UNIVERSIDADE FEDERAL DE OURO PRETO – UFOP

CIÊNCIA DA COMPUTAÇÃO

ENGENHARIA DE SOFTWARE I

LISTA 1

Marcus Vinícius Souza Fernandes

19.1.4046

Ouro Preto

2021

1

O termo engenharia de software traz os conceitos de engenharia (criação, construção, análise, desenvolvimento e manutenção) para a produção de softwares com o objetivo de produzi-los de maneira econômica, que seja confiável e que trabalhe em máquinas reais.

Os fundamentos científicos envolvem o uso de modelos abstratos e precisos que permitem ao engenheiro especificar, projetar, implementar e manter sistemas de software, avaliando e garantindo sua qualidade. Além disso, deve oferecer mecanismos para se planejar e gerenciar o processo de desenvolvimento.

2

O PMBOK conceitua um projeto como um esforço temporário, ou seja, finito. Tem, portanto, início e fim bem determinados e empreendidos para se alcançar um objetivo exclusivo, ou seja, um resultado específico que o torna único.

Os projetos são executados por pessoas e com limitações de recursos e são planejados, executados e controlados ao longo de seu ciclo de vida. De forma simples, é possível afirmar que os projetos se diferenciam dos processos e das operações, porque não são contínuos e repetitivos pois possuem caráter único.

Para que se tenha uma dimensão melhor da importância dos projetos, basta compreender que, para que qualquer organização alcance seus objetivos, ela precisará de esforços organizados. Isso é válido desde a construção de uma nova fábrica até a ampliação de uma unidade operacional, por exemplo.

Ele pode ser melhor compreendido por meio dos processos que o compõem, organizados em cinco grupos:

- Iniciação;
- Planejamento;
- Execução;
- Monitoramento e controle;
- Encerramento.

3

A arquitetura de software serve como uma estrutura através da qual se tem o entendimento dos componentes de um sistema e de seus inter-relacionamentos. Ela consiste na definição dos componentes de software, suas propriedades externas, e seus relacionamentos

com outros softwares. O termo também se refere à documentação da arquitetura de software do sistema. Essa documentação facilita a comunicação entre os stakeholders, registra as decisões iniciais acerca do projeto de alto-nível e permite o reuso do projeto dos componentes e padrões entre projetos.

4

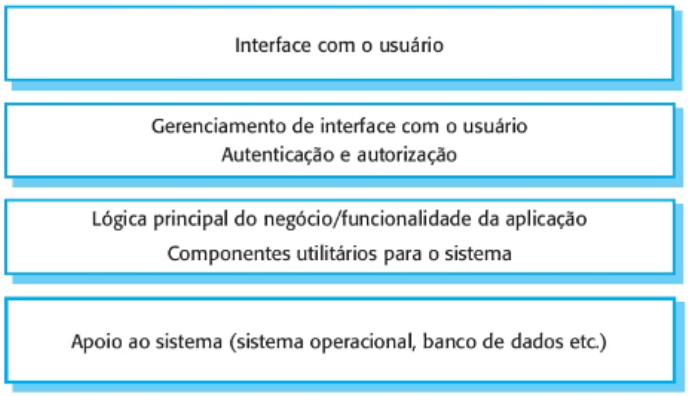
Componentes são unidades independentes de um software com o qual interagimos através de interfaces, estas são: Interface *Requires* e Interface *Provides (API)*.

Desenvolvimento baseado em componentes (DBC) é o desenvolvimento de um sistema com foco no desenvolvimento dos componentes independentes que irão compor o software quando interligados e que permitem uma fácil manutenção e alteração de parte do software.

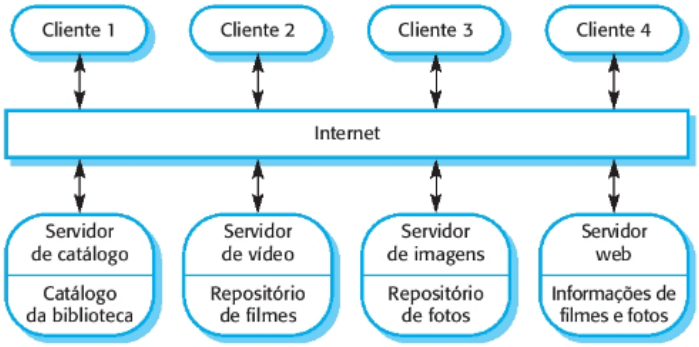
5

- Camadas: as funcionalidades do sistema estão organizadas em camadas, onde cada camada é implementada baseada no funcionamento das camadas inferiores, quanto mais internas forem as camadas, mais essencial é sua função, tendendo a ser utilizada em todo o sistema.
    - Para facilitar o entendimento, podemos olhar para o modelo de redes OSI, onde a primeira camada é a mais essencial e utilizada em toda comunicação com a rede.
    - Em suas vantagens, permite a substituição de camadas inteiras, contanto que a interface seja mantida. Recursos redundantes, como a autenticação, podem ser fornecidos em cada camada para aumentar a dependabilidade do sistema.
    - Já por outro lado, na prática, muitas vezes é difícil proporcionar uma separação clara entre as camadas, de modo que camadas dos níveis mais altos podem ter de interagir diretamente com as dos níveis mais baixos em vez das imediatamente inferiores a elas. O desempenho pode ser um problema por causa dos múltiplos níveis de interpretação de uma requisição de serviço à medida que essa requisição é processada em cada camada.

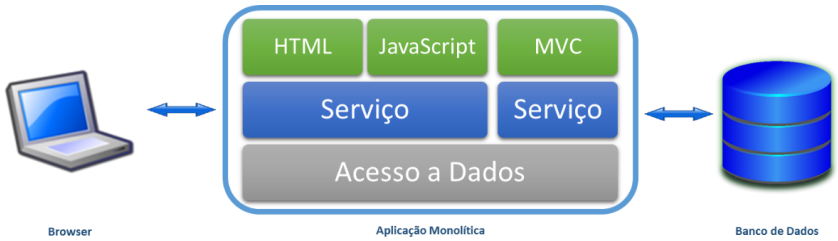
- Podemos ver um modelo geral de um sistema com arquitetura em camadas abaixo.



- Cliente-servidor: sistema organizado em um conjunto de servidores e serviços associados e também um cliente que acessa e usa os serviços. Geralmente são implementados como sistemas distribuídos.
  - Os servidores são componentes do software e geralmente podem ser executados da própria máquina.
  - Outro componente importante é uma rede que permite o acesso aos serviços pelo cliente.
  - A principal vantagem desse modelo é que os servidores podem ser distribuídos em rede. A funcionalidade geral (por exemplo, um serviço de impressão) pode estar disponível para todos os clientes e não precisa ser implementada por todos os serviços.
  - Já para as desvantagens, cada serviço é um único ponto de falha e, portanto, é suscetível a ataques de negação de serviço ou a falhas no servidor. O desempenho pode ser imprevisível porque depende da rede e também do sistema. Podem surgir problemas de gerenciamento se os servidores forem de propriedade de organizações diferentes.
  - Abaixo podemos ver um modelo de um sistema de uma biblioteca com arquitetura cliente-servidor, onde o cliente se comunica com o sistema através de uma interface.



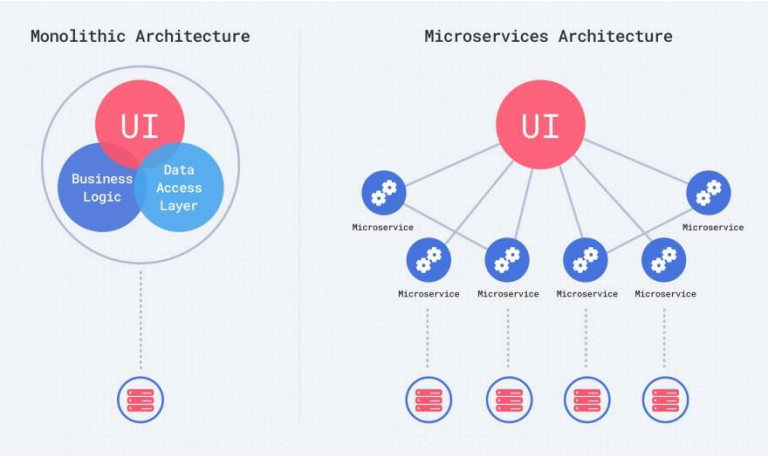
- Monolítica: é um sistema único, isto é, sem divisões de processos. Nesta organização, os componentes estão ligados a um único programa.
    - Algumas de suas vantagens incluem a facilidade de implementação devido ao fato de a organização ficar concentrada em um único sistema, o que facilita também os testes e deploy do sistema para o servidor.
    - Por outro lado, mesmo que seja mais fácil a implementação, testes e deploy, sua manutenção é mais complicada, por ser um único sistema interligado, facilitando erros que podem tornar o sistema inoperante. Sua alteração também é uma desvantagem, visto que é necessário refazer o deploy para o servidor de todo o sistema a cada alteração.



6

Arquitetura de Microsserviços projeta o sistema de forma a decompor as aplicações em propósitos, isto é, esta organização divide um monolito. Cada microserviço é responsável por uma funcionalidade no sistema, fazendo essa atividade com eficiência.

- Alguns princípios:
    - **Alta coesão**: Esse princípio indica que o serviço deve ter uma única responsabilidade no sistema.
    - **Autônomos**: Um microserviço deve ser autônomo, isto é, independente de outros serviços.
    - **Resiliência**: Ser capaz de validar os dados recebidos (mesmo que estes estejam corrompidos) e tratar a perda ou falha na comunicação com outro serviço da cadeia, sem quebrar o fluxo da aplicação.
    - **Observável**: Consultar o status de um sistema em tempo real.
    - **Automatização**: Automatizar algumas tarefas como envio de resultados de forma contínua.



7

Em um framework é definida a arquitetura de uma aplicação, como a divisão de classes e objetos, as responsabilidades chave de cada e como essas classes e objetos se relacionam. Dessa forma, é possível inferir que o framework lida com o reuso de design e controla o fluxo da aplicação. Enquanto que em uma biblioteca de funções são definidas funções em uma determinada linguagem de programação, para serem utilizadas em um programa, portanto uma biblioteca lida com o reuso de código e o programador que a utiliza define o fluxo da aplicação.

Assim a diferença entre um framework e uma biblioteca de funções se encontra na forma como lidam o fluxo de uma aplicação, em uma biblioteca o programador tem independência para controlá-lo, já em um framework, esse fica dependente das diretrizes do framework.

Alguns exemplos de softwares que utilizam a arquitetura de framework seriam: Facebook, Discord e Airbnb utilizam o framework React Native em suas aplicações mobile, enquanto isso temos as aplicações, por outro lado software como Google, Instagram e Wikipedia utilizam a biblioteca de funções JavaScript & jQuery em suas respectivas aplicações web.

Assim é possível concluir que em situações que necessitam de uma maior flexibilidade o ideal é utilizar a biblioteca de funções, visto que o programador tem maior controle, por outro lado em situações que necessitam treinar novos membros de equipe e criar aplicações em larga escala o ideal é utilizar um framework, devido a documentação para o treinamento e implementação prévia das normas de design.

8

API é um conjunto de definições e protocolos usados no desenvolvimento e na integração de software de aplicações. API é um acrônimo em inglês que significa *Application Programming Interface*.

As Interfaces de programação de aplicações simplificam a integração de novos componentes e serviços de uma aplicação a uma arquitetura preexistente, visto que nessa interface os vários elementos que compõem um software são abstraídos.

Em termos coloquiais a API seria um conjunto de regras que determina como o programador deve utilizar os serviços de uma plataforma ou software.

9

a) Fraco acoplamento: O fraco acoplamento pode ser definido como a baixa interdependência entre os componentes de um software.

b) Alta coesão: A alta coesão é definida como a capacidade de um componente dentro do software de realizar apenas um papel determinado pelo programador e de forma independente, ou seja, esse componente não realiza ou depende dos papéis desempenhados por outros componentes e não necessita de outros componentes para desempenhar sua função.

10

Separar as aplicações front-end e back-end (API) é uma grande vantagem, pois é importante para proteger o armazenamento de dados, possibilitando apenas a troca de informações, o processo de desenvolvimento da aplicação ocorre com mais facilidade, já que não há dificuldades para acoplar recursos uma vez que o código possui toda sua estrutura organizada e desacoplada de interfaces. A API permite que o banco de dados de diferentes servidores seja acessado pela aplicação, se tornando importante para o desenvolvimento em grandes aplicações. Portanto, sua utilização resulta em uma garantia de mais praticidade e confiabilidade.

11

Reuso de código é uma abordagem que procura reutilizar determinados trechos de código no contexto da aplicação. Essa prática promove muitos benefícios tais como: criação de códigos mais limpos e claros, redução significativa no volume do código, aumento na qualidade, produtividade no desenvolvimento, dentre várias outras. Algumas das desvantagens presentes neste processo se dão em manutenções em um determinado trecho/componente que está em grande uso na aplicação, isso implica em realizar ajustes e testes em todos os casos dependentes, outro ponto importante para ressaltar é que geralmente o processo de criação desses trechos são um pouco mais trabalhosos, pois devem ser mais genéricos a nível de contemplar vários casos de uso similares.

12

Fases do desenvolvimento de software:

1 - Fase de requisitos: Levantar os requisitos mínimos, estudar a viabilidade e definir o modelo a ser utilizado;

2 - Fase de projeto: Envolve atividades de concepção, especificação, design da interface, prototipação, design da arquitetura;

3 - Fase de implementação: Tradução para uma linguagem de programação das funcionalidades definidas durante as fases anteriores;

4 - Fase de testes: Realização de testes no que foi desenvolvido de acordo com os requisitos;

5 - Fase de produção: Implantação em produção do produto final;

13

A verificação de software pode ser realizada por uma equipe de desenvolvimento de software e consiste em analisar se o que foi implementado está correto.

A validação de software (também conhecida por *assurance* ou *quality assurance*) pode ser feita apenas pelos clientes e usuários e consiste em analisar se o software desenvolvido é aquele que se deseja, com o comportamento esperado e alta qualidade.

14

a) Teste unitário: testes de funções e métodos (menores unidades de código em linguagens imperativas e programação orientada a objetos, respectivamente), da maneira mais isolada possível, e analisando se, para um conjunto determinado de entradas, as funções ou métodos em questão retornam os valores corretos e esperados;

b) Teste funcional: testa se o uso coordenado dos métodos ou funções que compõem um determinado módulo ou classe atinge o objetivo determinado. Muitas vezes utilizado em módulos ou classes que possuem uma funcionalidade completa e cujas funções ou métodos não podem ser testados isoladamente (Ex.: API para a construção de um servidor, que envolve métodos de abrir e fechar uma conexão, enviar e receber dados, etc.);

c) Teste de integração: teste que cruza várias interfaces ou módulos, analisando a integração destes, e se estão comunicando de forma correta;

d/e) Teste sistêmico/Teste de aceitação: utiliza um programa ou sistema como um todo e analisa se ele atinge as necessidades do cliente.

15

a) Teste caixa branca: testes abertos em que se tem acesso ao código. Este tipo de teste é evitado pois, se um software já estiver funcionando na visão de um cliente, não existe razão para testes internos serem realizados, embora seja difícil implementar métodos e funções (nível mais baixo) sem saber se estão funcionando como o esperado;

b) Teste caixa preta: testes de software já empacotados e implantados no ambiente de produção (ou num ambiente que simula tal), em que não se tem acesso ao código. Este

tipo de teste é mais difícil de ser projetado e não se preocupa com a implementação. Existem ferramentas para realizar este tipo de teste, como o Appium (que gera testes para Android) e o Selenium (que gera testes para web);

c) Teste caixa cinza: nível intermediário entre o teste caixa branca e o teste caixa preta. Em testes funcionais, em que um módulo ou classe está sendo testado, não é necessário ter acesso à implementação, basta ter acesso à API ou interface e ser capaz de utilizar este componente. No entanto, ainda é necessário utilizar a mesma linguagem e tecnologia para esse tipo de teste.