UNIDADE 2 – ABORDAGENS DE DESENVOLVIMENTO BASEADO EM ARQUITETURA E ESTILOS DE ARQUITETURAS DE *SOFTWARE*

MÓDULO 1 – MODEL DRIVEN ARCHITECTURE MDA (ARQUITETURA ORIENTADA A MODELOS)

01

1 - O QUE É O MDA?

Após estudarmos o que é a Arquitetura de *Software*, sua importância para o desenvolvimento de *software* e os principais estilos de arquitetura, vamos nos aprofundar nos estilos mais utilizados atualmente. Nosso foco será a Arquitetura Orientada a Modelos, Modelo em 3 Camadas, MVC e Arquitetura Orientada a Serviço.

Um tema recorrente na evolução da engenharia de *software* é a utilização de linguagens mais abstratas na modelagem das soluções. Programas desenvolvidos em linguagens mais abstratas, como JAVA e C#, são transformados em executáveis através de ferramentas próprias para este fim, como os compiladores. A utilização de linguagens mais abstratas visava proporcionar mais produtividade e a diminuição de erros uma vez que estas linguagens, supostamente, estariam mais próximas da linguagem natural.

É evidente que poucas pessoas acreditam que JAVA e C# são exemplos de linguagens abstratas. Na verdade, a cada dia surgem novas linguagens de desenvolvimento, mas pouquíssimas ganham notoriedade. Mas isso não desencoraja os pesquisadores em tentar criar linguagens cada vez mais próximas da linguagem natural.

Model Driven Architecture (Arquitetura Orientada a Modelos) ou simplesmente MDA é uma visão de como o *software* pode ser desenvolvido colocando a modelagem no centro do processo de desenvolvimento.

Trata-se de uma tecnologia recente que lidera o grupo em termos de especificação mais abstrata e ferramentas de desenvolvimento.

Model Driven Architecture

No ano 2000, o Object Management Group (OMG) publicou o "Model Driven Architecture" (OMG 2000), um documento que descreve uma visão de desenvolvimento de *software* de como modelos de objetos devem ser interligados para construir sistemas completos.

02

Como seu nome sugere, o **modelo das aplicações** é a força motriz por trás do MDA. Um modelo no MDA é uma especificação formal de uma função, estrutura ou comportamento de uma aplicação ou de um sistema.

Na abordagem MDA, um sistema de TI é primeiramente analisado e especificado como um modelo computacional independente (CIM), também conhecido como modelo de domínio. O CIM foca no ambiente e nos requisitos do sistema. Os detalhes computacionais e de implementação são omitidos neste nível de descrição ou até mesmo ainda não foram definidos.

Como apresentado na figura a seguir, o modelo de domínio é transformado em um modelo independente de plataforma (PIM) que contém as informações computacionais da aplicação, mas nenhuma informação específica para a tecnologia da plataforma subjacente que será usado para, eventualmente, implementar o modelo independente de plataforma. Finalmente, o modelo é transformado em um modelo específico de plataforma (PSM), que incluem descrições detalhadas e elementos específicos da plataforma alvo da implementação.

1 - Modelo de Transformação no MDA

03

Uma plataforma no MDA é definida como um conjunto de subsistema e tecnologia que fornece um acervo coerente de funcionalidades através de interfaces e padrões de uso especificados.

Uma plataforma MDA é um conceito bastante amplo. Plataformas geralmente referem-se a um conjunto específico de tecnologia de subsistemas que são definidos como um padrão. Um exemplo deste tipo de padrão é o J2EE.

Plataformas também podem ser referenciadas como uma plataforma específica de um fornecedor que implementa um padrão, como a plataforma J2EE da WebLogic ou como Microsoft .Net.

O MDA é suportado por uma série de padrões da OMG, incluindo a UML, o MOF (Meta-Object Facility), o XMI (XML Metadata Interchange), e o CWM (Common Warehouse Metamodel). O MDA inclui padrões que definem como um sistema pode ser desenvolvido utilizando uma abordagem voltada a modelo. E estes modelos devem ser especificados por uma linguagem de modelagem que pode variar de linguagens de modelagem genéricas aplicáveis a múltiplos domínios (por exemplo, UML) para uma linguagem de modelagem específica de domínio.

04

2 - Por que utilizar MDA?

Como podemos perceber, modelos têm um papel central no MDA. Mas por que exatamente precisamos de modelos?

Modelos fornecem abstrações de um sistema que permitem aos participantes ou interessados raciocinar sobre o sistema de diferentes pontos de vista e sobre diferentes níveis de abstração.

Modelos podem ser utilizados de diversas maneiras, como por exemplo, para prever a qualidade de performance de um sistema, validar o *design* com relação aos requisitos, e para comunicar as características do sistema para os analistas de negócio, arquitetos e engenheiros de *software*.

Os três principais **objetivos do MDA** são:

- portabilidade,
- interoperabilidade e
- reuso.

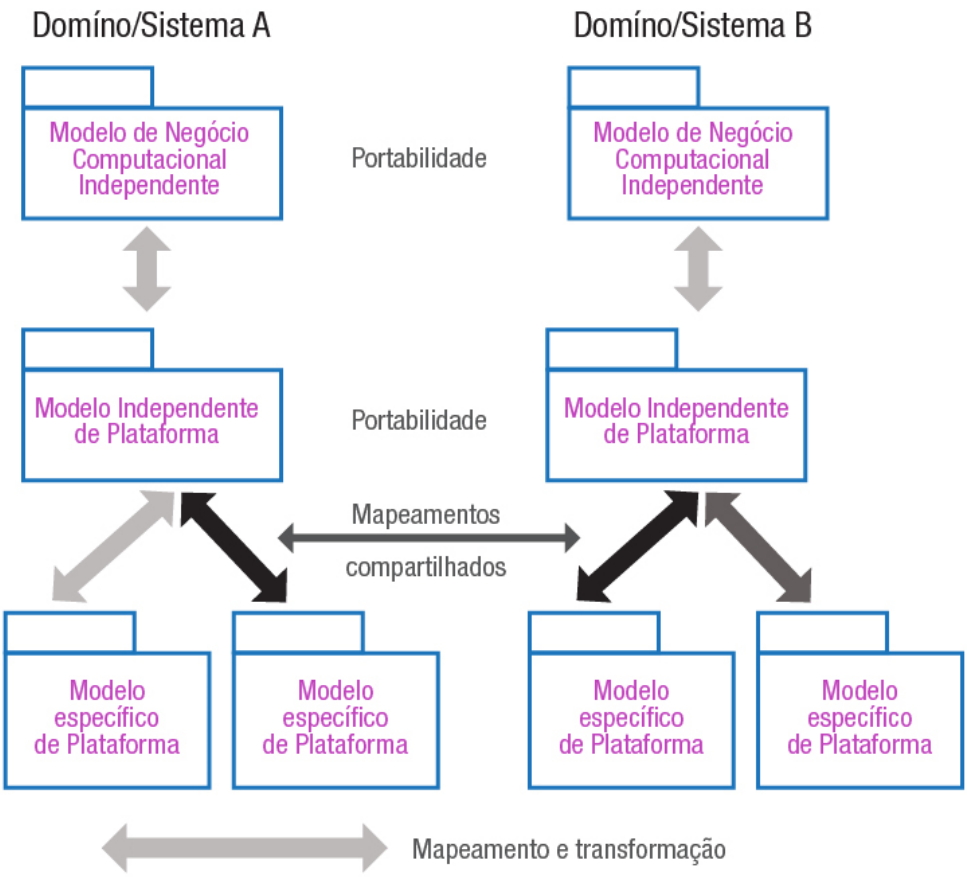
A seguir vamos verificar como cada objetivo é atendido.

05

2.1 - Portabilidade

Portabilidade pode ser atingida por meio da separação dos modelos e das transformações. Em modelos de alto nível não temos detalhes técnicos ou de baixo nível sobre a plataforma.

Como ilustrado na figura abaixo, quando a plataforma de baixo nível for alterada ou evoluída, os modelos de nível superior podem ser transformados para uma nova plataforma diretamente, sem a necessidade de manutenção.



2 - Mapeamento de Modelos MDA

A portabilidade também é alcançada elaborando modelos que podem ser movidos através de diferentes ferramentas.

Ferramentas

Os padrões MOF e XMI permitem que um modelo UML seja transformado em um documento XML e que este documento seja importado em uma nova ferramenta por questões de modelagem e análise.

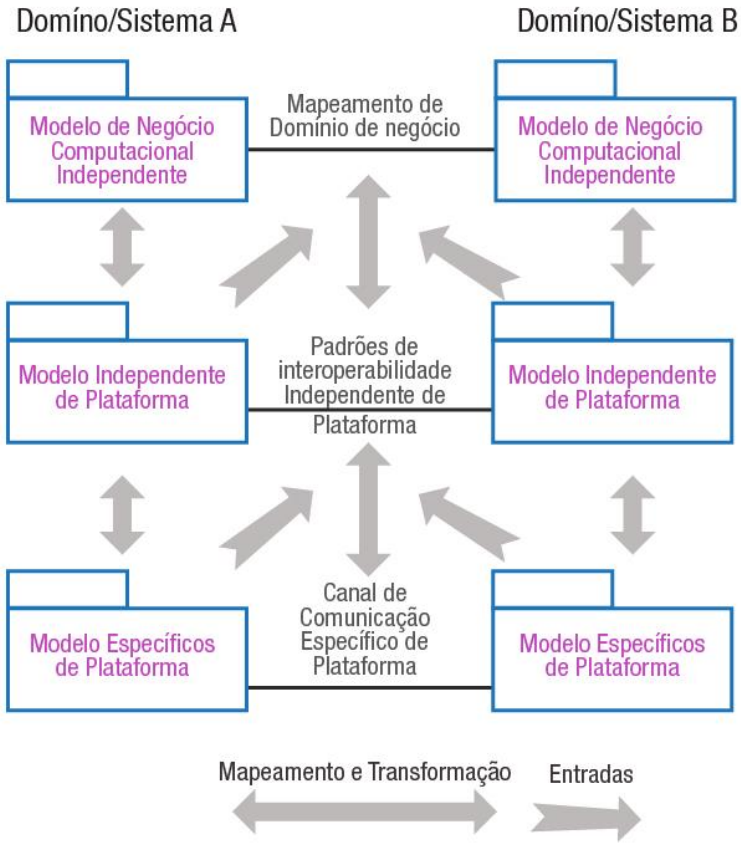
06

2.2 - Interoperabilidade

É raro encontrarmos uma aplicação que não se comunica com outras aplicações. Aplicações de nível corporativo particularmente necessitam se comunicar por meio das fronteiras internas e externas da organização de uma maneira heterogênea e distribuída. Na maioria das vezes tem-se um controle limitado sobre os outros sistemas com os quais se tem que interagir.

Utilizando o MDA, a interoperabilidade é atingida por meio de um mapeamento horizontal de modelo e interação.

Conforme apresentado a figura a seguir.



3 - Mapeamento horizontal de Modelo

A interoperabilidade pode ser encarada como um problema de **mapeamento de modelo horizontal e interações**. Para simplificar, vamos supor que temos dois conjuntos de modelos para dois sistemas diferentes, conforme apresentado na figura. A interação entre os níveis mais altos pode ser analisada e modelada. A interação entre os modelos pode então ser detalhada e os elementos envolvidos no mapeamento de alto nível podem ser facilmente rastreados ou mesmo automaticamente traduzidos em elementos de nível mais baixo.

O mesmo problema pode também ser visto como um problema de **refinamento** de um único modelo de alto nível em vários modelos de operação entre duas ou mais plataformas. **Saiba+**

Saiba+

Diferentes partes dos modelos de nível superior são refinadas em modelos específicos para diferentes plataformas. Associações nos modelos originais são refinadas em canais de comunicação ou bancos de dados entre modelos específicos de plataforma compartilhada.

07

2.2 - Reuso

Reuso é a chave para se alcançar maior produtividade e qualidade.

MDA encoraja o reuso de modelos e das melhores práticas na modelagem de aplicações, especialmente através da criação de famílias de aplicações, como em uma linha de produtos de *software*.

08

3 – As Ferramentas

Embora seja possível praticar partes do MDA sem suporte de ferramentas, isso não é recomendado para quem busca produtividade e qualidade no desenvolvimento. Uma grande parcela dos padrões é destinada às ferramentas e a interoperabilidade entre estas ferramentas. Alguns padrões são destinados a serem legíveis por máquina, dificultando a leitura sem o apoio de ferramentas de *software*.

Desde o surgimento dos padrões MDA que uma infinidade de ferramentas passou a suportar este padrão, todas com características e capacidades muito diferentes entre si. A falta de definição do MDA em certas características tem causado problemas em termos de interoperabilidade entre ferramentas e para a consequente reutilização.

Basicamente, uma ferramenta MDA é uma ferramenta utilizada para desenvolver, interpretar, comparar, alinhar, medir, verificar e transformar modelos ou metamodelos.

Em qualquer abordagem MDA temos essencialmente dois **tipos de modelos**:

- os **modelos iniciais**, criados manualmente por agentes humanos,
- os **modelos derivados**, criados automaticamente por programas.

Veja um exemplo.

Exemplo

Por exemplo, um analista pode criar um modelo UML inicial da sua observação de alguma situação de negócios enquanto um modelo Java pode ser automaticamente derivado a partir deste modelo UML por uma operação de transformação do modelo.

09

Uma ferramenta de MDA pode ser classificada de acordo com os seguintes **tipos**:

- Ferramenta de Criação;
- Ferramenta de Análise;
- Ferramenta de Transformação;
- Ferramenta de Composição;
- Ferramenta de Teste;
- Ferramenta de Simulação;
- Ferramenta de Gerenciamento de Metadados;
- Ferramenta de Engenharia Reversa.

Algumas ferramentas podem executar mais de uma função. Por exemplo, algumas ferramentas de criação também podem ter capacidade de transformação e de teste. Existem outras ferramentas que são exclusivamente para a criação, para fins de apresentação gráfica ou para transformação.

Discutiremos a seguir alguns **exemplos de ferramentas** que suportam o MDA. Estas ferramentas estão ligadas à plataforma J2EE/JAVA em virtude de sua ampla utilização do MDA.

Ferramenta de Criação

Ferramentas usadas para criar modelos iniciais ou editar modelos derivados.

Ferramenta de Analise

Ferramenta usada para verificar modelos para a completude, inconsistências ou erros e condições que necessitam de algum tipo de alerta. Também é usado para calcular as métricas para o modelo.

Ferramenta de Transformação

Ferramenta usada para transformar modelos em outros modelos ou em código e documentação.

Ferramenta de Composição

Ferramenta utilizada para compor vários modelos de origem, de preferência em conformidade com o mesmo metamodelo.

Ferramenta de Teste

A ferramenta utilizada para testar modelos.

Ferramenta de Simulação

Uma ferramenta usada para simular a execução de um sistema representado por um determinado modelo. Isto está relacionado com o tema da execução do modelo.

Ferramenta de Gerenciamento de Metadados

A ferramenta destina-se a lidar com as relações entre os diferentes modelos, incluindo os metadados de cada modelo, por exemplo, autor, data de criação ou modificação, método de criação.

Ferramenta de Engenharia Reversa

Ferramenta que se destina a transformar código-fonte legado em modelos permitindo identificar os componentes do sistema e o seus inter-relacionamentos e criar representações do sistema em níveis mais altos de abstração.

10

3.1 – AndroMDA

AndroMDA é um framework MDA de código aberto. Ele tem uma arquitetura de *plug-in* em que plataformas e componentes de suporte podem ser trocados em qualquer momento.

O AndroMDA explora fortemente projetos existentes de código aberto para questões específicas da plataforma e para serviços gerais de infraestrutura.

No AndroMDA, os desenvolvedores podem estender a linguagem de modelagem existente por meio de instalações conhecidas como "**metafacades**". A extensão é refletida como um perfil UML na modelagem de bibliotecas e modelos em ferramentas de transformação.

3.2 - ArcSyler

ArcStyler é uma das principais ferramentas comerciais do mercado para MDA. Ela suporta J2EE e .NET de forma nativa.

ArcStyler utiliza as suas próprias marcações MDA na UML como uma forma de introduzir informação sobre a plataforma sem poluir o modelo com detalhes do nível de plataforma.

Como AndroMDA, ArcStyler suporta cartuchos extensíveis para geração de código. Os próprios cartuchos também podem ser desenvolvidos dentro do ambiente ArcStyler seguindo os princípios do MDA. A ferramenta também suporta transformação de modelo por meio de arquivos externos de regra de transformação.

11

3.3 – Eclipse Modelling Framework (EMF)

A ligação inseparável entre modelos MDA e o código criado por meio de geração de código exige uma gestão consistente de modelos e código em um único IDE.

EMF é a sofisticada estrutura de metamodelagem e modelagem atrás da IDE Eclipse.

Embora o EMF só tenha sido lançado publicamente como um subprojeto Eclipse em 2003, tem uma longa herança como um mecanismo de gerenciamento de metadados da IDE VisualAge da IBM.

A forte integração do EMF com a IDE Eclipse suporta a integração de metadados diferentes em várias ferramentas que colaboraram em um ecossistema comum baseado em Eclipse. Isso aumenta o nível de interoperabilidade das ferramentas sendo amplamente compatível com as práticas de MDA.

12

4 – MDA E A ARQUITETURA DE *SOFTWARE*

A maioria dos modelos em MDA são essencialmente representações de uma arquitetura de *software*. Em um sentido amplo, os modelos de domínio e modelos de sistema são abstrações e diferentes pontos de vista dos modelos de arquitetura de *software*. Modelos de geração de código possuem as

características dos modelos de arquitetura, juntamente com detalhes de implementação. O código pode de fato ser usado em ferramentas de engenharia reversa para reconstruir a arquitetura da aplicação.

A arquitetura de *software* pode ser descrita em uma linguagem de descrição de arquitetura (ADL). Houve muitas ADLs desenvolvidas nos últimos anos, cada um com sua expressividade focada em diferentes aspectos dos sistemas de *software* e domínios de aplicação. Muitos recursos úteis de uma ADL foram absorvidos pela UML, ou especificados como extensões da UML. Assim, a UML é usada em MDA como um ADL.

13

4.1 – MDA e os Requisitos Não Funcionais

Requisitos não funcionais são uma grande preocupação da arquitetura de *software*. Estes requisitos contemplam exigências relacionadas a atributos de qualidade como:

- desempenho,
- reutilização,
- interoperabilidade e
- segurança.

Embora MDA não aborde cada atributo individualmente, ele promove e ajuda a alcançar estes atributos de qualidade devido aos seguintes fatores:

- Um certo grau de interoperabilidade, reusabilidade e portabilidade é construído em todos os modelos.
- O MOF e o mecanismo de UML Profiles permitem que a UML seja estendida para modelar requisitos e endereçando especificamente os requisitos não funcionais.
- Junto com as extensões para requisitos não funcionais para modelagem de projeto, modelos que mapeiam regras endereçam os atributos de qualidade durante a transformação destes modelos.

14

4.2 – Transformação de Modelos e a Arquitetura de *Software*

Uma grande parte da arquitetura de *software* diz respeito à forma de conceber e validar uma arquitetura que atenda aos seus requisitos e seja fiel ao *design*. Um obstáculo importante no desenho da arquitetura é a dificuldade de concepção que capta claramente como os vários aspectos do *design*

satisfazem os requisitos. Por esta razão, pode ser difícil sistematicamente validar se modelos de arquitetura satisfazem os requisitos, uma vez que a rastreabilidade entre os requisitos e os elementos de *design* não está formalizada.

No MDA, todas as linguagens de modelagem são bem definidas pela sintaxe e semântica em um metamodelo. O processo de transformação de um modelo (por exemplo, requisitos) para outro modelo (por exemplo, *design*) é um processo sistemático, seguindo regras de transformação explicitamente definidas. Esta potencial automatização poderia melhorar muito a qualidade e a eficiência de validar um modelo de arquitetura.

15

RESUMO

MDA é um padrão amplamente utilizado na indústria de *software* e continua a evoluir. O MDA impacta diretamente as práticas de arquitetura de *software*, uma vez que requer que a equipe de arquitetura crie modelos formais de sua aplicação usando linguagens de modelagem rigorosamente definidos e ferramentas de apoio.

Isso significa que é necessário elevar o nível de abstração para modelos de arquitetura. A indústria de *software* tem buscado elevar os níveis de abstração no desenvolvimento de *software* e o MDA, através de seus modelos, é uma iniciativa para deixar o desenvolvimento em um nível ainda mais abstrato.

Existe uma série de ferramentas que suportam MDA. Estas ferramentas podem ser classificadas em: Ferramenta de Criação, Ferramenta de Análise, Ferramenta de Transformação, Ferramenta de Composição, Ferramenta de Teste, Ferramenta de Simulação Ferramenta de Gerenciamento de Metadados e Ferramenta de Engenharia Reversa.

Os modelos de domínio e modelos de sistema, definidos na MDA, são abstrações e diferentes pontos de vista dos modelos de arquitetura de *software*. O MOF e o mecanismo de UML Profiles permitem que a UML seja estendida para modelar requisitos e endereçando especificamente os requisitos não funcionais.

UNIDADE 2 – ABORDAGENS DE DESENVOLVIMENTO BASEADO EM ARQUITETURA E ESTILOS DE ARQUITETURAS DE *SOFTWARE*

MÓDULO 2 – ARQUITETURA EM 3 CAMADAS

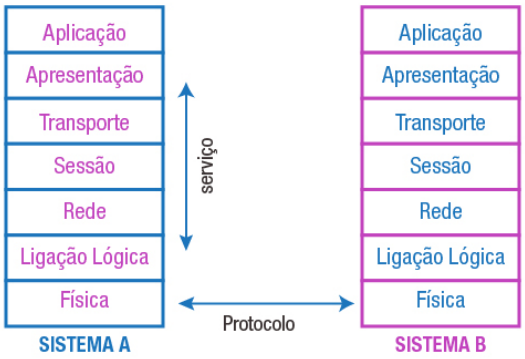
01

1 - O QUE É ARQUITETURA EM CAMADAS

A arquitetura em camadas existe desde os primórdios da computação, ou pelo menos desde o início dos anos 1960. Para evidenciar a importância da arquitetura em camadas para a tecnologia, vamos utilizar dois exemplos que estão a nossa volta.

Primeiro exemplo: Modelo OSI (Open Systems Interconnection) de 7 camadas da ISO (International Standards Organization).

Este modelo consiste em duas pilhas de camadas, cada camada provê um nível mais alto de funcionalidades que o seu nível inferior e cada camada tem uma camada correspondente na outra pilha. A comunicação entre estas pilhas ocorre entre as camadas mais abaixo.



Cada camada pode ter tamanhos diferentes e apresentar um número diferente de protocolos.

02

Segundo exemplo: Sistemas Operacionais.

Sistemas Operacionais são exemplos de arquitetura em camadas. Eles possuem um kernel central cercado por várias camadas de funcionalidades, como os drivers dos periféricos. A camada mais externa contém as aplicações executadas pelos usuários.

A arquitetura em camadas de um sistema operacional facilita a adaptação de um novo dispositivo, pois apenas o *software* controlador de Periférico deve ser alterado quando um novo dispositivo é adicionado. Já as camadas mais acima não precisam se preocupar com o novo hardware.

Visto como a arquitetura em camadas está presente na tecnologia, vamos ser mais específicos. Como a arquitetura em camadas se apresenta na engenharia de *software*?

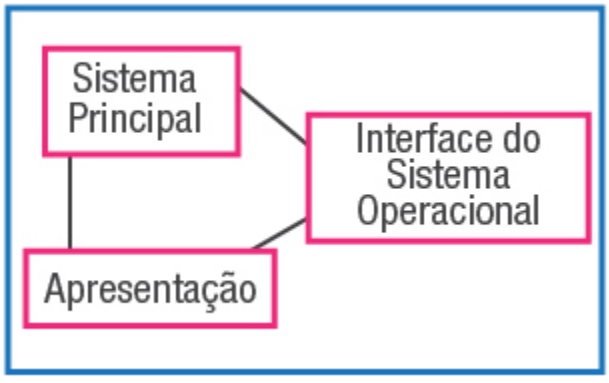
03

2- MOTIVAÇÃO PARA ARQUITETURA EM CAMADAS

Imagine que você é um arquiteto de *software* responsável pelo desenvolvimento de um sistema que apresenta o estado de um disco primário para usuário em seu computador. Este sistema deve criar um gráfico para o disco apresentando os blocos com cores diferentes para representar os blocos vazios, os que estão sendo utilizados, os que estão com algum tipo de problema e assim por diante.

Este sistema não deve ser difícil de ser implementado uma vez que você já tem o domínio de como esta apresentação funciona. Você rapidamente identifica quais as chamadas de sistemas devem ser utilizadas e desta forma todas as operações são rapidamente identificadas.

Pela facilidade que você encontrou para o desenvolvimento deste sistema, você opta por desenvolvê-lo de **forma monolítica**, ou seja, um grande sistema com todas as funcionalidades necessárias. O programa principal chama o componente de apresentação para apresentar as informações para o usuário, e este programa identifica cada bloco para apresentar o seu estado. Ao final você tem um grande programa que executa todas as operações. Este sistema pode ser representado de forma simplificada pela figura a seguir.



Entretanto, justamente quando você agendou a apresentação para o seu cliente, um novo requisito surgiu. Agora, o cliente gostaria de apresentar o mapeamento do disco para um outro sistema operacional. Com isso, todas as chamadas do programa principal para identificar o estado de cada bloco do disco deverão ser modificadas. Adicionalmente, o cliente ainda gostaria de apresentar o resultado

em um gráfico de pizza e não em um gráfico de barra como você havia implementado. O que fazer? Veja a seguir.

04

Diante deste cenário, você precisará dar manutenção em todo o seu programa principal, pois ele foi desenvolvido de forma monolítica.

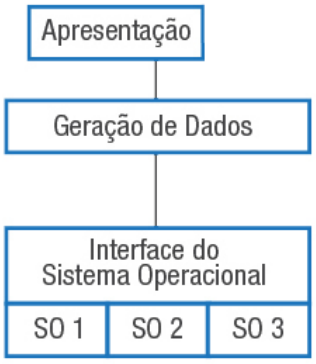
No momento de fazer nova apresentação para o cliente, novas alterações foram solicitadas e mais uma vez você precisará dar manutenção em seu programa principal para acomodar as mudanças solicitadas pelo cliente.

Entretanto, ao invés de sair alterando o programa principal, você resolve analisar a situação e fazer uma alteração estrutural em seu programa.

O seu programa monolítico agora será dividido em três partes distintas:

1. A parte mais alta apresenta as funções para a apresentação na tela do usuário.
2. A parte central é responsável pelo tratamento da informação, é ela que faz o acesso às funções do sistema operacional e com a apresentação.
3. A parte mais abaixo é responsável pelo serviço específico do sistema operacional.

O resultado desta sua alteração é apresentado na figura abaixo.



Note que a alteração fez com que o programa passasse de monolítico para uma **arquitetura em camada**, permitindo acomodar mudanças mais facilmente. Com isso, caso seja solicitada uma nova forma de apresentar as informações, basta que a camada superior seja alterada. Caso a alteração seja solicitada para o sistema operacional a ser exibido, basta que as camadas central e a mais baixa sejam alteradas, sem a necessidade de alterar a camada de apresentação. Assim, novas alterações não necessitarão que todo o sistema seja alterado.

05

3- Benefícios da Arquitetura em Camadas

O benefício mais importante da arquitetura em camadas é a **manutenibilidade**. Cada camada interage apenas com a camada adjacente, primordialmente com a camada abaixo dela. Esta característica permite que uma camada possa ser alterada, estendida e até mesmo trocada sem afetar as demais camadas.

Outros benefícios associados à utilização da arquitetura em camadas serão apresentados a seguir.

- **Facilita o reuso**

Todos os aspectos que tornam a arquitetura em camadas fácil de ser utilizada entendida e mantida contribuem para torná-la propícia para o reuso. Cada camada tem uma abstração bem definida tornando-a mais compreensível e permitindo que camadas possam ser reutilizáveis.

A funcionalidade fornecida por uma camada é bem determinada.

A interface bem definida de uma camada permite que ela seja reutilizável para resolver problemas similares.

Desenvolvedores resistem em reutilizar camadas existentes, pois eles preferem escrever um componente para resolver o seu problema. O argumento para isso é que uma camada existente não atende toda a necessidade dele. Desta forma, é importante ter em mente que o reuso é importante para beneficiar o desenvolvedor tornando menor o tempo para o desenvolvimento. Caso o tempo para adaptar uma camada existente seja maior do que construir um novo, é melhor construir um novo.

06

- **Fornecer agrupamentos padronizados de abstração e interfaces para uma arquitetura em camadas.**

Os padrões da indústria são facilmente adaptáveis às arquiteturas em camadas. Os padrões ajudam diferentes grupos ou empresas a produzir sistemas que irão trabalhar em conjunto. Camadas que são compatíveis aos padrões podem ser utilizados e reutilizados de forma intercambiável.

- **As dependências entre as camadas são minimizadas.**

Esta estrutura faz com que seja mais fácil de isolar alterações no código, quando os requisitos mudam. Todas as mudanças na camada de apresentação, por exemplo, são restritas a esta camada, as mudanças na camada de controle ficam restritas a esta camada e assim por diante.

- **Uma camada pode ser facilmente trocada com outras implementações desta camada.**

Implementações das camadas individuais que satisfaçam a mesma abstração e as mesmas interfaces são intercambiáveis sem muito esforço.

Se as interfaces são *hard coded*, você pode substituir os nomes antigos com os novos nomes com bastante facilidade. Se isso não puder ser feito, você ainda pode reutilizá-las usando um adaptador para ligar a camada existente com a nova camada.

- **Facilitam projetos de desenvolvimento complexos.**

A utilização de camadas permite dividir o trabalho entre vários desenvolvedores ou equipes de desenvolvimento para trabalhar em paralelo.

Padrões

A arquitetura em camadas, uma vez que tenham camadas claramente definidas juntamente com suas interfaces, também pode conduzir à definição de padrões. Os órgãos que definem os padrões necessitam de exemplos reais. Desta forma, um sistema bem estruturado pode servir de exemplo para os órgãos de definição de como os padrões devem ser utilizados.

07

Na decisão de adotar uma arquitetura em camadas, certas características devem ser levadas em consideração. Estas características devem ser consideradas e comparadas aos benefícios para saber se uma arquitetura em camadas irá realmente ser adequada na solução do problema que você está tentando resolver.

Estas **características** estão listadas a seguir:

- Camadas não são tão eficientes quanto às conexões codificadas dentro de uma solução monolítica.
- Mudanças em uma camada podem afetar outras camadas.
- Camadas podem representar um trabalho desnecessário.
- Arquiteturas em camadas não têm uma estrutura obrigatória para as camadas.

Camadas não são tão eficientes quanto as conexões codificadas dentro de uma solução monolítica.

Em uma solução monolítica, um componente no mais alto nível de abstração pode chamar uma função direta no nível mais baixo. Esta não é possível numa arquitetura em camadas, porque todas as camadas intermediárias estão envolvidas na chamada. Cada chamada realizada entre diferentes camadas incorre em uma ligeira perda de desempenho para o processamento.

Mudanças em uma camada pode afetar outras camadas.

Embora una arquitetura em camadas normalmente evite que alterações em uma camada comprometam o funcionamento das demais camadas, em alguns casos isso não é possível. Nestes casos, será necessário reestruturar outras camadas em virtude de uma mudança em outra camada.

Camadas podem representar um trabalho desnecessário.

Esta situação pode ocorrer quando várias camadas fornecem redundância para apoiar confiabilidade aos níveis superiores. Estas múltiplas camadas podem adicionar várias verificações nas mensagens, quando uma única verificação seria suficiente.

Arquiteturas em camadas não têm uma estrutura obrigatória para as camadas.

Consequentemente, alguns sistemas em camadas podem ter muita funcionalidade em poucas camadas, o que torna mais difícil de reutilizar as camadas e mais difícil de compreender o objetivo de cada camada. Outra desvantagem é que um designer pode criar muitas camadas, aumentando assim a sobrecarga associada com a arquitetura em camadas. Dividir um sistema no número de camadas mais adequado é o maior desafio desta arquitetura.

08

4- ARQUITETURA EM 3 CAMADAS

Você não pode simplesmente pegar o código fonte de um aplicativo, cortá-lo em diferentes partes e chamar cada parte camada. É necessário identificar áreas específicas de responsabilidade, em seguida, identificar o código que funciona dentro de cada uma dessas áreas.

As **três camadas** mais comuns são:

Apresentação lógica

•A apresentação lógica, interface de usuário ou, simplesmente, UI exibe dados e aceita entrada do usuário. Em um aplicativo de web é esta parte que recebe o pedido HTTP e retorna a resposta HTML.

Lógica de negócios

•A lógica de negócio ou camada de negócio manipula a validação de dados, regras de negócios e comportamento específico de tarefa.

Lógica de acesso a dados

•A Lógica de Acesso a Dados ou Camada de Persistência é responsável pela comunicação com o banco de dados através da construção de consultas SQL e executá-los através da API relevante.

Embora seja possível tomar qualquer das áreas acima e dividi-las em componentes ainda menores, o excesso de camadas pode ser imprudente, pois você pode afogar-se em pequenos detalhes e perder de vista o quadro geral.

09

Assim a figura a seguir apresenta a arquitetura típica em três camadas, onde a apresentação é a camada superior, a camada central trata da lógica do negócio e a camada de banco de dados é a mais abaixo. Cada camada tem uma responsabilidade distinta e bem definida.

Note também que você não deve inferir, a partir deste esquema, que o aplicativo inteiro pode ser construído com um único componente em cada uma dessas três camadas. Deve ter várias opções como se segue:

- Deve haver um componente separado na camada de apresentação para cada transação de usuário.
- Deve haver um componente separado na camada de negócios para cada entidade de negócios (tabela de banco de dados).
- Deve haver um componente separado na camada de acesso a dados para cada SGBD suportados.

Com essa estrutura, é fácil substituir o componente em uma camada com outro componente sem ter que fazer qualquer alteração em outras camadas.

Essa estrutura também fornece mais capacidade de reutilização, como: um único componente na camada de negócios pode ser compartilhado por vários componentes na camada de apresentação. Isso significa que a lógica de negócios pode ser definida em um lugar ainda compartilhado por vários componentes.

10

4.1- As regras da arquitetura de três camadas

Uma arquitetura em três camadas deve conter algumas regras. Estas regras não são complexas, mas devem ser seguidas para que você possa efetivamente se beneficiar deste tipo de arquitetura.

As **regras** estão listadas a seguir:

- Código de cada camada deve estar em arquivos separados, para que possam ser mantidos separadamente.
- Cada camada deve conter código que pertence a essa camada. Assim, a lógica do negócio só pode residir na camada de negócios, lógica de apresentação na camada de apresentação e lógica de acesso a dados na camada de acesso a dados.
- A camada de apresentação só pode receber solicitações e retornar respostas a um agente externo.
- A camada de apresentação só pode enviar solicitações e receber respostas da camada de negócios. Ele não pode ter acesso direto ao banco de dados ou à camada de acesso a dados.
- A camada de negócio só pode receber solicitações e retornar a resposta para a camada de apresentação.
- A camada de negócio só pode enviar solicitações e receber respostas da camada de acesso a dados. Ele não pode acessar o banco de dados diretamente.

- A camada de acesso a dados só pode receber solicitações e retornar respostas para a camada de negócios.
- Cada camada deve ser totalmente inconsciente do funcionamento interno das outras camadas.

11

4.2- Os benefícios da arquitetura de três camadas

Quando falamos de benefícios da arquitetura em 3 camadas não podemos nos limitar em analisar apenas pelo fato de tornar a nossa vida mais fácil quando precisamos alterar seu mecanismo de banco de dados ou linguagem de programação. Também é útil quando você quiser substituir a camada de apresentação ou criar uma camada de apresentação adicional.

As **principais vantagens** da arquitetura de três séries frequentemente citadas são:

- Flexibilidade;
- Facilidade de manutenção;
- Reutilização;
- Escalabilidade;
- Confiabilidade.

Outro benefício não tão óbvio que só pode vir de exposição real para ter desenvolvido vários aplicativos usando a arquitetura de três camadas é que se **torna possível a criação de um quadro para a construção de novas aplicações** em torno dessa arquitetura. Como cada uma das camadas é especializada em apenas uma área do aplicativo, é possível ter mais componentes reutilizáveis que lidam com cada uma dessas áreas. Esses componentes podem ser pré-construídos e entregues como parte do quadro, ou gerado pelo próprio *framework*. Isto reduz a quantidade de esforço necessária para criar uma nova aplicação, e também reduz a quantidade de esforço necessária para manter essa aplicação.

Flexibilidade

Ao separar a lógica de negócios de um aplicativo a partir de sua lógica de apresentação, uma arquitetura de 3 camadas torna o aplicativo muito mais flexível a mudanças.

Facilidade de manutenção

Alterações dos componentes em uma camada não deve ter nenhum efeito sobre quaisquer outras camadas. Além disso, se diferentes camadas exigem diferentes habilidades (como HTML / CSS é a camada de apresentação, PHP / Java na camada de negócios, SQL na camada de acesso a dados), então estes podem ser geridos por equipes independentes com competências nessas áreas específicas.

Reutilização

Separar o aplicativo em várias camadas torna mais fácil de implementar componentes reutilizáveis. Um único componente na camada de negócios, por exemplo, pode ser acessado por vários componentes na camada de apresentação, ou mesmo por várias camadas de apresentação diferentes (como o desktop e web), ao mesmo tempo.

Escalabilidade

A arquitetura de 3 camadas permite a distribuição de componentes do aplicativo em vários servidores, tornando assim o sistema muito mais escalável.

Confiabilidade

A arquitetura de 3 camadas, se implantada em vários servidores, torna mais fácil para aumentar a confiabilidade de um sistema através da implementação de vários níveis de redundância.

12

RESUMO

Uma das mais importantes e populares arquiteturas de software é a arquitetura em camada. Existe desde os primórdios da computação, ou pelo menos desde o início dos anos 60 esta arquitetura permitiu que os programas passassem de monolíticos para uma arquitetura em camadas.

O maior benefício da arquitetura em camadas é a manutenibilidade, ou seja, sistemas que utilizam esta arquitetura permitem absorver mudanças de forma mais fácil. Adicionalmente, outros benefícios na utilização desta arquitetura são:

- Facilita o reuso;
- Fornecer agrupamentos padronizados de abstração e interfaces para uma arquitetura em camadas;

- As dependências entre as camadas são minimizadas;
- Uma camada pode ser facilmente trocada com outras implementações desta camada;
- Facilitam projetos de desenvolvimento complexos;
- Camadas não são tão eficientes quanto às conexões codificadas dentro de uma solução monolítica;
- Mudanças em uma camada podem afetar outras camadas;
- Camadas podem representar um trabalho desnecessário.

Mesmo com estes benefícios algumas características adversas podem ser observadas em sistemas que utilizam a arquitetura em camadas. Estas características estão listadas a seguir:

- Arquiteturas em camadas não têm uma estrutura obrigatória para as camadas;
- Deve haver um componente separado na camada de apresentação para cada transação de usuário;
- Deve haver um componente separado na camada de negócios para cada entidade de negócios (tabela de banco de dados);
- Deve haver um componente separado na camada de acesso a dados para cada SGBD suportados.

A implementação mais popular da arquitetura em camadas é a arquitetura em 3 camadas, em que, tipicamente, a apresentação é a camada superior, a camada central trata da lógica do negócio e a camada de banco de dados é a mais abaixo.

Mas para atingir os benefícios desta arquitetura algumas regras devem ser seguidas. Entre os benefícios estão:

- Flexibilidade;
- Facilidade de manutenção;
- Reutilização;
- Escalabilidade;
- Confiabilidade.

UNIDADE 2 – ABORDAGENS DE DESENVOLVIMENTO BASEADO EM ARQUITETURA E ESTILOS DE ARQUITETURAS DE *SOFTWARE*

MÓDULO 3– ARQUITETURA MVC

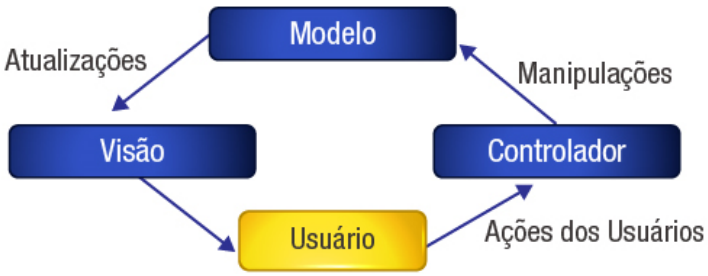
01

1 - O QUE É ARQUITETURA MVC

Uma das arquiteturas mais populares para o desenvolvimento de aplicações Web é a MVC.

MVC, que significa Model-View-Controller, ou Modelo-Visão-Controlador, é um padrão arquitetural que divide um *software* em partes interconectadas para separar a forma interna das informações de como elas são apresentadas para os usuários.

O conceito por trás do MVC é isolar a lógica de negócio da interface do usuário. O Modelo contém as regras de negócio utilizadas para manipular os dados e as informações da aplicação. A visão são os elementos da interface do usuário, tais como texto, itens de caixa de seleção, e assim por diante. Já o controlador gerencia detalhes envolvendo a comunicação entre o modelo e a visão. É o controlador que manipula as ações do usuário, como movimentos do mouse.



Típica colaboração entre os componentes do MVC

Os conceitos por trás desse padrão foram desenvolvidos por Trygve Reenskaug nos anos 70. Mas muitas pessoas tomaram conhecimento deste padrão através da língua Smalltalk já nos anos 80, quando Jim Althoff implementou uma versão do MVC para esta linguagem.

02

1.1- Entendendo o Problema

Para que possamos entender a arquitetura MVC vamos partir de um problema exemplo.

Você foi designado para desenvolver um sistema para apoiar a pesquisa de algumas espécies de animais selvagens. Nesta fase da pesquisa os pesquisadores coletaram uma série de informações sobre o lobo guará do cerrado brasileiro, um animal ameaçado de extinção. Estas informações incluem:

Registros de nascimento e morte

- Datas de nascimento e morte
- Identificação e referências cruzadas para irmãos e pais
- Referências cruzadas para locais de nascimento e morte

Informações de localização

- Território geral
- Localização
- Localização Local de nascimento e da morte
- Gama geral e padrões de viagem

Família informações agrupamento

- Pais, filhos, irmãos
- Companheiro

Encontros com pessoas

- Relatórios incômodo
- Onde foram avistados
- Relatórios de pessoas dentro do território dos lobos

População

- Informações gerais censo
- Tendências da população (taxa de fertilidade e taxa de mortalidade infantil)

03

Seus clientes, os pesquisadores, pediram-lhe para preparar um sistema de pesquisa para que eles possam examinar os dados coletados. Eles querem usar alguns cruzamentos de informações que eles já previamente identificaram, mas é importante você imaginar que eles não tenham pensado em todos os pontos de vista possíveis e até mesmo úteis para a pesquisa que eles estão realizando. Desta forma, o sistema que você irá construir deve ser extensível.

Os clientes querem também uma nova interface gráfica (GUI) para o sistema, que lhes permita selecionar o que eles veem e controlar praticamente todo o sistema através do clique do mouse.

Diante do problema proposto você precisa refletir sobre as principais **partes do sistema** que será construído. São elas:

- **Dados;**
- **Componente de Controle;**
- **Visualizações.**

Identificadas as principais partes do sistema a ser implementado, é iniciada a implementação do mesmo.

Dados

Os dados são os componentes primários. Você está familiarizado com a maneira com que os dados são armazenados: em um banco de dados simples. Alguns dos dados que os clientes desejam visualizar, no entanto, não são armazenados diretamente na base de dados; eles são calculados a partir de outros dados no banco de dados. Esta capacidade computacional é construída em componentes que ficam bem em cima do banco de dados.

Componente de Controle

Outra parte do sistema é o componente de controle de interface de usuário (UI). Esta parte interage com o usuário, tendo informações sobre os dados e o formato dos dados que o usuário deseja ver.

Visualizações

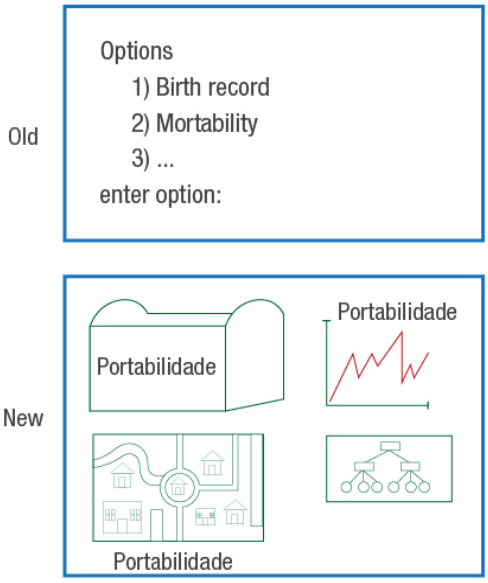
Você sabe sobre algumas das visualizações que os cientistas vão querer ver, e você já as projetou para serem componentes separados do sistema. Todas as visualizações são muito semelhantes, assim que você coletá-los em uma terceira parte do sistema: a visão.

04

1.2- Alterando a interface do usuário

Enquanto você estava implementando o sistema, os cientistas solicitaram uma grande mudança na interface do sistema. Eles pretendem adicionar uma nova maneira de interagir com todo o sistema. Eles

querem abandonar a visão principal com uma lista de menu para um novo sistema baseado em ícones, que lhes permite arrastar conjuntos de dados para exibir visualizações.



Mostra esses dois estilos de interface do usuário

Para fazer essa alteração, você precisa **mudar a estrutura superior** da interface do usuário. Todos os outros aspectos do sistema, incluindo a visualização de dados, não precisa mudar.

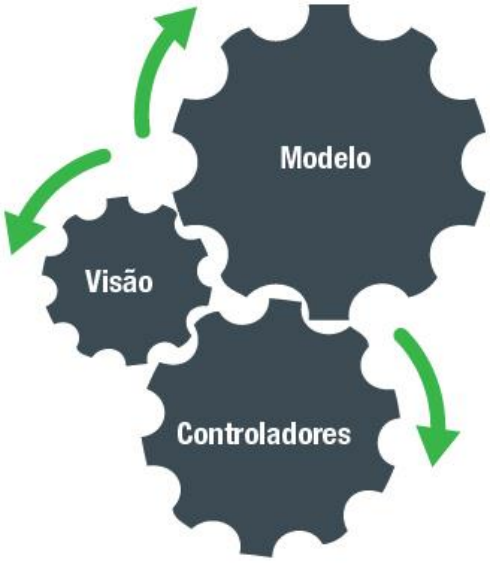
A arquitetura selecionada para o desenvolvimento do sistema pode ajudar, e muito, nas alterações solicitadas pelos usuários.

05

1.3- A Solução

Conforme vimos no início deste módulo, o MVC tem o objetivo de isolar a lógica de negócio da interface do usuário. Desta forma, construindo um Sistema Modelo-Visão-Controlador as mudanças solicitadas pelos usuários teriam um impacto menor no sistema.

Para construir um sistema MVC, divida seu aplicativo em três componentes:



Modelo

Que contém a funcionalidade e dados núcleo.

Visão

Que exibem informações para o usuário.

Controladores

Que lidam com a entrada do usuário e passam para a visualização o que o usuário quer exibir.

06

2- Os Benefícios do MVC

O padrão MVC tem sido usado muitas vezes para estruturar sistemas interativos. Os arquitetos que utilizam este padrão arquitetural citam a flexibilidade de dissociar os dados (Modelo) da saída (Visão) e entrada (Controladores) como principal benefício desta arquitetura.

Entretanto, podemos citar alguns **benefícios adicionais**:

- **O modelo é separado dos componentes de interface de usuário nesta arquitetura.**

Você pode usar os mesmos dados para suportar diferentes visões.

- **As alterações nos dados no modelo são refletidas em todas as visões de forma automática.**

Isso é possível porque existe uma única fonte de dados que estão sendo exibidos.

- **Você pode alterar a visão e o controlador sem alterar o modelo de dados.**

Esse recurso aumenta a flexibilidade do sistema. Você pode manter o elemento de modelo subjacente consistente e intacto, e trocar a visão e o controlador do sistema.

- **Como o código de interface do usuário é independente do modelo, quando você precisar fazer grandes mudanças na interface do usuário, os dados não precisam mudar.**

Este tipo de mudança pode ser resultante, por exemplo, da mudança do sistema para um novo hardware.

- **As visões não interagem.**

Como resultado, você pode alterar uma visão, sem ter que fazer mudanças em outra visão.

- **Arquiteturas MVC podem ser usadas como um framework a serem utilizadas e estendido em outras situações.**

Os três componentes estão relacionados, mas independentes, o que simplifica a manutenção e evolução.

07

3- CARACTERÍSTICAS ADVERSAS

Além dos benefícios, algumas características inerentes ao MVC podem ser encaradas como desvantagens. Desta forma, ao considerar a arquitetura MVC para o seu sistema, estas **características** devem ser equilibradas com os benefícios:

- Complexidade é aumentada através da separação dos três componentes do MVC.
- Alterações no modelo são publicadas em todas as visões relacionadas a estes modelos.

- O controlador e a visão crescem com o tempo.
- O controlador e a visão são muito dependentes do modelo.
- Acesso a dados de forma ineficiente causada pela separação da visão e do modelo onde sempre será necessário que a visão chame o modelo.
- MVC surgiu antes da criação de ferramentas de interface do usuário mais modernas.

Complexidade é aumentada através da separação dos três componentes do MVC.

Você tem mais componentes para construir e manter do que você teria se tivesse projetado o sistema como um monólito. A menos que você precise de flexibilidade na interface do usuário ou nas visões, o MVC pode adicionar mais sobrecarga do que sua aplicação realmente precisa.

Alterações no modelo são publicados em todas as visões relacionadas a estes modelos.

À medida que o sistema se torna maior, mais relacionamentos entre modelo e visões existem, e uma mudança em um modelo ou na assinatura da mensagem enviada pelo modelo todas as visões que interagem com ele deverão sofrer manutenção.

O controlador e a visão crescer com o tempo.

Mesmo que os componentes sejam únicos, eles têm relações fortes que limitam a sua capacidade de reutilização de um controlador com a sua visão. À medida que o sistema cresce e evolui, visões são adicionadas ao sistema, juntamente com as consequentes melhorias no controlador que permitirão que as visões possam ser utilizadas e controladas. Uma vez que os componentes são muito dependentes, reutilizar apenas o controlador ou apenas os a visão é mais difícil do que reutilizar o controlador e a visão juntos. Esta característica também limita a sua capacidade de inserir uma nova versão de qualquer um dos componentes, porque a nova versão deve ser adaptada para suportar o componente que não está sendo substituído.

O controlador e a visão são muito dependentes do modelo.

Mudanças no modelo podem exigir mudanças em ambos os outros dois componentes. A utilização de padrões de design, como o Command Processor, evita este forte acoplamento entre o modelo e os demais componentes.

Acesso a dados de forma ineficiente causado pela separação da visão e do modelo onde sempre será necessário que a visão chame o modelo.

Este problema é especialmente aparente se a visão tiver que acessar os mesmos dados, sem alteração, com frequência. O desempenho pode ser melhorado através de estratégia de cache de informações pela visão.

MVC surgiu antes da criação de ferramentas de interface do usuário mais modernas.

MVC é útil para aumentar a portabilidade. Se a portabilidade não é um requisito essencial, a utilização de um kit de ferramentas de interface pode ser uma solução mais apropriada. Estas duas soluções são incompatíveis porque toolkits que se especializam em, ao criar interfaces de usuário, incluem seu próprio fluxo de controle e seus próprios mecanismos de acesso ao modelo, ao passo que os controladores MVC querem controlar a maneira que a interação do usuário ocorre.

08

4 - Detalhando o MVC

Até agora, vimos as partes do MVC em termos gerais. Nesta seção, vamos detalhar as funções e responsabilidades das três partes do MVC:

- Modelo,
- Visões, e
- Controladores.

4.1- O Modelo

O Modelo contém o núcleo da aplicação – tanto os dados da aplicação como a funcionalidade relacionada a dados importantes. O modelo fornece procedimentos e métodos para acessar os dados.

Estes procedimentos e métodos são chamados pelo controlador em resposta a um comando do usuário. O modelo também fornece funções para acessar os dados armazenados no modelo que as visões precisam construir suas apresentações.

O modelo deve manter os dados que ele armazena atualizados, por isso deve ter mecanismos para atualizar os dados internamente e para reportar as atualizações para as visões que utilizam estes dados. Frequentemente, este mecanismo de mudanças de propagação é implementado através do padrão Publisher-Subscriber.

09

Uma variação do modelo é para que permaneça passivo e não publicar atualizações. Nesta variante, as visões e controladores que “perguntam” ao modelo por atualizações em lugar de subscrever e esperar por atualizações.

O cartão a seguir resume as principais características do Modelo:

Classe Modelo	**Colaboradores** Visões Controladores
Responsabilidades • Fornece as funcionalidades principais da aplicação. • Registram as visões e controladores interessados em seus dados. • Notifica os componentes registrados sobre atualização de dados.	

10

4.2- A Visão

Informações do modelo são exibidas para os usuários através dos componentes de visão. Um sistema pode ter mais de uma visão. Cada visão fornece ao usuário maneiras diferentes de visualizar os dados.

As visões recebem dados atualizados dos modelos assinando o modelo de publicação de mudanças. Quando os dados atualizados são recebidos, todas as visões atualizam o que estão mostrando ao usuário.

Durante a inicialização, todas as visões se registram no modelo de publicação de mudanças, garantindo que as visões estarão com os dados atualizados.

Existe uma relação de um-para-um entre as visões e os controladores – ou seja, cada visão tem um controlador. Cada visão também pode ter subvisões. Em um aplicativo comum, botões, barras de rolagem e menus são subvisões. Uma hierarquia de visões e controladores fornecem comportamentos e as interfaces que o usuário espera.

O cartão a seguir resume as principais características da Visão:

Classe Visão	**Colaboradores** Modelo Controladores
Responsabilidades • Cria e inicializa os seus controladores. • Exibem as informações para os usuários. • Atualiza-se quando dados atualizados são recebidos dos modelos. • Recupera dados do modelo.	

11

4.3- O Controlador

O controlador interage com o usuário e processa entradas do usuário como eventos. Quando os eventos chegam, o controlador verifica para ver se o evento lhe é aplicável; se isso acontecer, o controlador processa o evento. Se o evento não é relevante para o controlador, o controlador não toma nenhuma providência.

Em certas ocasiões, o comportamento do controlador depende do estado do modelo. Nesse caso, o controlador deve se registrar no método de propagação de mudança do modelo, assim como a visão o faz. Este registo é exigido quando a presença de certos dados no modelo pode permitir a criação de novos itens de menu, por exemplo.

Visões podem ter mais de um controlador. Alguns elementos de tela podem ser editados enquanto outros não podem. Nesse caso, os controles para esses elementos podem ser colocados em controladores separados.

O cartão a seguir resume as principais características do Controlador:

Classe	**Colaboradores**
Controlador	Visões Modelos
Responsabilidades • Recebe os *input*s dos usuários como eventos. • Transforma os eventos em requisições para os modelos e solicitações de exibição para as visões. • Atualiza-se quando recebe dados atualizados dos modelos.	

12

4.4- Trabalhando em conjunto

Vimos o objetivo de cada parte do MVC, agora vamos ver como cada parte trabalha em conjunto para atingir o objetivo da aplicação.

Vamos imaginar uma página web, esta página interage com o usuário através de um navegador web. Então, temos um desenho como o da figura a seguir:

Esta é uma visão simplificada, pois esta página pode conter uma infinidade de códigos, fazer uma série de decisões e ainda retornar informações para o navegador. Pode até mesmo interagir com o banco de dados para obter informações relevantes para o usuário. Desta forma, nosso desenho ficaria como a figura abaixo:

Você, como arquiteto de *software*, questiona se ao invés de ter uma única página com todo este código não seria melhor separar as responsabilidades. Mas, como isso pode ser feito?

Vejamos a seguir.

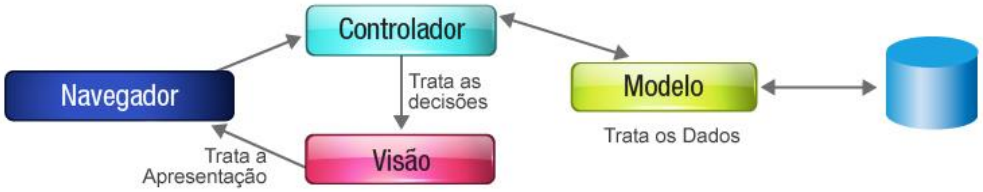
13

Através o MVC, o primeiro passo é ter um Controlador. Esta classe é responsável por receber as requisições do navegador e toma a decisão do que deve ocorrer com base nas requisições recebidas.

Caso estas requisições estejam relacionadas de alguma forma com os dados, o controlador aciona o Modelo, que fará a interação com o Banco de dados e resolverá toda a regra de negócio relacionada com o dado em questão e então retornará para o controlador.

Assim que o controlador estiver com todas as informações de retorno para o Navegador, ele acionará a visão. A visão tomará a decisão de que HTML, CSS e JavaScript deverá ser utilizado para apresentar as informações para o Navegador.

Temos, portanto, três componentes diferentes para tratar nossa interação com o Navegador, onde o Controlador trata as decisões, o Modelo trata os dados e a visão trata a apresentação. Com isso, nossa figura ficaria assim:



Desta forma, teremos o código separado nas partes corretas.

14

5- O MVC E A ARQUITETURA EM TRÊS CAMADAS SÃO A MESMA COISA?

Alguns programadores, ao ouvir que o sistema está dividido em três áreas de responsabilidade, automaticamente assumem que estas responsabilidades são as mesmas entre o Model-View-Controller (MVC) e a arquitetura em três camadas. Mas isso não é verdade!

Embora haja semelhanças, também existem algumas **diferenças** importantes:

- A Visão e o Controlador ambos se encaixam em uma única camada, a camada de apresentação.
- Embora o modelo e a camada de negócio pareçam ser idênticas, o padrão MVC não tem um componente separado dedicado para acesso a dados.

As sobreposições e as diferenças são mostradas na figura abaixo:

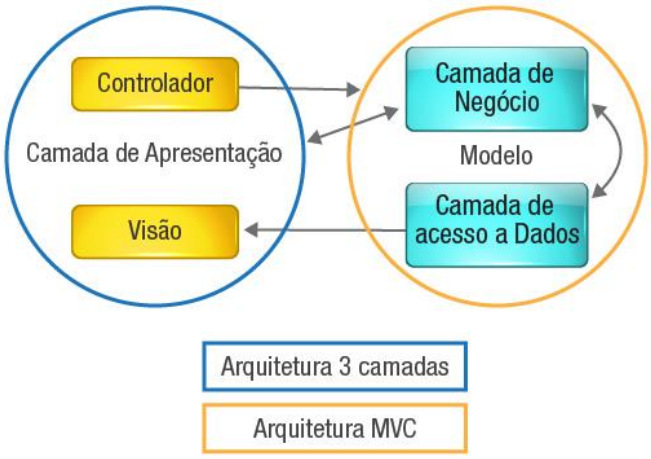


Figura 8 - A MVC e 3 camadas arquiteturas combinadas

Você pode pensar que qualquer implementação de MVC poderia automaticamente ser considerada uma implementação de três camadas, mas este não é o caso. Em cada implementação de MVC existe uma característica fundamental, que torna isso impossível, e é aí que a conexão do banco de dados é feita.

Na arquitetura de três camadas toda a comunicação com o banco de dados, e isso inclui a abertura de uma conexão, é feita dentro da camada de acesso a dados após o recebimento de um pedido da camada de negócios. A camada de apresentação não tem qualquer comunicação com o banco de dados, que só pode se comunicar com ela através da camada de negócios. Os frameworks MVC fazem a conexão do banco de dados dentro do controlador, e o objeto de conexão é então transmitido ao modelo que, em seguida, usa-o quando necessário.

15

Resumo

Neste módulo abordamos a MVC, uma das arquiteturas mais populares para o desenvolvimento de aplicações Web.

MVC, ou Modelo-Visão-Controlador, é um padrão arquitetural que divide um *software* em partes interconectadas para separar a forma interna das informações de como elas são apresentadas para os usuários.

O MVC é composto por 3 partes:

O **Modelo** que contém o núcleo da aplicação – tanto os dados da aplicação e a funcionalidade relacionada a dados importantes. É o modelo que fornece os procedimentos e métodos para acessar os dados.

As principais responsabilidades do modelo são:

- Fornecer as funcionalidades principais da aplicação.
- Registrar as visões e controladores interessados em seus dados.
- Notificar os componentes registrados sobre atualização de dados.

A **Visão** exibe as informações provenientes do modelo para o usuário. As principais responsabilidades da visão são:

- Criar e inicializar os seus controladores.
- Exibir as informações para os usuários.
- Atualizar-se quando dados atualizados são recebidos dos modelos.
- Recuperar dados do modelo.

O **Controlador** trata as entradas (ações) do usuário e passa para a visualização o que o usuário quer que seja exibido.

As principais responsabilidades do controlador são:

- Receber os inputs dos usuários como eventos.
- Transformar os eventos em requisições para os modelos e solicitações de exibição para as visões.
- Atualizar-se quando recebe dados atualizados dos modelos.

Foram abordadas as principais diferenças entre o modelo em camadas e o MVC com destaque para os seguintes aspectos:

- A Visão e o Controlador do MVC se encaixam na camada de apresentação do modelo em camada; e
- O padrão MVC não tem um componente separado dedicado para acesso a dados como no modelo em camadas.

UNIDADE 2 – ABORDAGENS DE DESENVOLVIMENTO BASEADO EM ARQUITETURA E ESTILOS DE ARQUITETURAS DE *SOFTWARE*

MÓDULO 4 – ARQUITETURA ORIENTADA A SERVIÇO

01

1 - O QUE É ARQUITETURA ORIENTADA A SERVIÇO - SOA

Um dos modelos arquiteturais que está ganhando bastante força é o SOA.

O Service-Oriented Architecture, ou **Arquitetura Orientada a Serviço** é um padrão arquitetural no qual os componentes da aplicação fornecem serviços para outros componentes através de um protocolo de comunicação. O serviço é uma funcionalidade ou operação que pode ser evocada diretamente através de sua interface.

Após esta breve definição, vamos aos conceitos mais formais.

O Open Group define a Arquitetura Orientada a Serviços (SOA) como um estilo de arquitetura que suporta a orientação a serviços. A orientação a serviço é uma forma de pensar em termos de desenvolvimento baseado em serviços e os resultados dos serviços.

Para o Open Group um **serviço** é uma representação lógica de uma atividade de negócios repetível que tem um resultado específico (por exemplo, verificar o crédito do cliente, fornecer dados meteorológicos, consolidar relatórios de perfuração).

Pode ser composto de outros serviços, mas, para os consumidores, o serviço funciona como uma "caixa preta".

02

Um estilo arquitetônico é a combinação de características distintas em que a arquitetura é realizada ou expressa.

Ainda segundo o Open Group, o estilo arquitetônico SOA tem as seguintes **características**:

- Ele é **baseado no *design* dos serviços** que espelham as atividades comerciais do mundo real que compõem os processos de negócios de uma empresa ou do relacionamento entre empresas.
- A representação de um serviço **utiliza descrições de negócios para fornecer contexto** (ou seja, processos de negócio, objetivo, regra, política, interface de serviço e componente de serviço) e implementa serviços usando a orquestração.
- SOA **requer requisitos únicos de infraestrutura**.
- **Implementações são específicas do ambiente.**

- **Exige uma forte governança** da representação serviço e implementação.
- Exige **testes bem realizados** para determinar um bom serviço.

requer requisitos únicos de infraestrutura

Recomenda-se que implementações utilizem padrões abertos para realizar a interoperabilidade e transparência de localização.

Implementações são específicas do ambiente

Estas implementações são restritas ou habilitadas pelo contexto e devem ser descritas dentro desse contexto.

03

1.1- Entendendo o Problema

Imagine um sistema desenvolvido em JAVA denominado ABC. Esta plataforma oferece uma API que permite que outras ferramentas acessem os dados armazenados neste sistema.

Esta abordagem foi bem-sucedida inclusive na integração com outras ferramentas, mas ainda tinha duas **limitações** importantes:

1. As outras ferramentas tiveram que ser escritas em Java (ou usaram alguma abordagem para chamar um programa Java a partir de outra linguagem);

2. As requisições para métodos EJB não foram autorizadas a serem feitas através do firewall da organização. Isso significava que as APIs poderiam ser acessadas apenas por ferramentas que funcionavam no mesmo domínio.

A equipe de desenvolvimento do sistema ABC queria eliminar essas duas restrições, a fim de tornar mais fácil a integração com outras ferramentas em implantação.

Era necessária também uma abordagem que permitisse que outras ferramentas fossem desenvolvidas em uma ampla gama de linguagens de programação como C ++ e C #, e essas ferramentas precisavam de uma maneira de acessar com segurança a API do sistema ABC através da Internet a partir de um local remoto.

Como resolver este problema? Veja a **solução**.

Solução

Felizmente para a equipe do sistema ABC, uma abordagem de arquitetura e tecnologias de suporte foram evoluindo ao longo dos anos, o que satisfez ambos os requisitos. As tecnologias de serviços Web Comunicação suporta a comunicação entre aplicações através da Internet utilizando documentos XML e o protocolo SOAP. Serviços Web também apoiam uma abordagem de *design* conhecido como arquitetura orientada a serviços.

04

2- BACKGROUND

As arquiteturas orientadas a serviços e serviços Web surgiram com o objetivo de tentar corrigir os problemas de interoperabilidade entre aplicações e fornecer uma base para aplicações distribuídas em escala na Internet.

Elas também marcam o fim da "guerra dos *middleware*", com todos os principais fornecedores finalmente concordando com um único conjunto de padrões de tecnologia para integração de aplicativos e computação distribuída.

Middleware de integração de aplicativos é usada para muitos propósitos, desde reunir componentes para uma solução local simples até para a construção de uma cadeia de suprimentos globais que abrangem a Internet.

Entretanto, tecnologias tradicionais, tais como aplicações J2EE, podem ser excelentes soluções para a construção de aplicações ou integração de aplicativos dentro da mesma organização. No entanto, eles estão muito aquém do que é necessário para unir processos de negócios geridos por organizações independentes que são conectadas através da Internet. Serviços web e arquiteturas orientadas a serviços são projetados para atender a essa necessidade.

De muitas maneiras, a computação orientada a serviço não é uma novidade. Como as tecnologias de computação distribuídas e arquiteturas mais tradicionais, o seu principal objetivo é **permitir que aplicações chamem funcionalidades fornecidas por outras aplicações**, assim como J2EE permite outros aplicativos chamar métodos fornecidos por componentes J2EE.

Middleware

O temo Middleware é usado para agrupar todas as tecnologias em *software* que estão entre a aplicação final e os fornecedores de dados para esta aplicação final. Assim, uma solução de Middleware fica entre a aplicação que o usuário enxerga e as fontes de informações. A solução de

Middleware intermedia a interação entre a aplicação final e as fontes de informações. Estas fontes de informações podem ou não estar na mesma máquina do servidor de aplicações, podendo inclusive, estar fora do ambiente físico desta máquina. Além disso, as fontes de informações podem estar em plataformas diferentes com sistemas operacionais diferentes.

05

A verdadeira diferença do foco do modelo baseado em serviços e suas tecnologias de apoio está na interoperabilidade e na resolução de problemas práticos que surgem por causa das diferenças de plataformas e linguagens de programação.

Esta ênfase na interoperabilidade pragmática resulta na aceitação da natureza diversa das empresas de hoje, com a percepção de que essa diversidade não vai diminuir no futuro. As empresas atualmente suportam uma série de plataformas, linguagens de programação e pacotes de *software*, incluindo aplicações legadas críticas para os negócios. Qualquer proposta de integração que necessite reescrever aplicações ou a migração de aplicações que já trabalham para novas plataformas irá falhar no primeiro obstáculo, pois os custos e riscos serão muito altos.

A realidade é que os aplicativos corporativos de larga escala são cada vez mais dependentes de aplicativos, pacotes e componentes que não foram projetados para trabalhar em conjunto e podem até mesmo ser executados em plataformas incompatíveis. Isto dá origem a uma necessidade crítica para a interoperabilidade, uma vez que as organizações começam a construir uma nova geração de aplicativos integrados que incorporam processos de negócio hospedados em parceiros comerciais e prestadores de serviços especializados.

Serviços web e arquiteturas orientadas a serviços são a resposta da indústria de tecnologia para esta necessidade de integração.

06

3- Sistemas Orientados a Serviços

A mudança para sistemas orientados a serviços está sendo impulsionada pela necessidade de integrar as aplicações e os sistemas que suportam o negócio. A maioria das tecnologias de integração existentes são fechadas ou proprietárias e só permitem a integração de aplicativos criados na mesma tecnologia, a menos que a organização esteja disposta a arcar com o custo de comprar ou escrever um adaptador complexo com esta finalidade. Estas restrições podem até ser aceitáveis dentro de uma organização, embora, mesmo nesta situação, as chances das aplicações e dos sistemas serem compatíveis são bastante raras.

A necessidade de integração de sistemas de negócios existe onde há sistemas de negócios. Esta integração tem sido tradicionalmente tratada através da troca de documentos em papel, tais como citações, faturas e encomendas. Estes documentos tradicionais ainda são usados até hoje, mas agora eles são quase sempre produzidos por sistemas informatizados.

A economia de custo e eficiência proporcionada por se livrar do papel e sistemas de negócios que integram diretamente sistemas baseados em computador é óbvia, mas não é uma tarefa simples. Mas, o advento dos serviços de Internet mudou totalmente esse quadro. A Internet agora potencialmente conecta cada sistema de computador em uma rede global permitindo que as empresas enviem documentos eletronicamente para seus parceiros e clientes em qualquer parte do mundo, rapidamente e com baixo custo.

Os serviços Web abordam a outra parte do problema, fornecendo um conjunto único de normas de integração de aplicativos que são implementados por todos os principais fornecedores e são enviados como parte integrante de todas as plataformas. O resultado deste cenário é que a integração em nível de negócio está se tornando relativamente fácil, barata e comum.

07

Os princípios fundamentais da arquitetura orientada a serviços não são novos e em grande parte apenas refletem anos de experiência na construção de sistemas integrados de larga escala que efetivamente funcionam e que necessitam ser mantidos. Alguns dos princípios, como a redução dos impactos da latência da rede fazendo o trabalho tanto quanto possível em cada chamada de serviço, já são prática comum em sistemas distribuídos de alto desempenho. Outros, tais como minimizar os custos da infraestrutura de serviços da Web, resultam de uma compreensão dos valores de uma solução complexa e da importância da simplicidade na obtenção de interoperabilidade.

Os **princípios básicos da arquitetura orientada a serviços** são:

- As fronteiras são claras;
- Serviços são autônomos;
- Compartilhar esquemas e contratos e não implementações;
- Compatibilidade de serviço é baseada em política.

Vamos detalhar cada um destes princípios.

08

3.1- As fronteiras são claras

O primeiro dos princípios reconhece que os serviços são **aplicações independentes**, não apenas o código que está ligado em seu programa, que pode ser criados a quase nenhum custo.

Acessar um serviço requer, pelo menos, atravessar as fronteiras que separam os processos e, provavelmente, atravessar diferentes redes e fazer a autenticação de usuário de domínio cruzado. Cada uma dessas fronteiras (processo, máquina, confiança) que tem de ser superada reduz o desempenho, aumenta a complexidade e aumenta as chances de falhas. Desta forma, é importante que as aplicações sejam conscientemente reconhecidas e manuseadas dentro do processo de *design*.

Os desenvolvedores e prestadores de serviços também podem estar separados geograficamente e isso implica limites, que podem se refletir no aumento do tempo de desenvolvimento, custo e robustez da solução.

A resposta a este desafio é concentrar-se na **simplicidade**, tanto na especificação de serviços como nos padrões dos serviços. Bons serviços têm interfaces simples e compartilham poucas abstrações e suposições com os seus clientes. Isso torna os serviços mais fáceis de serem compreendidos e utilizados pelos desenvolvedores.

09

3.2- Os serviços são autônomos

Serviços são aplicações independentes autônomas, não podem ser classes ou componentes que estão fortemente ligadas a aplicativos. Os serviços destinam-se a serem implantados em uma rede, possivelmente a Internet, onde podem ser facilmente integrados em qualquer aplicação que os achem úteis.

Os serviços não necessitam saber nada sobre aplicações de clientes, e podem aceitar solicitações de serviço recebidas a partir de qualquer lugar, contanto que as mensagens de solicitação estejam formatadas corretamente e atendam aos requisitos de segurança especificados.

Os serviços podem ser implantados e gerenciados de forma totalmente independente de outros serviços e quaisquer aplicações de clientes possíveis, e os proprietários destes serviços podem alterar as suas definições, implementações ou requisitos a qualquer momento. A **compatibilidade de versão** é um problema de longa data com todos sistemas e tecnologias distribuídas, e é agravado pela natureza aberta de serviços. Como evoluir um serviço quando você tem um grande número de clientes (possivelmente desconhecidos) que dependem dele? Veja um exemplo.

Parte da resposta para este problema reside na **simplicidade** deliberada e **extensibilidade** do modelo de serviços. Tudo o que os clientes sabem sobre um serviço é quais mensagens ele irá aceitar e retornar, e esta é a única dependência que existe entre um cliente e um serviço. Os proprietários de serviços podem alterar a implementação de um serviço à vontade, contanto que as mensagens válidas não sejam alteradas.

Veja aqui uma solução para o exemplo dado.

Como os serviços são autônomos, eles também são responsáveis pela sua própria segurança e têm de se proteger contra possíveis chamados maliciosos. Os sistemas implantados inteiramente num único sistema ou sobre uma rede fechada podem ser capazes de ignorar em grande medida a segurança ou simplesmente confiar nos *firewalls*. No entanto, serviços acessíveis através da Internet aberta têm que levar a segurança muito mais a sério.

Exemplo

Por exemplo, um banco executando um componente de servidor que só é chamado por um aplicativo contador interno pode conhecer a identidade e localização de todos sistemas cliente, deste modo a atualização do serviço é, pelo menos tecnicamente, viável. Mas o processador do cartão de crédito que pode aceitar solicitações de autorização de qualquer comerciante através da Internet não tem como saber como localizar todos os seus clientes, nem pode levá-los a atualizar suas aplicações para corresponder às novas definições de serviço.

Veja aqui

Nosso processador de cartão de crédito poderia totalmente mudar a forma como o serviço é implementado, talvez passando de CICS / COBOL para um C # /. NET, e essa mudança será imperceptível para todos os clientes que fazem chamadas ao serviço, desde que não haja mudanças na mensagem "autorizar o pagamento".

10

3.3- Compartilhar esquemas e contratos e não implementações

Anos de experiência têm mostrado que a construção de sistemas de grande escala integrados, robustos e confiáveis é uma tarefa difícil. Construir esses sistemas de componentes usando diferentes modelos de programação e correndo em diferentes plataformas é muito mais difícil. A tecnologia orientada a

serviços endereça este problema deliberadamente apontando para a simplicidade, tanto quanto possível.

Serviços não são objetos remotos com herança e métodos de execução complexa, nem são componentes que suportam eventos, propriedades e chamadas de método. Serviços são apenas **aplicações que recebem e enviam mensagens.** Clientes e serviços compartilham nada mais do que as definições dessas mensagens, e certamente não compartilham código de método ou ambientes de execução complexas.

Tudo o que um aplicativo precisa saber sobre um serviço é o seu contrato:

- a estrutura (esquema) das mensagens que vai aceitar e retornar,
- e se os serviços têm de ser enviados em uma ordem particular.

As aplicações cliente podem usar esse contrato para construir mensagens de pedido para enviar a um serviço, e serviços podem usar seus esquemas para validar as mensagens recebidas e verificar se elas estão corretamente formatadas.

11

3.4- Compatibilidade de Serviço é baseada em política

Os clientes têm de ser totalmente compatíveis com os serviços que desejam usar. Compatibilidade não significa simplesmente que os clientes estão seguindo o formato de mensagens e padrões de troca de mensagens especificados, mas também que eles cumpram com outros requisitos importantes, tais como se as mensagens devem ser encriptadas ou se necessitam ser rastreadas para assegurar que nenhuma delas tenha sido perdida no trânsito. No modelo orientado a serviços, estes requisitos não funcionais são definidos usando políticas, e não apenas como parte da documentação do serviço.

Por exemplo, nosso processador de cartão de crédito pode decidir que todos os comerciantes que apresentaram pedidos de autorização de pagamento devem provar a sua identidade usando tokens de autenticação baseados em X.509. Esta restrição de segurança pode ser representada simplesmente como uma declaração na política de segurança publicada para o serviço de autorização.

Políticas são conjuntos de instruções legíveis por máquina que permitem definir as suas exigências para itens como segurança e confiabilidade. Estas políticas podem ser incluídas como parte do contrato de um serviço, permitindo-lhe especificar completamente o comportamento e as expectativas de um serviço.

Políticas baseadas em contrato podem ser consideradas como apenas uma parte da documentação de um serviço, mas também podem ser usadas por ferramentas de desenvolvimento para gerar automaticamente o código compatível para ambos, clientes e serviços. Veja um exemplo.

A separação das políticas de contratos também permite que aplicativos clientes se **adaptem dinamicamente** para atender às exigências de um determinado serviço. Isso vai se tornar cada vez mais útil à medida que os serviços se tornarem padrão.

Veja outro exemplo.

Exemplo

Por exemplo, uma política de segurança do lado do servidor pode ser usada para gerar código que irá verificar que as peças necessárias de uma mensagem de entrada são criptografadas e, em seguida, descriptografadas, apresentando-as como texto simples para o aplicativo de serviço. Tudo isso é feito sem qualquer esforço de codificação do desenvolvedor.

Outro exemplo

Por exemplo, um varejista on-line pode usar duas formas de entrega que oferecem exatamente os mesmos serviços e usam os mesmos esquemas de mensagem, mas têm requisitos de autenticação diferentes. O uso de políticas dinâmicas permite que nossos desenvolvedores escrevam um único aplicativo que suporta ambos os métodos de autenticação e selecione dinamicamente qual usar para buscar a política do serviço de destino antes de construir e enviar todos os pedidos de entrega.

12

4- Web Services

Os serviços Web são um conjunto de padrões de tecnologia de integração que foram projetados especificamente para atender às exigências decorrentes da arquitetura orientada a serviços.

De muitas maneiras, os serviços da Web não são realmente muito diferentes de tecnologias de *middleware* existentes, mas eles diferem em seu foco na simplicidade e interoperabilidade. A característica mais importante oferecida pelos serviços Web é que todos os principais fornecedores de *software* concordaram em suportá-lo.

Os serviços Web fornecem quatro **funções básicas** que permitem que desenvolvedores (e programas) façam o seguinte:

- Encontrem o serviço adequado (usando UDDI ou outro diretório);
- Informe-se sobre um serviço (usando WSDL);
- Solicite a execução de um serviço (usando SOAP);
- Façam uso de serviços com segurança (usando padrões WS- *).

SOAP, WSDL e UDDI foram os primeiros padrões de serviços Web a serem publicados, mas eles só atendem aos requisitos mais básicos para a integração de aplicação. Esses padrões carecem de apoio para segurança, transações, confiabilidade e muitas outras funções importantes.

13

Os serviços Web são padrões XML. Os serviços são definidos usando XML, as aplicações solicitam serviços enviando mensagens XML e os padrões de serviços fazem uso extensivo de padrões XML sempre que possível. Os padrões são o mais simples possível, uma vez que eles precisam suportar aplicações seguras, robustas e interoperáveis. Há também cada vez mais ferramentas e bibliotecas que suportam estes padrões. Com estas ferramentas, os desenvolvedores só precisam compreender os recursos oferecidos, em vez da sintaxe detalhada do XML.

Um dos princípios que permeiam todos os padrões de serviços Web é que os **vários campos de mensagem e atributos são totalmente independentes uns dos outros**. As aplicações só precisam incluir apenas aqueles poucos campos e atributos necessários para os seus fins específicos, e pode ignorar todos os outros padrões.

Outro objetivo dos padrões de serviços Web é **fornecer um bom suporte para arquiteturas** de sistemas que fazem uso de "intermediários". Ao invés de permitir que os clientes sempre enviem pedidos diretamente aos prestadores de serviços, o modelo intermediário permite que essas mensagens passem ao longo de uma cadeia de outros aplicativos no seu caminho para o seu destino final. Esses intermediários podem fazer qualquer coisa com as mensagens que recebem, incluindo roteamento, registrando, verificando a segurança ou até mesmo adicionando ou subtraindo pedaços de conteúdo da mensagem.

14

4.1- Benefícios

Os defensores do SOA acreditam que esta arquitetura pode ajudar as empresas a **responderem mais rapidamente e de forma mais eficiente às mudanças** nas condições de mercado. Este estilo de arquitetura é capaz de promover a reutilização de serviços e simplificar a interligação de aplicações. Outra vantagem é que, como a integração entre as aplicações ocorre no nível das mensagens, a linguagem de programação e tecnologia utilizada não é relevante no uso desta arquitetura. O que

permite, inclusive, que **sistemas desenvolvidos em diferentes linguagens possam ser integrados** de maneira relativamente simples.

Esta arquitetura também permite a obtenção de **métricas de utilização** de forma facilitada.

Esta arquitetura proporciona uma **forma simplificada de comunicação** entre diferentes empresas. Uma vez que um serviço está exposto, é possível que outra empresa possa utilizar e se beneficiar de um serviço, bastando para isso seguir os padrões de segurança e utilizar a assinatura do serviço corretamente.

Outro benefício indireto do SOA é que **simplifica drasticamente os testes**. Serviços são autônomos, com interfaces plenamente documentadas e separadas das preocupações transversais da implementação.

Métricas de utilização

Por exemplo, é possível quantificar o número de vezes que um determinado serviço foi utilizado. Esta característica é capaz de fornecer informações relevantes para o negócio sem a necessidade de escrever relatórios complexos.

15

4.2- SOAP e a Mensagem

SOAP foi o padrão original de serviços para a Internet e ainda é o mais importante e o mais utilizado no momento. Este padrão especifica um protocolo de comunicação simples, baseado em XML, entre as aplicações. Como ele é baseado em XML, não é necessário passar objetos como referência.

Tudo o que o padrão SOAP faz é definir um protocolo orientado a mensagem simples, mas extensível para invocar serviços remotos, utilizando HTTP, SMTP ou UDP como a camada de transporte e XML para formatar os dados.

Mensagens SOAP tem a estrutura simples mostrada na figura a seguir:

O cabeçalho contém informações sobre a carga da mensagem, incluindo possivelmente elementos, como tokens de segurança e contextos de transação. O corpo mantém o conteúdo real da mensagem que está sendo passada entre as aplicações.

15

4.3- UDDI, WSDL e Metadados

Os Serviços SOAP são normalmente descritos usando WSDL (Web Services Description Language) e podem ser localizados através de pesquisa em um diretório UDDI (Universal Description, Discovery and Integration).

UDDI provou ser o menos utilizado até agora dos três padrões originais de serviços Web. UDDI é importante, dependendo da importância que você dá para a capacidade de descobrir dinamicamente e fazer o link do serviço à sua aplicação. As pessoas estão desenvolvendo serviços Web complexos sem o uso de diretórios UDDI globais, utilizando outro método de encontrar serviços como o contato pessoal ou listas publicadas de serviços em sites da Web.

WSDL é usado para descrever serviços Web, incluindo as suas interfaces, métodos e parâmetros. A descrição WSDL de um serviço chamado *StockQuoteService* que proporciona uma única operação denominada *GetLastTrade-price*. WSDL é bem suportado pelos ambientes de desenvolvimento, como Visual Studio e WebSphere. Estas ferramentas podem gerar WSDL automaticamente e tornar mais fácil para os desenvolvedores a escrever código que chama esses serviços. Um efeito colateral adverso deste apoio da ferramenta é que ela tende a incentivar os desenvolvedores a pensar em serviços como métodos remotos.

16

4.4 – SOA Manifesto

Em outubro de 2009, um grupo misto de 17 praticantes, entre profissionais independentes e fornecedores, anunciou a publicação do Manifesto SOA.

O SOA Manifesto é um conjunto de objetivos e princípios orientadores que visam proporcionar uma compreensão clara do que é a arquitetura orientada a serviço.

O manifesto fornece uma definição ampla de SOA, os valores que ela representa para os signatários e alguns princípios orientadores. O **manifesto prioriza**:

- Valor de negócio sobre a estratégia técnica
- Objetivos estratégicos mais benefícios específicos do projeto
- Interoperabilidade intrínseca sobre a integração personalizada
- Serviços compartilhados mais implementações de propósito específico
- Flexibilidade sobre otimização
- Refinamento evolutivo ao longo busca da perfeição inicial

Em setembro de 2010, o Manifesto SOA tinha sido assinada por mais de 700 signatários e que tinha sido traduzido para nove idiomas.

17

RESUMO

Neste módulo abordamos o Service-Oriented Architecture, ou **Arquitetura Orientada a Serviço.** Este padrão arquitetural define que os componentes da aplicação devem fornecer serviços para outros componentes através de um protocolo de comunicação.

Podemos definir um **serviço** como uma representação lógica de uma atividade de negócios que tem um resultado específico.

Segundo o Open Group, o estilo arquitetônico SOA tem as seguintes características:

- É baseado no design dos serviços, que espelham as atividades comerciais que compõem os processos de negócios de uma empresa;
- A representação de um serviço utiliza descrições de negócios para fornecer contexto e implementa serviços usando a orquestração;
- Requer requisitos únicos de infraestrutura;
- Implementações são específicas do ambiente;
- Exige uma forte governança da representação serviço e implementação;
- Exige testes bem realizados para determinar um bom serviço.

Arquitetura orientada a serviços traz os seguintes princípios básicos:

1. As fronteiras entre as diferentes aplicações são bem definidas, ou seja, são claras;

2. Os serviços são autônomos;

3. Compartilhar esquemas e contratos e não implementações;

4. Compatibilidade de serviço é baseado em política.

Os serviços Web são um conjunto de padrões de tecnologia de integração que foram projetados especificamente para atender às exigências decorrentes da arquitetura orientada a serviços.

18

Os benefícios da arquitetura orientada a serviço:

- Esta arquitetura pode ajudar as empresas a responderem mais rapidamente e de forma mais eficiente às mudanças nas condições de mercado.
- Promove a reutilização de serviços e simplificar a interligação de aplicações.
- Integração entre as aplicações ocorre no nível das mensagens, onde a linguagem de programação e tecnologia utilizada não são relevantes.
- Permite a obtenção de métricas de utilização de forma facilitada.
- Proporciona uma forma simplificada de comunicação entre diferentes empresas.
- Outro benefício indireto do SOA é que simplifica os testes.

Foi abordado também o **SOA Manifesto**, um conjunto de objetivos e princípios orientadores que visam proporcionar uma compreensão clara do que é a arquitetura orientada a serviço.

O manifesto fornece uma definição ampla de SOA, os valores que ela representa para os signatários e alguns princípios orientadores. O manifesto prioriza:

- Valor de negócio sobre a estratégia técnica.
- Objetivos estratégicos mais benefícios específicos do projeto.
- Interoperabilidade intrínseca sobre a integração personalizada.
- Serviços compartilhados mais implementações de propósito específico.
- Flexibilidade sobre otimização.
- Refinamento evolutivo ao longo busca da perfeição inicial.