Arquitetura de Softwares

Desenvolvimento Orientado Para Arquitetura de Software

Software, de modo genérico, é uma entidade que se encontra em quase constante estado de mudança. As mudanças ocorrem por necessidade de corrigir erros existentes no software ou de adicionar novos recursos e funcionalidades. Igualmente, os sistemas computacionais (isto é, aqueles que têm software como um de seus elementos) também sofrem mudanças frequentemente.

Essa necessidade evolutiva do sistema de software o torna 'não confiável' e predisposto a defeitos, atraso na entrega e com custos acima do estimado. Concomitante com esses fatos, o crescimento em tamanho e complexidade dos sistemas de software exige que os profissionais da área raciocinem, projetem, codifiquem e se comuniquem por meio de componentes de software.

Como resultado, qualquer concepção ou solução de sistema passa então para o nível arquitetural, onde o foco recai sobre os componentes e relacionamentos entre eles num sistema de software.

Arquitetura de Software

Quase cinco décadas atrás software constituía uma insignificante parte dos sistemas existentes e seu custo de desenvolvimento e manutenção eram desprezíveis. Para perceber isso, basta olharmos para a história da indústria do software (ver seção Links).

Encontramos o uso do software numa ampla variedade de aplicações tais como sistemas de manufatura, software científico, software embarcado, robótica e aplicações Web, dentre tantas. Paralelamente, observou-se o surgimento de várias técnicas de modelagem e projeto bem como de linguagens de programação. Perceba que o cenário existente, décadas atrás, mudou completamente.

Antigamente, os projetos de sistemas alocavam pequena parcela ao software. Os componentes de hardware, por outro lado, eram analisados e testados quase exaustivamente, o que permitia a produção rápida de grandes quantidades de subsistemas e implicava em raros erros de projetos.

Entretanto, a facilidade de modificar o software, comparativamente ao hardware, tem servido como motivador para seu uso. Além disso, a intensificação do uso do software numa larga variedade de aplicações o fez crescer em tamanho e complexidade. Isto tornou proibitivo analisá-lo e testá-lo exaustivamente, além de impactar no custo de manutenção.

Um reflexo dessa situação é que as técnicas de abstração utilizadas até o final da década de 1980 (como decomposição modular, linguagens de programação de alto nível e tipos de dados abstratos) já não são mais suficientes para lidar com essa necessidade.

Diferentemente do uso de técnicas que empregam algoritmos e estruturas de dados e das linguagens de programação que implementam tais estruturas, o crescimento dos sistemas de software demanda notações para conectar componentes (módulos) e descrever mecanismos de interação, além de técnicas para gerenciar configurações e controlar versões.

A Tabela 1 apresenta o contexto da arquitetura de software. Na programação estruturada, é feito uso de estruturas de seqüência, decisões e repetições como 'padrões' de controle nos programas. Já a ocultação de informações é um recurso do paradigma orientado a objetos que permite ao programador, por exemplo, ocultar dados tornando-os seguros de qualquer alteração acidental.

Além disso, na programação orientada a objetos, dados e funções podem ser 'encapsulados' numa entidade denominada objeto, o que resulta em mais simplicidade e facilidade na manutenção de programas. Por outro lado, os estilos arquiteturais capturam o 'padrão' de organização dos componentes de software num programa, caracterizando a forma na qual os componentes comunicam-se entre si.

Abordagem	Foco	Padrões
Programação estruturada	Sistemas de pequeno porte	Estruturas de controle
Abstração e modularização	Sistemas de médio porte	Encapsulamento e ocultação de informações
Componentes e conectores	Sistemas de grande porte	Estilos arquiteturais

Tabela 1. Contexto da arquitetura de software.

Perceba que a categorização, apresentada na Tabela 1, teve o objetivo de capturar uma visão geral de abordagens aplicadas a sistemas de software. Nada impede, por exemplo, o uso da programação estruturada em sistemas de grande porte ou da ênfase de um estilo arquitetural num sistema de pequeno porte. Entretanto, essa prática não é comum.

Note que à medida que tamanho e complexidade dos sistemas de software aumentam, o problema de projeto extrapola as estruturas de dados e algoritmos de computação. Ou seja, projetar a arquitetura (ou estrutura geral) do sistema emerge como um problema novo.

Questões arquiteturais englobam organização e estrutura geral de controle, protocolos de comunicação, sincronização, alocação de funcionalidade a componentes e seleção de alternativas de projeto. Por exemplo, nos sistemas Web, uma solução que tem sido empregada faz uso de múltiplas camadas separando componentes cliente, servidores de aplicações, servidores Web e outras aplicações (que possam ter acesso a esse sistema).

Essa estruturação em camadas objetiva facilitar a alocação da funcionalidade aos componentes. O uso de camadas oferece suporte à flexibilidade e portabilidade, o que resulta em facilidade de manutenção. Outro aspecto a destacar da arquitetura em camadas é o uso de interfaces padrões visando facilitar reuso e manutenção. Interfaces bem definidas encapsulam componentes (com funcionalidades definidas) já testados, prática que permite o reuso e também auxilia na manutenção, já que toda e qualquer alteração necessária estaria confinada àquele componente.

Importância da Arquitetura de Software

Todos esses fatores compreendem o projeto no nível arquitetural e estão diretamente relacionados com a organização do sistema e, portanto, afetam os atributos de qualidade (também chamados de requisitos não funcionais) como desempenho, portabilidade, confiabilidade, disponibilidade, entre outros.

Se fizermos uma comparação entre arquitetura de software (caracterizada, por exemplo, pelo estilo em camadas) e arquitetura ‘clássica’ (relativa à construção de edificações), podemos observar que o projeto arquitetural é determinante para o sucesso do sistema.

A Tabela 2 destaca aspectos da representação de projeto que captura elementos característicos da arquitetura enquanto que as restrições estão associadas a atributos de qualidade e, portanto, servem como determinantes nas decisões do projeto arquitetural.

Por exemplo, embora o uso de múltiplas camadas facilite a manutenção de um sistema de software, também contribui para degradar o desempenho do sistema. Uma tática tem sido reduzir o nível de acoplamento entre componentes para não comprometer o desempenho do sistema. Dessa forma, se adotarmos uma redução no nível de acoplamento dos componentes, eles terão menor necessidade de comunicação entre si, o que resulta num melhor desempenho.

Categorias arquiteturais	Representações de projeto	Restrições
Arquitetura Clássica	Modelos, desenhos, planos, elevações e perspectivas	Padrão de circulação, acústica, iluminação e ventilação
Arquitetura de Software	Modelos para diferentes papéis, múltiplas visões	Desempenho, confiabilidade, escalonamento e manutenibilidade

Tabela 2. Comparação de aspectos arquiteturais.

Hoje em dia, os processos de engenharia de software requerem o projeto arquitetural de software. Por quê?

É importante poder reconhecer as estruturas comuns existentes de modo que arquitetos de software (ou engenheiros de software realizando o papel de arquiteto de software – conforme Tabela 3) possam entender as relações existentes nos sistemas em uso e utilizar esse conhecimento no desenvolvimento de novos sistemas.

O entendimento das arquiteturas permite aos engenheiros tomarem decisões sobre alternativas de projeto.

Uma especificação arquitetural é essencial para analisar e descrever propriedades de um sistema complexo, permitindo o engenheiro ter uma visão geral completa do sistema.

O conhecimento de notações para descrever arquiteturas permite engenheiros comunicarem novos projetos e decisões arquiteturais tomadas a outros membros da equipe.

Cabe destacar que, para que haja o entendimento da arquitetura, faz-se necessário ao engenheiro de software conhecer os estilos arquiteturais existentes, conforme apresentado adiante. As propriedades de cada arquitetura, portanto, são dependentes do estilo arquitetural adotado. Por exemplo, o uso de uma notação padrão como a UML ajuda na representação de componentes e compartilhamento de informações do projeto.

Esses aspectos servem como indicadores de uma maturidade inicial de engenharia de software. Outros aspectos compreendem uso e reuso de soluções existentes no desenvolvimento de novos sistemas. Para tanto, a prototipagem tem sido usada em projetos de natureza inovadora (bem antes da implementação ou aceitação de um produto acontecer).

Além disso, o aumento da complexidade e quantidade de requisitos dos sistemas dificulta cada vez mais atender às restrições de orçamento e cronograma. Atualmente, empresas têm procurado incorporar estratégia de reuso de software, enfatizando o reuso centrado na arquitetura para obter melhores resultados no desenvolvimento de sistemas.

Note que 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. Em outras palavras, ela define a estrutura do sistema, de modo consistente para implementações, já que está diretamente relacionada aos atributos de qualidade como confiabilidade e desempenho.

A organização dos componentes num sistema de software impacta sobre a qualidade apresentada por ele. Por exemplo, a adoção de uma arquitetura em camadas serve para modularizar o sistema bem como facilitar modificações. Entretanto, um número elevado de camadas (4 ou 5) pode degradar o desempenho do sistema se houver um elevado grau de acoplamento entre os componentes.

Diversos benefícios decorrem da incorporação da arquitetura de software como ‘elemento norteador’ do processo de desenvolvimento de software. Cabe destacar que a arquitetura pode:

Prover suporte ao reuso – seus componentes definidos e testados podem ser reaproveitados em novas aplicações.

Servir de base à estimação de custos e gerência do projeto – a existência de uma arquitetura bem definida permite ao gerente de projeto adequadamente alocar tarefas de, por exemplo, implementação de componentes e melhor estimar o tempo e tamanho de equipe necessária para realização de um projeto.

Servir de base para análise da consistência e dependência – o arquiteto de software pode verificar se a arquitetura de software adotada suporta os atributos de qualidade desejados de modo consistente e avaliar o nível de dependência dos atributos de qualidade em relação à arquitetura.

Para tanto, ele faz a análise arquitetural que verifica o suporte oferecido pela arquitetura a um conjunto de atributos de qualidade (como desempenho, portabilidade e confiabilidade).

Ser utilizada para determinar atributos de qualidade do sistema – o arquiteto de software faz a análise arquitetural a fim de determinar os atributos de qualidade. Trata-se de um processo iterativo.

Atuar como uma estrutura para atender os requisitos do sistema – a arquitetura ajuda a definir os requisitos funcionais, que compreendem o conjunto de funcionalidades do sistema de software, e requisitos não funcionais (ou atributos de qualidade) que determinam as características visíveis ao usuário como desempenho e confiabilidade.

Uma questão que você deve estar se fazendo é: Por que apenas recentemente houve o foco na arquitetura de software?

A resposta é simples: economia e reuso.

Anteriormente não havia forte ênfase na disciplina de engenharia de software, fato este ocorreu com o amadurecimento desta nova área ao longo de toda a década de 1990. Tudo motivou o surgimento de um novo profissional: o arquiteto de software.

Habilidades do Arquiteto de Software

Perceba que o arquiteto de software tem um papel de suma importância para estratégia adotada pela empresa.

Ele precisa ter profundo conhecimento do domínio, das tecnologias existentes e de processos de desenvolvimento de software. Uma síntese de um conjunto desejado de habilidades para um arquiteto de software e das tarefas atribuídas a ele são apresentados na Tabela 3.

Note que a prototipagem é uma tarefa comum onde o arquiteto desenvolve um protótipo para 'testar' uma possível solução. Já a simulação pode ser empregada quando ele necessita avaliar o suporte oferecido a determinado atributo de qualidade como o desempenho. Por outro lado, a experimentação pode ocorrer quando o arquiteto precisa testar um novo componente recém implementado.

Habilidades desejadas	Tarefas atribuídas
Conhecimento do domínio e tecnologias relevantes	Modelagem
Conhecimento de questões técnicas para desenvolvimento de sistemas	Análise de compromisso e de viabilidade
Conhecimento de técnicas de levantamento de requisitos, e de métodos de modelagem e desenvolvimento de sistemas	Prototipagem, simulação e experimentação
Conhecimento das estratégias de negócios da empresa	Análise de tendências tecnológicas
Conhecimento de processos, estratégias e produtos de empresas concorrentes	'Evangelizador' de novos arquitetos

Tabela 3. Habilidades e tarefas de um arquiteto de software.

Entendo o Estilo Arquitetural

O estilo arquitetural serve para caracterizar a arquitetura de software de um sistema, possibilitando a:

Identificação de componentes o arquiteto identifica quais os principais elementos que tem funcionalidades bem definidas como, um componente de cadastro de (informações de) usuários e um componente de autenticação de usuário numa aplicação Web.

Identificação de mecanismos de interação a comunicação entre objetos por meio da troca de mensagens constitui uma forma através da qual os componentes de software interagem entre si.

Identificação de propriedades o arquiteto pode analisar as propriedades oferecidas por cada estilo baseado na organização dos componentes e nos mecanismos de interação, conforme discutido abaixo.

O estilo arquitetural considera o sistema por completo, permitindo o engenheiro ou arquiteto de software determinar como o sistema está organizado, caracterizando os componentes e suas interações. Em outras palavras, ele determina uma estrutura para todos os componentes do sistema. O estilo arquitetural compreende o vocabulário de componentes e conectores, além da topologia empregada. Mas, você pode estar se questionando: Por que saber o estilo arquitetural é importante?

Os sistemas de grande porte exigem níveis de abstração mais elevados (justamente onde se têm os estilos) que servem de apoio à compreensão do projeto e comunicação entre os participantes do projeto. Ele é determinante no entendimento da organização de um sistema de software.

Mas, o que se ganha em saber o estilo arquitetural? Ele oferece:

Suporte a atributos de qualidade (ou requisitos não funcionais);

Diferenciação entre arquiteturas;

Menos esforço para entender um projeto;

Reuso de arquitetura e conhecimento em novos projetos.

Conhecer o estilo arquitetural permite ao engenheiro antecipar, por meio da análise (arquitetural), o impacto que o estilo (isto é a classe de organização do sistema) terá sobre atributos de qualidade. Adicionalmente, facilita a comunicação do projeto, além do reuso da arquitetura (da solução).

A caracterização e existência de estilos arquiteturais constituem sinais de amadurecimento da engenharia de software, uma vez que permite ao engenheiro organizar e expressar o conhecimento de um projeto de modo sistemático e útil.

Note que uma forma de codificar conhecimento é dispor de um vocabulário de um conjunto de conceitos (terminologia, propriedades, restrições), estruturas (componentes e conectores) e padrões de uso existentes.

Conectores são empregados na interação entre componentes como, tubo (pipe) no estilo tubos e filtros, e mensagens no estilo de objetos.

Modelando a Arquitetura Usando a UML

A arquitetura é essencialmente um modelo abstrato que descreve a estrutura de componentes que compõe um software e o seu funcionamento. Este modelo deve ser descrito através de notações e linguagens apropriadas.

Diversas linguagens de descrição de arquitetura têm sido propostas, entretanto nenhuma ainda se firmou como uma linguagem padrão.

Vimos que a UML foi proposta como uma linguagem para especificação, visualização, construção e documentação de sistemas de software e pode ser utilizada em diversas etapas do desenvolvimento. A UML não foi elaborada com a finalidade de descrever a arquitetura do software.

Entretanto, ela apresenta diversos diagramas que permitem representar a estrutura lógica e física do software em termos de seus componentes, bem como a descrição do seu comportamento.

Vamos utilizar alguns dos diagramas UML para descrever a arquitetura lógica do software. É preciso ressaltar, porém, que a UML foi desenvolvida para a modelagem de software desenvolvido no paradigma de orientação a objetos. Para representarmos componentes funções é preciso fazer algumas adaptações. Outro aspecto importante é que na UML o termo componente refere-se a componentes físicos.

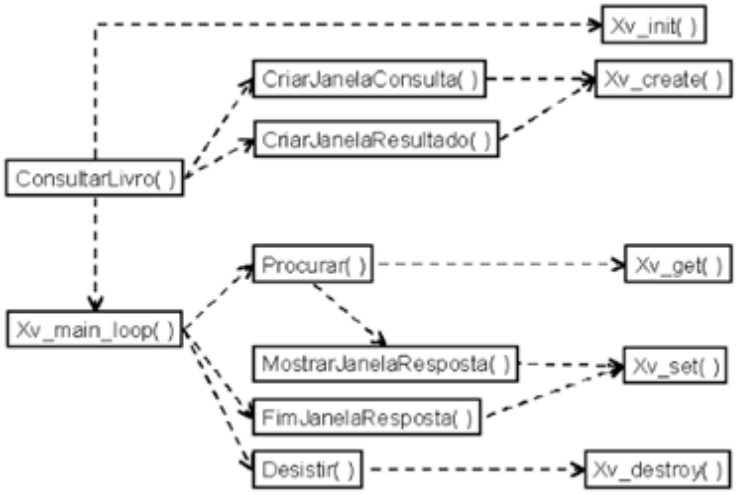
Na UML existe uma notação específica para a representação de componentes físicos e diagramas que descrevem a arquitetura em termos destes componentes. Os componentes lógicos são representados como através de diagramas de classes e de objetos.

Modelando a Dependência de Funções

As funções que compõem um software podem ser modeladas usando um diagrama de objetos simplificado. O diagrama de funções é semelhante ao diagrama de objetos e descreve a relação de dependência entre as funções.

Neste diagrama cada função é representada por um retângulo e a dependência entre elas é indicada por uma seta tracejada. Esta dependência indica que uma determinada função USA uma outra função.

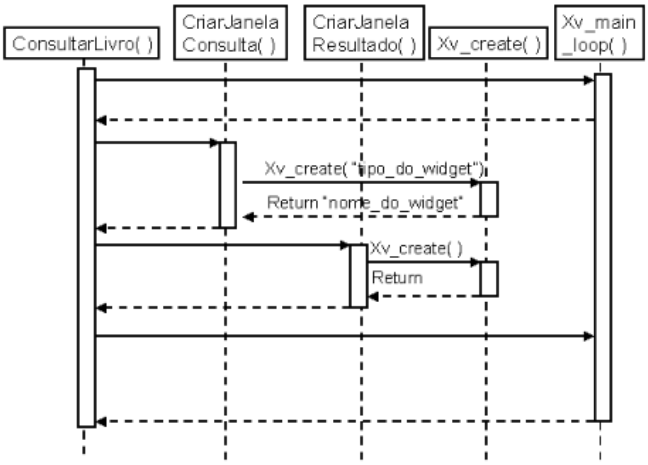
A solução B dos exemplos é composta por diversas funções que possuem dependência entre si, como mostra a figura abaixo.



Este diagrama mostra apenas que uma função usa uma ou outras funções. A função ConsultarLivro( ) usa as funções xv_init( ), xv_main_loop( ), CriarJanelaConsulta( ) e CriarJanelaResposta( ). Estas duas últimas funções usam xv_create( ).

Usando o Diagrama de Interação

Para mostrar como as funções interagem entre si podemos utilizar o diagrama de interação da UML. Este diagrama mostra como as funções interagem entre si. Este diagrama possui duas formas: o diagrama de seqüência e o diagrama de colaboração. o diagrama de seqüência mostrar como as interações entre as funções ocorrem ao longo do tempo. A figura abaixo ilustra um diagrama de seqüência .

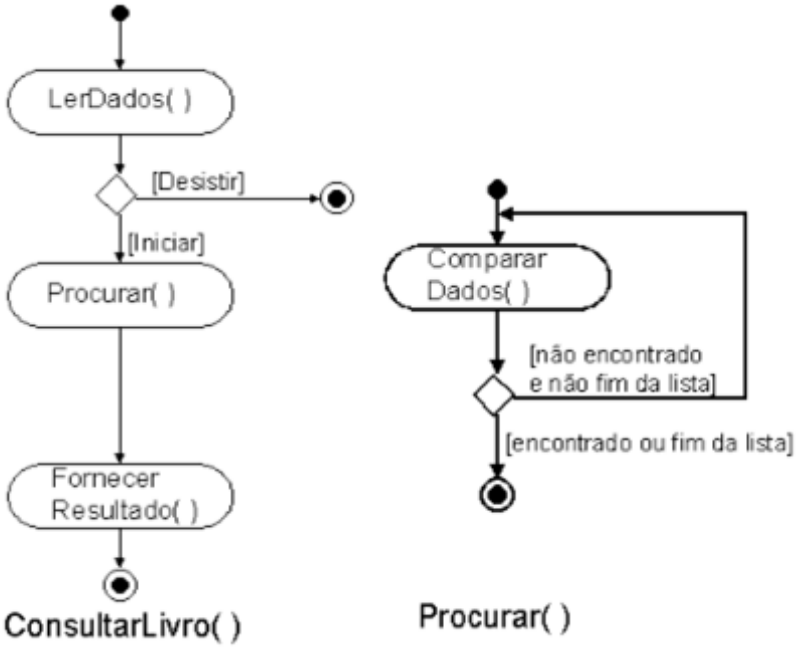


Cada função é representada por um retângulo e são alinhadas na parte superior. De cada retângulo parte uma linha pontilhada indicando a passagem do tempo de cima para baixo. O tempo no qual cada função está ativa é representado por um retângulo vertical sobre a linha do tempo.

Quando uma função chama uma outra função ela ativa esta função. Isto é representado no diagrama por uma linha saindo da função que chama e chegando na linha de tempo da função chama. O retêngulo vertical que indica a ativação da função começa a partir deste ponto. O retorno à função que fez a chamada é representado por uma seta tracejada. Cada uma das setas horizontais de chamada e retorno pode indicar quais valores (dados) são passados ou retornados entre as funções.

Usando Diagrama de Atividades

A atividade interna a cada função pode ser descrita através de diagramas de atividades. Estes diagramas descrevem cada passo da função e permitem representar estruturas de decisões e repetições internas a cada função. As figuras abaixo descrevem os diagrama sde atividades para as funções ConsultarLivro( ) e Procurar( ), da solução A dos exemplos.



Estilos e Padrões de Design Arquitetural

Padrões são soluções para problemas específicos que ocorrem de forma recorrente em um determinado contexto que foram identificados a partir da experiência coletiva de desenvolvedores de software. A proposta original de padrões veio do trabalho de Christopher Alexander na área de arquitetura (para construções). Sua definição para padrões:

Cada padrão é uma regra (esquema) de três partes que expressa uma relação entre um certo contexto, um problema, e uma solução.

O contexto descreve uma situação no desenvolvimento na qual existe um problema. O problema, que ocorre repetidamente no contexto, deve também ser descrito bem como as forças (requisitos, restrições e propriedades) associadas a ele. A solução descreve uma configuração ou estrutura de componentes e suas interconexões, obedecendo às forças do problema.
As forças, denominação dada por [Alexander], descrevem os requisitos que caracterizam o problema e que a solução deve satisfazer, as restrições que devem ser aplicadas às soluções e propriedades desejáveis que a solução deve ter.

Padrões no Desenvolvimento de Software

Os padrões são desenvolvidos ao longo dos anos por desenvolvedores de sistemas que reconheceram o valor de certos princípios e estruturas organizacionais de certas classes de software.

O uso de padrões e estilos de design é comum em várias disciplinas da engenharia. Eles são codificados tipicamente em manuais de engenharia ou em padrões ou normas

Em software podemos ter padrões a nível conceitual, a nível de arquitetura de software ou a nível de algoritmos e estruturas de dados. O padrões podem ser vistos como tijolos-de-construção mentais no desenvolvimento de software. Eles são entidades abstratas que apresentam soluções para o desenvolvimento e podem ser instanciadas quando se encontra problemas específicos num determinado contexto.

Os padrões se diferenciam de métodos de desenvolvimento de software por serem dependentes-do-problema ao passo que os métodos são independentes-do-problema. Os métodos apresentam passos o desenvolvimento e notações para a descrição do sistema, mas não abordam como solucionar certos problemas específicos.

Um sistema de software não deve ser descrito com apenas um padrão, mas por diversos padrões relacionados entre si, compondo uma arquitetura heterogênea.

Os padrões podem ser descritos em sistemas ou catálogos de padrões. Num sistema os padrões são descritos de maneira uniforme, são classificados. Também são descritos os seus relacionamentos. Num catálogo os padrões são descritos de maneira isolada.

Categorias de Padrões e Relacionamentos

Cada padrão depende de padrões menores que ele contém e de padrões maiores no qual ele está contido.

Padrões arquiteturais - expressam o esquema de organização estrutural fundamental para um sistema de software. Assemelham-se aos Estilos Arquiteturais descritos por [Shaw & Garlan 96].

Padrões de design - provê um esquema para refinamento dos subsistemas ou componentes de um sistema de software.

Idiomas - são padrões de baixo nível, específicos para o desenvolvimento em uma determinada linguagem de programação, descrevendo como implementar aspectos particulares de cada componente.

Os diversos padrões podem ser relacionados entre si. Três relações são identificados: refinamento, variante e combinação.

Refinamento: Um padrão que resolve um determinado problema pode, por sua vez, gerar novos problemas. Componentes isolados de um padrão específico podem ser descritos por padrões mais detalhados.

Variante: Um padrão pode ser uma variante de um outro. De uma perspectiva geral um padrão e suas variantes descrevem soluções para problemas bastante similares. Estes problemas normalmente variam em algumas das forças envolvidas e não na sua caracterísitica geral.

Combinação: Padrões podem ser combinados em uma estrutura mais complexa no mesmo nível de abstração. Tal situação pode ocorrer quando o problema original inclui mais forças do que podem ser balanceadas em um único padrão. Neste caso, combinar vários padrões pode resolver o problema, cada um satisfazendo um conjunto particular de forças.

Descrição de Padrões

Cada padrão pode ser descrito através do seguinte esquema:

Nome - o nome do padrão e uma breve descrição.

Também conhecido como - outros nomes, se algum for conhecido.

Exemplo - um exemplo ilustrativo do padrão.

Contexto - as situações nas quais o padrão pode ser aplicado.

Problema - o problema que o padrão aborda, incluindo uma discussão das forças associadas.

Solução - o princípio fundamental da solução por trás do padrão

Estrutura - um especificação detalhada dos aspectos estruturais.

Dinâmica - cenários descrevendo o comportamento em tempo-de-execução.

Implementação - diretrizes para implementar o padrão.

Exemplo resolvido - discussão de algum aspecto importante que não é coberto nas descrições da Solução, Estrutura, Dinâmica e Implementação.

Variantes - Uma breve descrição das variantes ou especializações de um padrão.

Usos conhecidos - Exemplos do uso obtidos de sistemas existentes.

Conseqüências - Os benefícios que o padrão proporciona e potenciais vantagens.

Ver também - Referências a padrões que resolvem problemas similares e a padrões que ajudam refinar os padrões que está sendo descrito.

Modelo de Uma Camada

Também chamado de sistemas centralizados ou de arquitetura uni processada, o modelo de uma camada era caracterizado por manter todos os recursos do sistema (banco de dados, regras de negócios e interfaces de usuário) em computadores de grande porte, os conhecidos mainframes.

Os terminais clientes não possuíam recursos de armazenamento ou processamento, sendo conhecidos como terminais burros ou mudos.

Nesta arquitetura, o mainframe tinha a responsabilidade de realizar todas as tarefas e processamento.

Modelo de Duas Camadas

Com o passar do tempo e com o surgimento dos computadores pessoais, cada vez mais microcomputadores estavam disponíveis nas mesas dos usuários, fato que foi tornando necessária a utilização do poder de processamento destas máquinas dentro do sistema.

Também devido à grande expansão das redes de computadores, os métodos de desenvolvimento de software foram aos poucos evoluindo para uma arquitetura descentralizada, na qual não somente o servidor é o responsável pelo processamento, mas as estações clientes também assumem parte desta tarefa.

Dentro deste contexto que surgiu o modelo de duas camadas, justamente com o objetivo de dividir a carga de processamento entre o servidor e as máquinas clientes.

Igualmente conhecido como modelo cliente e servidor de duas camadas, esta técnica é composta por duas partes distintas: uma executada na estação cliente e outra no servidor.

A camada cliente tem a função de prover a interface para que os usuários possam manipular as informações, ou seja, através dela realiza-se a interação entre o usuário e o sistema.

É desenvolvida para se conectar diretamente ao banco de dados, tendo como responsabilidade fazer as solicitações dos dados necessários ao servidor, sendo que este os processa e devolve o resultado.

Neste modelo, as regras de negócios (tais como funções, validações entre outros) podem ficar armazenadas no cliente, no servidor ou em ambos.

Quando contidas no cliente, apresentam-se na forma de códigos da linguagem de programação que está sendo utilizada.

Já quando localizadas no servidor, estão na forma de recursos do banco de dados, como triggers e stored procedures, por exemplo.

O cliente recebe a denominação de “cliente gordo” quando a maior parte das regras são nele implementadas, enquanto que o servidor recebe a qualificação de “servidor gordo” quando as regras são nele desenvolvidas em maior número.

Em suma, a base do funcionamento desta técnica consiste em armazenar determinado volume de dados em um computador central e deixa-lo encarregado de manipulá-los e devolve-los à estação cliente que os requisitou.

A Figura 1 mostra a arquitetura de duas camadas.

Figura 1. Arquitetura de duas camadas

Como se pode observar na figura, existem três estações clientes que fazem as requisições diretamente ao servidor de banco de dados

Modelo Multicamadas

Também conhecido como modelo cliente e servidor de várias camadas, este método é uma evolução da tecnologia de duas camadas e tem como princípio básico o fato de que a estação cliente jamais realiza comunicação direta com o servidor de banco de dados, mas sim com uma camada intermediária, e esta, com o banco de dados. Isto proporciona uma série de vantagens sobre a técnica de duas camadas, as quais serão explanadas adiante.

Um sistema multicamadas faz uso de objetos distribuídos aliados à utilização de interfaces para executar seus procedimentos, o que torna o sistema independente de localização, podendo estar tanto na mesma máquina como em máquinas separadas.

Desta forma, a aplicação pode ser dividida em várias partes, um bem definida, com suas características e responsável por determinadas funções. Em um aplicativo nestes moldes, pelo menos três camadas são necessárias: apresentação, regras de negócios e banco de dados.

A seguir, cada uma das partes do modelo é explicada.

Apresentação

A camada de apresentação fica fisicamente localizada na estação cliente e é responsável por fazer a interação do usuário com o sistema. É uma camada bastante leve, que basicamente executa os tratamentos de telas e campos e geralmente acessa somente a segunda camada, a qual faz as requisições ao banco de dados e devolve o resultado. É também conhecida como cliente, regras de interface de usuário ou camada de interface.

Regras de Negócios

Em um sistema seguindo este modelo, a aplicação cliente nunca acessa diretamente a última camada que é a do banco de dados, pois quem tem essa função é a camada de regras de negócios, na qual podem se conectar diversas aplicações clientes.

Esta parte do sistema é responsável por fazer as requisições ao banco de dados e todo o seu tratamento, ou seja, somente ela que tem acesso direto ao banco de dados. É também conhecida como lógica de negócios, camada de acesso a dados, camada intermediária ou servidor de aplicação por geralmente se tratar de um outro computador destinado somente ao processamento das regras.

O servidor de aplicação é, geralmente, uma máquina dedicada e com elevados recursos de hardware, uma vez que é nele que ficam armazenados os métodos remotos (regras de negócios) e é realizado todo o seu tratamento e processamento.

Banco de Dados

É a última divisão do modelo, na qual fica localizado o sistema gerenciador de banco de dados. É também conhecida como camada de dados.

Adicionalmente a essas três divisões, também pode ser implementada uma camada somente para validação, na qual são executados todos os procedimentos necessários para garantira integridade dos dados digitados na camada de apresentação.

A Figura 2ilustra o esquema de comunicação de um sistema multicamadas.

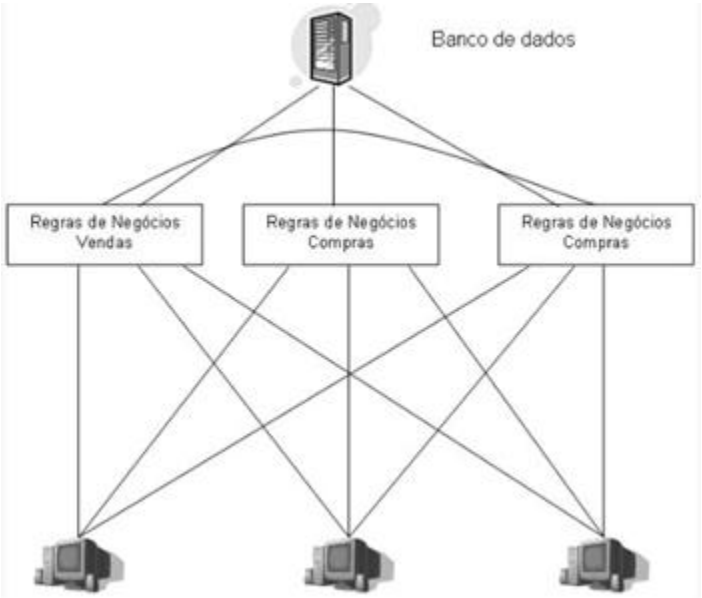


Figura 2.Esquema de comunicação de um sistema multicamadas

Na figura, na porção superior está localizado o servidor de banco de dados, o qual se comunica com os servidores de aplicação através de algum protocolo de rede (TCP/IP, por exemplo) e o acesso aos dados é realizado por meio da linguagem SQL (Structured Query Language). Na parte inferior estão as estações clientes, que fazem a comunicação com a camada intermediária através da utilização de interfaces. Este é basicamente o esquema de comunicação desta arquitetura e o mesmo não pode ser alterado.

Ainda pode-se observar na figura anterior que também é possível haver a interação entre os servidores de aplicação. Com isso é possível obter o recurso de escalabilidade que será detalhado adiante.

Outro recurso que pode ser utilizado e não está representado na figura é uma estação cliente, que possui melhores recursos de hardware, agir tanto como cliente quanto como servidor de aplicação. Este fato é plausível desde que a máquina possua os protocolos necessários para realizar tal função.

Vantagens do Desenvolvimento em Multicamadas

Uma aplicação desenvolvida neste modelo apresenta várias vantagens sobre a técnica de duas camadas, dentre elas pode-se destacar a modularização, a facilidade de redistribuição, os clientes leves, a economia de licenças de acesso ao banco de dados, a economia de conexões no servidor, a escalabilidade e a independência de localização, de linguagem de programação e de sistema gerenciador de banco de dados. A seguir será detalhado cada um desses benefícios.

Modularização

A modularização refere-se a separara lógica do negócio e regras de acesso ao banco de dados da camada de apresentação. Desta maneira, várias aplicações clientes podem compartilhar as mesmas regras, que ficam encapsuladas em uma camada de acesso comum.Assim sendo, as regras ficam centralizadas em um único local, ao contrário de em uma aplicação desenvolvida em duas camadas;

na qual geralmente existe redundância nestas regras e uma mudança mesmo que pequena acarretará na redistribuição do aplicativo em cada estação cliente.

Um exemplo prático deste fato é a construção de um simples cadastro de clientes que deve ser disponibilizado com uma interface padrão baseada em formulários e outra baseada em um browser para acesso pela Internet.

No modelo de duas camadas, se determinada validação for implementada na aplicação feita com formulários, esta deverá ser recodificada na outra aplicação justamente por não estar centralizada. Neste exemplo, a camada de regras de negócios poderia executar o papel de centralizadora, atendendo as duas situações descritas e solucionando a questão.

Outro problema comum que pode ocorrer é no controle de versão, pois se determinado usuário possui uma versão mais antiga do que outro, ocorrerá erros de dados lógicos no processamento das regras de negócios.

Facilidade De Redistribuição

Como as estações clientes acessam uma mesma camada em comum, qualquer alteração realizada nas regras de negócios (geralmente um EXE ou uma DLL no servidor de aplicação) será vista por todas as aplicações clientes.

Clientes Leves (Thin-Clients)

Ao contrário de em uma aplicação duas camadas na qual há a divisão das regras de negócios entre o cliente e o servidor, em multicamadas isto não ocorre, pois como a camada intermediária é a responsável por fazer todo o processamento das solicitações de dados no servidor de banco de dados, cabe à camada de apresentação somente exibir estes dados, tendo no máximo os códigos de tratamento de telas e campos.

Com isso, a aplicação cliente apresenta grande diminuição de código e todo o trabalho de instalação é bastante reduzido, possuindo somente uma configuração para o cliente ter acesso à camada intermediária. Por esta razão, há diminuição de custos, uma vez que não existe necessidade de fazer upgrade nas estações clientes que apresentam poucos recursos de hardware ou que são computadores antigos.

Economia de Licenças de Acesso ao Banco de Dados

Em um modelo construído em duas camadas, a estação cliente faz acesso direto ao servidor de banco de dados através de um conjunto de bibliotecas que ficam localizadas no computador cliente e que têm a função de viabilizar a comunicação entre ambos. Visto que muitos fabricantes de sistemas gerenciadores de banco de dados cobram taxas por licenças adicionais para utilização dessas bibliotecas, com o modelo multicamadas elas ficam localizadas somente na camada de acesso a dados, eliminando assim custos extras com licenças.

Economia de Conexões no Servidor

No modelo de duas camadas, se existirem, por exemplo, quinhentas estações clientes conectadas simultaneamente no servidor, os mesmos números de conexões no banco de dados serão realizados, uma para cada cliente. Numa arquitetura multicamadas isso não ocorre, porque se uma conexão for realizada pelo servidor de aplicação, está será compartilhada por todas as máquinas que nele se conectarem.

Através desta característica,é possível solucionar eventuais problemas com o número de conexões no banco de dados desejadas maior que a quantidade de licenças de acesso disponíveis.

Escalabilidade

Com a utilização do modelo de duas camadas, é comum que ocorra uma queda de desempenho quando um grande número de máquinas clientes simultâneas se conecta ao servidor. Este fato é conhecido como gargalo de rede e mesmo que o servidor seja um computador potente e localizado em uma rede veloz, pode ocorrer o problema de gargalo de I/O (Input/Output) na máquina servidora.

Com o modelo multicamadas este problema pode ser evitado, uma vez que é possível ter a mesma regra de negócio dividida entre vários servidores através do balanceamento de carga, ou seja, quando algum deles ficar sobrecarregado o outro entra em ação para ajudá-lo. Se ocorrer algum problema com algum servidor e este não puder mais responder as requisições (ficar off-line, por exemplo), outro servidor poderá entrar em seu lugar.

Pode-se observar na figura anterior que existem dois servidores de aplicação com as regras de negócios do módulo de compras. Através disso, se um deles estiver sobrecarregado ou ficar desconectado, o outro entrará em ação como descrito anteriormente.

Outra característica importante é que se o sistema for de grande porte, pode-se dividi-lo em vários servidores de aplicação, um para cada setor como mostrado na figura (vendas e compras), evitando assim o gargalo na rede e melhorando o desempenho.

Independência de Localização

Visto que esta arquitetura utiliza objetos distribuídos, o servidor de banco de dados e o servidor de aplicação podem estar fisicamente distantes da aplicação cliente. Se alguma empresa, por exemplo, possuir cinco filiais geograficamente distribuídas, todas podem acessar o mesmo servidor de aplicação.

Independência de Linguagem de Programação

Como são utilizadas interfaces na construção da arquitetura, uma camada de regras de negócios construída sobre o protocolo COM, por exemplo, pode ser acessada por aplicações clientes desenvolvidas em diversas linguagens de programação que possuem suporte ao COM.

Independência de Sistema Gerenciador de Banco de Dados

Numa arquitetura multicamadas, o banco de dados é utilizado somente como um contêiner para armazenar as tabelas e dados, pois quando recursos como triggers e stored procedures são implementados, ocorre uma ligação direta da aplicação com o banco de dados, fato que pode tornar bastante trabalhoso o processo de migração no caso da aplicação mudar de banco de dados.

Isto ocorre porque cada solução possui suas particularidades, ou seja, a construção de uma trigger ou stored procedure em determinada ferramenta geralmente será diferente de outra. Por isso, em sistemas multicamadas deve-se evitar usar tais recursos, deixando o servidor de aplicação encarregado deste controle.

Aplicações multicamadas podem ser utilizadas normalmente como um substituto do habitual modelo de duas camadas, pois como observado anteriormente, apresenta vantagens bastante significativas, principalmente no que diz respeito à organização, manutenção, custos, desempenho e portabilidade.

Se, por exemplo, determinada aplicação desenvolvida em duas camadas apresentar problemas relacionados à dispersão das regras de negócios entre cliente e servidor, à dificuldade de redistribuição do aplicativo nas estações clientes, à problemas em migrar de banco de dados devido às regras estarem armazenadas em formas de stored procedures ou triggers, e, principalmente, à queda de desempenho por causa do gargalo na rede; são fortes indícios de que a aplicação deve ser mudada para o modelo multicamadas, pois este, como visto anteriormente, possui recursos adequados para resolver os problemas supracitados e ainda, oferecer outras facilidades.

O desenvolvimento através do modelo multicamadas vem crescendo constantemente, sendo que seu uso é mais comumente indicado para sistemas complexos e de grande porte, que requerem grande volume de dados, alto grau de processamento, grande quantidade de usuários conectados simultaneamente e utilização em ambientes heterogêneos.

Introdução ao Padrão Model View Controller (MVC)

Nesse artigo vamos entender um pouco mais sobre o que é o padrão MVC e por que trabalhar com uma arquitetura bem definida faz total diferença no desenvolvimento de um app (aplicativo, software desenvolvido para rodar em um celular). Vale lembrar que o MVC é aplicado para diversas tecnologias não somente para o mobile, no caso do artigo, o foco principal é no desenvolvimento para iOS vamos focar mais em como usá-lo em nosso ambiente.

Em todo projeto existe a parte onde estamos esboçando o problema, essa etapa é mais teórica onde estudamos como entender o problema para encontrarmos a solução e só assim escolher a melhor. Por exemplo, temos um problema em pedir um táxi em nossa cidade. Depois de horas de discusão com a equipe decidimos criar um app para chamar táxis.

Uma vez que decidimos como resolver, é hora de reunir a equipe e analisar a melhor arquitetura para trabalhar no app, essa etapa eu acho de extrema importância a participação de todos do time de desenvolvimento e exige muita experiência e é exatamente aí que o trabalho em equipe contribui muito.

Voltando ao MVC, a arquitetura de um app possui várias camadas como: camadas de persistência, camadas conexão, camadas de interação, camadas utilitários, etc. Sendo assim, é muito importante na arquitetura definirmos as camadas que precisarão ser utilizados no app e como vão se conectar entre elas.

No mercado existem vários (Design Patterns) e MVC é apenas um deles. Como developer você deve entender as características de cada Design Patterns para poder escolher aquele que melhor atende as necessidades do seu projeto.

Model View Controller

Como ja falamos o MVC é um Design Pattern muito antigo descrito pela primeira vez em 1979, ele é de alto nível e pode ser usado tanto para a arquitetura global do app quanto para uma pequena parte dele. Ele classifica os objetos de acordo com as funções que eles possuem no app. Basicamente trabalha delegando atribuições a sua aplicação em três partes: O Modelo, a View e o Controlador.

Nota: A tradução correta da segunda parte seria Visão, porém é comumente utilizado como View. Em diversos materiais você vai encontrar essas três partes separadas em inglês (Modal, View, Controller).

No início o MVC foi desenvolvido para mapear a entrada, processamento e saída que os programas baseados em GUI (Graphical User Interface) utilizavam. Na imagem abaixo fazemos uma comparação de como isso fica no padrão MVC.

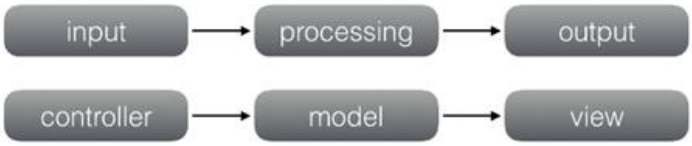


Figura 1: Exemplo de mapeamento do modelo MVC.

A maioria das aplicações orientadas a objeto pode se beneficiar trabalhando com MVC, o código passa a ser mais reutlizável e as interfaces melhor definidas uma vez que cada camada cuida da sua parte.Na prática se você usa MVC em um app, e em algum momento no futuro você precise alterar ou implementar alguma funcionalidade em seu código, essa modificação é muito mais flexível, mais fácil de ser implementada sem causar grande impacto no seu código.

Ok, mas como de fato um developer implementa isso em sua aplicação? Como isso pode me ajudar a ser um developer melhor ? Vamos chegar lá, primeiramente para exemplificar melhor vamos ilustrar a interação de um usuário através de uma modelagem. Veja a imagem abaixo

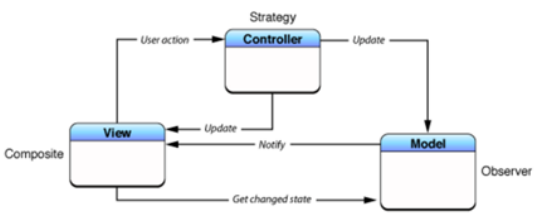


Figura 2: Exemplo das camadas do MVC utilizadas em uma aplicação.

A Figura 2 representa a concepção original do MVC onde o usuário insere uma informação na view, um evento é gerado nesse momemento e a View notifica a camada Controller que pode interpretar esse

evento e notificar a camada Model que por sua vez modifica o objeto Model se necessário e devolve essa alteração para a aplicação, é exatamente aqui que o fluxo começa a mudar.

No MVC original quando a camada Model termina de fazer as suas alterações que foram solicitadas pelo Controller ou pela View ele notifica todos os objetos que tenham se registrado como observers dessas altereções de estado, e nesse caso isso pode passar ou não pelo Controller, mas você consegue atualizar quem precisa ser atualizado sem problemas basta cadastrar o objeto como observer. Dentro de iOS você também pode trabalhar dessa forma, porem a própria Apple não recomenda a utilização do MVC no padrão original.

No MVC adotado pela Apple as coisas funcionam um pouco diferente, na maioria das aplicações os objetos Model notificam o Controller que por sua vez notifica a View para fazer as devidas atualizações. Vamos olhar a figura abaixo pra entender melhor como funciona:

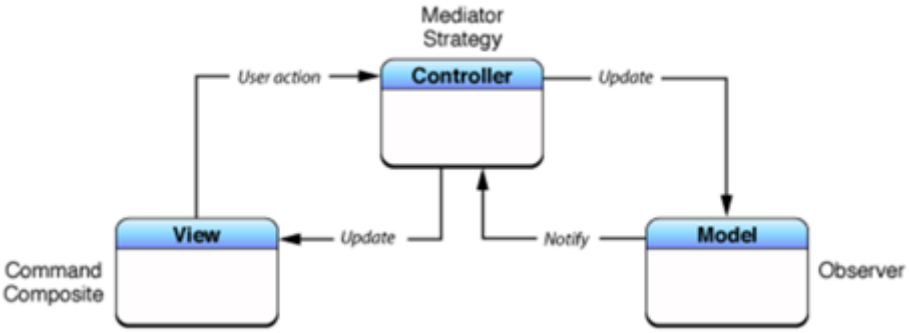


Figura 3: Modelo MVC recomendado pela Apple.

Vamos facilitar mais ainda com um exemplo, vamos imaginar a seguinte situação, temos um app que possui apenas um botão na tela que exibe as informações de um usuário como (nome,sobrenome,idade).

Então quando o usuário clicar na tela ele vai exibir essas informações. Agora vamos pensar no modelo MVC desse app, teremos na camada View a interface com o botão e uma tabela onde as informações serão exibidas, nós teremos uma classe (Controller) que terá o método (exibirUsuário) e por fim teremos a nossa classe Pessoa (Model) que será responsável por montar o objeto Pessoa que será apresentado na Interface.

Agora que ja temos a nossa estrutura montada, vamos ao fliuxo, o Usuário em algum momento clica no botão, quando isso acontece a camada View notifica a nossa Controller que dispara o método (exibirUsuário) esse cara precisa montar o objeto Pessoa para exibi-lo na tela, sendo assim a Controllernotifica a classe Pessoa que nesse caso é nossa camada Model, nessa classe nós montamos o objeto Pessoa e quando esse processo acaba a camada Modelnotifica a Controller que pode verificar se está tudo OK e notificar a camada View para finalmente atualizar a interface para o usuário poder ver o objeto Pessoa sendo exibido.

Talvez ainda não tenha ficado muito claro, não se assuste com isso, esse modelo exige muito estudo para aplicá-lo de forma que beneficie a sua aplicação, mas um ótimo começo é você entender bem o que cada camada deve fazer, e por isso vamos ajudar descrevendo cada uma delas.

Model

Essa camada é responsável por manipular os dados que serão processados no app. Uma boa prática ao desenvolver um aplicativo usando MVC é encapsular todos os dados importantes nessa camada, criando um ou vários objetos model. Um objeto model não possui nenhuma ligação explícita com a User Interface (View), que é usada para apresentar esse objeto ou até mesmo editá-lo. Por exemplo, temos um objeto model que representa um (Carro) que será usado para alimentar uma View com dados desse carro. Podemos armazenar nesse objeto Model diversas informações do carro como (cor,modelo,ano) e você pode querer armazenar a quiilometragem do carro, não tem problema nenhum usar o objeto Model para armazenar essa informação, no entanto a forma como essa informação será exibida não

deve ser tratada nesse objeto model, você deve utlizar outra camada como a View por exemplo para fazer esse tipo de tratamento.

É importante saber que aqui cabe um pouco de flexibilidade pois depende muito de cada app, essa separação nem sempre é a melhor coisa a se fazer, mas na maioria dos casos o objeto model não deve se preocupar com será apresentado na View. Ele deve trabalhar apenas com a manipulação dos dados importantes para o correto funcionamento do app.

View

Essa camada é responsável por exibir as informações para o usuário e permitir que eles editem essas informações quando necessário, assim como a camada model é responsável por apenas armazenar as informações e não deve se preocupar em como elas serão exibidas, essa separação também vale para a camada View, mas nesse caso ela não deve se preocupar em como os dados que estão sendo exibidos ou editados serão armazenados. Mais uma vez vale ressaltar que isso não é uma regra absoluta, em algumas situações você pode precisar armazenar dados na View, e isso não irá comprometer seu aplicativo, mas sempre que possível procure separar bem as tarefas de cada camada.

A View deve garantir que ela está exibindo o objeto Model corretamente, como consequência ela vai precisar saber quando esse objeto sofre alterações, como os objetos Model não devem ser vinculados aos objetos View de forma específica, para que a View saiba quando as alterações ocorreram é necessário termos uma forma genérica de notifica-la e é ai que entra a nossa próxima camada Controller, mas é importante saber que ela não é a unica forma de a View se comunicar com a Model e vice versa.

Controller

Essa camada é a responsável por fazer a comunicação entre a camada View e a Model, Essa camada também pode controlar o ciclo de vida de alguns objetos, no modelo MVC quando um usuário clica em um botão na tela essa toque é feito na camada View que notifica a Controller que interpreta essa ação, que por exemplo poderia ser disparando um método que está conectado a esse botão, a Controller dispara essa ação que altera alguma propriedade do objeto Model, esse por sua vez faz a alteração e notifica novamente a Controller para ela avisar a camada View que deve atualizar a interface para o usuário.

Lembre-se de que estamos abordando o MVC focado para iOS e a documentação da Apple recomenda que a comunicação entre objetos View e objetos Model seja feita atravéz da camada Controller para que esses objetos tenham uma maior flexibilidade para reutilização.

Combinando os Papéis

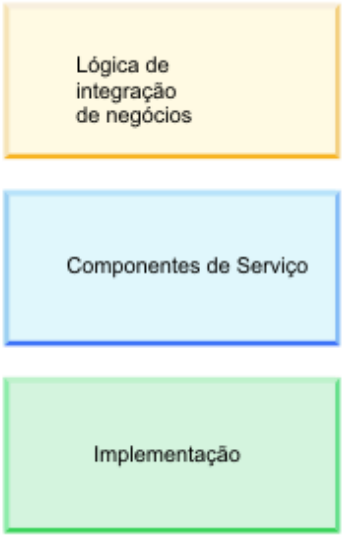
Ainda falando sobre o modelo MVC usado pela Apple existe mais um recurso muito utilizado que é a junção dos papéis como por exemplo as classes ViewControllers que tem como principal responsabilidade gerir a interface e se comunicar com a classe model. Por exemplo as actions dos botões são geralmente implementadas na ViewController, pois elas precisão ser conectadas em um objeto de interface (botão) e em alguns casos necessitam acessar um objeto model que está em outra classe para poder exibir essas informações na interface.
Você pode encontrar mais informações na própria documentação da Apple que foi uma das referências utilizadas nesse artigo.

Dicas de um Developer

Esse artigo apenas arranha a superfície do assunto, eu recomendo que você pesquise mais sobre o assunto, se você é um iniciante na área de programação mobile recomendo que faça um projeto e aplique o que aprendeu no artigo, a melhor maneira de evoluir nessa área é pondo a mão na massa. Lembre-se também que existem outros Design Patterns que você pode estudar e que certamente podem te ajudar algum dia.

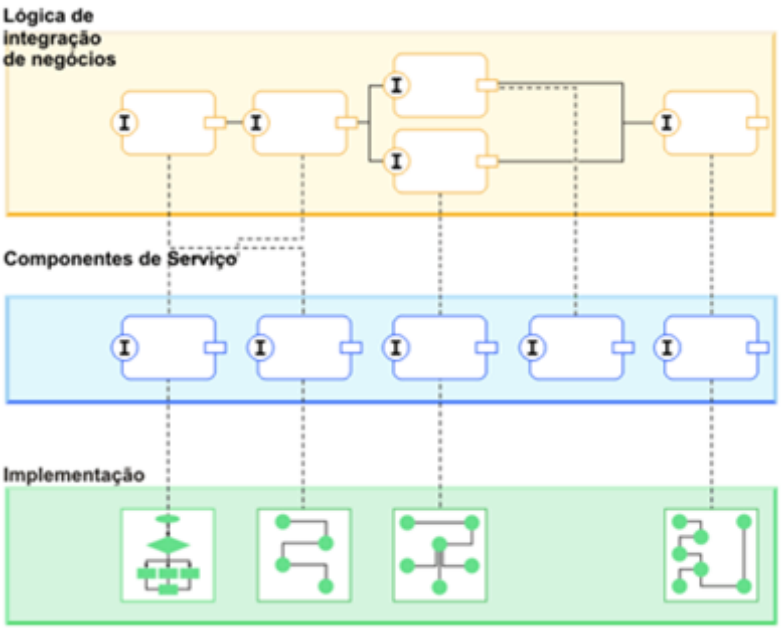
A Service-oriented architecture (SOA) é um padrão de mercado vagamente definido que apresenta todos os processos de negócios de uma maneira voltada para serviço. Dependências entre serviços como serviços da web, ativos de serviço do Enterprise Information System (EIS), fluxos de trabalho e bancos de dados são minimizados e a implementação de qualquer serviço é oculta.

O objetivo da arquitetura orientada a serviços é separar a lógica de integração de negócios da implementação para que um desenvolvedor de integração possa focar na montagem de um aplicativo integrado em vez de nos detalhes da implementação. Para alcançar esse objetivo, os componentes de serviço que contêm a implementação de serviços individuais requeridos pelos processos de negócios são criados. O resultado é uma arquitetura de três camadas (lógica de integração de negócios, componentes de serviço e implementação) conforme mostrado no diagrama a seguir:



Porque os componentes de serviço contêm a implementação, eles podem ser montados graficamente pelo desenvolvedor de integração sem o conhecimento de detalhes da implementação. Os componentes de serviço oferecem também a opção de permitir que o desenvolvedor de integração ou alguém que trabalhe para o desenvolvedor de integração inclua a implementação posteriormente. Componentes são montados visualmente juntos.

Em outras palavras, você não é exposto ao código dentro dos componentes. No nível da lógica de negócios, mostrado no diagrama a seguir, os componentes são montados independente de sua implementação. A arquitetura orientada a serviços então permite que você se concentre em solucionar seus problemas de negócios utilizando e reutilizando componentes em vez de desviar sua atenção para a tecnologia que está implementando os serviços que você está utilizando.



Benefícios-chave da arquitetura orientada a serviços

Com o aumento da taxa de negócios modernos, uma arquitetura orientada a serviços fornece a agilidade para reagir e capitalizar sobre as condições de negócios que mudam. Ela também permite tornar o seu software sustentável ao longo do tempo. Alcança esses objetivos das seguintes maneiras:

Consolidação da lógica de negócios e dos dados de negócios. Componentes usados por vários grupos em uma corporação ou mesmo compartilhados entre um conjunto de corporações podem ser usados por todos como os componentes em conformidade com padrões de mercado como a Web Services Descriptive Language (WSDL) e a Business Process Execution Language (BPEL), que são independentes de plataforma e fornecedor. Os dados são consistentemente representados da mesma forma, permitindo que eles sejam compartilhados entre componentes de um aplicativo de arquitetura orientada a serviços.

Alavancagem de aplicativos e sistemas existentes. Quando você coloca aplicativos e sistemas dentro de código WSDL, eles se tornam universalmente disponíveis a qualquer desenvolvedor de aplicativos na corporação que esteja desenvolvendo um aplicativo atual.

Quais são os problemas de design fundamentais de uma arquitetura orientada a serviços?

Os componentes estão fracamente acoplados. Um componente que acessa outro componente não requer conhecimento das estruturas de dados, das chamadas para outros componentes, do gerenciamento de transações e assim por diante nesse outro componente.

Os componentes são configuráveis. Olhar um aplicativo de arquitetura orientada a serviços como no diagrama anterior é igual a olhar um diagrama de configuração. Os componentes podem ser incluídos, excluídos e configurados de diferentes maneiras para criar novos aplicativos.

Os componentes são interoperáveis. Qualquer componente pode interoperar com outro componente, incluindo componentes criados por ambientes de desenvolvimento de diferentes fornecedores.

Os Componentes São Independentes de Locais.

Em conjunto, esses princípios de desenvolvimento criam uma arquitetura flexível capaz de adaptar-se e prosperar nas mudanças rápidas das condições de negócios.