Marcos Geraldo Braga Emiliano 19.1.4012

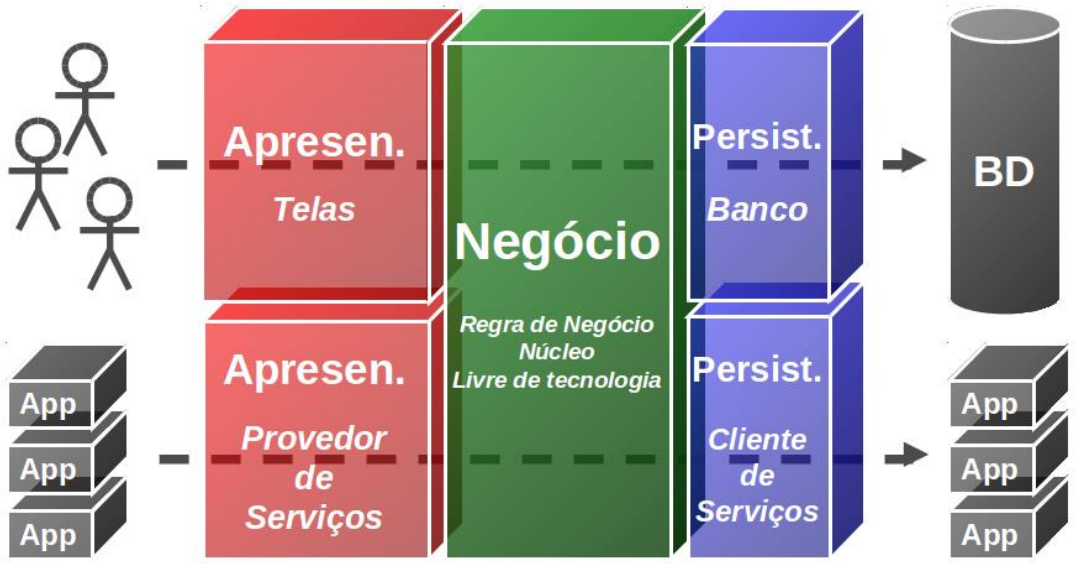
Lista 1

1)A Engenharia de Software é a área que tem como objetivo categorizar, aplicar, avaliar, entre outros, softwares em geral, com o intuito de torná-los impessoais, onde os más costumes dos programadores não se tornem um problema, é visado também tornar a manutenção mais simplificada.

2)Um projeto que possui inicio e fim bem determinados, tem como objetivo um fim exclusivo único. *"Um projeto é um esforço temporário empreendido para criar um produto, serviço ou resultado exclusivo. Os projetos e as operações diferem, principalmente, no fato de que os projetos são temporários e exclusivos, enquanto as operações são contínuas e repetitivas."*

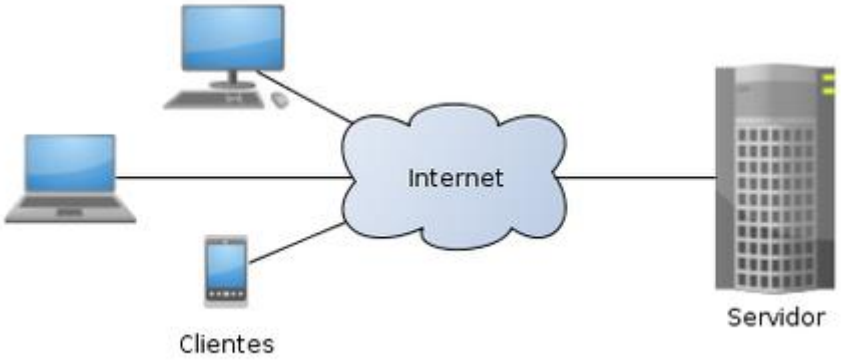
3) A arquitetura de software de um sistema é a forma como ele foi estruturado, padronizado, como é seu comportamento, como as funções são efetuadas por seus componentes, resumidamente é um modelo que pode ser usado para produzir algo.

4) Arquitetura em camadas visa a criação de aplicativos modulares, de forma que a camada mais alta se comunica com a camada mais baixa e assim por diante, fazendo com que uma camada seja dependente apenas da camada imediatamente abaixo.

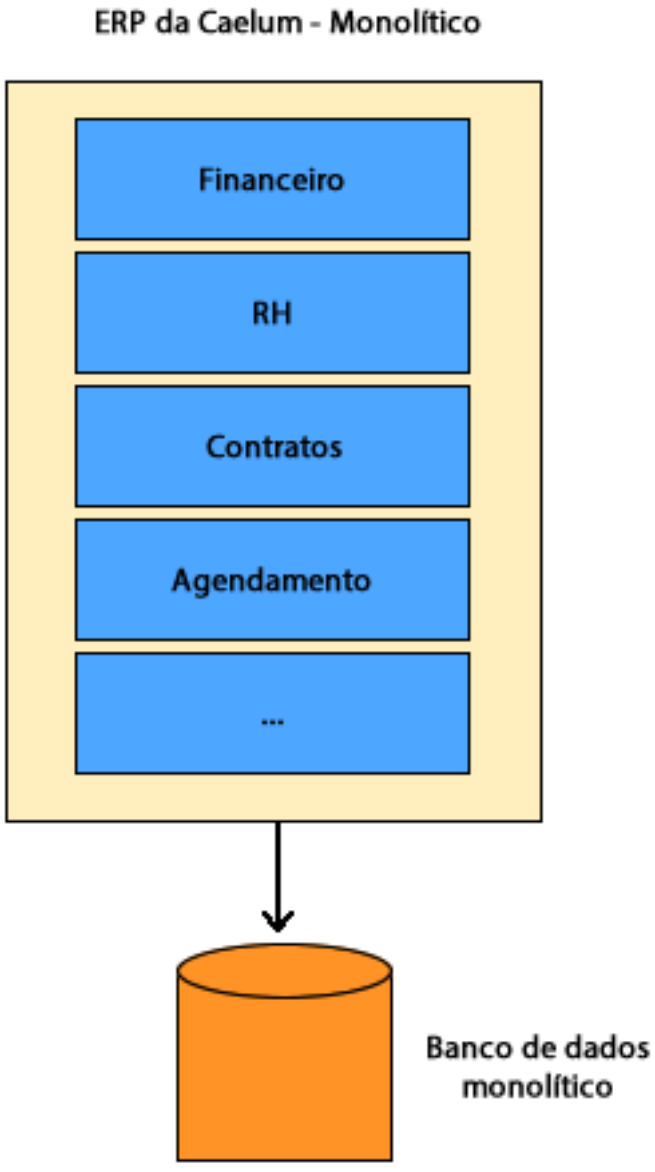


Arquitetura cliente-servidor ou modelo **cliente-servidor** é uma **arquitetura** na qual o processamento da informação é dividido em módulos ou processos distintos. Existe um processo que é responsável pela manutenção da informação (servidores) e outro responsável

pela obtenção dos dados (os clientes)



Arquitetura Monolítica é um sistema único, não dividido, que roda em um único processo, uma aplicação de software em que diferentes componentes estão ligados a um único programa dentro de uma única plataforma.



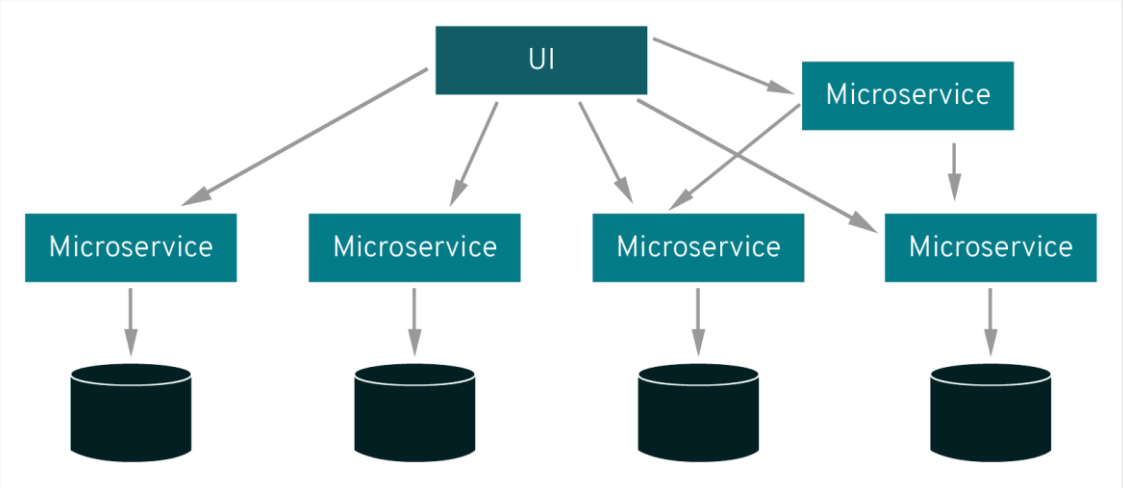
Vantagens e desvantagens:

Um sistema monolítico é mais fácil de entender, possui uma curva de aprendizado menor e, obviamente, existem mais desenvolvedores familiarizados com este modelo. Difícil de escalar: a medida que a aplicação vai crescendo a tendência é sempre duplicar a infraestrutura e colocar os servidores embaixo de um load balancer ou simplesmente aumentar o tamanho do servidor. Por ser construído como um único código, se algo quebrar todo o sistema fica indisponível. O ambiente de desenvolvimento fica muito mais simples quando a arquitetura é monolítica.

O modelo cliente/servidor é particularmente recomendado para redes que requerem um grande nível de fiabilidade com as seguintes vantagens: recursos centralizados, já que o servidor está no centro da rede, ele pode gerenciar recursos comuns a todos os usuários como, por exemplo, um banco de dados centralizado, a fim de evitar os problemas de redundância e contradição. Esse modelo oferece também maior segurança pois o número de pontos de entrada que permitem o acesso aos dados é menor, o gerenciamento do servidor e uma rede evolutiva. Graças a esta arquitetura, é possível remover ou adicionar clientes sem perturbar o funcionamento da rede e sem modificar o essencial. No entanto, a arquitetura cliente/servidor tem algumas lacunas, entre as quais o custo elevado devido ao tecnicismo do servidor e o elo fraco, isto é, o servidor é o único elo fraco da rede cliente/servidor, já que toda a rede está estruturada em função dele. Felizmente, o servidor tem uma grande tolerância a avarias, principalmente graças ao sistema RAID.

Separando em camadas, pode-se fazer o desenvolvimento de um sistema em etapas. Com isso, além da organização, ganha-se tempo e performance, desenvolvendo desta maneira, o software das camadas tende a ser mais limpos, facilitando o entendimento futuro do software, caso algum bug surgir, neste tipo de organização de software é mais fácil encontrá-lo e eliminá-lo, pois como as camadas são bem “isoladas” (focadas em funcionalidades específicas), fica relativamente simples descobrir em qual camada o bug está, organizando em camadas é possível substituir uma camada inteira por outra sem comprometer o sistema todo. Um exemplo disso é o próprio TCP/IP. Se for necessário trafegar os dados via Wi-Fi ou cabo (ethernet, por exemplo), basta alterar a camada mais especialista, já que as camadas acima comunicam-se com mensagens / métodos / funções que independem do canal de transmissão. Com isso, o código ganha versatilidade.

6)



Os microsserviços são uma arquitetura e uma abordagem para escrever programas de software. Com eles, as aplicações são desmembradas em componentes mínimos e independentes. Diferentemente da abordagem tradicional monolítica em que toda a aplicação é criada como um único bloco, os microsserviços são componentes separados que trabalham juntos para realizar as mesmas tarefas. Cada um dos componentes ou processos é um microsserviço. Essa abordagem de desenvolvimento de software valoriza a granularidade, a leveza e a capacidade de compartilhar processos semelhantes entre várias aplicações. Trata-se de um componente indispensável para a otimização do desenvolvimento de aplicações para um modelo nativo em nuvem.

7)A biblioteca é simplesmente um com junto de implementações com suas funcionalidades bem definidas e que podem ser importadas para o código e um framework é um modelo que pode ser posteriormente utilizado, sua principal função é servir como suporte. A biblioteca e preferível quando a função implementada por ela pode diretamente ser usada em seu código e o framework e preferível quando é exigido um nível maior de personalização, onde o reuso de código é mais difícil.

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 interface de programação de aplicações. Uma API permite que sua solução ou serviço se comunique com outros produtos e serviços sem precisar saber como eles foram implementados. Isso simplifica o desenvolvimento de aplicações, gerando economia de tempo e dinheiro. Ao desenvolver novas ferramentas e soluções (ou ao gerenciar aquelas já existentes), as APIs oferecem a flexibilidade necessária para simplificar o design, a administração e o uso, além de fornecer oportunidades de inovação.

9) Acoplamento fraco é aquele em que cada um dos seus componentes tem ou faz uso de pouco ou nenhum conhecimento das definições de outros componentes separados. As subáreas incluem o acoplamento de classes, interfaces, dados e serviços.[1] Acoplamento fraco é o oposto de acoplamento forte. E o conceito de alta coesão esta voltado para o conceito de encapsulamento, ou seja, uma alta coesão implica em uma alta especificação de operação, onde é avaliado um maior aproveitamento de código.

10)Por motivos de facilidade de manutenção, não a necessidade de fazer alterações na API, logo mantê-la juntamente com a implementação apenas iria deixar o código mais extenso.

11)O reuso de código é a pratica de programar sempre tomando o cuidado de tornar alguma função reutilizável por alguma outra aplicação, a principal vantagem e a economia de tempo onde não se torna necessário fazer novamente aquela “parte” do código e a principal desvantagem é o possível não acesso ao código fonte daquela função, logo não é possível fazer nenhuma manutenção ou modificação. O reuso de código é importante como ferramenta de economia de tempo, onde não será necessário implementar todo um segmento de código.

12) Fase de diagnóstico, levantamento e análise de requisitos, fase de desenvolvimento e etapa de implantação. Fase de Diagnóstico, nesta fase o time de desenvolvimento (ou o time comercial, no caso de muitas empresas) é responsável por conhecer o cliente a fundo. Dessa forma, é essencial que o problema seja extremamente detalhado e explicado para que o software atenda todas as necessidades. Além disso, quanto mais claro e nítidos estiverem as exigências do problema, mais completa e relevante será a solução.

Levantamento e Análise de requisitos, os requisitos podem ser lidos como sendo as “necessidades” do problema. Nesta fase de Levantamento e Análise de requisitos, deve-se pensar em alternativas de solução. Pois, as alternativas devem sempre se basear em um diagnóstico previamente feito na fase anterior (daí a importância de um diagnóstico bem feito).

Fase de Desenvolvimento, após essas fases, chegamos à etapa de Desenvolvimento. É aqui o momento em que o código será realmente desenvolvido, criado. Os grupos se organizam, as tarefas são organizadas, e os responsáveis iniciam o desenvolvimento do software. Neste momento, é muito comum ouvirmos falar sobre uma grande gama de “métodos ágeis” de desenvolvimento de software. Estes métodos seriam nada mais do que processos e etapas que, caso seguidas, agilizam e otimizam o desenvolvimento. E com isso, o produto final é entregado muito mais rápido e de maneira muito mais satisfatório.

Etapa de Implantação, e por fim, com o software em mãos e pronto para uso, iniciamos a fase de implantação. Nela, o código é instalado no ambiente do cliente (sistema operacional, servidor específico, entre outros). Além disso, são criados Manuais do Sistema, que auxiliam o entendimento do produto final. Estes materiais facilitam um possível treinamento para as pessoas que usarão futuramente o novo software

13) A Verificação é uma atividade, a qual envolve a análise de um sistema para certificar se este atende aos requisitos funcionais e não funcionais. Já a Validação, é a certificação de que o sistema atende as necessidades e expectativas do cliente. O processo de Validação e Verificação, não são processos separados e independentes.

14)

A) Testes de Unidade ou teste unitário é a fase de testes onde cada unidade do sistema é testada individualmente. O objetivo é isolar cada parte do sistema para garantir que elas estejam funcionando conforme especificado.

B) O teste funcional, ou de caixa-preta, é baseado nos requisitos funcionais do software. Esta técnica não está preocupada com o comportamento interno do sistema durante a execução do teste, mas sim com a saída gerada após a entrada dos dados especificados.

C)Teste de integração é a fase do teste de software em que módulos são combinados e testados em grupo. Ela sucede o teste de unidade, em que os módulos são testados

individualmente, e antecede o teste de sistema, em que o sistema completo é testado num ambiente que simula o ambiente de produção.

D) Na fase de teste de sistema, o objetivo é executar o sistema sob ponto de vista de seu usuário final, varrendo as funcionalidades em busca de falhas em relação aos objetivos originais. Os testes são executados em condições similares – de ambiente, interfaces sistêmicas e massas de dados – àquelas que um usuário utilizará no seu dia-a-dia de manipulação do sistema. De acordo com a política de uma organização, podem ser utilizadas condições reais de ambiente, interfaces sistêmicas e massas de dados.

E) Teste de aceitação é uma fase do processo de teste em que um teste de caixa-preta é realizado num sistema antes de sua disponibilização. Tem por função verificar o sistema em relação aos seus requisitos originais, e às necessidades atuais do usuário. É geralmente realizado por um grupo restrito de usuários finais, num ambiente parecido com o deles. Há três estratégias de implementação de testes de aceitação: a aceitação formal, a aceitação informal (ou teste alfa) e o teste beta.

15)

A) Também chamada de teste estrutural ou orientado à lógica, a técnica de caixa-branca avalia o comportamento interno do componente de software. Essa técnica trabalha diretamente sobre o código fonte do componente de software para avaliar aspectos tais como: teste de condição, teste de fluxo de dados, teste de ciclos, teste de caminhos lógicos, códigos nunca executados. Os aspectos avaliados nesta técnica de teste dependerão da complexidade e da tecnologia que determinarem a construção do componente de software, cabendo, portanto, avaliação de mais aspectos que os citados anteriormente. O testador tem acesso ao código fonte da aplicação e pode construir códigos para efetuar a ligação de bibliotecas e componentes. Este tipo de teste é desenvolvido analisando o código fonte e elaborando casos de teste que cubram todas as possibilidades do componente de software. Dessa maneira, todas as variações relevantes originadas por estruturas de condições são testadas.

B) Também chamada de teste funcional, teste comportamental, orientado a dado ou orientado a entrada e saída, a técnica de caixa-preta avalia o comportamento externo do componente de software, sem se considerar o comportamento interno do mesmo. Dados de entrada são fornecidos, o teste é executado e o resultado obtido é comparado a um resultado esperado previamente conhecido. Como detalhes de implementação não são considerados, os casos de teste são todos derivados da especificação. Quanto mais entradas são fornecidas, mais rico será o teste. Numa situação ideal todas as entradas possíveis seriam testadas, mas na ampla maioria dos casos isso é impossível. Outro problema é que a especificação pode estar ambígua em relação ao sistema produzido, e como resultado as entradas especificadas podem não ser as mesmas aceitas para o teste. Uma abordagem mais realista para o teste de caixa-preta é escolher um subconjunto de entradas que maximize a riqueza do teste. Pode-se agrupar subconjuntos de entradas possíveis que são processadas similarmente, de forma que testar somente um elemento desse subconjunto serve para averiguar a qualidade de todo o subconjunto.

C) A técnica de teste de caixa-cinza é uma mescla do uso das técnicas de caixa-preta e de caixa-branca. Esta técnica analisa a parte lógica mais a funcionalidade do sistema, fazendo uma comparação do que foi especificado com o que está sendo realizado. Usando esse método, o testador comunica-se com o desenvolvedor para entender melhor o sistema e otimizar os casos de teste que serão realizados. Isso envolve ter acesso a estruturas de dados e algoritmos

do componente a fim de desenvolver os casos de teste, que são executados como na técnica da caixa-preta. Manipular entradas de dados e formatar a saída não é considerado caixa-cinza pois a entrada e a saída estão claramente fora da caixa-preta. A caixa-cinza pode incluir também o uso de engenharia reversa para determinar por exemplo os limites superiores e inferiores das classes, além de mensagens de erro.