

MANUAL

DO TÉCNICO

Construção de Software!

Uma abordagem sobre a arquitetura na Engenharia de Software

VITOR CAETANO

Sumário

Arquitetura de Software	2
<i>O que fazer para construção da Arquitetura</i>	3
Projeto de Interface	3
Projeto de Dados	3
Projeto de componentes/classes/algoritmos	3
Princípios	3
Artefatos	3
<i>Arquitetura de Software e Valores de Negócio</i>	5
Comportamento x Estrutura/Arquitetura	5
Sobre os Problemas na Equipe	5
Sobre os Padrões Arquiteturais	5
Tipos de Padrões ou Estilos	6
Arquitetura em camadas	6
Cliente-Servidor	7
MVC (Model View Controller)	8
Camadas + MVC	9
Repositório	10
Duto e filtro	11
Serviços - SOA (Arquitetura orientada a serviços)	12
Arquitetura de Microsserviços	15
<i>Abordagem Tradicional de Desenvolvimento</i>	16
Metodologias Ágeis	16
Princípios de Projeto	16
Integridade Conceitual	16
Ocultamento de Informação	17
Coesão (alta)	17
Acoplamento (baixo)	17
SOLID	18
Extra: Demeter Principle	19
Projeto de Componentes e Classes	19
Decisões de Projeto	20
Padrões	21
Características do Padrão	21
Tipos de Padrões	22
Padrões GRASP (General Responsibility Assignment Software Patterns)	22
Padrões básicos	22
Padrões avançados	24

Arquitetura de Software

Todas as construções humanas precisam de um rascunho, um desenho, um esquema geral antes de se concretizarem. A construção de um software segue este mesmo princípio: é possível criar uma abstração do software para guiar o desenvolvimento para que os objetivos definidos sejam alcançados. A fase de construção dessas abstrações é chamada de Projeto.

Desenho de software, é mais que um esboço ou rascunho.

Exemplo de projeto de engenharia civil: planta baixa, hidráulica e elétrica.

- Modelos mais abstratos apresentam uma visão geral, mais alto nível, mais próxima da especificação: funcionalidades e regras de negócio.
- Modelos mais concretos apresentam detalhes para sua implantação, mais baixo nível, mais próximo da codificação: código que implementa funcionalidades.

Funcionalidades, algoritmos, arquivos, classes e componentes, instruções, linguagem e código.

O que fazer para construção da Arquitetura

1. Decompor o software em grandes blocos de funcionalidades:
 - Requisitos funcionais (códigos)
 - Requisitos não-funcionais (arquitetura)

Projeto de Interface

Pensar como a interação do usuário cumprirá seus objetivos.

Projeto de Dados

Como os dados são organizados e armazenados

Projeto de componentes/classes/algoritmos

Como os códigos permitirão a implementação do comportamento do sistema frente às entradas/estímulos dadas pelo usuário.

Princípios

Integridade Conceitual, ocultamento de informação, coesão (alta), acoplamento (baixo)

Artefatos

- Arquitetura ou até projeto: **Ordenar** e **organizar** o espaço para determinada **finalidade** visando determinada **intenção**. Relacionado à organização e estrutura em um alto nível de abstração.
- Envolve:
 - funcionalidade
 - usabilidade
 - escalabilidade (resiliência)
 - performance

- reusabilidade
- compreensividade
- restrições tecnológicas e econômicas
- trade-offs
- estética
- Componentes, classes e algoritmos,
- Dados
- Interfaces

Design bom: Se o esforço necessário para satisfazer as demandas do cliente for baixo e se mantiver assim ao longo da vida do sistema. Os esforços aumentam a cada release, caso contrário. Sua significância é medida pelo custo da mudança. Um conjunto de decisões que deveriam ter sido tomadas logo no início do projeto.

- Esforço para gerir o caos = baixa produtividade
- Seguir rápido é seguir bem, de forma organizada e prudente e antecipada.

O software foi inventado para ser um “produto” “suave”, de forma que ele possa ser facilmente alterado, diferente do hardware.

A falta de atenção ao valor agregado pela estrutura do software é que faz com que o custo de desenvolvimento de certos produtos seja exorbitante e cresça ao longo do tempo.

A impressão que se tem é que, a cada nova release ou solicitação de mudança, as equipes recebem peças de um quebra-cabeça que ficam cada vez mais difíceis de encaixar. Ou, talvez, uma analogia mais interessante seja o jogo do Tetris

Arquitetura de Software e Valores de Negócio

Comportamento x Estrutura/Arquitetura

- Um programa que funciona perfeitamente, mas seja impossível (ou muito difícil de mudar), não funcionará quando as exigências mudam, o programa será inútil.
- Um programa que não funciona, mas seja fácil de mudar, posso fazê-lo funcionar, e posso mantê-lo funcionando à medida que as exigências mudam. Portanto, o programa permanece continuamente útil.
- É claro que um programa que não funciona como não deveria é inútil em um certo momento do tempo, e pode representar uma oportunidade de negócio perdida ou o pioneirismo em um nicho de mercado.
- Entretanto, considerando todo o ciclo de vida de um software, é preciso entender que software funcionando, apesar de ser medida primária de progresso, precisa estar em uma estrutura que permita sua evolução de forma sustentável

Sobre os Problemas na Equipe

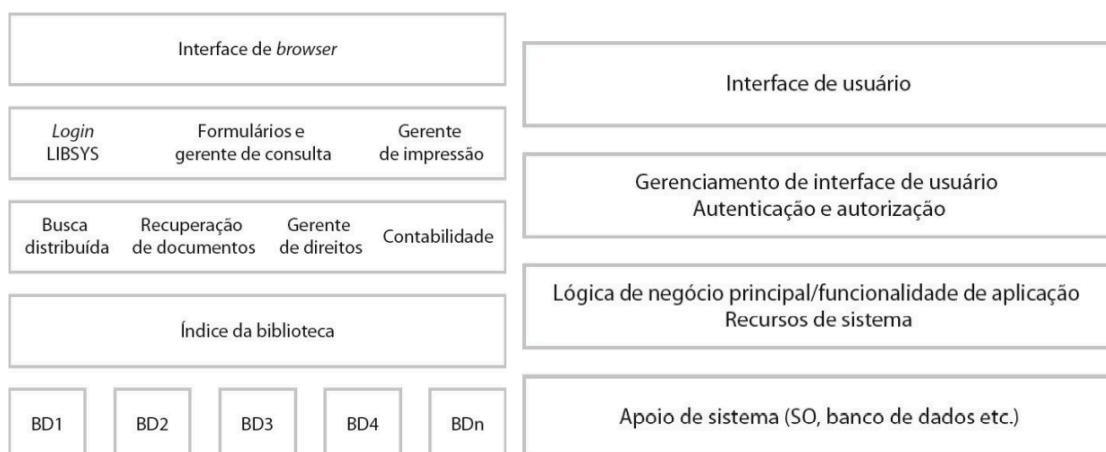
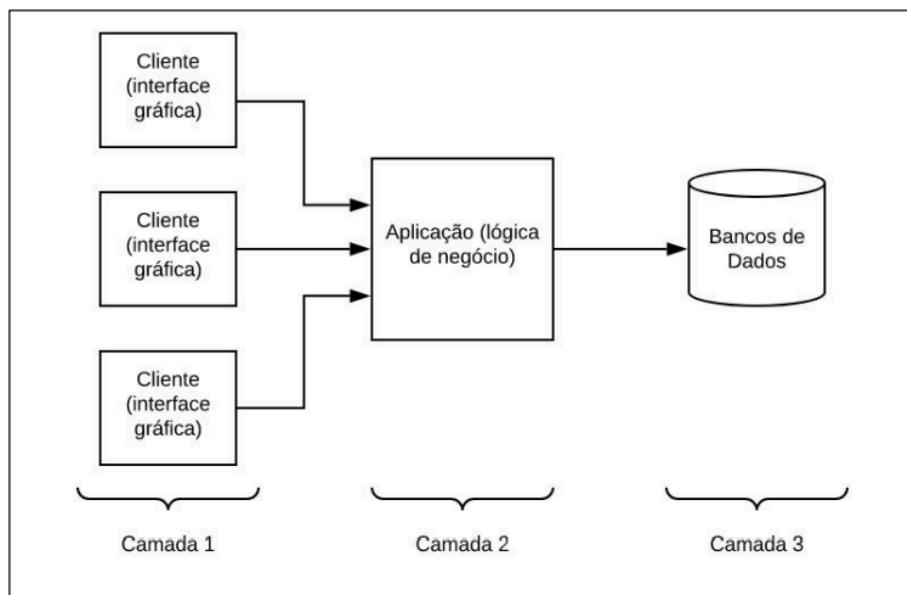
- As gerências não são capazes de avaliar a importância da arquitetura.
- Vira responsabilidade da equipe de desenvolvimento a organização de baixo e alto nível
- Implica Tempo para cuidar do comportamento e da estrutura do produto

Sobre os Padrões Arquiteturais

- Princípios e regras que formam a base do software que refletem conceitos e princípios comuns
- essência da organização dos elementos
- meios de representar, partilhar e reusar conhecimento
- descrição estilizada das boas práticas de projeto
- define um template pronto que soluciona problemas arquiteturais recorrentes
- expressa um esquema fundamental de organização para sistemas de softwares
- conjunto de subsistemas pré-definidos
- especifica suas responsabilidades, regras, diretrizes e organização

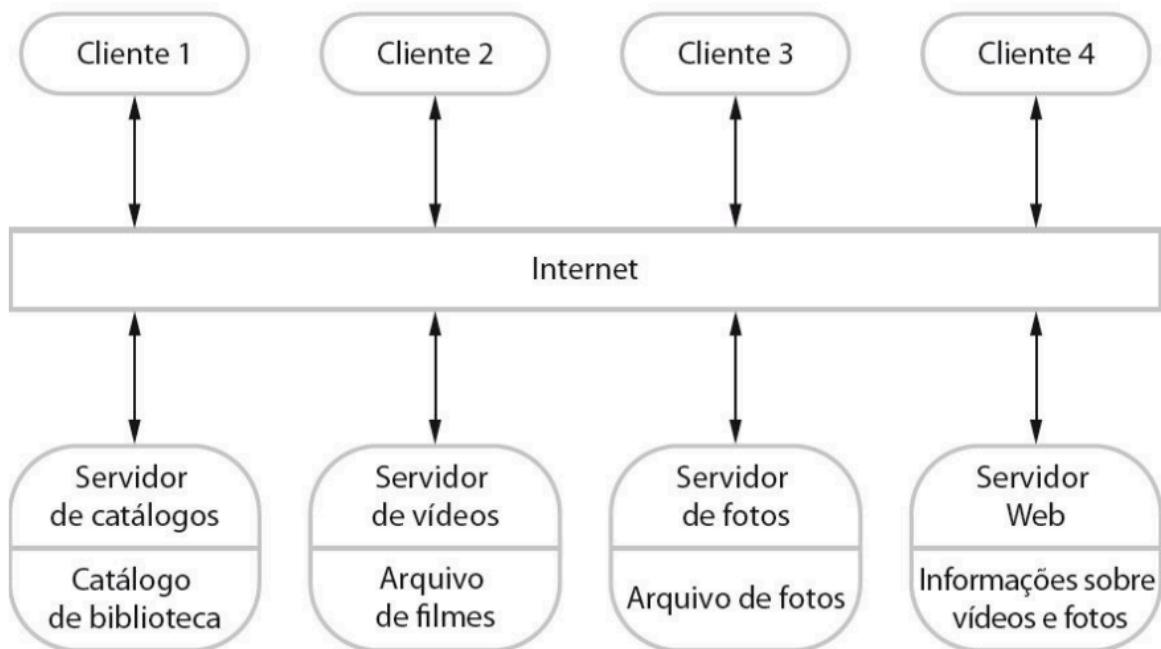
Tipos de Padrões ou Estilos

Arquitetura em camadas



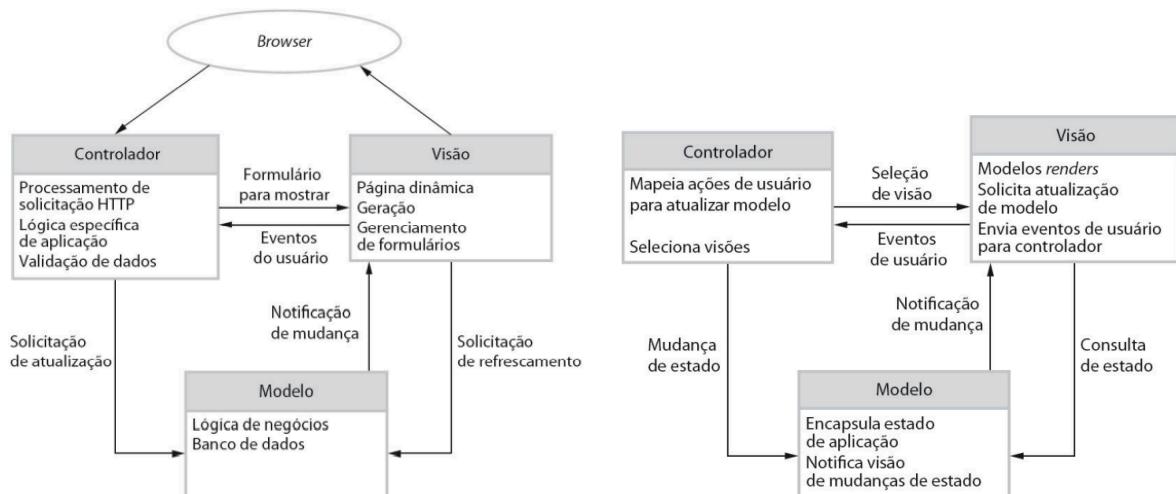
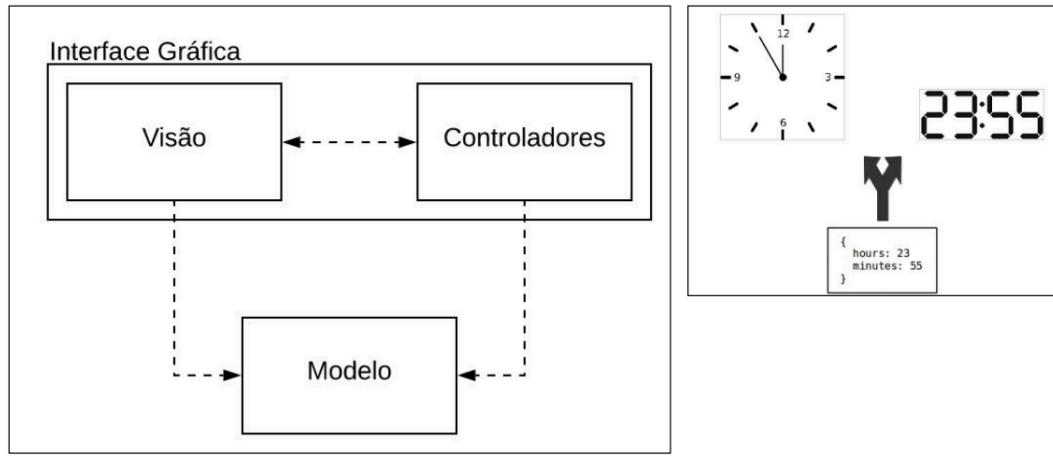
Nome	Arquitetura em camadas
Descrição	Organiza o sistema em camadas com a funcionalidade relacionada associada a cada camada. Uma camada fornece serviços à camada acima dela; assim, os níveis mais baixos de camadas representam os principais serviços suscetíveis de serem usados em todo o sistema. Veja a Figura 6.4.
Exemplo	Um modelo em camadas de um sistema para compartilhar documentos com direitos autorais, em bibliotecas diferentes, como mostrado na Figura 6.5.
Quando é usado	É usado na construção de novos recursos em cima de sistemas existentes; quando o desenvolvimento está espalhado por várias equipes, com a responsabilidade de cada equipe em uma camada de funcionalidade; quando há um requisito de proteção multinível.
Vantagens	Desde que a interface seja mantida, permite a substituição de camadas inteiras. Recursos redundantes (por exemplo, autenticação) podem ser fornecidos em cada camada para aumentar a confiança do sistema.
Desvantagens	Na prática, costuma ser difícil proporcionar uma clara separação entre as camadas, e uma camada de alto nível pode ter de interagir diretamente com camadas de baixo nível, em vez de através da camada imediatamente abaixo dela. O desempenho pode ser um problema por causa dos múltiplos níveis de interpretação de uma solicitação de serviço, uma vez que são processados em cada camada.

Cliente-Servidor

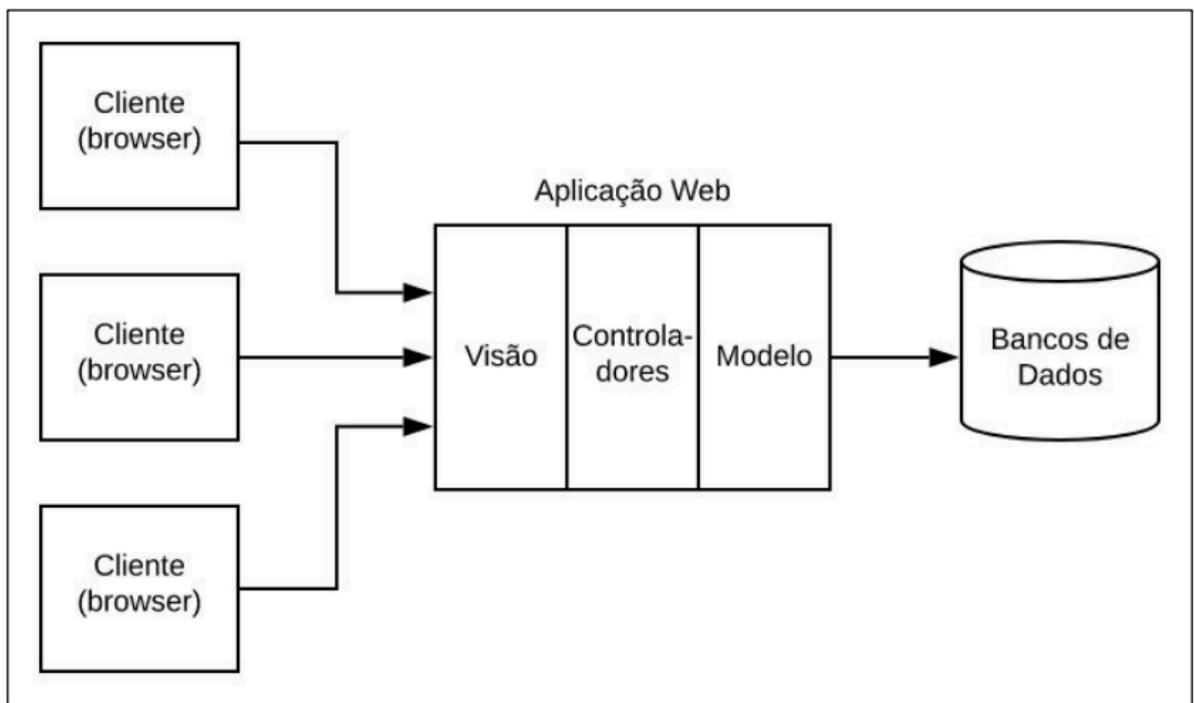


Nome	Cliente-servidor
Descrição	Em uma arquitetura cliente-servidor, a funcionalidade do sistema está organizada em serviços — cada serviço é prestado por um servidor. Os clientes são os usuários desses serviços e acessam os servidores para fazer uso deles.
Exemplo	A Figura 6.7 é um exemplo de uma biblioteca de filmes e vídeos/DVDs, organizados como um sistema cliente-servidor.
Quando é usado	É usado quando os dados em um banco de dados compartilhado precisam ser acessados a partir de uma série de locais. Como os servidores podem ser replicados, também pode ser usado quando a carga em um sistema é variável.
Vantagens	A principal vantagem desse modelo é que os servidores podem ser distribuídos através de uma 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.
Desvantagens	Cada serviço é um ponto único de falha suscetível a ataques de negação de serviço ou de falha do servidor. O desempenho, bem como o sistema, pode ser imprevisível, pois depende da rede. Pode haver problemas de gerenciamento se os servidores forem propriedade de diferentes organizações.

MVC (Model View Controller)



Camadas + MVC

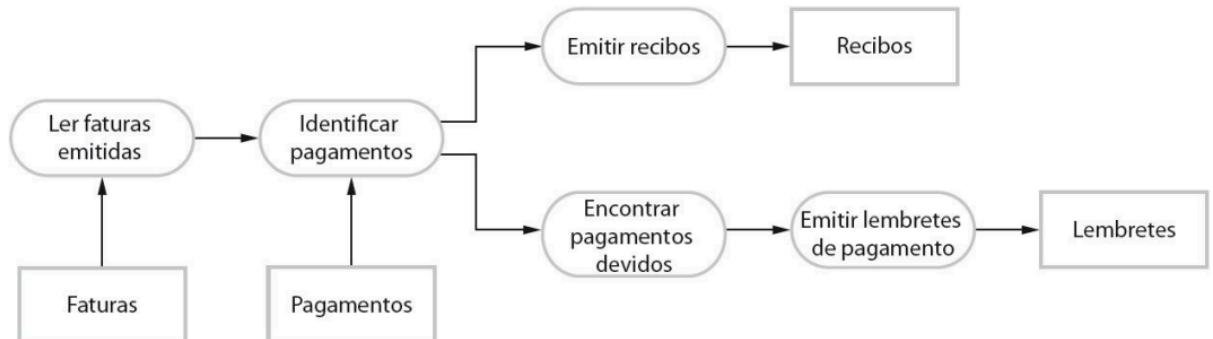


Repositório



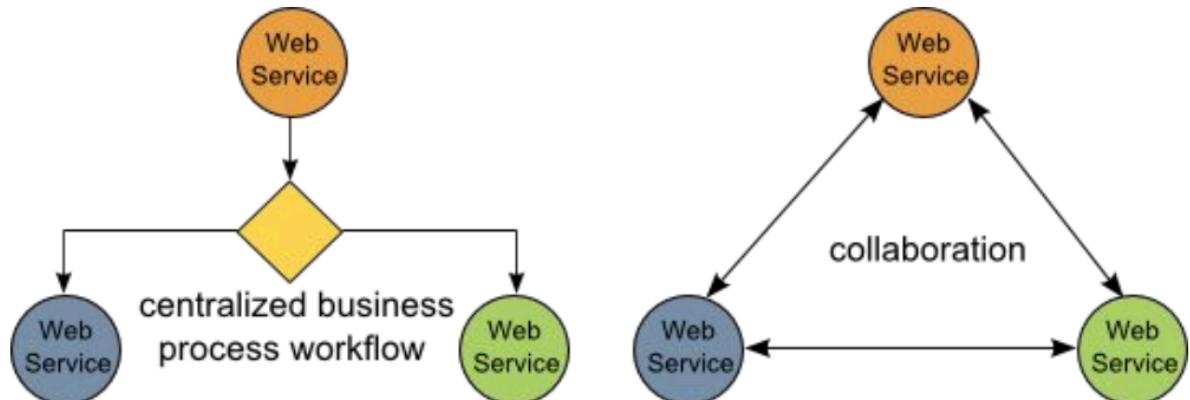
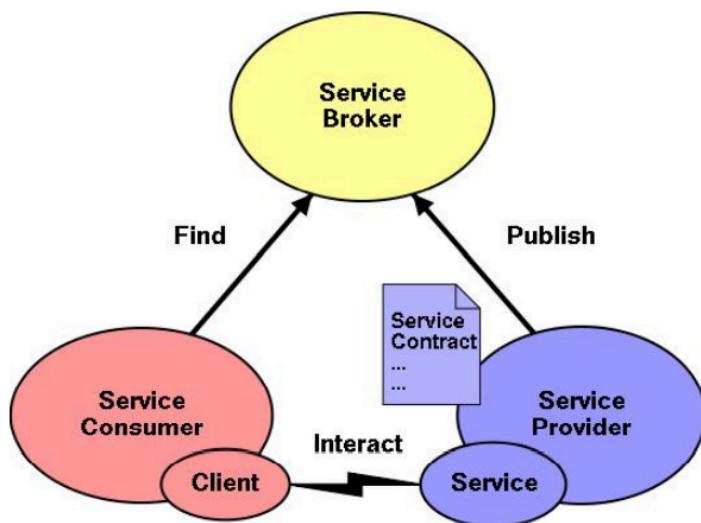
Nome	Repositório
Descrição	Todos os dados em um sistema são gerenciados em um repositório central, acessível a todos os componentes do sistema. Os componentes não interagem diretamente, apenas por meio do repositório.
Exemplo	A Figura 6.6 é um exemplo de um IDE em que os componentes usam um repositório de informações sobre projetos de sistema. Cada ferramenta de software gera informações que ficam disponíveis para uso por outras ferramentas.
Quando é usado	Você deve usar esse padrão quando tem um sistema no qual grandes volumes de informações são gerados e precisam ser armazenados por um longo tempo. Você também pode usá-lo em sistemas dirigidos a dados, nos quais a inclusão dos dados no repositório dispara uma ação ou ferramenta.
Vantagens	Os componentes podem ser independentes — eles não precisam saber da existência de outros componentes. As alterações feitas a um componente podem propagar-se para todos os outros. Todos os dados podem ser gerenciados de forma consistente (por exemplo, <i>backups</i> feitos ao mesmo tempo), pois tudo está em um só lugar.
Desvantagens	O repositório é um ponto único de falha, assim, problemas no repositório podem afetar todo o sistema. Pode haver ineficiências na organização de toda a comunicação através do repositório. Distribuir o repositório através de vários computadores pode ser difícil.

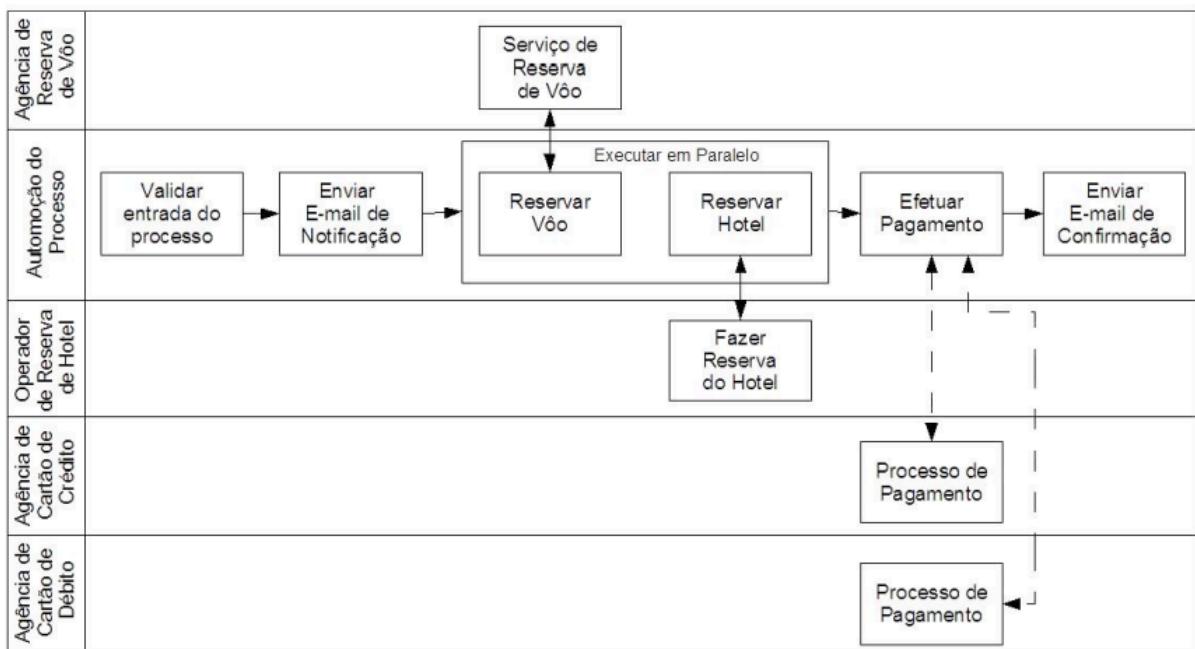
Duto e filtro



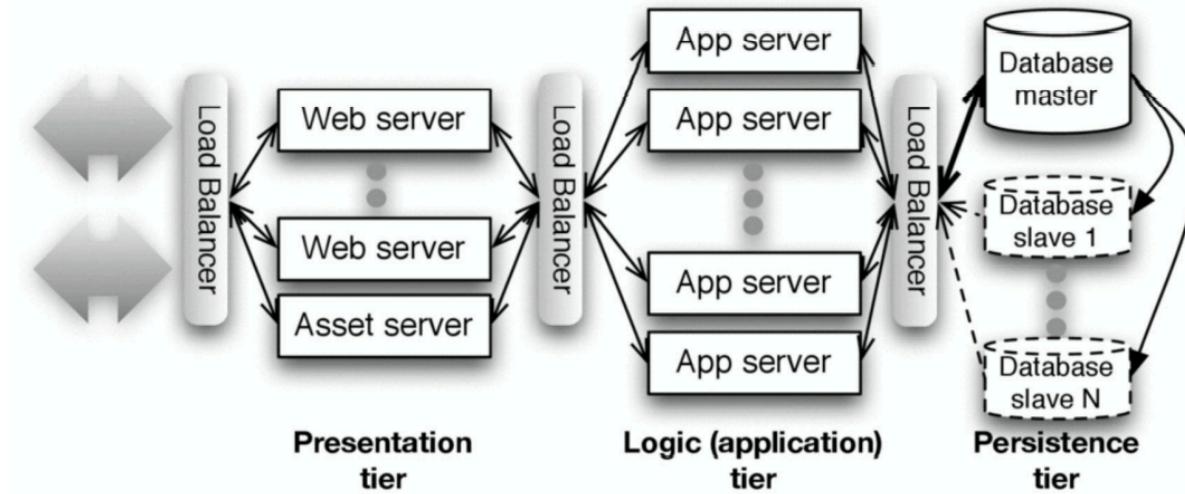
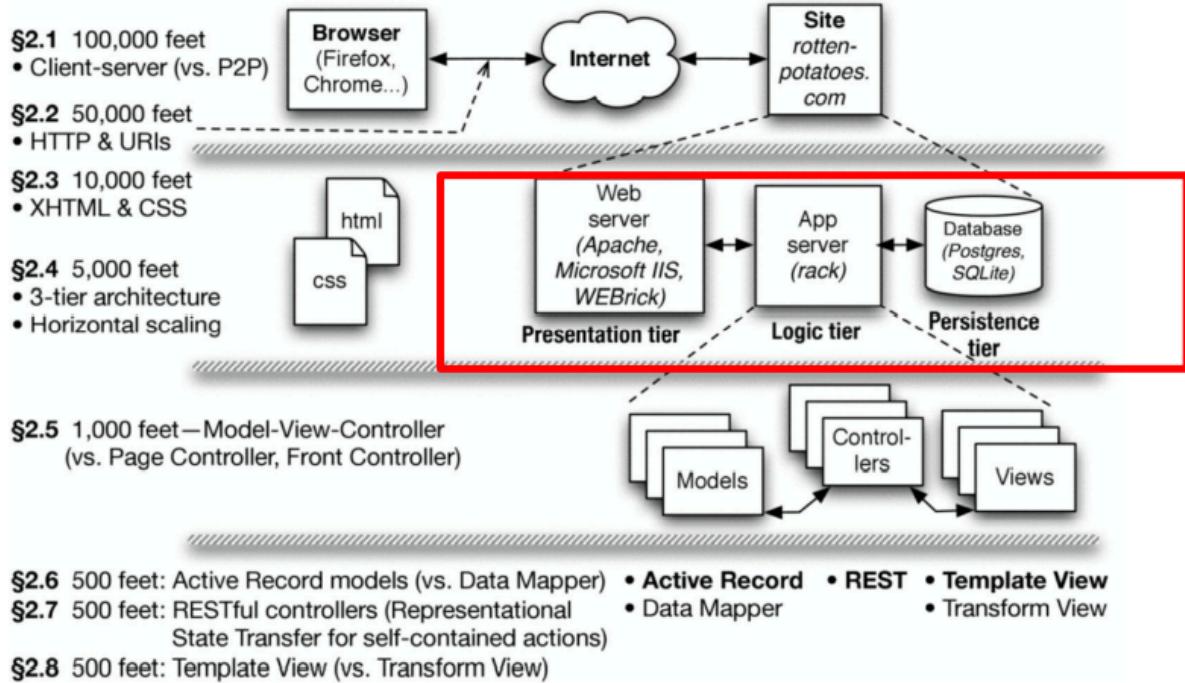
Serviços - SOA (Arquitetura orientada a serviços)

- O serviço é o bloco de construção principal de um software orientado a serviços
- Serviços são módulos independentes e auto contidos que oferecem funcionalidades de negócio específicas.
- Os serviços são descritos de forma padronizada, possuem interfaces publicadas e se comunicam com outros serviços por meio de invocações remotas.

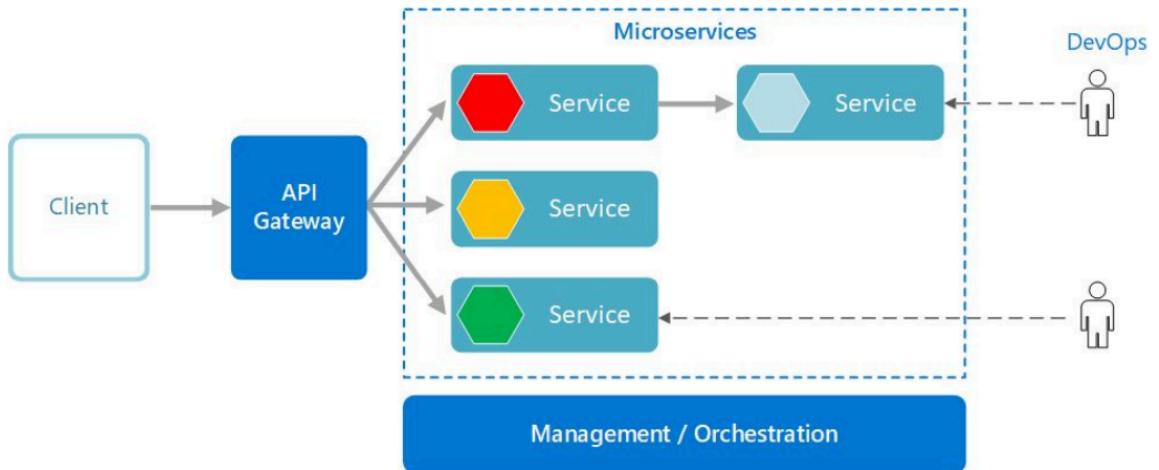




Exemplo de Arquitetura de uma aplicação SaaS



Arquitetura de Microsserviços



Benefícios

- Agilidade
- Equipes pequenas e focadas
- Base de código pequena
- Combinação de tecnologias
- Isolamento de falha
- Escalabilidade
- Isolamento de dados

Desafios

- Complexidade (combinação de serviços)
- Desenvolvimento e teste (dependências)
- Falta de governança (falta de padronização)
- Latência e congestionamento de rede
- Integridade de dados (cada serviço persiste e mantém seus próprios dados)
- Gerenciamento
- Controle de versão
- Conjunto de qualificações da equipe

Abordagem Tradicional de Desenvolvimento

- Nos métodos tradicionais, o projeto da arquitetura é realizado quando os requisitos estão bem claros: up front architectural design
- Define-se antecipadamente quais componentes do sistema serão criados e como eles vão interagir entre si e com os componentes/outros sistemas que já existem.
- A arquitetura fornece o modelo sob qual o software sera desenvolvido

Metodologias Ágeis

- Ágeis : o projeto da arquitetura é mais informal e não requer tanto rigor em sua especificação e documentação inicial.
- Como o software está em constante mudança, a arquitetura deve evoluir e amadurecer conforme os desenvolvedores entendem melhor o sistema
- Comece a pensar na arquitetura assim que o backlog do produto estiver pronto
- Faça o esboço de uma arquitetura inicial para guiar o desenvolvimento das primeiras funcionalidades
- Quantas Camadas? Tecnologias? Classes e componentes e subdivisões? Qual o Fluxo de Informações?
- A cada sprint feedbacks sobre a arquitetura, revisões, modificações e familiarização da equipe com o produto.

Princípios de Projeto

Integridade Conceitual

Um sistema não pode ter partes distintas com padrões e regras distintas para sua organização

Ocultamento de Informação

Vantagens:

Desenvolvimento em paralelo. Quando essas classes ocultam suas principais informações, fica mais fácil implementá-las em paralelo, por desenvolvedores diferentes

Flexibilidade a mudanças. Quando detalhes de implementação de Ci são ocultados do resto do sistema, fica mais fácil trocar sua implementação por uma classe Ci', que use estruturas de dados e algoritmos mais eficientes.

Facilidade de entendimento. Um novo desenvolvedor contratado pela empresa pode ser alocado para trabalhar em algumas classes apenas

Coesão (alta)

A implementação de qualquer classe deve ser coesa, isto é, toda classe deve implementar uma única funcionalidade ou serviço. Toda classe deve ter uma única responsabilidade no sistema. Uma classe deve implementar apenas um interesse.

Vantagens:

Facilita a implementação de uma classe, bem como o seu entendimento e manutenção.

Facilita a alocação de um único responsável por manter uma classe.

Facilita o reúso e teste de uma classe, pois é mais simples reusar e testar uma classe coesa do que uma classe com várias responsabilidades.

Acoplamento (baixo)

É a força de conexão entre duas classes. Uma classe não impacta fortemente as mudanças em outra classe. Dependência estáveis indicam um acoplamento bom ou aceitável.

SOLID

- Single Responsibility Principle (Princípio da Responsabilidade Única)

Um módulo deve ter uma, e apenas uma, razão para mudar

Um módulo deve ser responsável por um, e apenas um, ator

Sintoma 1 - Duplicação acidental

Sintoma 2: Fusões (Merges)

- Open/Closed Principle (Princípio do Aberto/Fechado)

Um artefato de software deve ser aberto para extensão, mas fechado para modificação

Para atingir este objetivo, particionamos o sistema em componentes e organizamos esses componentes em uma hierarquia de dependência que proteja os componentes de nível mais alto das mudanças em componentes de nível mais baixo

- Liskov Substitution Principle (Princípio da Substituição de Liskov)

Para criar sistemas de software a partir de partes intercambiáveis, essas partes devem aderir a um contrato que permita que elas sejam substituídas umas pelas outras

- Interface Segregation Principle (Princípio da Segregação da Interface)

Não se deve depender de coisas que não são usadas

- Dependency Inversion Principle (Princípio da Inversão de Dependência)

Módulos de alto nível não devem depender de módulos de baixo nível e ambos devem depender de abstrações. Abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações.

Abstrações estáveis

- Em uma interface abstrata, toda mudança corresponde a uma mudança em

susas implementações concretas

- As mudanças nas implementações concretas normalmente ou nem sempre

requerem mudanças nas interfaces que implementam

- Existe um esforço para definir interfaces que pouco mudam
- As interfaces são menos voláteis que as implementações
- Não se refira a classes concretas voláteis
- Não derive de classes concretas voláteis
- Não sobrescreva funções concretas
- Nunca mencione o nome de algo que seja concreto e volátil

Extra: Demeter Principle

Lei de Demétrio: “Converse com seus amigos – não fique íntimo de estranhos”

Um método pode chamar métodos de sua própria classe e métodos das classes de seus atributos, mas não de outras classes

Projeto de Componentes e Classes

O projeto de componentes, classes e algoritmos especificam a estrutura e o comportamento do software em um nível mais detalhado do que a arquitetura do software

- Responsabilidades (quem faz o quê)
- Estruturas (elementos que constituem o software de acordo com a decomposição escolhida e como eles se relacionam)
- Comportamento

Exemplo:

The screenshot shows the 'Create account' form from the IMDb website. It includes fields for 'Your name', 'Email', 'Password' (with a note 'at least 8 characters'), 'Re-enter password', and a 'Create your IMDb account' button. There is also a link 'Already have an account? [Sign-in](#)'.

Objetivo desta funcionalidade:

- Captar os dados do novo usuário para a criação de uma nova conta
- Processar os dados recebidos
- Armazenar os dados em um banco de dados para que o usuário possa se autenticar no sistema

O que determina o sucesso desta funcionalidade:

- Os dados do novo usuário estão cadastrados no banco de dados

Decisões de Projeto

Os impactos das decisões de projeto durante o desenvolvimento podem não ser fáceis perceber no início do projeto

Com o tempo, um projeto de má qualidade fará o software se deteriorar porque vai ficar mais difícil ler, entender, testar, corrigir e evoluir

É comum que, no início, o projeto não seja ótimo. E não há problemas nisso, afinal, “o ótimo é inimigo do bom”. Mas não pode ser uma bagunça. Com o tempo, o software vai evoluir e o que era um bom projeto passa a ser insuficiente diante das modificações impostas ao software:

“Cada padrão descreve um problema que ocorre repetidamente de novo e de novo em nosso ambiente, e então descreve a parte central da solução para aquele problema de uma forma que você pode usar esta solução um milhão de vezes, sem nunca implementá-la duas vezes da mesma forma.”

Padrões

- Encapsulamento: um padrão encapsula um problema ou solução bem definida.
- Generalidade: deve permitir a construção de outras realizações a partir deste padrão.
- Abstração: representam abstrações da experiência empírica ou do conhecimento cotidiano.
- Abertura: deve permitir a sua extensão para níveis mais baixos de detalhe.
- Combinatoriedade: os padrões são relacionados hierarquicamente.

Padrões de alto nível podem ser compostos ou relacionados com padrões que endereçam problemas de nível mais baixo.

Características do Padrão

- Nome: uma descrição da solução, mais do que do problema ou do contexto.
- Exemplo: uma ou mais figuras, diagramas ou descrições que ilustrem um protótipo de aplicação.
- Contexto: a descrição das situações sob as quais o padrão se aplica.
- Problema: uma descrição das forças e restrições envolvidas e como elas interagem.
- Solução: relacionamentos estáticos e regras dinâmicas descrevendo como construir artefatos de acordo com o padrão, frequentemente citando variações e formas de ajustar a solução segundo as circunstâncias. Inclui referências a outras soluções e o relacionamento com outros padrões de nível mais baixo ou mais alto.

Vantagens:

- Padrões reduzem a complexidade da solução
- Padrões promovem o reuso
- Padrões facilitam a geração de alternativas
- Padrões facilitam a comunicação

Tipos de Padrões

Padrões GRASP (General Responsibility Assignment Software Patterns)

Os padrões GRASP fornecem uma abordagem sistemática para a atribuição de responsabilidades às classes do projeto

Responsabilidades

- De conhecimento: sobre dados privativos e encapsulados; sobre objetos relacionados; sobre coisas que pode calcular ou derivar.
- De realização: fazer alguma coisa em si mesmo; iniciar uma ação em outro objeto; controlar e coordenar atividades em outros objetos

Padrões básicos

- Criador (Creator) : Quem deve ser responsável por criar uma nova instância de uma classe?
- Especialista (Information Expert): Precisa-se de um princípio geral para atribuir responsabilidades a objetos. classe que possui a informação necessária para cumpri-la; Qual é o princípio mais básico de atribuição de responsabilidades a objetos ? Atribuir responsabilidade ao especialista da informação.

Benefícios:

- Mantém encapsulamento e favorece o acoplamento fraco
- O comportamento fica distribuído entre as classes que têm a informação necessária (classes “leves”) e favorece alta coesão
- Favorece o reuso

- Alta coesão (High Cohesion)

Como manter os objetos focados, comprehensíveis, gerenciáveis e, em consequência, com Baixo Acoplamento? Atribua responsabilidades de modo que a coesão da classe permaneça alta. Quem deve ser o responsável por lidar com um evento de uma interface de entrada? Atribuir a responsabilidade de receber ou lidar com um evento do sistema para uma classe que representa todo o sistema (controlador de fachada – front controller), um subsistema e um cenário de casos de uso (controlador de caso de uso ou sessão)

- Baixo acoplamento (Low Coupling) : Como prover baixa dependência entre classes, reduzir o impacto de mudanças e obter alta reutilização? Atribua as responsabilidades de modo que o acoplamento entre classes permaneça baixo.

- Controlador (Controller): Quem deve ser o responsável por lidar com um evento de uma interface de entrada? Atribuir a responsabilidade de receber ou lidar com um evento do sistema para uma classe que representa todo o sistema (controlador de fachada – front controller), um subsistema e um cenário de casos de uso (controlador de caso de uso ou sessão)

Sinais de Mau Projeto: falta de foco e tratamento de muitas responsabilidades (coesão baixa)

- uma única classe controladora tratando todos os eventos, que são muitos.
- o próprio controlador executa as tarefas necessárias para atender o evento, sem delegar para outras classes (coesão alta, não especialista)
- controlador tem muitos atributos e mantém informação significativa sobre o domínio, ou duplica informações existentes em outros lugares

Padrões avançados

- Polimorfismo (Polymorphism)
- Fábrica pura (Pure Fabrication)
- Indireção (Indirection)
- Variações protegidas (Protected Variations)

Padrões GoF (Gang of Four)