Análise de Sistemas e Engenharia de Software

Respostas às Perguntas

Capítulo 1 - Introdução ao Software e à Engenharia de Software

1. **Pergunta:** Defina o que é software e cite os três principais tipos.
   Resposta: Software é o conjunto de instruções que diz ao hardware o que fazer; é tudo aquilo que não se consegue tocar num computador, mas que o faz funcionar. Os três principais tipos são: software de sistema (como Windows, macOS), software de aplicação (como Word, Spotify) e software de desenvolvimento (ferramentas para criar outros softwares). (Página 4)

2. **Pergunta:** Quais são as principais diferenças entre software genérico e software personalizado?
   Resposta: Software genérico é feito para o mercado em geral, tem baixo custo por utilizador, mas pode não se ajustar às necessidades específicas (ex: Microsoft Excel). Software personalizado é desenvolvido à medida para uma organização específica, responde exatamente ao que é preciso, mas é mais caro e demorado de desenvolver (ex: sistema de gestão para um hospital específico). (Página 4-5)

3. **Pergunta:** Por que a Engenharia de Software é importante no desenvolvimento de sistemas?
   Resposta: A Engenharia de Software é importante porque: evita o caos organizando o desenvolvimento; reduz falhas, o que é crítico em sistemas como aviões ou hospitais; controla custos e prazos; e garante qualidade com testes, documentação e boas práticas. (Página 5-6)

4. **Pergunta:** Explique por que "Engenharia de Software ≠ Programar".
   Resposta: Programar é apenas uma parte da engenharia de software. Esta também inclui levantamento de requisitos, desenho de soluções, testes, validação, manutenção e gestão de equipas e projetos. A engenharia de software é uma disciplina mais ampla que envolve todo o ciclo de vida do desenvolvimento. (Página 6)

Capítulo 2 - Processos de Desenvolvimento de Software

5. **Pergunta:** O que é um processo de software e quais são as fases comuns independentemente do modelo adotado?
   Resposta: Um processo de software é um conjunto estruturado de atividades, tarefas e decisões que guiam o desenvolvimento de software do início ao fim. As fases comuns são: análise (o que é necessário), desenho (como será construído), implementação (codificar e integrar), testes (verificar se funciona) e manutenção (corrigir e melhorar após entrega). (Página 8-9)

6. **Pergunta:** Descreva o Modelo em Cascata (Waterfall), suas vantagens e desvantagens.
   Resposta: No Modelo em Cascata, cada fase só começa depois da anterior terminar, como uma linha de montagem. Vantagens: é fácil de entender e gerir, e tem boa documentação. Desvantagens: é pouco flexível a mudanças e o cliente só vê o produto no final. É ideal para projetos bem definidos, com poucos riscos e mudanças. (Página 9-10)

7. **Pergunta:** Explique como funciona o Modelo de Prototipagem e em que situações ele é mais útil.
   Resposta: No Modelo de Prototipagem, constrói-se uma maquete funcional para validar ideias com o utilizador antes de investir na solução final. Vantagens: melhor entendimento dos requisitos e envolvimento do utilizador. Desvantagens: pode criar falsas expectativas e aumentar o custo se mal gerido. É muito útil quando os requisitos estão pouco claros ou o cliente tem dificuldade em expressar o que quer. (Página 10)

8. **Pergunta:** Descreva o Modelo Espiral e suas características principais.
   Resposta: O Modelo Espiral combina o rigor do cascata com a flexibilidade da prototipagem e foca-se em gestão de riscos. É organizado por ciclos (iterações), onde a cada volta: analisa-se os riscos, define-se objetivos, cria-se uma versão e avalia-se para planejar a próxima iteração. Vantagens: foco na gestão de riscos e adaptabilidade a projetos grandes. Desvantagens: requer gestão experiente e é mais difícil de planear. (Página 10-11)

9. **Pergunta:** O que é o Unified Process (UP) e quais são suas fases principais?
   Resposta: O Unified Process é uma abordagem híbrida que une boas práticas de diferentes modelos num processo estruturado e iterativo. É dirigido por casos de uso e centrado na arquitetura. Suas fases principais são: Iniciação (definir visão e objetivos), Elaboração (definir arquitetura e requisitos principais), Construção (desenvolver funcionalidades em iterações) e Transição (preparar para entrega e uso real). (Página 11-12)

10. **Pergunta:** Quais são as atividades fundamentais de um processo de software?
    Resposta: As atividades fundamentais são: Especificação (levantar e documentar requisitos), Desenho e Implementação (criar soluções e programar), Validação (verificar se o software cumpre os requisitos) e Evolução (corrigir, adaptar e melhorar ao longo do tempo). (Página 12)

Capítulo 3 - Metodologias Ágeis de Desenvolvimento

11. **Pergunta:** Quais são os quatro valores fundamentais do Manifesto Ágil?
    Resposta: Os quatro valores do Manifesto Ágil são: valorizar indivíduos e interações mais que processos e ferramentas; software funcional mais que documentação extensa; colaboração com o cliente mais que negociação de contratos; e responder a mudanças mais que seguir um plano rígido. (Página 14)

12. **Pergunta:** Liste e explique três princípios ágeis fundamentais.
    Resposta: Três princípios ágeis fundamentais são: 1) Entregar software funcional frequentemente; 2) Acolher mudanças, mesmo em fases tardias do desenvolvimento; 3) Trabalhar em equipa com motivação e confiança. O foco está em entregar valor rapidamente, ouvindo o cliente e adaptando-se. (Página 14-15)

13. **Pergunta:** Quais são as práticas principais do Extreme Programming (XP) e como ele contribui para a qualidade do software?
    Resposta: As práticas principais do XP incluem: programação em par (dois programadores partilham um computador), testes automáticos, integração contínua, design simples, refatoração constante, propriedade coletiva do código, cliente sempre presente e iterações curtas. Estas práticas contribuem para um código mais limpo e testado, alta colaboração, flexibilidade extrema e rápido feedback do cliente. (Página 15-16)

14. **Pergunta:** Descreva a estrutura do Scrum, seus papéis principais e reuniões.
    Resposta: O Scrum organiza equipas em ciclos curtos chamados sprints, com papéis bem definidos. Estrutura: Sprint (ciclo de trabalho de 2-4 semanas), Product Backlog (lista priorizada de funcionalidades) e Sprint Backlog (tarefas para a sprint atual). Papéis: Product Owner (define o que fazer), Scrum Master (facilita o processo) e Equipa de desenvolvimento (implementa). Reuniões: Daily Scrum, Sprint Planning, Sprint Review e Sprint Retrospective. (Página 16-17)

15. **Pergunta:** Como as metodologias ágeis devem ser adaptadas ao contexto organizacional?
    Resposta: As metodologias ágeis devem ser adaptadas considerando: tamanho da equipa, nível de experiência, cultura da organização, frequência de mudanças nos requisitos, envolvimento do cliente e regulamentações do setor. É comum ver práticas de Scrum combinadas com Kanban, XP ou elementos do cascata, formando modelos "ágeis à medida". O importante é aplicar os princípios ágeis para criar software útil, com qualidade e no tempo certo, não seguir fórmulas fixas. (Página 17-18)

Capítulo 4 - Análise e Elicitação de Requisitos

16. **Pergunta:** O que é Engenharia de Requisitos e quais são suas fases principais?
    Resposta: A Engenharia de Requisitos é o processo de descobrir, analisar, documentar e gerir as funcionalidades e restrições de um sistema. Suas fases principais são: Elicitação (identificar necessidades), Análise e negociação (refinar e resolver conflitos), Documentação (formalizar os requisitos), Validação (verificar se estão corretos) e Gestão de requisitos (acompanhar mudanças). (Página 19)

17. **Pergunta:** Quais são algumas técnicas eficazes para elicitação de requisitos?
    Resposta: Algumas técnicas eficazes de elicitação são: entrevistas (estruturadas ou informais), questionários, workshops, observação direta, brainstorming, análise de sistemas existentes e prototipagem. Não se trata apenas de "ouvir" o cliente, mas de interpretar e transformar necessidades vagas em requisitos claros e úteis. (Página 19-20)

18. **Pergunta:** Explique o que são personas e como elas ajudam no desenvolvimento de software.
    Resposta: Personas são personagens fictícias que representam tipos de utilizadores reais, incluindo dados como nome, idade, função, objetivos, motivações e frustrações. Elas ajudam a criar soluções centradas no utilizador, tornando mais tangíveis as necessidades e expectativas dos utilizadores finais. Por exemplo, "Joana, 42 anos, enfermeira, usa o sistema hospitalar para registar dados durante turnos apertados". (Página 20)

19. **Pergunta:** Como são estruturadas as histórias de utilizador e qual o seu papel no desenvolvimento ágil?
    Resposta: As histórias de utilizador são frases simples que expressam funcionalidades do ponto de vista do utilizador, seguindo o formato: "Como [persona], quero [funcionalidade], para [benefício]". Por exemplo: "Como enfermeira, quero registar um novo paciente com poucos campos obrigatórios, para poupar tempo e evitar erros". Elas são fundamentais no desenvolvimento ágil por serem curtas, centradas no valor para o utilizador e facilitarem o planeamento e priorização. (Página 21)

20. **Pergunta:** Por que a validação de requisitos é importante e que técnicas podem ser usadas?
    Resposta: A validação de requisitos é importante para garantir que são corretos, compreensíveis, testáveis e relevantes. Técnicas incluem: revisões com stakeholders, prototipagem rápida para feedback, testes de consistência e ambiguidade, e verificação da rastreabilidade (cada requisito liga-se a código e testes). Requisitos mal definidos geram retrabalho, custos e frustração — quanto mais cedo forem corrigidos, melhor. (Página 22-23)

Capítulo 5 - Modelação de Sistemas

21. **Pergunta:** Por que a modelação é importante no desenvolvimento de sistemas?
    Resposta: A modelação é importante porque permite: redução da complexidade (foca nos aspetos relevantes); comunicação eficaz entre analistas, clientes, programadores e testers; documentação do sistema (essencial para manutenção futura); e serve como base para a implementação, ligando a análise ao código. Modelar é como desenhar o mapa antes de iniciar a viagem. (Página 24-25)

22. **Pergunta:** Quais são os principais tipos de modelos na UML e sua finalidade?
    Resposta: Os principais tipos são: Diagramas de Casos de Uso (mostram funcionalidades e utilizadores), Diagramas de Classes (representam estruturas e relações de dados), Diagramas de Sequência (mostram interação entre objetos no tempo), Diagramas de Atividades (fluxos de trabalho e decisões), Diagramas de Estados (ciclo de vida de um objeto) e Diagramas de Componentes (arquitetura modular). A escolha depende da fase do projeto e do público-alvo da modelação. (Página 26)

23. **Pergunta:** Explique os conceitos-chave da modelação orientada a objetos.
    Resposta: Os conceitos-chave são: Classe (modelo de um objeto, ex: Pessoa), Objeto (instância concreta, ex: Pedro), Atributos (características, ex: nome), Métodos (ações, ex: falar()), Associação (relação entre classes), Herança (uma classe "filha" herda de uma "pai") e Encapsulamento (esconder o funcionamento interno). A modelação orientada a objetos reflete a realidade através destes elementos. (Página 27-28)

24. **Pergunta:** Descreva o ciclo prático da modelação orientada a objetos com um exemplo simples.
    Resposta: O ciclo prático é: 1) Identificar objetos relevantes no domínio do problema; 2) Criar as classes correspondentes; 3) Definir atributos e métodos; 4) Representar relações entre classes; 5) Criar diagramas para ilustrar comportamentos. Exemplo: Classe Paciente (atributos: nome, número de utente; métodos: agendarConsulta()), Classe Consulta (atributos: data, médico; métodos: confirmar()), com a relação "um Paciente pode ter várias Consultas". (Página 27-28)

Capítulo 6 - Arquitetura de Software

25. **Pergunta:** O que é arquitetura de software e por que é importante?
    Resposta: Arquitetura de software é o conjunto de decisões estruturais fundamentais que determinam como os módulos se organizam, interagem, que tecnologias se usam e que padrões se seguem. É importante porque define a base técnica para o desenvolvimento, afeta diretamente a qualidade (desempenho, segurança, escalabilidade), facilita a manutenção e evolução, e suporta decisões de negócio. (Página 30)

26. **Pergunta:** Quais são as características desejáveis numa boa arquitetura de software?
    Resposta: As características desejáveis são: modularidade (componentes independentes e reutilizáveis), flexibilidade (permite mudanças com impacto mínimo), escalabilidade (fácil de crescer sem reestruturar tudo), desempenho (responde bem sob carga), segurança (protege dados e acesso) e manutenibilidade (fácil de corrigir e evoluir). (Página 31)

27. **Pergunta:** Explique o que são arquiteturas distribuídas e seus principais desafios.
    Resposta: Arquiteturas distribuídas dividem o sistema em componentes que podem estar em máquinas diferentes, comunicando pela rede (ex: frontend no navegador, backend e base de dados em servidores separados). Vantagens: melhor desempenho, escalabilidade horizontal e tolerância a falhas. Desafios: comunicação remota (latência, falhas de rede), sincronização de dados e segurança na transmissão de informação. (Página 31-32)

28. **Pergunta:** Descreva o padrão MVC (Model-View-Controller) e seu funcionamento.
    Resposta: O MVC divide a aplicação em três camadas: Modelo (representa dados e lógica de negócio), Vista (interface com o utilizador) e Controlador (coordena ações entre modelo e vista). Funcionamento: 1) O utilizador interage com a vista; 2) O controlador interpreta e atualiza o modelo; 3) O modelo altera os dados e notifica a vista; 4) A vista mostra os dados atualizados. Este padrão ajuda a separar responsabilidades. (Página 32-33)

29. **Pergunta:** O que são trade-offs em decisões arquiteturais e dê exemplos.
    Resposta: Trade-offs são situações onde, ao optar por uma solução, ganha-se algo mas perde-se outra coisa. Exemplos: usar microserviços traz escalabilidade e independência, mas aumenta a complexidade; armazenar dados localmente no cliente aumenta a rapidez, mas cria desafios de sincronização e segurança; adotar arquitetura em camadas melhora a organização, mas pode aumentar a latência. A chave está em equilibrar os fatores com base no contexto do projeto. (Página 33)

Capítulo 7 - Computação em Nuvem

30. **Pergunta:** O que é computação em nuvem e quais são suas características principais?
    Resposta: Computação em nuvem é um modelo que permite acesso remoto a recursos informáticos através da internet, sem necessidade de gerir a infraestrutura localmente. Características: on-demand (acesso quando necessário), auto-serviço (configuração própria de ambientes), escalabilidade (aumenta ou diminui conforme o uso), pagamento conforme o consumo e multitenancy (múltiplos utilizadores com segurança isolada). (Página 35)

31. **Pergunta:** Compare os três principais modelos de serviço na cloud: IaaS, PaaS e SaaS.
    Resposta: IaaS (Infrastructure as a Service) fornece infraestrutura básica (servidores, redes), com alta responsabilidade do utilizador (ex: Azure). PaaS (Platform as a Service) fornece ambiente de desenvolvimento pronto a usar, o utilizador apenas desenvolve a aplicação (ex: Heroku). SaaS (Software as a Service) oferece aplicações completas prontas a usar via navegador, o utilizador apenas usa o serviço (ex: Gmail). Estes modelos representam diferentes níveis de controlo e abstração. (Página 36-37)

32. **Pergunta:** Explique a diferença entre arquiteturas multi-tenant e multi-instance na nuvem.
    Resposta: Em multi-tenant, todos os clientes usam a mesma instância da aplicação, mas os dados são isolados (ex: um servidor de email serve milhares de utilizadores). Vantagens: menor custo e atualizações centralizadas. Desvantagens: complexidade de isolamento e personalização limitada. Em multi-instance, cada cliente tem sua própria instância da aplicação. Vantagens: maior isolamento e personalização. Desvantagens: maior custo de gestão. Multi-tenant é ideal para muitos clientes com necessidades semelhantes, multi-instance para poucos clientes com exigências específicas. (Página 37-38)

Capítulo 8 - Arquitetura de Microserviços

33. **Pergunta:** O que são microserviços e quais suas características principais?
    Resposta: A arquitetura de microserviços divide uma aplicação em vários serviços pequenos, autónomos e especializados, que comunicam entre si através de APIs. Características: autonomia (cada serviço funciona independentemente), responsabilidade única (cada serviço tem função específica), desenvolvimento descentralizado (equipas paralelas), escalabilidade independente, resiliência (se um falhar, outros continuam) e implantação contínua (cada serviço pode ser atualizado separadamente). (Página 40)

34. **Pergunta:** Como é feita a comunicação entre microserviços e quais os padrões comuns?
    Resposta: Os microserviços comunicam através de chamadas HTTP/REST (síncrona, quando se precisa de resposta imediata) ou mensagens assíncronas via filas (quando se pode esperar ou processar em segundo plano). Como o sistema é distribuído, técnicas para gestão de falhas incluem timeouts (evitar espera indefinida), retries (tentar novamente após falha temporária), circuit breaker (desativar chamadas a serviços instáveis) e fallbacks (fornecer resposta alternativa). (Página 41-42)

35. **Pergunta:** Explique como a arquitetura de microserviços facilita a implementação contínua.
    Resposta: Cada microserviço tem seu próprio repositório de código, pipeline de CI/CD, testes e ciclo de vida autónomo. Isto permite publicação frequente de atualizações sem impactar o resto do sistema, redução de riscos (alterações pequenas são mais fáceis de testar) e resposta rápida a falhas ou necessidades. Ferramentas comuns incluem containers (Docker), orquestração (Kubernetes) e pipelines de CI/CD (Jenkins, GitLab CI). (Página 42-43)

Capítulo 9 - Segurança e Privacidade em Software

36. **Pergunta:** Quais são os três princípios básicos da segurança da informação (triângulo CIA)?
    Resposta: Os três princípios básicos são: Confidencialidade (proteger informação contra acessos não autorizados), Integridade (garantir que a informação não foi alterada indevidamente) e Disponibilidade (assegurar que a informação está acessível quando necessário). Estes formam o chamado triângulo CIA (Confidentiality, Integrity, Availability). (Página 45)

37. **Pergunta:** Liste e explique três ameaças comuns à segurança de software.
    Resposta: Três ameaças comuns são: 1) Malware - software malicioso como vírus ou ransomware; 2) SQL Injection - injeção de comandos maliciosos em campos de input para manipular a base de dados; 3) Cross-Site Scripting (XSS) - injeção de scripts maliciosos em páginas web que são executados no navegador da vítima. Outras ameaças incluem phishing, ataques de força bruta e DDoS. (Página 45)

38. **Pergunta:** Diferencie autenticação e autorização em sistemas de software.
    Resposta: Autenticação é o processo de verificação da identidade do utilizador (saber quem está a aceder), podendo usar username/password, autenticação multifator ou biometria. Autorização é o processo de atribuição de permissões e acessos com base no papel do utilizador (decidir o que cada utilizador pode fazer), geralmente implementada com sistemas de controlo de acesso baseados em papéis (RBAC) ou atributos (ABAC). (Página 46)

39. **Pergunta:** Explique os tipos de encriptação e suas aplicações em software.
    Resposta: A encriptação torna os dados ilegíveis para quem não tem a chave certa, aplicando-se em trânsito (dados circulando pela internet, ex: HTTPS) e em repouso (dados armazenados, ex: bases de dados encriptadas). Tipos: encriptação simétrica (mesma chave para encriptar e desencriptar) e assimétrica (par de chaves: pública para encriptar, privada para desencriptar). (Página 47)

40. **Pergunta:** Quais são os princípios fundamentais do RGPD (Regulamento Geral sobre a Proteção de Dados)?
    Resposta: Os princípios do RGPD são: Consentimento (o utilizador deve autorizar o tratamento dos seus dados), Finalidade (os dados só podem ser usados para os fins comunicados), Minimização (só se recolhe o necessário), Transparência (o utilizador tem direito a saber como e porquê os dados são tratados) e Direito ao esquecimento (o utilizador pode pedir a eliminação dos seus dados). (Página 47-48)

Capítulo 10 - Programação Confiável

41. **Pergunta:** Quais são algumas boas práticas essenciais para programação confiável?
    Resposta: Boas práticas incluem: definir requisitos bem claros (evita ambiguidade), seguir convenções de codificação (facilita leitura e manutenção), evitar duplicação de código (reduz erros), dividir em funções/módulos pequenos (facilita testes e reutilização) e documentar intenções (ajuda a compreender decisões). Exemplos de boas decisões de design incluem usar tipos fortes, preferir código explícito e limitar efeitos colaterais. (Página 50)

42. **Pergunta:** Por que a validação de entradas é crucial e que tipos de validação devem ser feitos?
    Resposta: Toda entrada deve ser tratada como potencialmente maliciosa ou incorreta. Para texto livre, deve-se validar tamanho, caracteres inválidos e injeções; para datas, formato e intervalos válidos; para números, intervalos permitidos e tipo correto; para ficheiros, tipo de conteúdo, tamanho e extensões seguras. A validação adequada protege o sistema contra erros e ataques. (Página 51)

43. **Pergunta:** Explique a importância da gestão de erros e as boas práticas associadas.
    Resposta: A gestão de erros garante que os erros são detetados, tratados com segurança e comunicados adequadamente. Boas práticas incluem: try/catch estruturado (isolar blocos suscetíveis a falhas), logs de erro (registar o que falhou, onde e porquê), mensagens amigáveis (não mostrar exceções cruas ao utilizador) e fallbacks seguros (fornecer alternativas quando possível). Um erro bem tratado não compromete a segurança nem a experiência do utilizador. (Página 51)

44. **Pergunta:** O que são padrões de desenho (design patterns) e dê exemplos.
    Resposta: Padrões de desenho são soluções recorrentes para problemas comuns de design, servindo como guias para estruturar código de forma robusta e reutilizável. Exemplos: Singleton (garantir que só existe uma instância de uma classe), Factory (criar objetos sem acoplar a lógica de criação), Observer (notificar partes do sistema quando algo muda), Strategy (trocar algoritmos em tempo de execução) e MVC (separar dados, lógica e interface). (Página 52)

45. **Pergunta:** O que é refatoração e qual a sua importância no desenvolvimento de software?
    Resposta: Refatorar é melhorar o código sem alterar o seu comportamento, tornando-o mais limpo, compreensível, reduzindo duplicações, melhorando desempenho e eliminando "débitos técnicos". Exemplos incluem dividir funções longas em funções pequenas, substituir condições complexas por guard clauses, e eliminar código duplicado com funções reutilizáveis. Ferramentas úteis incluem linters, analisadores estáticos e testes automatizados. (Página 53)

Capítulo 11 - Testes de Software

46. **Pergunta:** Quais são os principais tipos de testes de software e seus objetivos?
    Resposta: Os principais tipos são: Testes unitários (testam uma função, método ou componente isolado, ex: testar calcularIVA()), Testes de integração (testam interações entre componentes, ex: verificar se login acede corretamente à base de dados) e Testes de sistema (testam o sistema como um todo, ex: validar processo de compra online). Os objetivos são deteção precoce de erros, garantir que as partes "falam a mesma língua" e validar o comportamento final. (Página 55)

47. **Pergunta:** Explique o ciclo do TDD (Test-Driven Development) e suas vantagens.
    Resposta: O ciclo do TDD é: 1) Escrever um teste (que falha); 2) Escrever o código mínimo para passar no teste; 3) Executar o teste (deve passar); 4) Refatorar o código mantendo os testes a passar; 5) Repetir para a próxima funcionalidade. Vantagens: força a clareza dos requisitos, garante cobertura de testes desde o início, torna a refatoração mais segura e serve como documentação viva do comportamento esperado. (Página 56)

48. **Pergunta:** Qual a importância das revisões de código (code reviews) e como devem ser conduzidas?
    Resposta: Revisões de código são a leitura crítica do código por outro programador antes de ser integrado. Ajudam a detetar erros, violações de boas práticas, códigos difíceis de manter e vulnerabilidades. Boas práticas incluem: usar ferramentas de apoio (GitHub, GitLab), focar em mudanças pequenas e frequentes, e discutir com empatia e foco na melhoria coletiva. (Página 56-57)

49. **Pergunta:** O que são testes de segurança e que tipos existem?
    Resposta: Testes de segurança procuram detetar vulnerabilidades que possam ser exploradas maliciosamente. Tipos incluem: testes de injeção (verificam se o sistema bloqueia comandos maliciosos), testes de autenticação (verificam se login e permissões funcionam corretamente), testes de exposição (verificam se mensagens de erro revelam informação sensível) e scans automatizados (procuram vulnerabilidades conhecidas no código e bibliotecas). Ferramentas populares incluem OWASP ZAP, SonarQube e Snyk. (Página 57-58)

Capítulo 12 - DevOps e Gestão de Código

50. **Pergunta:** O que é DevOps e quais são os conceitos-chave de CI/CD?
    Resposta: DevOps é a prática que aproxima o desenvolvimento (Dev) da operação (Ops), promovendo automação, colaboração contínua e entrega frequente de software com qualidade. Conceitos-chave: CI (Continuous Integration) - programadores integram frequentemente seu código; CD (Continuous Delivery) - o código está sempre pronto para ser lançado; CD (Continuous Deployment) - o código é lançado automaticamente após testes. (Página 60)

51. **Pergunta:** Explique o funcionamento prático de um pipeline de CI/CD e seus benefícios.
    Resposta: Na prática: 1) Cada programador faz alterações; 2) O código é automaticamente testado e integrado num repositório comum; 3) Se passar nos testes, é preparado para ser lançado (ou lançado imediatamente). Benefícios: deteção precoce de erros, publicações mais frequentes e seguras, redução de retrabalho e feedback quase imediato sobre a qualidade do código. (Página 60-61)

52. **Pergunta:** Quais são os conceitos principais de controlo de versões e como funciona um fluxo de trabalho comum?
    Resposta: Os conceitos principais são: Commit (uma alteração registrada no código), Branch (uma linha paralela de desenvolvimento), Merge (combinação de alterações entre branches), Pull/Merge Request (proposta de integração) e Repositório (onde o código e histórico são armazenados). Um fluxo de trabalho comum segue estas etapas: 1) Criar uma branch para desenvolver uma nova funcionalidade; 2) Fazer commits progressivos; 3) Abrir um pull request para revisão; 4) Após aprovação, fazer merge com a branch principal. As ferramentas mais usadas são Git, GitHub, GitLab e Bitbucket. (Página 61-62)

53. **Pergunta:** Quais são as métricas importantes para monitorização de sistemas em produção?
    Resposta: As métricas importantes são: Disponibilidade (o sistema está acessível e funcional), Tempo de resposta (quanto tempo demora uma ação), Taxa de erro (quantos pedidos falham), Uso de recursos (CPU, memória, armazenamento) e Logs e eventos (o que está acontecendo no sistema). A monitorização é essencial para detectar problemas antes que afetem os utilizadores. (Página 62-63)

54. **Pergunta:** Quais são os tipos de manutenção em sistemas de software e como funciona o ciclo de melhoria contínua?
    Resposta: A manutenção pode ser: Corretiva (corrigir erros), Evolutiva (adicionar novas funcionalidades), Adaptativa (ajustar a mudanças externas) e Preventiva (melhorar a estrutura interna). O ciclo de melhoria contínua consiste em: 1) Monitorizar; 2) Detetar problemas; 3) Corrigir rapidamente; 4) Atualizar o sistema sem afetar os utilizadores. Ferramentas de monitoramento incluem New Relic, Datadog, ELK Stack e Grafana. (Página 63-64)

Capítulo 13 - Desenho de Interfaces Humanas

55. **Pergunta:** Quais são os princípios fundamentais de usabilidade?
    Resposta: Os princípios fundamentais são: Consistência (elementos devem comportar-se da mesma forma), Feedback imediato (o sistema deve reagir às ações), Minimizar a carga de memória (o utilizador não deve ter de lembrar informações), Controlo pelo utilizador (deve sentir-se no comando) e Prevenção de erros (melhor do que lidar com erros é evitá-los). As heurísticas de Nielsen incluem também visibilidade do estado do sistema, correspondência com o mundo real, entre outras. (Página 65-66)

56. **Pergunta:** Descreva as boas práticas para desenho de formulários e relatórios.
    Resposta: Formulários devem ser claros, organizados, com validação imediata, compatíveis com dispositivos móveis e com etiquetas explicativas (ex: em vez de "Submeter", usar "Guardar Dados do Cliente"). Relatórios devem usar gráficos quando apropriado, permitir filtros e ordenação, usar títulos claros e datas visíveis, e destacar valores críticos. Mesmo sistemas intuitivos precisam de sistemas de ajuda como tooltips, ajuda contextual, FAQs e guias passo a passo. (Página 66-67)

57. **Pergunta:** O que são diagramas de diálogo e qual sua importância no desenho de interfaces?
    Resposta: Diagramas de diálogo são representações visuais que mostram as interações possíveis entre o utilizador e o sistema. Incluem elementos como estado do sistema (tela ou ecrã), ação do utilizador (clique, preenchimento) e resposta do sistema (mensagens, transições). Formatos comuns incluem diagramas de fluxo, diagramas de estado e protótipos de baixa fidelidade. Estes diagramas ajudam a alinhar expectativas, prever erros de usabilidade e melhorar o design antes de implementá-lo. (Página 67-68)

Capítulo 14 - Implementação e Operação de Sistemas

58. **Pergunta:** Quais são as boas práticas de codificação para desenvolvimento de software confiável?
    Resposta: Boas práticas incluem: seguir padrões de codificação (nomes consistentes, indentação), usar comentários úteis sem excesso, escrever código modular (funções pequenas com responsabilidade única), reutilizar código para evitar duplicações e escrever testes automáticos desde o início. O objetivo é escrever código claro, eficiente, legível e fácil de manter. (Página 70)

59. **Pergunta:** Quais são os tipos de testes que devem ser integrados à codificação?
    Resposta: Durante o desenvolvimento, devem ser aplicados: Testes unitários (testam partes isoladas como funções), Testes de integração (verificam a interação entre componentes) e Testes de sistema (avaliam o comportamento global). Muitas equipas adotam TDD (Test-Driven Development), escrevendo testes antes do código propriamente dito. (Página 70-71)

60. **Pergunta:** Quais são os tipos essenciais de documentação no desenvolvimento de software?
    Resposta: Os tipos essenciais são: Documentação técnica (arquitetura, APIs, instalação, configuração), Documentação do utilizador (como usar o sistema, funcionalidades) e Documentação de manutenção (erros conhecidos, planos de atualização). Boas práticas incluem linguagem clara e direta, capturas de ecrã, tutoriais passo a passo e atualização contínua com as versões do software. (Página 71-72)

61. **Pergunta:** Como deve ser estruturado o suporte ao utilizador após a implementação de um sistema?
    Resposta: O suporte deve ajudar a resolver dúvidas e problemas através de canais como email, telefone, chat e sistemas de tickets. Deve incluir repositórios de conhecimento (FAQs, tutoriais, fóruns) e monitorização proativa. Um ciclo de feedback eficaz envolve: recolher dados (erros, pedidos), analisar padrões, priorizar correções, implementar atualizações e comunicar as mudanças aos utilizadores. (Página 73)

Capítulo 15 - Gestão de Projetos de Software

62. **Pergunta:** Explique os desafios da estimativa em projetos de software e as técnicas utilizadas.
    Resposta: Estimar o esforço necessário em projetos de software é um desafio, mas essencial. Técnicas incluem: Expert Judgment (baseado na experiência), Decomposição de tarefas (dividir em partes menores) e Modelos paramétricos (como COCOMO). É melhor ter uma estimativa aproximada e revista frequentemente do que uma "certeza" inflexível. (Página 75)

63. **Pergunta:** Como se deve abordar a gestão de riscos em projetos de software?
    Resposta: Riscos são eventos que podem afetar negativamente o projeto, como atrasos, saída de membros da equipa ou mudanças nos requisitos. Para gerir riscos deve-se: 1) Identificar os riscos; 2) Avaliar a probabilidade e impacto; 3) Definir estratégias de mitigação (evitar, transferir, aceitar ou reduzir). Uma gestão de riscos eficaz prepara o projeto para o imprevisto. (Página 76)

64. **Pergunta:** Explique os princípios W5HH de gestão eficaz segundo Barry Boehm.
    Resposta: Os princípios W5HH são sete perguntas essenciais: 1) Why - Qual é o objetivo do projeto? 2) What - Quais são as funcionalidades principais? 3) When - Qual é o calendário? 4) Who - Quem faz o quê? 5) Where - Como se distribuem as responsabilidades? 6) How - Qual será o processo adotado? 7) How much - Que recursos são necessários? Este conjunto de questões serve como bússola para qualquer gestor de projeto. (Página 77-78)

Capítulo 16 - Qualidade de Software

65. **Pergunta:** O que define a qualidade de software segundo o modelo ISO/IEC 25010?
    Resposta: Segundo este modelo, a qualidade de software é definida por características como: Funcionalidade (faz o que é suposto), Fiabilidade (continua funcionando sob diferentes condições), Usabilidade (é fácil de usar), Eficiência de desempenho (responde rápido e usa bem os recursos), Manutenibilidade (é fácil de alterar) e Portabilidade (funciona em diferentes ambientes). Estes critérios ajudam a definir objetivos claros de qualidade durante o desenvolvimento. (Página 80)

66. **Pergunta:** O que é Garantia da Qualidade (QA) e quais são as técnicas mais comuns?
    Resposta: Garantia da Qualidade é um conjunto de práticas que assegura que o software atinge o nível de qualidade desejado. Técnicas incluem: Revisões técnicas (programadores revêem o trabalho uns dos outros), Inspeções formais (análise detalhada de documentos com base em checklists), Auditorias (verificações independentes) e aplicação de Normas de qualidade (como ISO 9001, CMMI). O princípio é que feedback frequente e imediato ajuda a evitar erros mais tarde. (Página 81)

67. **Pergunta:** Explique os diferentes tipos de custos associados à qualidade de software.
    Resposta: Os custos incluem: Custos de prevenção (evitar erros, ex: formação), Custos de deteção (encontrar erros antes da entrega, ex: testes), Custos de falha interna (corrigir erros antes da entrega) e Custos de falha externa (erros que afetam clientes - os mais caros). A regra de ouro é: "Quanto mais cedo detetares o erro, mais barato será corrigi-lo." O equilíbrio envolve definir objetivos realistas, priorizar critérios críticos e avaliar o custo-benefício de cada medida. (Página 82-83)

Capítulo 17 - Melhoria de Processos e Tendências Futuras

68. **Pergunta:** O que é Melhoria de Processos de Software (SPI) e como funciona o ciclo de melhoria contínua?
    Resposta: SPI visa tornar o desenvolvimento mais eficiente, previsível e com qualidade. Reduz custos, diminui atrasos e aumenta a satisfação do cliente. O ciclo de melhoria contínua segue etapas iterativas: 1) Avaliar o estado atual; 2) Definir metas claras; 3) Implementar mudanças; 4) Medir resultados; 5) Repetir o ciclo. Este processo é inspirado no ciclo PDCA (Plan-Do-Check-Act). (Página 85)

69. **Pergunta:** Descreva os níveis de maturidade do CMMI (Capability Maturity Model Integration).
    Resposta: O CMMI tem 5 níveis de maturidade: 1) Inicial (caótico e imprevisível, depende de heróis individuais); 2) Gerido (gestão básica de projetos e requisitos); 3) Definido (processos padronizados em toda a organização); 4) Quantitativamente Gerido (métricas e análise estatística); 5) Otimização (melhoria contínua com base em inovação e dados). Subir nesta escada significa maior previsibilidade, menos erros e mais confiança dos clientes. (Página 86)

70. **Pergunta:** Quais são as tendências emergentes em desenvolvimento de software e por que a adaptabilidade é crucial?
    Resposta: Tendências incluem DevOps e automação total, Inteligência Artificial no apoio à programação, Low-code/no-code e ética no software. A adaptabilidade é crucial porque os sistemas são cada vez mais complexos (distribuídos, interdependentes, críticos). As equipas precisam colaborar de forma ágil, tomar decisões baseadas em dados, aprender rapidamente e adaptar processos ao contexto. Como disse Darwin, "não é o mais forte que sobrevive, mas o que melhor se adapta à mudança". (Página 86-87)