01:

Identify three main axes in digital transformation:

User experience , processos operacionais e business model

Explain what the Systems Development Cycle is:

Processo de determinar como um sistema de informação pode ser criado. Tem quatro fases principais – planeamento, análise, design e implementação. Cada fase tem uma sequência de ações e no final de cada fase há entregas do que foi feito. O membro “principal” da equipa é o analista, que analisa a situação de negócio e identifica oportunidades de melhoria – que preparará para implementação.

02:

Justify the use of models in systems engineering

Modelos facilitam a visualização do sistema high level. Para além disso, especificam e documentam a estrutura e o comportamento do sistema antes da implementação, o que tornará a implementação em si mais smooth. Assim, também servem de referência para orientar a construção, também documentando as decisões feitas ao longo do processo de design.

Enumerate advantages of visual models

Ao criar modelos visuais, a comunicação tornar-se-á mais clara e sucinta. Para além disso, torna o desenho e a implementação coerentes. Além disso, tendo modelos visuais, ter-se-á um nível de detalhe apropriado para cada fase. Não só, certas ferramentas tornam a construção um processo automático através de modelos.

Identify the main diagrams in UML and their modeling viewpoint

Diagrama de use case - organiza as funcionalidades do sistema em episódios de utilização

Diagrama de atividade – explica procedimentos do domínio

Modelo de domínio – usa classes para representar os conceitos do problema, tornando as ligações entre eles mais claras

Diagrama de estados – análogo ao diagrama de atividade, mas com foco no estado de um componente

02b:

Distinguish structured activities from actions; control flows from data flows

Ações são algo que é feito ao sistema, enquanto que atividades estruturadas são, habitualmente, reações a ações, sendo intrínsecas a este. Control flows diferem de data flows por natureza – control flows ditam como algo vai acontecer e data flows apenas ditam fluxos de informação.

Identify when to use Activity modeling

Como diagramas de atividade mostram o fluxo de ações e dados, são úteis para modelar fluxos de trabalho/processos de negócio e processos computacionais/algoritmos. Também são úteis na descrição de sequências de interações entre atores num caso de utilização.

03:

Identificar atividades comuns a todos os projetos (ciclo de vida)

Identificação do valor que o sistema gerará para a organização, pedido do sistema e a análise de viabilidade são apresentadas às entidades que dão o “okay” a novos projetos, criação de plano de trabalho, análise dos sistemas existentes, recolha de requisitos, conceitualização de uma solução, definição de estratégia de desenvolvimento, desenho de arquitetura do sistema, modelo de dados e programas, implementação do sistema, instalação e transição para produção e, finalmente, execução do plano de suporte.

Distinguir projetos sequenciais de projetos evolutivos

Em projetos sequenciais, todas as atividades do processo são planeadas com antecedência e o progresso do projeto é comparado com o plano original, enquanto que num projeto evolutivo o planeamento é incremental, sendo possível adaptar caso os requisitos do cliente mudem.

Descrever a estrutura do Unified Process (fases, objetivos, iterações)

Inception – iteração curta que produz o documento relativo à visão e um business case inicial. Também reduz o risco, identificando os requirements chave

Elaboration – faz “sacrifícios” no que toca a riscos de business e produção de valor. Também produz e valida a arquitetura.

Construction - construção do que não foi feito na elaboração, criando um sistema operacional

Transition – validação final e deplyment

Mapear disciplinas técnicas nas fases do OpenUP

Na inception, as duas disciplinas técnicas com maior foco são business modelling e requirements. Na elaboração, o foco é colocado nos requirements e na análise e design. Na construção, é na implementação e configuração e management de alterações. Estas mesmas fases mantém o foco na transição.

04:

>write in the form of subject-verb-direct object

>make sure it’s clear who the initiator of the step is

>write from an independent observer’s perspective

>write at about the same level of abstraction

>ensure the use case has a sensible set of steps

>apply the KISS principle liberally (keep it simple, stupid)

>write repeating instructions after the set of steps to be repeated

What happens? Then what? What if one of the steps fails/doesn’t go that way?

05:

Distinguish functional and non-functional requirements

Requisitos funcionais são serviços e funções que o sistema tem de ser capaz de fazer, enquanto que requisitos não funcionais são restrições/compromissos no software em si.

Enumerate requirement gathering techniques and argument when to use each one

Entrevistas com o cliente são um bom método de encontrar requirements surface-level, numa fase inicial do projeto será bom estabelecer com o cliente quais são as necessidades base. Questionários com utilizadores futuros também dão origem a requirements surface-level ao nível funcional. Análise de documentos será útil para não cometer os erros que outros já cometeram, tal como na observação.

Describe requirements documentation techniques

Requisitos são, por norma, documentados em tabelas e são bem formados, isto é, podem ser verificados mensuravelmente.

Understand the relation between requirements and use cases

Um use case irá, por norma, assentar nos requisitos já estabelecidos, ou, na criação dele, irá tornar mais claros alguns requisitos que ainda não tinham sido considerados.

06:

Describe the structure of the Unified Process (phases, milestone objectives, iterations)

A primeira iteração é a conceção, que tem como objetivo atingir um consenso com os objetivos dos stakeholders e garantir as condições necessárias para a viabilidade do projeto. As atividades associadas são elaboração de modelos de requisitos, identificação de interações necessárias com entidades externas, delineamento de casos de utilização e planeamento de fases futuras. Os produtos resultantes são a visão geral do problema, modelos de casos de utilização, uma avaliação de risco inicial, uma justificação da viabilidade do projeto, um plano de projeto e protótipos iniciais.

A segunda é a elaboração, cujos objetivos são validar cenários e definir a arquitetura. Nesta fase, é detalhado o modelo de casos de utilização, analisado o domínio, é definida a arquitetura e essa mesma é validada via uma implementação exploratória. Os produtos são modelos de casos de utilização com especificações abrangentes, requisitos tanto funcionais quanto não funcionais, descrição da arquitetura do software, protótipos, uma implementação exploratória, medidas de mitigação de risco e um plano de projeto revisto.

A última é a construção, cujo objetivo é entregar valor. As atividades são definir realização dos casos de utilização, especialmente no que toca a interações entre objetos, desenvolvimento do UI/UX, implementação de código e validação do incremento. Os produtos são a implementação parcial e a demonstração de testes funcionais.

08:

Distinguish user centric and product centric requirements specification

Requisitos definidos via um estudo centrado no utilizador ir-se-ão focar no que o utilizador deseja fazer, enquanto que uma recolha de requisitos via um estudo centrado no produto irá definir o que o produto deve fazer.

Processos de especificação de requisitos baseados no utilizador irão ser obtidos via entrevistas, casos de uso, entre outros, enquanto requisitos baseados no produto serão obtidos, maioritariamente, num estudo do mercado e do produto em si.

Describe requirements documentation techniques

A documentação de requisitos é feita via modelos de casos de uso, sendo que nestes os requisitos definidos são, maioritariamente, funcionais. Requisitos também são documentados em especificações suplementares, glossários e regras de negócio.

As especificações suplementares conterão tudo o que não se encontra contemplado nos casos de uso, sendo que por norma são requisitos ao nível do sistema e não-funcionais.

Requisitos definidos no glossário são requisitos relativos a dados, como regras de validação, entre outros.

Requisitos definidos em regras de negócio são requisitos que estão relacionados com elementos externos ao sistema, como leis ou políticas de empresa.

Understand the relation between requirements and use cases

A maioria dos requisites funcionais são estabelecidos nos use-cases, uma vez que estes são definições da grande maioria das interações que poderão ocorrer com o sistema.

Identify the meaning of evolutionary requirements concept

Requisitos evolutivos são requisitos que, mesmo depois de definidos, são mutáveis. Esta capacidade de alterá-los é importante para desenvolvimento de software pois, ao longo de um projeto, é natural que se veja que o sistema tem outras necessidades. Para além disso, a vontade do cliente pode mudar.

Describe the types of information the Analyst collects in the requirements engineering activities

Nesta fase do projeto o Analista irá recolher requisitos de negócio, requisitos de utilizador, regras de negócio, requisitos funcionais, atributos de qualidade, requisitos externos, limitações, requisitos de dados e ideias para soluções.

Explain how to implement requirements tracking activities

Atividades de atualização de requisitos passam pela revisão de todas as atividades necessárias para os definir, e documentação associada. É possível que seja criada nova documentação e casos de uso sejam alterados para um melhor encaixe nos novos requisitos do sistema.

15:

Façade

STRUCTURAL

A façade is a new interface for a library. A library is composed within a façade. A façade also shields variations in the implementation from the client. The objective of a façade is to simplify the interface used. The interface is not necessarily given; however, the implementation is chosen statically.

Adapter

STRUCTURAL

An adapter is a design application to a common interface, adapting other libraries to it. An adapter object is interposed between client and implementation. An adapter also shields variations in the implementation from the client. The objective of an adapter is to make existing interfaces connect. The interface is always given. An adapter is also dynamic to several implementations.

Strategy

BEHAVIORAL

A Strategy can be implemented with inheritance, being so that every new Context has a specific algorithm associated. Every variable that varies needs to be incapsulated. It’s applicable when many classes only differ in behavior and a client needs different variants of an algorithm. Using a Strategy will make code more extensible with new strategies. Strategies also separate algorithm from context and add objects and dynamism.

Abstract Factory

CREATIONAL

An abstract factory is appliable when the system should be independent of product creation, you want to configure multiple families of products or you want to ensure that a product family is used together. It isolates concrete classes, makes it easier to change product families, helps ensure consistent use of family and makes it hard to support new kinds of products.

Singleton

CREATIONAL

Applicable when there must be exactly one instance of a class, it must be accessible to clients from a well-known place and when the sole instance should be extensible by subclassing, with unmodified clients using the subclass. It gives controlled access to a sole instance, a reduced name space and tends to be more flexible than class operations. For it to be implemented, the constructor must be protected, the instance variable must be private and public operations must return singleton.

Observer

BEHAVIORAL

Applicable when an abstraction has two aspects, one dependent on the other, and you want to reuse each. Changing one object requires changing others, and you have no way to know how many those others would be. An object should also be able to notify others without knowing who they are. Using observers also makes coupling looser between subject and observer, enhancing reuse. It also supports broadcast communication and notifications can lead to further updates, causing a cascade effect.

Bridge

STRUCTURAL

A bridge is used when you want to define multiple abstractions and need to implement in multiple ways. By using one, you avoid an explosion in the number of classes, decouple abstraction from implementation, hide implementation from clients and get a fixed implementation interface.

Template Method

BEHAVIORAL

Template methods are useful when an algorithm consists of varying parts that must be customized, and common behavior should be factored and localized to avoid code duplication. Controlling subclass extensions to specific operations is also a valid reason to use template methods. Using template methods usually leads to code reutilization, more control on the client side, and a guarantee that the invariant parts of the algorithm aren’t changed by subclasses.

Decorator

STRUCTURAL

Used when it is necessary to add responsibilities to individual objects dynamically and transparently, but those responsibilities can be withdrawn too. Plus, it’s good to use decorators when extending by subclassing is impractical. Decorators are more flexible than static inheritance, they avoid monolithic classes, break object identity and create lots of little objects.

Proxy

STRUCTURAL

A proxy can be useful when you need a more sophisticated object reference than a simple pointer, and using a proxy adds a level of indirection, hiding distribution and optimizations from the client, and adding housekeeping tasks.

Mediator

BEHAVIORAL

Mediators are applicable when there is a set of objects that communicate in well-defined but complex ways. Reusing an object can be difficult because of these communications. A behavior distributed between several classes should be customizable without lots of subclassing too. Mediators avoid excessive subclassing to customize behavior, decouple colleagues, enhance reuse, simplify object protocols, the way objects cooperate is abstracted and control is centralized.

Command

BEHAVIORAL

Applicable when you need to parametrize objects by an action to perform, or you need to specify, queue and execute requests at different times. Commands are also useful when you structure a system around high-level operations built out of primitives. Commands support undo and logging crashes (that can be reapplied after a crash). Commands also decouple the object that invokes the operation from the one that performs it. Commands can be explicitly manipulated and, not only that, but they can also be grouped to create composite commands and it is easy to add new ones.

Visitor

BEHAVIORAL

Visitors are useful when there is a structure with many classes, being that the set of classes is stable, and the programmer wants to perform operations that depend on classes or wants to define new operations. Visitors make adding new operations easy, however, they make adding elements hard. You can group behavior into a Visitor and even store a state, but elements must expose their interface.