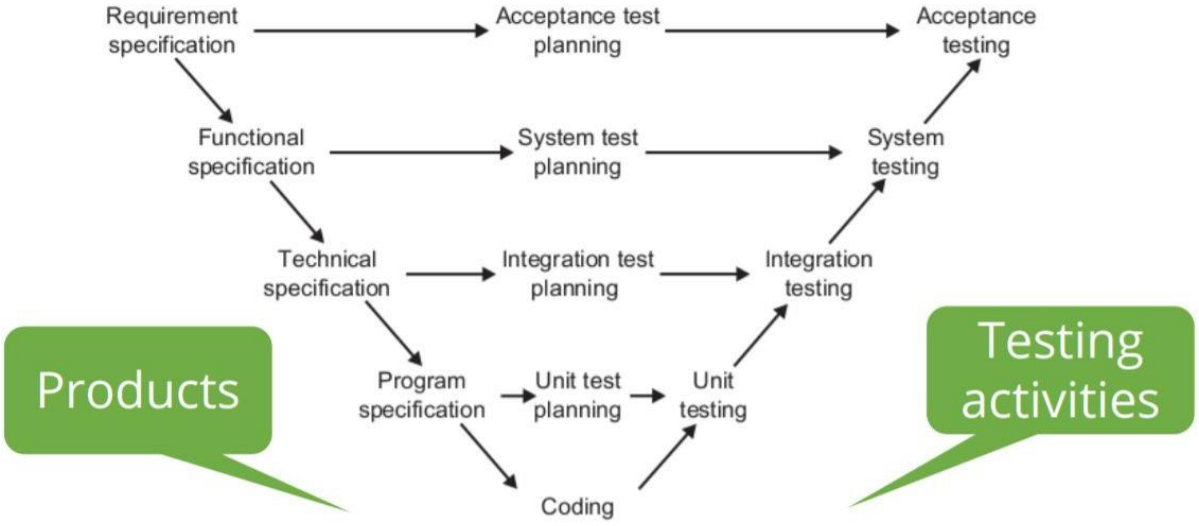


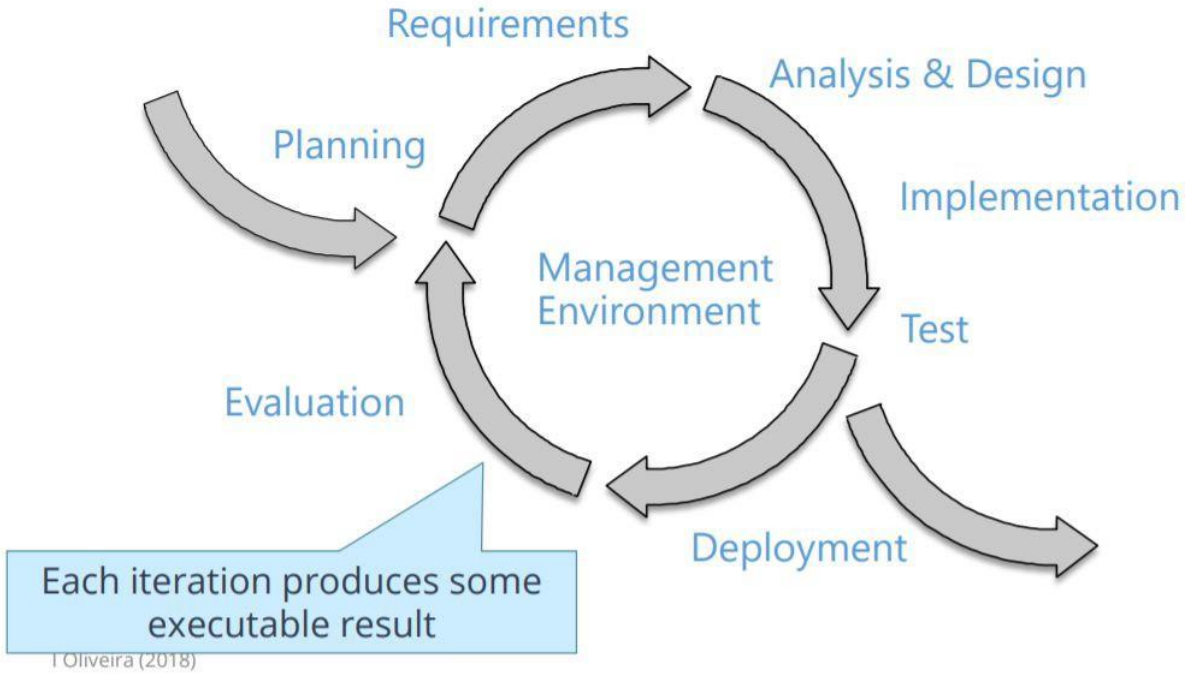
Figure 1.2: Physics of Test-Driven Development

Quando o tempo para descobrir um erro (Td) se aproxima de zero, o tempo para encontrar a causa raiz do erro (Tfind) 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 (Tfix) 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"?**
- **Relacione os critérios de aceitação da história (user-story) com o teste Agile.**
  RESPONDIDAS NA ANTERIOR

Abordagens complementares

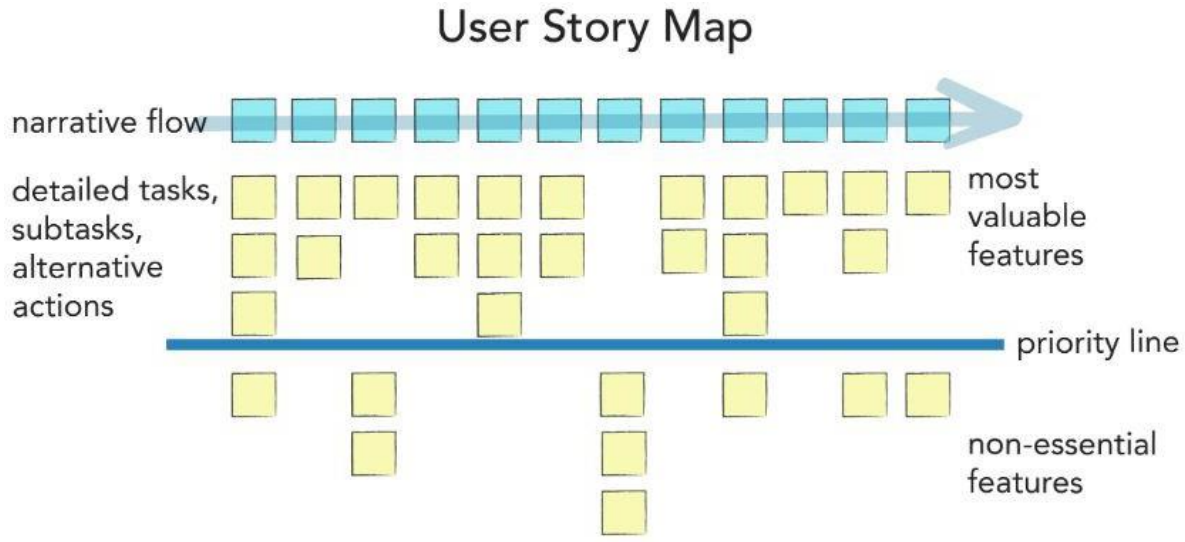
Histórias e métodos ágeis

- **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.

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

  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.**



- **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.

- **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 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, respetivamente (um modelo detalhado de Caso de Uso pode conter outros elementos).

  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 porquê 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.

- **O que é a pontuação de uma história e como é que é determinada?**
  A pontuação corresponde duma forma informal a dificuldade e velocidade que a equipa acha que levará a implementar tal funcionalidade.

- **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 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 subequipa 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.