UNIDADE 3 – DEFININDO A ARQUITETURA DE *SOFTWARE*

MÓDULO 1 – PROCESSO PARA A DEFINIÇÃO DA ARQUITETURA

01

1 - ELEMENTOS DO PROCESSO

O papel de um arquiteto é muito mais do que simplesmente a realização de uma atividade de projeto de *software*. O arquiteto deve normalmente:

- Trabalhar com a equipe requisitos

A equipe de requisitos é responsável por elicitar e documentar os requisitos funcionais da aplicação junto aos interessados. O arquiteto desempenha um papel importante reconhecendo dentre estes requisitos as necessidades globais dos sistemas e assegurando que os atributos de qualidade adequados são explicitados e compreendidos.

- Trabalhar com várias partes interessadas da aplicação

Arquitetos desempenham um papel de ligação fundamental, garantindo que todas as necessidades dos interessados são compreendidas e incorporadas ao projeto. Por exemplo, além dos requisitos de negócio dos usuários, os administradores do sistema necessitam que o aplicativo possa ser facilmente instalado, monitorado, gerenciado e atualizado.

- Liderar a equipe de projeto técnico

Definir a arquitetura do aplicativo é uma atividade de *design*. O arquiteto lidera uma equipe de projeto, compreendendo projetistas de sistemas e líderes técnicos, a fim de produzir o projeto de arquitetura.

- Trabalhar com a gestão do projeto

O arquiteto trabalha em estreita colaboração com o gerente de projeto, contribuindo com o planejamento, estimativa e alocação de tarefas.

02

Como vimos, são várias as atribuições e responsabilidades de um arquiteto de *software*. A fim de orientar o trabalho deste profissional foi estabelecido um processo básico para a definição da arquitetura do aplicativo.

A figura a seguir mostra um processo de três passos simples que norteiam a definição da arquitetura durante o *design* da aplicação.

Um processo de projeto de arquitetura de três passos

Este processo iterativo é inerente da arquitetura. Uma vez que um projeto é proposto, validá-lo pode mostrar que o projeto precisa de modificação, ou que certos requisitos precisam ser mais bem definidos e compreendidos. Estes levam a melhorias para o projeto, validação subsequente, e assim por diante, até que a equipe de *design* considere que as exigências foram atendidas.

É importante observar a **flexibilidade** deste processo. A definição da arquitetura é taxada como onerosa pela comunidade de métodos ágeis, mas na realidade ele não precisa ser. Se você estiver trabalhando em um projeto usando métodos ágeis, você pode querer ter algumas iterações iniciais (*sprints*, por exemplo) que incidem sobre o estabelecimento de sua arquitetura geral. O resultado dessas iterações será uma linha de base arquitetural que encarna e valida as decisões importantes de *design* do sistema. Iterações subsequentes detalham e complementam esta arquitetura base para atender às novas funcionalidades. Com a arquitetura definida no início do projeto, a refatoração se torna mais simples, proporcionando uma base sólida para a aplicação.

Determinar os requisitos de arquitetura

Isso envolve a criação de uma declaração ou modelo dos requisitos que irão conduzir o projeto de arquitetura.

Projeto de arquitetura

Trata-se de definir a estrutura e as responsabilidades dos componentes que irão compor a arquitetura.

Validação

Trata-se de "testar" a arquitetura, tipicamente, caminhando através do *design*, em relação às necessidades existentes e qualquer conhecido ou possíveis necessidades futuras.

03

1.1- Determinar requisitos arquitetônicos

Antes de uma solução arquitetônica ser concebida, é necessário ter uma boa ideia dos requisitos para a arquitetura do aplicativo.

Requisitos de arquitetura, também chamados de **requisitos arquiteturalmente significativos**, são, essencialmente, a qualidade e os requisitos não funcionais definidos para a aplicação.

04

1.2- Identificando os Requisitos de Arquitetura

Como a figura abaixo apresenta, as principais fontes de requisitos de arquitetura são o documento de requisitos funcionais, e outros documentos que capturam as diferentes necessidades dos *stakeholders*. O resultado dessa etapa é um documento que estabelece os requisitos de arquitetura para o aplicativo.

Um risco que deve ser observado é que parte da informação relevante para a definição da arquitetura não está documentada. Para obter estas informações é conveniente conversar com os vários *stakeholders*. Esta pode ser uma tarefa lenta e cuidadosa, especialmente se o arquiteto não é um especialista no domínio do negócio da aplicação.

Entradas e saídas para determinar os requisitos de arquitetura

Vejamos alguns exemplos de requisitos:

- Desempenho
- Segurança
- Gestão de Recursos de hardware
- Usabilidade
- Disponibilidade

- Resiliência
- Escalabilidade.

Desempenho

O desempenho do aplicativo deve fornecer resposta abaixo de 4 segundos para 90% das requisições.

Segurança

Toda comunicação deve ser autenticada e criptografada utilizando certificados.

Gestão de Recursos de hardware

O componente do servidor deve ser executado em um servidor com memória de 4GB.

Usabilidade

O componente de interface de usuário deve ser executado em um navegador da Internet para oferecer suporte a usuários remotos.

Disponibilidade

O sistema deve executar 24 × 7 × 365, com disponibilidade global de 0,99.

Resiliência

Não é permitida a perda de mensagens, e todas as saídas resultantes da entrega de mensagens deve ser de 30s.

Escalabilidade

A aplicação deve ser capaz de lidar com uma carga de pico de 500 usuários simultâneos durante o período de inscrição.

05

1.3 Priorização dos Requisitos de Arquitetura

É uma coisa rara quando todos os requisitos da arquitetura de um aplicativo são iguais. Muitas vezes, a lista de requisitos de arquitetura contém itens que são de baixa prioridade, ou "isso seria bom ter, mas não é necessário". Por este motivo, é importante identificar e classificar os requisitos da arquitetura usando prioridades. Inicialmente, em geral é suficiente classificar cada requisito em três categorias:

1. Alta:	2. Média:	3. Baixa:
• a aplicação tem de suportar esta exigência. Estes requisitos devem conduzir o projeto de arquitetura.	• este requisito terá de ser suportado em algum momento, mas não necessariamente na primeira versão.	• isso faz parte da lista de requisitos desejados. As soluções que podem acomodar estes requisitos são desejados, mas eles não são condutores do projeto.

Priorização fica mais complicada em face das exigências conflitantes e restrições que muitas vezes não têm como serem atendidas. Diante deste cenário, não há solução simples. Faz parte do trabalho do arquiteto discutir com os *stakeholders*, e chegar a possíveis soluções. É responsabilidade do arquiteto considerar possíveis trocas e tentar encontrar soluções que satisfaçam de forma adequada requisitos sem grandes consequências indesejáveis sobre o requisito. Lembre-se: bons arquitetos sabem como dizer "não".

Em um projeto com muitos *stakeholders*, uma boa ideia é separar os *stakeholders* em conjuntos para assinar esta priorização. Isto é especialmente verdade em face das exigências conflitantes. Assim que a priorização for aprovada, o projeto de arquitetura pode ter início.

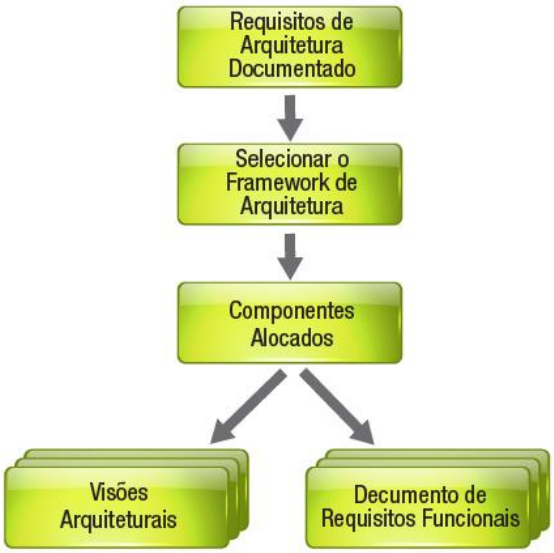
06

2- Projeto de Arquitetura de *Design*

Todas as tarefas executadas por um arquiteto são importantes. Entretanto, é a qualidade do projeto de arquitetura que realmente importa. Produzir documentos maravilhosos e um excelente relacionamento com os *stakeholders* não significam nada se uma má concepção for produzida.

Não é surpreendente que o *design* seja, tipicamente, a tarefa mais difícil que um arquiteto realiza. Bons arquitetos têm vários anos de experiência em *design* e engenharia de *software*. Não há outro substituto para a experiência.

Como mostra a figura a seguir, as entradas para a etapa de *design* são os requisitos de arquitetura. O próprio estágio de projeto tem **duas etapas**, que são iterativos por natureza. O primeiro envolve a **escolha de uma estratégia** global para a arquitetura, fundada em torno de padrões de arquitetura comprovados. O segundo envolve **especificar os componentes individuais** que compõem o aplicativo, mostrando como eles se encaixam no quadro geral e alocá-los responsabilidades.



Entradas e saídas de projeto de arquitetura

Observe que a saída é um **conjunto de pontos de vista** de arquitetura que capturam o projeto de arquitetura, e um **documento de *design*** que explica o projeto, as principais razões para algumas das principais decisões de projeto e identifica os riscos envolvidos em levar o projeto adiante.

07

2.1 Selecionando o Framework de Arquitetura

A maioria das aplicações de sucesso baseia-se em relação a um pequeno número de arquiteturas. Há uma boa razão para isso – elas funcionam. Aproveitando soluções conhecidas minimiza os riscos que uma aplicação irá falhar devido a uma arquitetura inadequada.

Assim, a etapa inicial do projeto envolve a **seleção de uma estrutura de arquitetura** que possa satisfazer os requisitos essenciais. Para pequenas aplicações, um único padrão de arquitetura pode ser suficiente. Para aplicações mais complexas, o projeto pode incorporar um ou mais padrões conhecidos, com o arquiteto especificando como esses padrões se integrarão e formarão a arquitetura geral.

Não há qualquer fórmula mágica para projetar a estrutura da arquitetura. Um pré-requisito, no entanto, é entender como cada um dos principais padrões arquiteturais endereçam os requisitos de qualidade.

Alguns dos principais padrões utilizados e como eles abordam os requisitos comuns de qualidade estão listados abaixo e serão descritos a seguir. São eles:

- Arquitetura em Camadas cliente-servidor
- Mensageria
- Publicador-Assinante
- Broker
- Coordenador de Processo

08

a) Arquitetura em Camadas cliente-servidor

As propriedades principais deste modelo são:

- **A separação de interesses**

Apresentação, negócios e manipulação de dados são claramente divididos em diferentes camadas.

- **Comunicações síncronas**

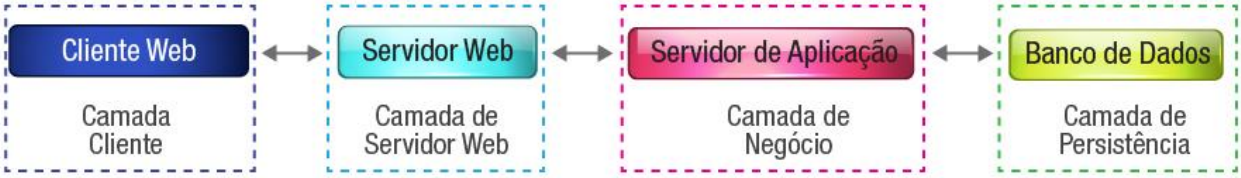
Comunicação entre os blocos é síncrona. Os pedidos emanam em uma única direção a partir da camada do cliente, através da web e da camada de negócio para a camada de gerenciamento de dados. Cada camada aguarda uma resposta da outra camada antes de prosseguir.

- **Implementação flexível**

Não há restrições sobre como uma aplicação em camadas é implantada. Todos os níveis poderiam ser executados na mesma máquina. Em outro extremo, cada camada pode ser implantada em uma máquina separada. Em aplicações web, a camada de cliente é geralmente um navegador rodando em uma área de trabalho do usuário, comunicando remotamente através da Internet com componentes da camada web.

09

A figura a seguir apresenta um exemplo de uma aplicação web de acordo com o padrão em camadas



Exemplo de arquitetura em camada cliente-servidor

Os atributos de qualidade para o padrão Cliente-Servidor são apresentados a seguir:

- Disponibilidade
- Tratamento de Erro
- Modificabilidade
- Desempenho
- Escalabilidade

Como cada atributo de qualidade serão considerados e utilizados na solução depende da tecnologia usada para implementar a aplicação. JAVA, .NET e outros servidores de aplicação proprietários têm diferente características que influenciam em como estes atributos de qualidade podem ser utilizados. As características de cada servidor de aplicação precisam ser compreendidas durante o projeto da arquitetura de modo que não haja surpresas desagradáveis mais tarde no projeto.

O padrão cliente-servidor é comumente usado. As tecnologias de servidores de aplicação tornam relativamente fácil de implementar aplicativos usando o padrão. Este padrão arquitetural é apropriado quando o aplicativo necessita suportar um grande número de clientes e solicitações simultâneas, e cada pedido leva um intervalo relativamente curto para processo.

Disponibilidade

Servidores em cada camada podem ser replicados de modo que, se um falhar, os outros permanecem disponíveis.

Tratamento de Erro

Se um cliente está se comunicando com um servidor que falhar, a maioria dos servidores web e de aplicações implementam *failover* transparente. Isto significa que um pedido do cliente é redirecionado para um servidor de réplica capaz de atender à requisição.

Modificabilidade

A separação em camadas permite que uma camada seja totalmente alterada sem que as alterações afetem outra camada.

Desempenho

Esta arquitetura já provou ser de alto desempenho. Questões-chave a serem consideradas são: a quantidade de *threads* simultâneas suportadas em cada camada, a velocidade de conexões entre as camadas e a quantidade de dados que são transferidos. Faz sentido também minimizar as chamadas entre camadas.

Escalabilidade

Como os servidores em cada camada podem ser replicados, a arquitetura pode ser escalada com relativa facilidade.

10

b) Mensageria

As propriedades principais do modelo de mensageria são:

- Comunicação assíncrona

Clientes enviam as solicitações para uma fila, onde a mensagem é armazenada até que o aplicativo a remova. Após o cliente colocar a mensagem na fila, ele continua a sua execução sem esperar que esta mensagem seja retirada da fila.

- QoS Configurável

A fila pode ser configurada para a alta velocidade ou mais lenta para atender as necessidades da qualidade do serviço. Operações de fila podem ser coordenadas com as operações de banco de dados.

- Fraco Acoplamento

Não há ligação direta entre clientes e servidores. O cliente é alheio a qual servidor atenderá a mensagem, assim como o servidor é alheio a de qual cliente a mensagem veio.

A figura abaixo apresenta um exemplo de uma aplicação com o padrão de mensageria.

Anatomia do padrão de mensageria

11

Os atributos de qualidade para o padrão de mensageria são apresentados a seguir:

- Disponibilidade
- Tratamento de Falha
- Modificabilidade
- Desempenho
- Escalabilidade

Mensageria é especialmente recomendada quando o cliente não precisa de uma resposta imediata diretamente após o envio de um pedido.

Por exemplo, uma cliente pode formatar um e-mail, e colocá-lo em uma fila de mensageria para processamento. O servidor irá, em algum momento no futuro, remover a mensagem da fila e enviar o e-mail usando um servidor de correio. O cliente realmente não precisa saber quando o servidor irá processar a mensagem.

Mensageria também fornece uma solução flexível para aplicações em que a conectividade a um aplicativo de servidor é transitória, devido a rede ou a falta de confiabilidade do servidor. Em tais casos, as mensagens são mantidas na fila até que o servidor se conecta e remove mensagens.

Disponibilidade

As filas podem ser replicadas em diferentes instâncias de servidor de mensagens, bastando para isso que o mesmo nome lógico seja atribuído para as diferentes filas. Assim, quando um falhar, os clientes podem enviar mensagens para uma outra fila ativa de forma transparente.

Tratamento de Falha

Se um cliente está se comunicando com uma fila que falhar, ele pode encontrar uma outra fila e postar sua mensagem.

Modificabilidade

O padrão de mensageria é por definição fracamente acoplado, e isso permite alta capacidade de modificação do cliente e servidor. Isso também ocorre porque o cliente e o servidor não estão diretamente ligados através de uma interface. Entretanto, as alterações no formato das mensagens enviadas pelos clientes podem causar alterações nas implementações do servidor.

Desempenho

A tecnologia de mensageria pode entregar milhares de mensagens por segundo. Contudo, é importante atentar que a diferença de desempenho irá depender da qualidade da tecnologia de mensagens utilizada.

Escalabilidade

Filas podem ser hospedadas em um terminal de comunicação, ou podem ser replicadas em clusters de servidores de mensagens. Isso faz com que a mensageria seja uma solução altamente escalável.

12

c) Publicador-Assinante

As propriedades principais do modelo Publicador-Assinante são:

- Mensagens “Muitos-para-Muitos”

As mensagens publicadas são enviadas para todos os assinantes que são registrados no tópico. Muitos editores podem publicar sobre o mesmo tema, e muitos assinantes podem receber mensagens de um mesmo tema.

- QoS configuráveis

Além de mensagens confiáveis e não confiáveis, o mecanismo de comunicação subjacente pode ser ponto-a-ponto ou broadcast/multicast.

- Acoplamento fraco

Tal como acontece com mensageria, não há ligação direta entre editores e assinantes. Os publicadores não sabem quem recebe suas mensagens e os assinantes não sabem qual editora enviou a mensagem.

A figura a seguir apresenta um exemplo de uma aplicação com o padrão Publicador-Assinante.

Anatomia do padrão de Publicador-Assinante

13

Os atributos de qualidade para o padrão Publicador-Assinante são apresentados a seguir:

- Disponibilidade
- Tratamento de Falha
- Modificabilidade
- Performance
- Escalabilidade

Arquiteturas baseadas na publicação-assinatura são altamente flexíveis e adequados para aplicações que requerem comunicação assíncrona de um-para-muitos, muitos-para-um ou de mensagens entre componentes de muitos-para-muitos.

Disponibilidade

Tópicos com o mesmo nome lógico podem ser replicados em diferentes instâncias de servidor que podem ser gerenciados como um cluster. Quando um falhar, o publicador pode enviar mensagens para as filas de réplicas.

Tratamento de Falha

Se um publicador está se comunicando com um host de tópico que falhar, ele pode encontrar uma fila de réplica e postar a mensagem lá.

Modificabilidade

O padrão publicador-assinante é, de forma inerente, fracamente acoplado, e isso promove alta modificabilidade. Novos publicadores e assinantes podem ser adicionados ao sistema sem que seja necessária uma alteração na configuração da arquitetura.

Performance

O padrão publicador-assinante pode entregar milhares de mensagens por segundo.

Escalabilidade

Os tópicos podem ser replicados através de clusters de servidores. Um cluster de servidores pode escalar para proporcionar um maior volume de mensagens sendo enviadas.

14

d) Broker

As propriedades principais do modelo Broker são:

- Arquitetura *hub-and-spoke*

O Broker atua como um hub de mensagens, no qual emissores e destinatários conectam-se entre si. As conexões para o Broker são realizadas através de portas que estão associadas com um formato de mensagem específica.

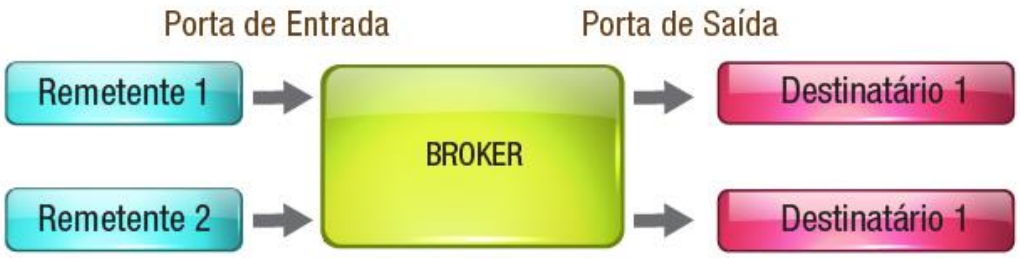
- Roteamento de mensagens

O broker incorpora a lógica de processamento para entregar uma mensagem recebida em uma porta de entrada para uma porta de saída. O caminho de entrega pode ser codificado ou depender de valores na mensagem de entrada.

- Transformação de mensagens

A lógica do Broker transforma o tipo de mensagem recebida na porta de entrada para o tipo de mensagem exigido no destino da porta de saída.

A figura abaixo apresenta um exemplo de uma aplicação com o padrão Broker.



15

Os atributos de qualidade para o padrão Broker são apresentados a seguir:

- Disponibilidade
- Tratamento de Falha
- Modificabilidade
- Performance
- Escalabilidade

Brokers são adequados para aplicações, nas quais componentes trocam mensagens que exigem grande transformação entre fonte e formatos de destino. O broker separa o emissor e o receptor,

permitindo-lhes produzir ou consumir o seu formato de mensagem nativa, e centraliza a definição da lógica de transformação no broker para facilitar a compreensão e a modificação.

Disponibilidade

Para construir uma arquitetura de alta disponibilidade para este padrão, os brokers devem ser replicados. Isto é suportado por um mecanismo similar ao da mensageria ou de publicador-assinante.

Tratamento de Falha

Como os brokers possuem tipos de porta de entrada eles validam e descartam quaisquer mensagens que sejam enviadas em um formato errado.

Modificabilidade

Brokers separam a transformação da lógica de roteamento das mensagens. A mudança destas duas características pode ser realizada sem afetar tanto o remetente como o destinatário das mensagens.

Performance

O pode ser tornar um gargalo, principalmente se eles tiverem que tratar um grande número de mensagens.

Escalabilidade

É possível clusterizar o broker para que possa ser atendido um grande número de requisições.

16

e) Coordenador de Processo

As propriedades principais do modelo Coordenador de Processo são:

- **Encapsulamento de Processo**

O coordenador de processo encapsula a sequência de passos necessários para cumprir o processo de negócio. A sequência pode ser arbitrariamente complexa. O coordenador é um único ponto de definição para o processo de negócio, tornando-o mais fácil de compreender e de modificar. Ao receber uma requisição de iniciação de um processo, ele chama os servidores definidos no processo e emite o resultado.

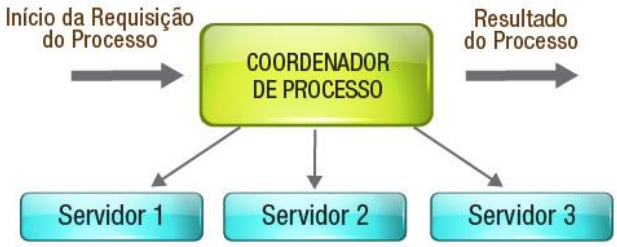
- **Acoplamento fraco**

Os componentes do servidor não estão cientes do seu papel no processo global de negócios, e da ordem das etapas do processo. Os servidores simplesmente definem um conjunto de serviços que podem ser executados, e o coordenador chama-os quando necessário, como parte do processo de negócio.

- **Comunicações flexíveis**

As comunicações entre o coordenador e os servidores podem ser síncronas ou assíncronas. Para comunicações síncronas, o coordenador aguarda até que o servidor responda. Para a comunicação assíncrona, o coordenador fornece um *callback* ou uma fila de resposta e espera até que o servidor responda usando o mecanismo definido.

A figura abaixo apresenta um exemplo de uma aplicação com o padrão Coordenador de Processo.



Exemplo de arquitetura com o padrão Coordenador de Processo

17

Os atributos de qualidade para o padrão Coordenador de Processo são apresentados a seguir:

- **Disponibilidade**

O coordenador é um ponto único de falha. Por isso, precisa ser replicado para criar uma solução de alta disponibilidade.

- **Tratamento de Falha**

O tratamento de falhas é complexo, pois pode ocorrer em qualquer fase na coordenação de processos de negócios. A falha de um passo do processo poderá exigir que etapas anteriores sejam

desfeitas. Manipulação de falhas precisa de um projeto cuidadoso para garantir que os dados mantidos pelos servidores permaneçam consistentes.

- **Modificabilidade**

Processo modificabilidade é reforçada porque a definição do processo é centralizada no coordenador de processo. Os servidores podem mudar sua implementação sem afetar o coordenador ou outros servidores desde que a sua definição de serviço externo não muda.

- **Performance**

Para alcançar um alto desempenho, o coordenador deve ser capaz de lidar com várias solicitações simultâneas e gerir o estado de cada requisição à medida que avança no processo. Além disso, a realização de qualquer processo será limitada pelo passo mais lento, isto é, o servidor mais lento.

- **Escalabilidade**

O coordenador pode ser replicado para permitir a escalabilidade da solução.

18

2.2- Alocar os Componentes

Uma vez que um quadro global arquitetural foi selecionado, com base em um ou mais padrões de arquitetura, a próxima tarefa é definir os componentes que irão compor o projeto.

O framework define os **padrões globais de comunicação** para os componentes que devem levar em consideração as seguintes orientações:

- Identificar os principais componentes de aplicação, e como eles serão conectados ao framework.
- Identificar a interface ou serviços que cada componente suporta.
- Identificar as responsabilidades dos componentes.
- Identificar as dependências entre os componentes.
- Identificar as partições na arquitetura que são candidatas à distribuição sobre os servidores da rede.

Os componentes da arquitetura são as principais abstrações que existirão na aplicação. É muito comum utilizar técnicas de *design* orientadas a objetos para representar os componentes. De fato, diagramas de classe e pacotes são muitas vezes utilizados para descrever componentes em uma arquitetura.

19

3 - VALIDAÇÃO

Durante a definição da arquitetura de um projeto de software, o objetivo da fase de validação é aumentar a confiança da equipe de design que a arquitetura atende às necessidades.

Validar um projeto de arquitetura traz alguns desafios. Se a arquitetura é para uma nova aplicação, ou uma evolução de um sistema existente, o projeto proposto é apenas um *design*. Ela não pode ser executada ou testada para ver se ela preenche os seus requisitos. Ele também irá, provavelmente, considerar os novos componentes que têm de ser construídos. Todas estas peças têm que ser integradas e obrigadas a trabalhar juntas.

Existem **duas principais técnicas** que se revelaram úteis.

A primeira envolve essencialmente o teste manual da arquitetura usando cenários de teste.

O segundo envolve a construção de um protótipo que cria um simples arquétipo do aplicativo desejado, de modo que a sua capacidade de satisfazer os requisitos possa ser avaliada de forma mais detalhada através de testes do protótipo.

O objetivo de ambos é identificar possíveis falhas e pontos fracos no projeto, de modo que eles possam ser melhorados antes da implementação começar. Essas abordagens devem ser usadas para identificar explicitamente áreas de risco potenciais para rastreamento e monitoramento durante as atividades de construção.

20

RESUMO

Neste módulo, vimos os elementos do processo para a definição da Arquitetura de um *Software*.

Este processo é composto por 3 passos:

- Determinar os requisitos de arquitetura
- Projeto de arquitetura e

- Validação.

Requisitos de arquitetura, também chamados de **requisitos arquiteturalmente significativos**, são, essencialmente, a qualidade e os requisitos não funcionais definidos para a aplicação e, para determiná-los, eles devem ser identificados.

O projeto de arquitetura é tipicamente a tarefa mais difícil que um arquiteto realiza. É neste momento que o arquiteto deve selecionar o *framework* mais adequado para atender os requisitos identificados no momento anterior.

Vimos alguns *frameworks* de arquitetura e suas características, como:

- Arquitetura em camadas cliente-servidor
- Mensageria
- Publicador- assinante
- Broker
- Coordenador de processo.

Por fim, vimos que são basicamente duas as técnicas para a validação da arquitetura. A primeira são os cenários de teste e a segunda são os protótipos.

UNIDADE 3 – DEFININDO A ARQUITETURA DE *SOFTWARE*
MÓDULO 2 – ATRIBUTOS DE QUALIDADE DE *SOFTWARE*
01

1- QUALIDADE NO CONTEXTO DA ARQUITETURA DE *SOFTWARE*

Vimos anteriormente, de forma superficial, os atributos de qualidade de *software*. No início desta unidade, exploramos os passos do processo para a definição da arquitetura de um projeto de *software*. Estes passos são definidos como:

Assim, tudo inicia com a determinação dos requisitos de *software*. Para isso, o conceito de qualidade é fundamental. Desta forma, vamos explorar nesta unidade a qualidade com um pouco mais de detalhe.

Grande parte da vida de um arquiteto de *software* é gasto projetando sistemas de *software* para atender a um conjunto de requisitos de atributos de qualidade. Em geral os **atributos de qualidade de *software*** incluem:

- escalabilidade
- segurança,
- desempenho e
- confiabilidade.

Requisitos de qualidade fazem parte de requisitos não funcionais de um aplicativo, que captura as muitas facetas de como os requisitos funcionais de uma aplicação devem ser alcançados. Mesmo as aplicações mais triviais terão requisitos não funcionais que podem ser expressos em termos de atributos de qualidade.

02

Para serem significativos, os requisitos de atributos de qualidade devem ser específicos sobre como um aplicativo deve atingir uma determinada necessidade. Um problema comum encontrado regularmente em documentos de arquitetura são requisitos muito genéricos, tais como "A requisição deve ser escalável".

Este tipo de requisito é demasiadamente impreciso e realmente não é útil a ninguém. Os requisitos de escalabilidade são muitos e variados, e referem-se a diferentes características da aplicação. Definir quais medidas de escalabilidade devem ser suportadas pelo sistema é crucial do ponto de vista arquitetônico. É, portanto, vital definir os requisitos de atributos de qualidade de forma mais concreta possível.

Um **exemplo** de uma correta especificação de um requisito de qualidade seria:

- *Deve ser possível escalar a implantação de um montante inicial de 100 desktops de usuários geograficamente dispersos para 10.000 sem um aumento do esforço/custo para instalação e configuração.*

Este requisito é **preciso** e **significativo**. Com este tipo de requisito, um arquiteto pode apontar um conjunto de soluções e tecnologias concretas que facilitem a instalação e minimize o esforço de implantação.

03

Note, no entanto, que muitos atributos de qualidade são realmente um pouco difíceis de serem validados e testados. No nosso exemplo, seria improvável que nos testes da versão inicial existisse um caso de teste para instalar e configurar o aplicativo em 10.000 desktops.

Este é o lugar onde bom senso e experiência entram. A solução adotada deve, obviamente, funcionar para a implantação inicial de 100 usuários. Caso não haja falhas críticas, é provavelmente seguro assumir que a solução irá escalar.

A seguir vamos fazer uma descrição de alguns dos atributos mais relevantes de qualidade para aplicações de TI, e alguma discussão sobre mecanismos de arquitetura que são amplamente utilizados para fornecer soluções para os atributos de qualidade exigidos. Estes irão dar-lhe um bom lugar para começar quando se pensa sobre as qualidades de um aplicativo que você está trabalhando deve possuir.

04

2 - Desempenho

Embora para muitas aplicações de TI, o desempenho não seja um problema muito grande, este quesito se tornou o centro das atenções da comunidade no que se refere à qualidade de *software*. Isto se deve ao fato de ser uma das qualidades do aplicativo que podem ser facilmente quantificadas e validadas. Além de afetar diretamente o usuário final.

A exigência de desempenho define uma métrica que indica a quantidade de trabalho que um aplicativo deve executar em um determinado **tempo** e o **prazo** que deve ser cumprido para que a operação seja concluída.

Poucas aplicações de TI têm **restrições** críticas de tempo real, como as encontradas em sistemas de controle de aeronaves, nas quais se alguma saída for produzida alguns milésimos de segundo mais tarde, coisas indesejáveis podem acontecer.

Por outro lado, existem os aplicativos que precisam processar centenas, às vezes milhares e dezenas de milhares de transações a cada segundo que são encontrados em grandes organizações, especialmente nos mundos no mercado financeiro, telecomunicações e governo.

Desempenho geralmente se manifesta nas seguintes **medidas**:

- Throughput;
- Tempo de Resposta;
- Prazos.

Essas medidas serão apresentadas a seguir.

05

2.1- *Throughput*

Throughput é a medida da **quantidade de trabalho** que uma aplicação necessita executar em uma unidade de tempo.

O trabalho é normalmente medido em **transações por segundo** (TPS), ou mensagens processadas por segundo (mps). Por exemplo, um aplicativo para um banco on-line pode ter que garantir executar 1000 tps dos clientes. Um sistema de gerenciamento de inventário para um grande supermercado pode precisar processar 50 mensagens por segundo de parceiros comerciais solicitando ordens de serviço.

É importante compreender exatamente o que se entende por *Throughput*. Ou seja, o que se deseja é a vazão **média** ao longo de um determinado período de tempo (por exemplo, um dia útil), ou taxa de transferência de pico? Esta é uma distinção crucial.

Para exemplificar a importância deste conceito, vamos imaginar um sistema de apostas em corridas de cavalos. Para a maioria das vezes, esta aplicação faz muito pouco trabalho. Portanto, tem um requisito de rendimento médio baixo e facilmente realizável.

No entanto, cada vez que há um evento de corrida importante, talvez duas vezes ao ano, em que minutos antes de cada corrida milhares de apostas estão sendo registradas a cada segundo. Se o aplicativo não for capaz de processar essas apostas à medida que elas são realizadas poderá gerar um grande prejuízo tanto para a empresa de apostas como para os apostadores. Para este cenário, o *software* deve ser projetado para atender o **pico** e não a média. Na verdade, projetar a solução para suportar somente a média provavelmente seria fatal para o *software* (e, consequentemente, para o arquiteto).

06

2.2- Tempo de Resposta

Esta é uma medida da latência em que a aplicação processa uma transação comercial. O tempo de resposta é frequentemente (mas não exclusivamente) associado com o tempo necessário para uma aplicação responder a alguma entrada.

Um tempo de resposta rápido permite aos usuários trabalharem de forma mais eficaz. Um excelente exemplo é um aplicativo de ponto de venda de uma grande loja de departamento. Quando um código

de barra de um item é “lido” pelo leitor de código de barra, uma resposta rápida com o preço do item exibido pelo sistema favorece que o cliente possa ser atendido rapidamente melhorando a experiência do usuário e, consequentemente, as vendas da loja.

Muitas vezes é importante fazer a distinção entre os tempos de resposta **garantido** e **médio**. Alguns aplicativos podem precisar que *todas as solicitações* sejam atendidas dentro de um limite de tempo especificado; este é um **tempo de resposta garantida**. Outros podem especificar um **tempo médio de resposta**, permitindo maior latência quando o aplicativo é submetido a condições mais extremas. Por exemplo, 95% de todas as requisições devem ser processadas em menos de 4s, e nenhuma solicitação deve levar mais de 15s.

07

2.3- Prazos

Você imagina um sistema de previsão do tempo que precisa de 36 horas para produzir a previsão para o dia seguinte? Este é um excelente exemplo da obrigação de cumprir um prazo.

Prazos, no mundo da TI, são comumente associadas a sistemas de lotes.

Um sistema de pagamento de segurança social deve completar a tempo de depositar os pagamentos nas contas dos segurados em um determinado dia. Se isso terminar a tempo, os segurados não são pagos quando eles esperam, e isso pode causar inconvenientes extremos, e não apenas para os segurados. Em geral, qualquer aplicação que tenha uma janela de tempo limitado para concluir terá um prazo de exigência de desempenho.

Estes três atributos de desempenho podem ser claramente especificados e validados. Ainda assim, há uma armadilha comum para evitar. Encontra-se na definição do que é uma **transação**. Essencialmente, esta é a definição de uma carga de trabalho do aplicativo.

Assim, não há nenhuma medida de carga de trabalho genérico, ou seja, que sirva para todo e qualquer sistema. Isso depende inteiramente do trabalho que o aplicativo deve executar. Desta forma, uma determinada medida de desempenho é uma medida específica do aplicativo.

08

3 - ESCALABILIDADE

Podemos definir escalabilidade como:

O quão bem uma solução para um determinado problema irá trabalhar caso o tamanho do problema aumente.

Isto é útil no contexto de uma arquitetura. Ela nos diz que a escalabilidade trata de como um projeto pode lidar com alguns aspectos da aplicação no que se refere às exigências cada vez maiores em termos de tamanho. Para se tornar um requisito concreto de qualidade, nós precisamos entender exatamente o que é esperado para a escalabilidade.

A seguir veremos alguns exemplos.

09

3.1- Pedido de carregamento

Com base nas requisições definidas para uma plataforma de *hardware*, a arquitetura pode ser concebida para suportar 100 TPS na carga de pico, com uma média de 1s de tempo de resposta. ***Se um pedido para crescer em dez vezes esta carga, a arquitetura pode apoiar este aumento da carga?***

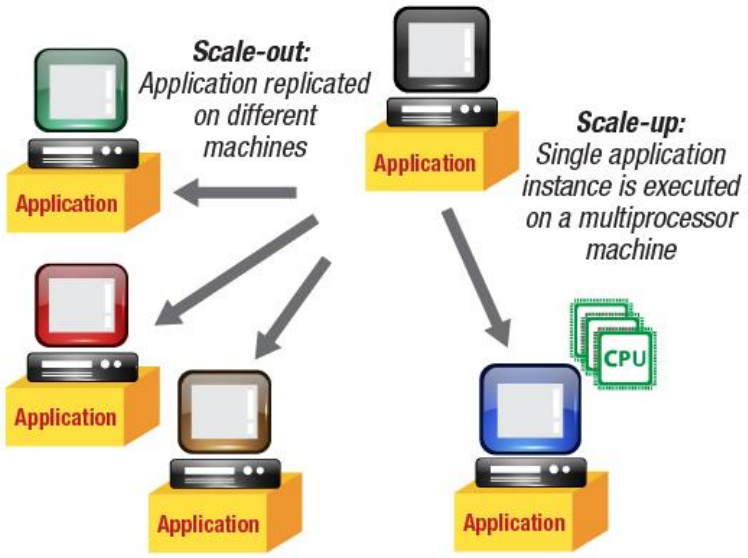
No mundo perfeito e sem capacidade de *hardware* adicional, como a carga aumenta, o rendimento do aplicativo deve permanecer constante (ou seja, 100 TPS), e o tempo de resposta por solicitação deve crescer apenas de forma linear (ou seja, 10 s). Uma solução escalável, então, é permitir que a capacidade de processamento adicional a ser implantado para aumentar o rendimento e diminuir o tempo de resposta.

Esta capacidade adicional pode ser implementada de duas maneiras diferentes:

1- adicionando mais CPUs (e provavelmente memória) para a máquina,

2- distribuir a aplicação em várias máquinas (*scale-out*).

Isto é ilustrado na figura abaixo:



Adição de mais CPUs e memória para a máquina (Scale-up) e distribuição da aplicação em várias máquinas (Scale-out)

10

Escalar através de adição de mais *hardware* até funciona bem se um aplicativo é *multithreaded*, ou se várias instâncias podem ser executadas em conjunto em uma mesma máquina. Sendo que adicionar mais *hardware* significa um acréscimo do consumo de memória e demais recursos associados principalmente se os processos forem pesados.

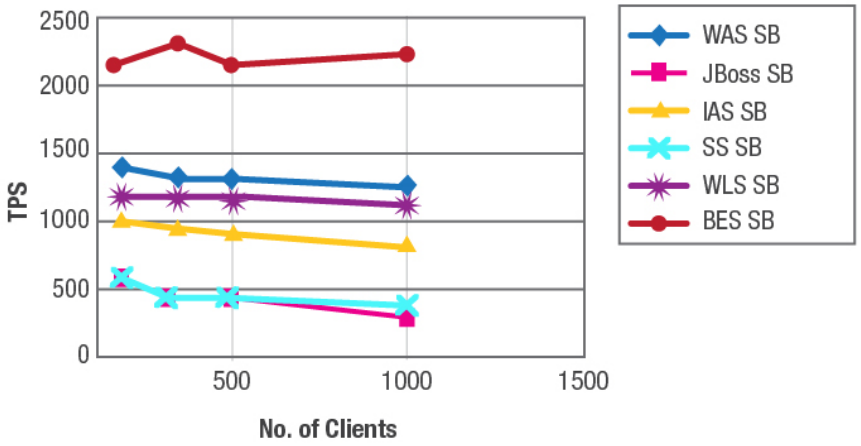
Distribuir o processamento em diferentes servidores funciona bem se o trabalho necessário para gerenciar a distribuição das requisições entre as várias máquinas for pequeno. O objetivo é manter cada máquina igualmente ocupada, como o investimento em mais *hardware* é desperdiçado se uma máquina estiver totalmente ocupada enquanto as demais estão ociosas. Distribuir a carga uniformemente entre várias máquinas é conhecido como **balanceamento de carga**.

É importante ressaltar que, para ambas as abordagens, a escalabilidade deve ser alcançada sem modificações na arquitetura da aplicação.

Na realidade, como a carga aumenta, as aplicações irão apresentar uma diminuição no *throughput* e um aumento exponencial no tempo de resposta. Isto acontece por duas razões.

Em primeiro lugar, com o aumento de carga, aumenta a disputa por recursos como CPU e memória por parte dos processos e threads na arquitetura do servidor.	Em segundo lugar, cada solicitação consome algum recurso adicional (espaço de buffer, locks, e assim por diante) e, eventualmente, estes recursos tornam-se esgotados o que limita a escalabilidade.

Apenas para ilustrar, a figura abaixo mostra como seis diferentes versões do mesmo aplicativo implementado usando diferentes servidores de aplicação JEE se comportam com um aumento de carga de 100 para 1.000 clientes.



Efeitos do aumento da carga de solicitação de cliente em plataformas JEE

11

3.2- Conexões simultâneas

Uma arquitetura pode ser projetada para suportar 1.000 usuários simultâneos. ***Como é que a arquitetura responde se esse número cresce significativamente?***

Se um usuário conectado consome alguns recursos, então provavelmente haverá um limite para o número de conexões que podem ser efetivamente suportados.

Um exemplo clássico desta situação ocorreu em um Internet Service Provider (ISP). Cada vez que um usuário se conectava ao serviço, um aplicativo do provedor gerava um novo processo em seu servidor responsável por apresentar anúncios para o usuário. Isso funcionou muito bem, mas cada requisição consumia recursos de memória e processamento consideráveis. Testes na infraestrutura de *hardware* revelaram que as máquinas deste ISP só poderiam suportar cerca de 2.000 conexões antes de sua memória virtual ficar sobrecarregada. Entretanto, para o modelo de negócio deste ISP a infraestrutura disponível deveria suportar 100.000 usuários.

Na época em que isso ocorreu, aumentar os recursos de *hardware* era uma proposta proibitivamente cara. Apesar dos esforços do *redesign* frenéticos, esta foi uma das causas deste ISP sair do negócio.

Este exemplo ilustra a importância da preocupação com as conexões simultâneas para uma solução de *software*.

12

3.3- Tamanho dos Dados e do Banco de Dados

Como é que uma aplicação se comporta se os dados que ela processa aumentam de tamanho?

Por exemplo, uma aplicação de intermediário de mensagem, talvez uma sala de bate-papo, pode ser concebida para processar mensagens com um tamanho médio esperado. Como a arquitetura irá reagir se o tamanho das mensagens cresce significativamente? De um modo um pouco diferente, uma solução de gestão de informação pode ser concebida para consultar dados a partir de um repositório de um tamanho especificado. Como a aplicação iria se comportar, se o tamanho do repositório crescesse 100 vezes mais? Este último está se tornando um problema tão grande que gerou toda uma área de pesquisa e desenvolvimento conhecido como *data intensive computing*.

3.4- Implantação

Qual é o esforço envolvido na implantação ou modificação de um aplicativo para uma crescente base de usuários?

Isto incluiria esforço para distribuição, configuração e atualização de novas versões. Uma solução ideal seria fornecer mecanismos automatizados que possam ser implantados de forma dinâmica e configurar um aplicativo para um novo usuário, capturando informações de registo no processo. Esta é, de fato, exatamente como muitos aplicativos são distribuídos hoje na Internet.

13

3.5- Algumas Reflexões sobre Escalabilidade

Projetar arquiteturas escaláveis não é fácil. Na maioria das vezes, a necessidade de escalabilidade não é evidente no início do projeto e não é especificada como parte da qualidade Requisitos de Arquitetura. Por isso se faz necessária a participação de um arquiteto mais experiente e atento para garantir que abordagens não escalonáveis não sejam introduzidas como componentes do núcleo da arquitetura da solução.

Mesmo quando a escalabilidade é um atributo de qualidade exigido, apenas avaliar se ela está sendo satisfeita, muitas vezes não é prático em termos de cronograma ou custo. É por isso que é importante para um arquiteto contar com modelos experimentados e testados e tecnologias.

14

4 - MODIFICABILIDADE

Todos os arquitetos de *software* estão cientes de que modificações em um sistema de *software* durante a sua vida útil são simplesmente um fato da vida. E, por este motivo, deve-se considerar a probabilidade de alteração no aplicativo uma boa prática durante definição da arquitetura. Quanto mais flexibilidade puder ser considerada no projeto inicial, menos dolorosas e caras serão as alterações.

A modificabilidade é uma medida de quão fácil pode ser para mudar uma aplicação para atender a novos requisitos funcionais e não funcionais.

Observe o uso de "pode" na frase anterior. Prever modificações requer uma estimativa de esforço e custo para fazer uma alteração e a realidade é que você só saberá ao certo quanto vai custar uma mudança depois de ter sido feita.

A medida de modificabilidade só é relevante no contexto de uma determinada solução arquitetônica. Esta solução deve ser expressa como um conjunto de componentes, as relações entre estes componentes e uma descrição de como os componentes interagem com o ambiente. Avaliar a modificabilidade requer que o arquiteto defina prováveis cenários de mudanças que capturam como os requisitos podem evoluir. Às vezes estes serão conhecidos com um grau razoável de certeza.

Na verdade, as mudanças podem até ser especificadas no plano de projeto para versões subsequentes. Na maioria das vezes, porém, as possíveis modificações terão de ser elicitadas pelos *stakeholders*, e definidas pela experiência do arquiteto.

15

Para cada cenário da mudança, o seu previsível impacto na arquitetura deve ser avaliado. Este impacto raramente é fácil de quantificar. Em muitos casos, o melhor que se pode atingir durante a análise do impacto é **identificar os componentes da arquitetura que necessitam de modificação**, ou uma demonstração de como a solução pode acomodar a modificação sem alterações.

Finalmente, com base nas estimativas de custo, tamanho ou do esforço para os componentes afetados, algumas quantificações úteis do custo de uma mudança podem ser feitas. Alterações isoladas para componentes individuais ou subsistemas fracamente acoplados são susceptíveis de serem menos onerosas do que aqueles que causam efeitos em toda a arquitetura. Se uma mudança provável parece difícil e complexa para ser feita, isso pode destacar uma fraqueza na arquitetura que pode justificar uma análise mais aprofundada.

Mesmo sendo uma coisa boa, um projeto com alta modificabilidade precisa ser pensado com cuidado. Uma arquitetura altamente modular pode tornar-se excessivamente complexa, incorrer em desempenho adicional e requerer significativamente mais esforço de concepção e construção.

Um risco que deve ser evitado a todo custo é incorporar complexidade desnecessária na arquitetura. Isso significa investir mais esforço em um sistema do que é essencialmente necessário. Os arquitetos pensam que sabem as necessidades futuras do seu sistema e decidem que é melhor fazer um projeto mais flexível ou sofisticado, para que ele possa acomodar as necessidades futuras. Isso soa razoável, mas requer uma bola de cristal confiável. Se as previsões estiverem erradas, muito tempo e dinheiro pode ser desperdiçado.

O relato a seguir apresenta um exemplo do que significa incorporar complexidade desnecessária para o projeto.

16

Exemplo

Em um projeto para um grande cliente, o arquiteto passou 5 meses, para estabelecer um projeto de arquitetura baseado no padrão de injeção de dependência para o tratamento de mensagens. O objetivo era fazer com que esta arquitetura extremamente robusta e criar um modelo de dados flexível para o serviço de mensagens e armazenamento de dados.

A teoria era que a arquitetura poderia ser reutilizada novamente em outro projeto com o mínimo de esforço, e seria fácil de injetar novos componentes devido à flexibilidade oferecida pela dependência injeção.

No entanto, a palavra **teoria** na frase anterior foi cuidadosamente escolhida. Os *stakeholders* do sistema ficaram impacientes, querendo saber por que tanto esforço estava sendo gasto em uma solução tão sofisticada, e pediu para ver algum progresso demonstrável. O arquiteto resistiu, insistindo que sua equipe não devia ser desviada e continuou a abraçar os benefícios em longo prazo da arquitetura.

Os *stakeholders* do sistema perderam a paciência e substituíram o arquiteto por alguém que estava promovendo uma solução muito mais simples baseada em servidor.

Este foi um caso clássico de *overengineering*. Enquanto a solução original era elegante e poderia ter colhido grandes benefícios em longo prazo, adotar uma abordagem ágil e que pudesse ser demonstrada ao longo do projeto era a chave para o sucesso aqui.

O segredo para o sucesso é concentrar-se nos requisitos conhecidos e evoluir e refatorar a arquitetura através de iterações regulares, enquanto a produção de código em execução, faz muito sentido em quase todas as circunstâncias.

Como parte deste processo, você pode analisar continuamente seu projeto para ver que melhorias podem ou não ser acomodadas no futuro. Trabalhar em estreita colaboração com as partes interessadas pode ajudar a desencadear necessidades futuras e eliminar aquelas que parecem improváveis.

17

5 - SEGURANÇA

A segurança é um assunto técnico complexo que só pode ser tratada superficialmente um pouco aqui. Em se tratando de arquitetura de *software*, a segurança se resume a entender os requisitos de segurança para uma aplicação e da elaboração de mecanismos para atendê-los. Os requisitos mais comuns relacionados à segurança são:

- **Autenticação**

Os aplicativos podem verificar a identidade de seus usuários e outras aplicações com as quais eles se comunicam.

- **Autorização**

Autenticando os usuários e aplicações definiriam direitos de acesso aos recursos do sistema. Por exemplo, alguns usuários podem ter acesso somente à leitura e aos dados do aplicativo, enquanto outros têm de leitura e escrita.

- **Criptografia**

As mensagens enviadas para e a partir da aplicação são criptografadas.

- **Integridade**

Isso garante que os conteúdos de uma mensagem não sejam alterados em trânsito.

- **O não-repúdio**

O remetente de uma mensagem tem prova de entrega e o receptor tem a garantia de identidade do remetente. Isto significa que nem posteriormente pode refutar sua participação na troca de mensagens.

Existem tecnologias bem conhecidas e amplamente utilizadas que suportam estes elementos de segurança do aplicativo. O Secure Socket Layer (SSL) e Infraestruturas de chave pública (PKI) são comumente usados em aplicações de Internet para fornecer autenticação, criptografia e não-repúdio.

18

6 - DISPONIBILIDADE

Disponibilidade está relacionada com a confiabilidade de um aplicativo. Se um aplicativo não está disponível para uso quando necessário, então é improvável que possa cumprir os seus requisitos funcionais.

Disponibilidade é relativamente fácil de especificar e medir. Em termos de especificações, muitas aplicações de TI devem estar disponíveis pelo menos durante o horário comercial. Como não há horário comercial on-line, a maioria dos sites da Internet desejam 100% de disponibilidade.

Falhas em aplicações podem levá-las a estar indisponíveis. Falhas geram impacto sobre a confiabilidade de um aplicativo, que geralmente é medida pelo tempo médio entre falhas. O período de tempo de indisponibilidade é determinado pela quantidade de tempo que leva para detectar falhas e reiniciar o sistema. Consequentemente, as aplicações que exigem alta disponibilidade, devem procurar minimizar ou, de preferência, eliminar o único ponto de falha. Pode-se procurar mecanismos que detectam automaticamente a falha e reinicia os componentes com falha.

Replicar componentes é uma estratégia testada e comprovada para alta disponibilidade. Quando um componente replicado falhar, a aplicação pode continuar executando usando réplicas que ainda funcionam. Isso pode levar a degradação do desempenho, enquanto um componente está com falha, mas a disponibilidade é não comprometida.

A **capacidade de recuperação** está intimamente relacionada com a disponibilidade. Um aplicativo é recuperável se ele tem a capacidade de restabelecer os níveis de desempenho requeridos e recuperar dados afetados após uma falha de aplicativo ou sistema.

Um sistema de banco de dados é o exemplo de um sistema recuperável. Saiba+

Saiba+

Quando um servidor de banco de dados falhar, ele não estará disponível até que ele se recupere. Isto significa reiniciar o aplicativo do servidor, e resolver quaisquer transações que estavam em execução

no momento da falha. Questões interessantes para aplicações recuperáveis são como as falhas são detectadas e a recuperação começa (de preferência automaticamente), e quanto tempo leva para se recuperar antes de o serviço ser restabelecido completamente. Durante o processo de recuperação, a aplicação é indisponível, e, portanto, o tempo médio de recuperação é uma métrica importante a considerar.

19

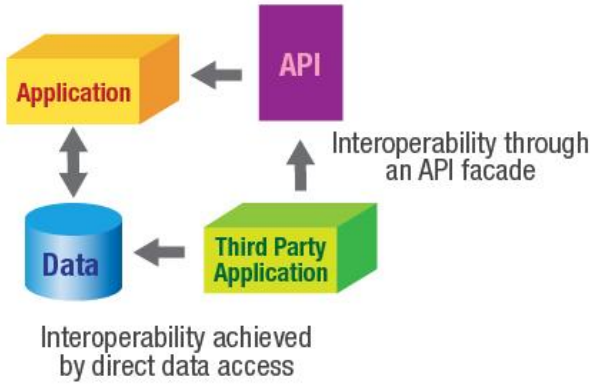
7 - INTEGRAÇÃO

Integração está relacionada com a facilidade com que uma aplicação pode ser incorporada em um contexto de outra aplicação.

O valor de uma aplicação ou componente de frequência pode ser aumentado se sua funcionalidade ou dados puderem ser usados de maneiras diferentes das que originalmente foram previstas pelo *design*er. As estratégias mais comuns para possibilitar a integração são através da integração de dados ou fornecendo uma API.

Integração de dados envolve armazenar os dados que um aplicativo manipula de forma que outras aplicações possam acessar. Isto pode ser tão simples quanto utilizar uma base de dados relacional padrão para armazenamento de dados, ou, talvez, a implementação de mecanismos para extrair os dados para um formato conhecido, tal como XML ou um arquivo de texto separado por vírgulas que outras aplicações podem tratar.

Com a integração de dados, as formas com que os dados são usados por outros aplicativos fica praticamente fora de controle do proprietário dos dados. Isso ocorre porque as regras de integridade de dados e de negócios impostas pela lógica do aplicativo são contornados.



Opções de integração

A alternativa é a **interoperabilidade** a ser alcançada através de uma API. Neste caso, os dados brutos que a aplicação possui estão escondidos atrás de um conjunto de funções que facilitam o acesso controlado aos dados. Desta forma, as regras de negócios podem ser garantidas através da implementação da API.

A escolha da estratégia de integração não é simples. Saiba+

Saiba+

Integração de dados é flexível e simples. Os aplicativos escritos em qualquer linguagem de programação podem processar texto ou acessar bancos de dados relacionais usando SQL. Construir uma API requer mais esforço, mas fornece um ambiente muito mais controlado, em termos de exatidão e segurança, para a integração. Também é muito mais robusto, a partir de uma perspectiva de integração, como os clientes da API são isolados a partir de muitas das mudanças nas estruturas de dados subjacentes. Eles não quebram toda vez que o formato é modificado, como os formatos de dados não são diretamente expostas e acessados. Como sempre, a melhor escolha da estratégia depende do que você deseja alcançar, e que limitações existem.

20

8 - Outros atributos de qualidade

Existem numerosos outros atributos de qualidade que são importantes em vários contextos de aplicação. Alguns deles são:

- **Portabilidade**

Pode uma aplicação ser facilmente executada em uma plataforma de *software* ou *hardware* diferente daquela em que foi desenvolvida? Portabilidade depende das opções de tecnologia de *software* utilizadas para implementar a aplicação e das características das plataformas que ela precisa executar. Códigos portáteis terão suas dependências com a plataforma, isoladas e encapsuladas em um pequeno conjunto de componentes que podem ser substituídos, sem afetar o resto da aplicação.

- **Testabilidade**

Quão fácil ou difícil é uma aplicação para ser testada? Decisões iniciais do projeto podem afetar significativamente a quantidade de casos de teste que são requeridos. Como regra geral, quanto mais complexo for um projeto, mais difícil é testá-lo. Simplicidade tende a promover a facilidade de teste. Da mesma forma, escrevendo menos de seu próprio código através da incorporação de componentes pré-testados reduz o esforço de teste.

- **Suportabilidade**

Esta é uma medida de quão fácil uma aplicação é suportada, uma vez que é implantada. Suporte tipicamente envolve o diagnóstico e correção dos problemas que ocorrem durante o uso do aplicativo. Sistemas suportáveis tendem a fornecer instalações explícitas para diagnóstico, como a aplicação logs de erro que registra as causas de falhas. Eles também são construídos de forma modular, de modo que correções de código podem ser implantadas sem comprometer o uso do aplicativo.

21

8.1- Trocas de *design*

Se a vida de um arquiteto fosse simples, *design* seria apenas envolver a construção de políticas e mecanismos para uma arquitetura para satisfazer os requisitos de qualidade para uma dada aplicação.

Infelizmente, a realidade não é bem essa. Atributos de qualidade não são ortogonais. Eles interagem de formas sutis, ou seja, um projeto que satisfaz um requisito de qualidade pode ter um efeito prejudicial sobre outro. Por exemplo, um sistema altamente seguro pode ser difícil ou impossível integrar em um ambiente aberto. Um aplicativo altamente disponível pode ter um desempenho inferior para atender a maior disponibilidade. Um aplicativo que exige alta performance pode ser amarrado a uma plataforma específica, e, portanto, não ser facilmente transportável.

Compreender as trocas entre os requisitos de qualidade, e projetar uma solução que faz compromissos sensatos é uma das mais difíceis partes do papel arquiteto. É simplesmente impossível satisfazer plenamente todos os requisitos concorrentes.

22

RESUMO

Os arquitetos devem despender um grande esforço para compreender os requisitos de qualidade e compreender o que deve ser feito para resolvê-los. Parte da dificuldade é que os atributos de qualidade não são sempre explicitamente indicados nos requisitos, ou adequadamente capturados pelos analistas de requisitos da equipe. É por isso que um arquiteto deve ser envolvido com os requisitos, para que eles possam fazer as perguntas certas para capturar os requisitos de qualidade que devem ser atendidos. Em geral os atributos de qualidade de *software* incluem:

- Desempenho
- Escalabilidade
- Modificabilidade
- Segurança

- Disponibilidade
- Integração

Naturalmente, a compreensão dos requisitos de atributos de qualidade é apenas um pré-requisito necessário para projetar uma solução satisfatória.

Atributos de qualidade em conflito são uma realidade em todas as aplicações de complexidade até mesmo simples. Criando soluções que satisfaçam estes requisitos de forma adequada é extremamente difícil, tanto técnica como socialmente. Este último envolve comunicações com as partes interessadas para discutir tolerâncias do projeto, descobrindo cenários quando certas exigências de qualidade podem ser seguramente atendidas, e comunicando constantemente estas ações com as partes interessadas de modo que estes possam entendê-las e para que não ocorra problemas no projeto.

UNIDADE 3 – DEFININDO A ARQUITETURA DE *SOFTWARE*
MÓDULO 3 – PRINCÍPIOS DO PROJETO DE ARQUITETURA
01

1 - CONTEXTO DOS PRINCÍPIOS DO PROJETO ARQUITETURA DE *SOFTWARE*

Vimos anteriormente que o Projeto de Arquitetura é o passo fundamental no processo de definição da arquitetura de um *software*. É neste passo do processo que a estrutura do *software* é concebida e as responsabilidades dos componentes são definidas.

Agora vamos entender os operadores gerais de um projeto de arquitetura e como eles podem ajudar o arquiteto a decompor um sistema em componentes e atingir os atributos de qualidade desejado.

Cada operador do projeto modifica efetivamente um conjunto de atributos de qualidade associado a um determinado projeto que pode afetá-lo positiva ou negativamente. Operadores de *design* mais comuns são:

- decomposição,
- replicação,
- compressão,
- abstração e
- compartilhamento de recursos.

Neste módulo vamos explorar o nível de projeto arquitetural, a elaboração da arquitetura com operadores de projeto e as estratégias funcionais de *design*.

02

2 - NÍVEL DE PROJETO ARQUITETURAL

O nível de projeto arquitetural incide sobre o sistema no nível de componentes e conectores e seus arranjos que abordam requisitos de qualidade.

2.1- Aplicando princípios de *design*

Métodos de *design* por si só são difíceis de aplicar. Por exemplo, quando se aplica o método de variação sistemática, como você decide qual atributo para se concentrar e como transformar o projeto em alguma variação desse projeto? Compreender atributos de qualidade ajuda a raciocinar sobre características específicas do sistema. Princípios de *design* orientam as decisões que devem ser feitas durante a aplicação de métodos de concepção.

Variação sistemática é uma técnica que ajuda o arquiteto descobrir ou criar um campo de solução para um determinado problema. A partir desta solução, o arquiteto pode avaliar os candidatos e escolher o projeto mais adequado ou, pelo menos, reduzir a escolha a um conjunto menor de candidatos.

Mas como é que o arquiteto cria e avalia cada opção de projeto? Vejamos a seguir.

03

O arquiteto aplica alguma operação de **criação** ou **transformação** para gerar um novo projeto ou para refinar um projeto existente. Esta transformação pode decompor a solução em mais componentes, ou abstrair uma parte do sistema para fazer seu funcionamento interno transparente para outros componentes. Cada operação preserva a funcionalidade representada pela arquitetura, mas muda sua arquitetura estrutural e assim, mudam alguns dos atributos de qualidade do sistema que está sendo descrito.

Cada projeto transformado pode sofrer novas transformações, produzindo assim um caminho através de um conjunto de soluções potenciais. O arquiteto também pode definir vários caminhos que dão forma a uma árvore de solução.

A qualquer momento o arquiteto pode aplicar técnicas de avaliação de projeto para testar os atributos de qualidade do projeto. Assim, podemos ver como os métodos e princípios da arquitetura do projeto podem ser aplicados para a resolução de problemas.

As operações não são necessariamente transitivas; isto é, a ordem na qual elas são aplicadas pode ser importante. Aplicando um padrão de *design* como MVC seguido por uma abstração para melhorar a

modificabilidade pode resultar em um *design* diferente com atributos de qualidade diferentes do que aplicar a abstração primeiro seguido pelo padrão MVC. Felizmente, muitos *design*ers têm percorrido o mesmo caminho e documentaram esses padrões.

04

2.2- Usando Systems *Thinking*

Esta seção sobre o **pensamento sistêmico** nos ajudará a entender o escopo do projeto de arquitetura. Uma coisa que distingue a arquitetura de *software* e o *design* do projeto é o **escopo** do problema a ser abordado.

Arquitetura *Software* requer pensamento sistêmico, considerando a visão inteira do problema, não apenas subproblemas. Subproblemas nos ajudam a resolver problemas complexos, mas o arquiteto deve continuamente sintetizar as soluções para os subproblemas, a fim de avaliar a solução global do sistema.

O pensamento sistêmico requer um equilíbrio entre olhar para a visão macro e olhar para os detalhes. É fácil ficar sobrecarregado com a visão macro. O arquiteto de *software* experiente pode mudar as perspectivas de nível de sistema para nível do componente conforme necessário.

Isto é semelhante ao processo de escrita. O aspecto mais importante do primeiro passo é expor suas ideias. Uma vez que suas ideias estão escritas em papel (ou em uma ferramenta de modelagem), você pode começar a formalizar as ideias. É mais fácil ver a visão macro desta maneira.

05

3 - Arquitetando com Operadores de *Design*

Operador de *design* é uma ferramenta de *design* fundamental para a criação de um projeto arquitetônico. A aplicação de operadores de *design* inicia com uma representação de um sistema como um único componente que tem características e funções visíveis externamente.

O arquiteto inicialmente decompõe o sistema em dois componentes, cada um com um propósito bem definido e que interagem através de alguns conectores. Em termos de estrutura modular, o sistema agora aparece como dois módulos que compartilham um conjunto de regras de *design* global, que definem os conectores.

Como já dissemos no início do nosso módulo, os operadores de *design* de *software* comuns são:

- Decomposição;
- Replicação;
- Compressão;
- Abstração; e
- Compartilhamento de recursos.

O método de projeto é uma **abstração técnica** que permite que você se concentre no essencial de um problema ignorando características não essenciais na resolução do problema.

Abstração como um princípio de *design* representa abstrações na solução, a fim de melhorar os atributos de qualidade, tais como modificabilidade e integrabilidade.

06

Existem outras **categorias** de princípios de *design*, tais como:

- *design* de interface gráfica de usuário (GUI),
- *design* de interação do usuário,
- *design* orientado a objeto,
- programação Java,
- programação de Thread Java,
- análise de domínio de aplicativo e
- análise de requisitos.

Você provavelmente irá identificar os seus princípios iniciais, que são específicos para seu domínio ou arquitetura do aplicativo. Estas declarações de primeiros princípios formam as metas ou objetivos do projeto, aos quais todos os membros da equipe devem obedecer.

07

3.1- Decomposição

A **decomposição** é a operação de separar a funcionalidade em componentes distintos que têm interfaces bem definidas.

Por exemplo, a decomposição de um sistema em um cliente e um servidor usando Open Data Base Connectivity (ODBC) ou Java Conectividade de Dados Base (JDBC) como interface para o servidor.

A decomposição é o princípio mais importante e é, muitas vezes, utilizada em qualquer campo de projeto de engenharia. A partir de uma visão sistêmica, a decomposição separa um sistema em dois ou mais sistemas (chamados de subsistemas). A decomposição é usada para conseguir uma variedade de qualidades e atributos incluindo modificabilidade, portabilidade e desempenho.

Os dois tipos de decomposição são:

Parte-todo

- A decomposição **Parte-todo** divide um sistema em um conjunto de subcomponentes que não se sobrepõem funcionalmente. Esta decomposição pode ser efetuada de forma recursiva para formar uma hierarquia de componentes funcionais. A decomposição uniforme divide o sistema em um conjunto de componentes que são eles próprios estruturalmente semelhantes. Por exemplo, cada componente tem um componente de apresentação, um componente lógico do aplicativo e um componente de acesso de armazenamento de dados.

Generalização-especialização

- Em uma decomposição **generalização-especialização**, há potencial funcionalidade de sobreposição entre os componentes. Cada aplicação do operador de decomposição divide um componente em dois subcomponentes. A escolha de onde traçar a linha entre os componentes é impulsionada por atributos de qualidade que você está tentando melhorar.

08

3.1.1- Identificando Componentes Funcionais

A identificação dos componentes do sistema é feita com base na especificação funcional inicial do sistema que utiliza decomposição.

Os componentes do sistema formam o mais alto nível de decomposição funcional da arquitetura estrutural do sistema. Esta abordagem começa a partir da visão do sistema como um componente único no interior de alguns ambientes.

Os componentes são identificados de várias maneiras:

- **Interfaces;**
- **Domínios;**
- **Camadas de abstração funcionais.**
- **Entidades de domínio.**

Interfaces.

As interfaces entre o sistema e as entidades externas devem ser associadas a componentes. Você pode associar um componente por interface ou possivelmente associar mais de uma interface para um determinado componente, mas você não deve fragmentar uma única interface em vários componentes.

Domínios.

Existem dois tipos de domínios: domínios de aplicação e domínios de solução. Ambos são fontes para a identificação de componentes. UMA aplicação de *software* pode dirigir mais de um domínio de aplicação. Cada um desses domínios pode ser modelado por um componente separado.

Por exemplo, em um sistema de publicação você pode identificar um componente de criação de conteúdo e um componente que publica o conteúdo, sendo cada domínio separado, mas relacionados. No domínio da solução, você pode identificar os componentes como repositório de conteúdo, Entrega de Conteúdo e Gerenciamento de Projetos.

Camadas de abstração funcionais.

Camadas de abstração funcionais decompõem o sistema em uma hierarquia funcional. A partir desta hierarquia funcional, é possível identificar as funções comuns que podem ser representadas por componentes compartilhados.

Entidades de domínio.

Assim como podemos representar um domínio como um componente, podemos representar entidades de domínio específicas como componentes. Essas entidades podem ser encontradas na literatura do domínio da aplicação e no modelo de objeto de aplicação (se foi criado) como parte da atividade de análise de requisitos. No entanto, isso pode não ser sempre adequado para modelar essas entidades diretamente como componentes. Os modelos de domínio de aplicativo descrevem o domínio do aplicativo ou um problema e não pode ser apropriado nesta solução modelo de domínio.

09

3.1.2- Composição / Agregação

A composição é relacionada à decomposição, mas não é exatamente o oposto de decomposição.

A **decomposição** envolve subdividir um componente em, possivelmente, dois componentes funcionais distintos. **Composição** envolve a montagem de componentes para formar novos componentes.

Por vezes, as duas não se distinguem uma da outra. Você pode perceber que o seu componente pode ser montado a partir de um banco de dados relacional, servidor Web, e alguns componentes personalizados (a lógica do aplicativo). Neste sentido você compôs o sistema usando três componentes: dois com propriedades bem conhecidas e uma terceira que é a funcionalidade restante não alcançada com os dois primeiros. Mas esta composição pode também ser pensada como uma decomposição do sistema em três componentes.

3.2.3- Comunicação de Componentes

Quando um componente é dividido em dois componentes, existe um **canal de comunicação implícita** introduzido entre eles. A comunicação realizada através da interface pode ser síncrona ou assíncrona. Comunicação assíncrona dissocia o processamento de dois componentes, o que os tornam concorrentes. Isso pode melhorar o desempenho e a confiabilidade. No entanto, isso aumenta a complexidade do sistema, tornando-o mais difícil de criar e depurar, e requer funções de domínio adicionais.

10

3.2- Replicação

Replicação, também conhecida como redundância, é a operação de duplicação de um componente a fim de aumentar a confiabilidade e o desempenho.

A replicação pode ser alcançada de duas formas:

Redundância

- onde existem várias cópias idênticas de um componente que executam simultaneamente.

Programação N-versões

- onde existem várias implementações diferentes da mesma funcionalidade.

Programação N-versões envolve a execução **simultânea de componentes redundantes** para executar uma função e usando um algoritmo para determinar qual dos vários resultados está correto.

Redundância está relacionada com o padrão de **automonitoramento**. O sistema de autocontrole é capaz de detectar algumas falhas de componentes e pode notificar um operador ou realizar uma alteração de componente de forma automática. Diferentes estratégias de replicação podem ser usadas com diferentes conjuntos de componentes num sistema.

11

3.3- Compressão

A compressão é o oposto de decomposição, embora a composição não seja exatamente o oposto de decomposição.

Compressão envolve agrupar os componentes em um único componente ou remoção de camadas ou interfaces entre os componentes.

Composição envolve o acoplamento ou a combinação de dois componentes para formar um novo sistema. No primeiro caso, não há nenhum vestígio de um ou mais componentes originais. Neste último, a componentes individuais ainda existem como subcomponentes. Composição é uma forma de síntese utilizada para encontrar novos comportamentos ou funções, combinando diferentes componentes em uma tentativa de simplificar uma solução ou encontrar novas soluções através da reutilização de soluções existentes.

Compressão não tem de ser aplicada de maneira uniforme. Por exemplo, você pode remover alguns componentes de camada intermediária, permitindo à camada de apresentação acesso direto ao banco de dados para melhorar o desempenho. Este tipo de compressão não uniforme pode fazer a solução mais complexa porque existem dois modos de interagir com a base de dados. Além disso, faz que o componente mais difícil seja mantido.

Uma forma de compressão chamada **camada** *straddling* é frequentemente usado em protocolos de comunicação para reduzir a sobrecarga de desempenho associada protocolos em camadas. Isto é diferente da utilização do termo de compressão para melhorar o desempenho de comunicação através da redução da quantidade de dados transmitidos, o que é chamado *de* **compressão de dados**.

12

3.4- Abstração

A **abstração** oculta informações através da introdução de uma camada semanticamente rica de serviços, ao mesmo tempo que esconde os detalhes de implementação.

Ela também é referida como a criação de uma máquina virtual. Uma máquina virtual não é necessariamente um intérprete complexo da linguagem de programação. No sentido geral, qualquer coisa que esconde a implementação de algum sistema é considerada uma máquina virtual. Este inclui interfaces, como JDBC ou ODBC.

A máquina virtual Java (JVM) abstrai o sistema operacional e o *hardware*, fazendo programas em Java portável entre ambos. JDBC abstrai a interface para um banco de dados relacional, fazendo um programa Java portátil em todas as plataformas de banco de dados. O Document Object Model (DOM XML) abstrai uma implementação parser XML.

Como você pode ver, a portabilidade se aplica a muitos aspectos de um sistema. Um atributo de qualidade única portabilidade é específico a um componente ou tecnologia.

13

3.4.1- Máquinas Virtuais e Adaptabilidade

Uma máquina virtual faz um sistema mais adaptável aos processos de negócios dos usuários finais. A ideia é converter requisitos de qualidade em uma linguagem para descrever a semântica de um aplicativo ou sistema.

As máquinas virtuais podem ser implementadas com uma linguagem baseada em texto interpretado ou compilado (como Java ou Visual Basic), ou um mecanismo de script simples que tem um ambiente de *design* GUI. Esta abordagem aumenta a complexidade de um sistema por adição de não apenas as funções da própria máquina virtual, mas também funciona para criar ou modificar configurações ou

scripts.

Uma variação desta técnica é o **interpretador de comandos**. Um intérprete de comandos permite que um sistema seja controlado pela execução de Comandos. Por exemplo, iniciar, parar, suspender e reiniciar são funções típicas do sistema que não têm nada a ver com o domínio da aplicação, mas são necessárias a fim de alcançar outros atributos de qualidade.

Estes tipos de comandos ajudam a tornar um sistema mais utilizável porque um operador pode reiniciar depois de fazer algumas mudanças de configuração importantes para melhorar o desempenho ou a segurança.

14

3.5- Compartilhamento de recursos

Compartilhamento de recursos é o encapsulamento de dados ou serviços a fim de compartilhá-los entre vários componentes de clientes independentes, melhorando a integração, portabilidade e modificabilidade.

O compartilhamento de recursos é útil quando o recurso em si pode ser escasso.

Por exemplo, um repositório de informações, tais como segurança do usuário, pode ser acessado por vários aplicativos independentes. Várias aplicativos Web podem compartilhar clusters de servidores Web para tirar proveito dos recursos computacionais e até mesmo um único endereço IP. Isso pode simplificar a implantação de aplicações, uma vez que não é necessária a instalação de um novo servidor Web, cada vez que um novo aplicativo Web é criado.

Um servidor de banco de dados relacional também é um recurso computacional compartilhado. Saiba+

Compartilhamento de recursos é comumente aplicado com a abstração, como vimos em bancos de dados e com interfaces de programação de aplicações (APIs) como JDBC e ODBC. Um fornecedor de aplicativo corporativo também pode expor alguns componentes como recursos compartilhados de modo que um cliente possa construir os seus próprios aplicativos para alavancar os recursos e dados dos componentes comerciais.

Saiba+

Diferentes aplicações podem armazenar seus próprios dados em um único servidor de banco de dados. Partilha de recursos pode simplificar alguns aspectos de configuração e gerenciamento de sistemas.

15

4 - ESTRATÉGIAS DE *DESIGN* FUNCIONAIS

Estratégias de *design* funcionais são mais complexos que os princípios de *design* individuais. Elas sugerem uma **estratégia funcional de decomposição** para atingir atributos de qualidade específicos. Estas estratégias podem ajudar a orientar o arquiteto em como ele decompõe um sistema com base nas necessidades funcionais e não funcionais do sistema.

Duas estratégias de **design** funcionais serão a seguir apresentadas:

- automonitoração e
- recuperação.

16

4.1 Automonitoração

Um sistema que tem automonitoramento é capaz de detectar certos tipos de falhas e reagir a elas de forma adequada, possivelmente, sem envolver um operador ou notificando um operador sobre uma condição específica.

A funcionalidade adicional para o sistema não está no domínio de aplicação, e deve ser documentada como tal. Existem duas **abordagens básicas** para a automonitorização:

- O monitoramento do processo
- O monitoramento de componentes.

Um monitoramento de processo é uma camada "acima" do aplicativo ou sistema que vigia o sistema. Na monitorização dos componentes, cada monitor é responsável por monitorar o seu próprio componente e relatar problemas para o próximo componente monitor de nível superior. Um exemplo simples é um

script que reinicia um servidor Web quando um evento detecta que a Web servidor não está em execução.

Automonitoração destina-se a melhorar a confiabilidade, mas introduz uma sobrecarga computacional adicional que pode afetar o desempenho. A capacidade de manutenção pode também ser afetada negativamente pelo aumento de tamanho e complexidade do sistema.

17

4.2 Recuperação

As funções de recuperação estão relacionadas com a capacidade de recuperação. Recuperabilidade é geralmente uma qualidade que é introduzida com base em determinadas decisões de *design*; normalmente não é um requisito de produto com base no domínio de aplicação.

A recuperação está relacionada com confiabilidade porque pode afetar o tempo médio de recuperação (MTTR), uma medida comum de confiabilidade pode ser necessária para um sistema restaurar-se a um estado estável antes do problema ocorrer.

18

RESUMO

Neste módulo exploramos os operadores de *design* mais comuns, que são: decomposição, replicação, compressão, abstração e compartilhamento de recursos. Os seguintes tópicos foram abordados:

Nível de projeto arquitetural. O nível de projeto arquitetural é o desenho do sistema em termos de componentes e conectores e seus arranjos. É neste nível de *design* que os operadores de projeto de arquitetura são aplicados.

Elaborando a Arquitetura com operadores de projeto. Os operadores de *design* são transformações explícitas de um projeto. Os operadores de *design* comuns são: decomposição, replicação, compressão, abstração e compartilhamento de recursos.

Estratégias funcionais de *design*. Operadores de projeto nos ajudam a alcançar os primeiros princípios de *design* (como a modularidade, ocultação de informações, e gestão global de complexidade). *Design*s arquitetônicos também podem ser transformados mediante a aplicação de alto nível de operações "macro", tais como padrões de projeto e outras estratégias de *design*.

UNIDADE 3 – DEFININDO A ARQUITETURA DE *SOFTWARE*
MÓDULO 4 – *SOFTWARE* PARTNERS

01

1- *PATTERNS* NO CONTEXTO DE ARQUITETURA DE *SOFTWARE*

Vimos anteriormente os passos do processo para a definição da arquitetura de um projeto de *software*. Estes passos são definidos como:

1 – Determinar os Requisitos de *software*;

2 – Projeto de Arquitetura; e

3 – Validação da Arquitetura.

Este processo não é cascata, onde cada passo é executado apenas uma única vez. Ou seja, a definição da arquitetura é um **processo iterativo**. Assim, um projeto é elaborado com base nos requisitos conhecidos, propostos e validados. Entretanto, durante a validação, modificações podem surgir ou os requisitos podem necessitar ser mais bem definidos e compreendidos. Assim, estes passos podem ser repetidos até que a equipe de *design* considere que as exigências do projeto foram atendidas.

Entendido este processo, percebemos que um dos passos mais críticos é o projeto de arquitetura. É neste passo do processo que a estrutura do *software* é concebida e as responsabilidades dos componentes são definidas.

Para a definição do projeto de arquitetura, em linhas gerais, o *framework* a ser utilizado é selecionado, os principais componentes são definidos e são identificadas as responsabilidades, as dependências e as interfaces destes componentes.

02

Desta forma, no cenário de definição de uma arquitetura, é crítico o entendimento do que é um *framework*.

Um *framework* pode ser definido como uma arquitetura reutilizável que fornece estruturas genéricas para um projeto de *software*.

A adaptação de um *framework* pode ser realizada para atender a diferentes necessidades. Ele não tem toda a funcionalidade específica de uma aplicação, por isso não pode ser considerada uma aplicação completa. Mas através de mecanismo como o de heranças ou como o de instanciações dos seus

componentes, as aplicações podem ser construídas, adicionando-se as funcionalidades necessárias a um *framework*.

Um conceito intimamente ligado ao do *framework* é o de ***design patterns***.

Diferente dos *framework*s, os ***design patterns*** possibilitam o reuso de microarquiteturas mais abstratas sem a necessidade de uma implementação. Para ficar mais claro, enquanto os *framework*s são implementados em uma linguagem de programação, os padrões, ou *patterns*, apenas definem as formas de utilizar estas linguagens. Em geral, os *design patterns* podem ser utilizados em um número maior de aplicações que os *framework*s justamente por serem menos especializados.

Em resumo, os *design patterns* ajudam na definição e na documentação de um *framework* que podem englobar dezenas de *design patterns*.

03

2 - O QUE SÃO *PATTERNS* (PADRÕES)

Um **padrão** é um elemento de *design* recorrente, seja em projetos de *software* ou no mundo real. Um padrão de *software* é uma solução para um projeto de *software* ou um problema de codificação que tem sido útil em pelo menos três oportunidades – uma exigência conhecido como Regra de Três.

É através desta recorrência que o padrão mostra que é uma solução que funciona em diferentes cenários.

Patterns é o resultado de quando diferentes pessoas olham para um código e percebem semelhanças na forma como este código é estruturado. Entretanto, é necessário que seja elaborada a documentação deste padrão para que outros que não tiveram a oportunidade de olhar para esses projetos iniciais possam entender e utilizar este padrão.

Você começa a notar semelhanças na maneira como algo é implementado e usado quando você identificar algo acontecer, pelo menos, três vezes. A mesma estrutura básica é vista em todos os casos, mas podem existir variações. Um padrão pode ser visto e utilizado em centenas de lugares, mas nunca ser precisamente o mesmo em todos os locais.

Para melhor entendermos o que são os *Patterns* vamos explorar nas seções a seguir alguns de seus atributos.

04

- **Modelos reutilizáveis**

É ótimo quando é possível reutilizar um *software* que você escreveu em algum momento no passado, reutilizar a classe ou função que você criou para outro projeto, ou usar algum *software* de código aberto que tem toda uma comunidade por trás dele, corrigir bugs e adicionar novas funcionalidades. Se as interfaces funcionaram, você pode usar o *software* sem qualquer alteração. Mas mesmo se você tiver que realizar algum tipo de adaptação, ainda assim é fácil perceber o benefício de não ter que escrever toda a solução a partir do zero.

É desta forma que os padrões funcionam, porque eles fornecem soluções reutilizáveis para os problemas. Às vezes você pode reutilizar um padrão como ele é apresentado; em outros momentos é necessário fazer algumas pequenas alterações. Na maioria dos padrões, a reutilização é em nível de *design*, pois eles não contêm o código que pode ser copiado e colado. Em vez disso, eles contêm informações de projeto que podem ser utilizados no *design* de um projeto de *software*. Mesmo reutilizando apenas o projeto, ainda assim o trabalho é agilizado.

Elementos de *design* reutilizáveis devem ser:

- modulares,
- flexíveis e
- capazes de serem utilizados mais de uma vez.

Sistemas de código aberto contêm *software* reutilizável – um fato de que muitos profissionais se aproveitam. A maioria das licenças de código aberto permitem que pequenas partes do *software* possam ser reutilizadas ao invés do todo, e você pode personalizá-lo para atender a sua situação específica.

05

Outro atributo de um *design* reutilizável é que **pode ser comunicado**, ou melhor, há necessidade de comunicação. Quando você cria um projeto reutilizável, você deseja que outras pessoas o utilizem também e uma boa comunicação é essencial para que isso ocorra. É necessário informar como usá-lo, como configurá-lo, e como ele funciona. *Patterns* ajudam na descrição do projeto. Um padrão contém informações suficientes para ajudá-lo a recriar o projeto e entender por que a solução é a melhor para o problema que está sendo solucionado.

Assim, os Padrões permitem que duas coisas sejam feitas ao mesmo tempo: **descrever algo** e **descrever como essa coisa é feita**.

Um padrão de *software* é a descrição de uma solução comprovada para um problema de *design* com informação suficiente para que, quem desejar utilizá-lo, possa adaptá-lo para atender às suas necessidades de projeto. Um elemento-chave desta definição é que o padrão contenha informações suficientes para que você possa ler e entender o problema e a solução, e ver quando e como você pode adaptá-lo à sua própria situação.

06

- **Soluções comprovadas**

Patterns descrevem soluções comprovadas, ou seja, aquelas soluções que foram testadas e comprovadas ao longo do tempo.

Os *patterns* ajudam você a verificar o que funcionou no passado evitando que você tenha que "reinventar a roda". O objetivo é evitar que você perca tempo tendo que definir uma nova solução para problemas que já têm uma solução comprovada, concentrando seu tempo e esforço sobre os novos problemas que ainda não foram resolvidos.

Se você reutilizar uma solução comprovada, você tem uma boa chance de alcançar o sucesso mais rapidamente do que se você tentar inventar uma solução a partir do zero.

Assim como os padrões ajudam você a evitar resolver o mesmo problema várias vezes, eles ajudam a evitar cometer os mesmos erros de forma recorrente. Um padrão explica por que é a solução correta evitando que você utilize soluções menos eficazes. Um padrão também informa quando a solução não é apropriada para o seu problema o que ajuda você saber quando deve olhar para um padrão diferente.

07

- **Ferramenta Educacional**

Cada nova área de programação em que você começar a trabalhar terá algumas informações básicas que você precisa saber a fim de tornar-se eficaz.

Os padrões podem ajudar você a explorar um novo domínio de computação e diminuir o tempo de aprendizado. Ler os padrões lhe dá a compreensão das questões mais interessantes e úteis. Os padrões ajudam entender as soluções mais eficazes e entender o vocabulário apropriado para a tecnologia utilizada.

Domínios tão diferentes como banco, *e-commerce*, telecomunicações, computação de alto desempenho, computação empresarial têm o seu próprio conjunto de padrões, mas todos eles usam os mesmos padrões arquitetônicos comuns. Alguns dos padrões são utilizados em mais de um destes domínios.

- **Guias do Sistema**

Patterns fornecem orientação no nível de arquitetura, mostrando-lhe como estruturar seu sistema.

08

- **Vocabulários arquitetônicos**

Uma das grandes vantagens dos Padrões é que eles definem um **vocabulário comum**. Quando alguém diz "Singleton" ou "MVC", por exemplo, você vai saber o que esses termos significam.

Um vocabulário arquitetônico comum ajuda a todos na equipe de *design* falar a “mesma língua”. Explicar o seu projeto para alguém é mais fácil quando você utiliza princípios e padrões comuns que você e seu ouvinte sabem. *Patterns* tornar a explicação mais fácil.

Padrões também oferecem um vocabulário mais rico, de modo que você não tem que fazer o projeto com base nas construções primitivas de uma linguagem ou metodologia, tais como ponteiros ou classes. Este tipo de linguagem é especialmente útil em código (OO) orientado a objetos, em que as relações entre as classes e objetos exigem cuidado *design*.

Fonte: http://vidadeprogramador.com.br/

09

- **Base de conhecimento**

Padrões são bons como fonte de experiências e servem como base de conhecimento, porque eles descrevem por que algo deve ser feito de determinada maneira.

Especialistas adquiriram seus conhecimentos através de anos de experiência, e eles invocam este conhecimento quando precisam resolver um problema. Quando esse conhecimento é capturado em um padrão, permite que você veja um problema através dos olhos de um especialista.

Para alcançar os benefícios da especialização contidos nos padrões, você precisa estar familiarizado com diferentes padrões – especialmente os que abordam os tipos de problemas que normalmente você enfrenta. Há muitas maneiras de fazer isso, incluindo a leitura da literatura sobre padrão e fazer o seu próprio manual ou catálogo observando novos padrões que você encontrar.

10

3 - O QUE PADRÕES NÃO SÃO

Às vezes, saber o que algo **não é** ajuda você a vê-lo mais claramente, e isso particularmente é verdade quando se trata de padrões. Aqui estão algumas das coisas que as pessoas frequentemente confundem com padrões:

- Padrões não são *framework*s
- Padrões não são algoritmos
- Padrões não são patentes.
- Padrões não são exclusivamente para o projeto OO.
- Padrões não são solucionadores de problemas universais.

Os padrões de oferecem soluções para os problemas recorrentes, mas os problemas que enfrentamos nos projetos de *software* não são idênticos para os problemas descritos em cada padrão. Assim, você tem que saber quando um padrão se aplica à sua situação e quando deverá ser adaptado.

"Se a única ferramenta em sua caixa de ferramentas é um martelo, todo mundo parece um prego." Por esse motivo, os padrões não devem ser as únicas ferramentas em sua caixa de ferramentas. Você também deve saber algoritmos relevantes e ter outros recursos os quais você pode utilizar para resolver seus problemas de *software*.

Padrões não são *frameworks*

*Framework*s são pedaços de código reutilizáveis, e os padrões são explicações textuais de *framework*s, mostrando como eles foram construídos e como eles podem ser personalizados.

Padrões não são algoritmos

Um algoritmo descreve um processo repetitivo de etapas bem definidas que produzem algum resultado, mas não explicam quando eles devem ser usados ou porque eles são a solução adequada. Um padrão inclui esse raciocínio.

Padrões não são patentes.

Estes dois termos podem ser confundidos porque eles têm sons semelhantes (quando falados em inglês: *patterns* e *patents*). A concessão de patentes é o direito exclusivo para produzir um produto útil, enquanto que um padrão é uma descrição de como resolver um problema de uma forma que tem provado ser eficaz.

Padrões não são exclusivamente para o projeto OO.

Padrões de *software* primeiro ganhou destaque através da comunidade OO, mas muitos padrões não OO estão disponíveis.

Padrões não são solucionadores de problemas universais.

Padrões fornecem soluções para problemas conhecidos. Entretanto, eles não vão fazer de você um expert instantaneamente. Você ainda precisa aplicar sua própria criatividade e inteligência para determinar como padrões se encaixam em seu projeto de *software*.

11

4 - ENTENDENDO A ESTRUTURA DOS *PATTERNS*

Agora que você sabe o que os padrões são e o que eles não são, vamos dar uma olhada na estrutura de um padrão. Cada padrão deve conter as seguintes informações:

- O título do padrão;
- Descrição do problema;
- O contexto em que o problema existe; e
- A solução comprovada.

A seguir vamos detalhar as **informações necessárias de um Padrão**:

- **Título**

A primeira coisa que você vê quando olha um padrão é o título.

Um bom título para um padrão lhe fornece uma noção do *que* o padrão faz e *como* ele faz isso, e pode até envolver o vocabulário arquitetônico.

12

- **Declaração do problema**

Cada padrão deve conter uma declaração clara do problema que você está resolvendo. O problema deve ser específico para o contexto, e deverá ser sucinto.

Um bom padrão resolve **pequenos problemas** em vez de tentar resolver grandes problemas. Estes padrões menores podem ser combinados para resolver os grandes problemas. Você provavelmente não terá todos os ingredientes necessários para resolver o grande problema.

A declaração do problema deve ser:

- relevante para o contexto,
- específica e
- fácil de entender.

Ao rever padrões, você deve revisar as declarações de problemas para ver se elas se encaixam ao problema que você está tentando resolver. Declarações de problemas genéricas como "Faça a coisa certa" não são muito úteis. Se você se deparar com padrões que têm declarações muito genéricas, eles provavelmente não serão suficientes para ajudá-lo a resolver o seu problema. Lembre-se a declaração do problema explica o que é o problema e que precisa ser resolvido.

13

- **Contexto**

O contexto é uma das seções mais importantes em um padrão, porque problemas de *design* não existem isoladamente; eles existem em algum contexto.

Padrões não existem isoladamente, eles são construídos sobre o ambiente do problema e uns sobre os outros. Às vezes não vai fazer sentido aplicar a solução de um padrão até que outro padrão tenha sido aplicado. O contexto de um padrão descreve qualquer pré-condição existente para a sua utilização, bem como quaisquer outros requisitos necessários.

O contexto inclui os seguintes padrões:

- Ambiente;
- Pré-condições;
- Premissas;
- Restrições.

Ambiente

O contexto explica o ambiente no qual o problema foi encontrado. Se o contexto não descrever seu ambiente, você pode não ser capaz de usar o padrão, embora o problema resolvido com o padrão seja o mesmo que o seu.

O contexto é muito importante para a definição de onde existe um problema. Para exemplificar a importância do contexto, imagine a seguinte solução: Garantir a consistência dos dados ao atualizar um banco de dados. Agora imagine a solução se o contexto for de uma única CPU. Mas e se a solução tiver que atender a um sistema distribuído de baixo acoplamento? A solução será bem diferente uma da outra.

Pré-condições

As pré-condições são muito importantes para a definição de um padrão, pois elas podem limitar a aplicabilidade do padrão. Aqui estão alguns exemplos de condições prévias:

- Ambiente distribuído;

- Ambiente heterogêneo;
- Linguagem de programação (ex: Java, C ++, C #, Haskell, Lisp, Ruby,...); e
- Restrições de memória.

Premissas

Às vezes o mesmo problema tem soluções diferentes no mesmo sistema. Os desenvolvedores podem ter acesso a ferramentas internas de um sistema, enquanto os testadores ou engenheiros de suporte não têm esse mesmo acesso. Em outras palavras, os contextos de ferramentas disponíveis são diferentes. Hipóteses a respeito do meio ambiente e do público-alvo devem aparecer no contexto do padrão.

Restrições

As restrições sobre o sistema que você não pode mudar ou controle também aparecem no contexto do padrão. Um problema pode existir apenas em uma linguagem de programação particular.

14

- **Solução**

A seção “Solução” de um padrão explica a forma de resolver o problema que existe no contexto mencionado.

A seção começa apresentando as medidas que se deve tomar para resolver o problema. Em seguida, são fornecidas as informações necessárias para que você possa construir algo.

A solução explica como resolver o problema no contexto indicado. A declaração da solução deve ser **específica** em como ela resolve o problema. Ela não deve ser genérica, oferecendo orientações vagas como "Faça a coisa certa".

15

- **Outras Seções**

Os padrões também podem conter outras seções. Diferentes autores de padrão descobriram que outras seções funcionam melhor para seus estilos de escrita. Eles também descobriram que os padrões que

têm como alvo diferentes públicos podem exigir alguma variação no tipo de informação que eles fornecem.

Aqui estão alguns exemplos de "outras" seções usadas em padrões:

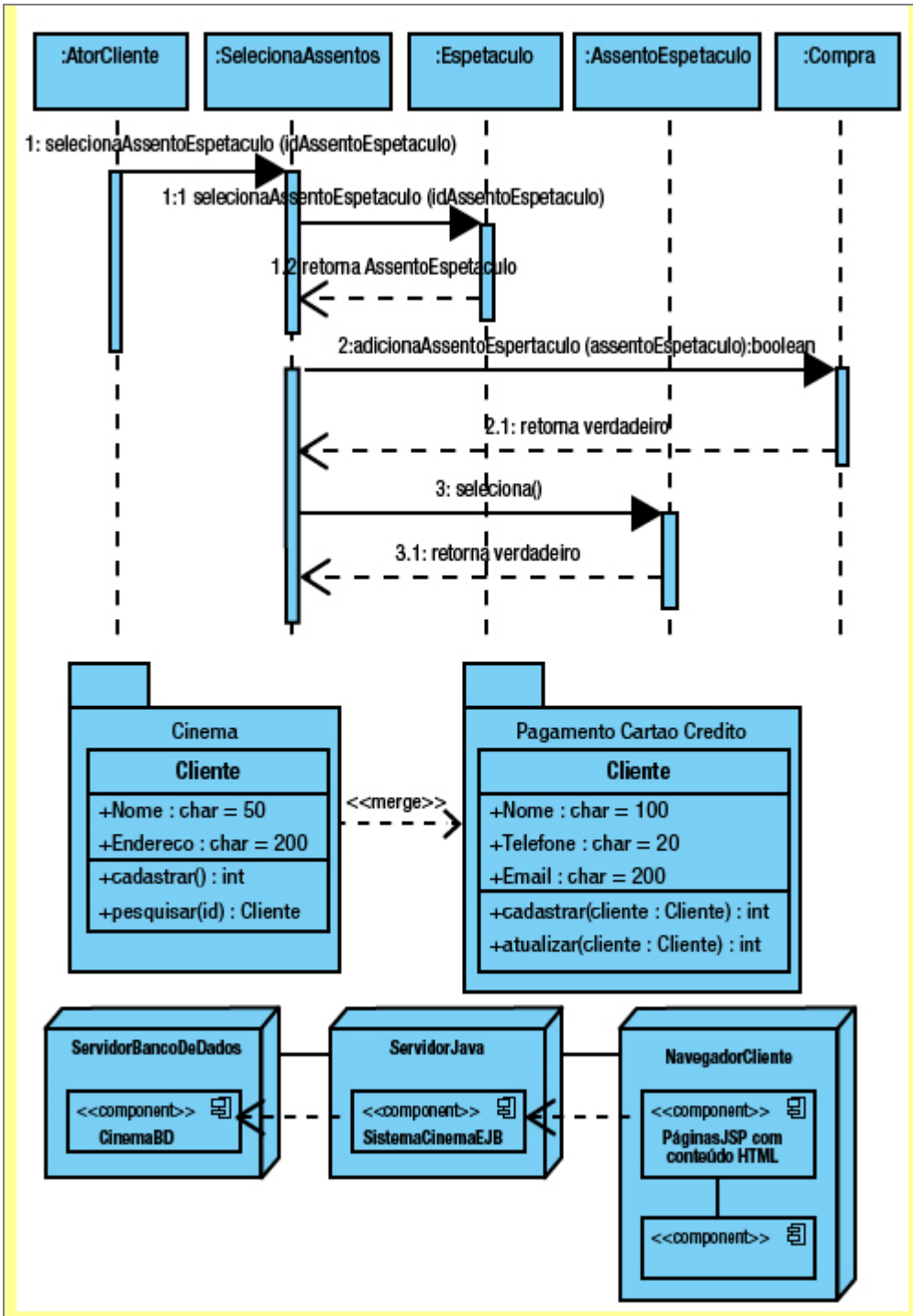
- Consequências
- Esboço
- Contexto do Resultando
- Justificativa
- Implementação
- Código de exemplo
- Utilizações conhecidas

Consequências

As consequências da aplicação da solução para o problema podem ser explicitadas na seção solução, ou elas podem aparecer em uma seção separada. Cada solução tem algumas consequências. A boa consequência é que o problema já está resolvido. A solução também pode fornecer alguns outros benefícios para o sistema, por exemplo, tornando-o mais fácil para expandir no futuro, ou talvez mais fácil de manter. Mas soluções podem introduzir exigências também.

Esboço

Muitos padrões contêm um esboço ou dois. É muito útil se o padrão contém um esboço. Ele pode ser da solução, ou ele pode ser um esboço do problema. O esboço pode ser diagramas UML ou diagramas de blocos simples da solução. A figura abaixo mostra alguns desenhos típicos.



Exemplos de esboço de arquitetura orientada a objeto

A ideia de um esboço é para você pensar visualmente sobre como a solução pode ser estruturada. O esboço também fornece outra visão da solução, o que ajuda a torná-lo mais claro.

Contexto do Resultando

Assim como o problema existia dentro de um contexto, a solução cria um novo contexto, o contexto resultante. O contexto resultante também descreve o que o sistema parece após o padrão ter sido aplicado e que o problema foi resolvido. Esta seção está intimamente relacionada com a seção consequências e pode ser apresentada em uma única seção.

Justificativa

Alguns padrões incluem uma seção lógica que ajuda a explicar em linguagem simples por que a solução do padrão é a melhor solução para o problema.

Implementação

Muitos padrões incluem uma seção de implementação, o que lhe dá instruções sobre como implementar o padrão, às vezes com um passo a passo que você pode seguir.

Código de exemplo

Muitos padrões incluem um código de exemplo. Esta seção pode mostrar código relacionado com qualquer parte do padrão.

Utilizações conhecidas:

Utilizações conhecidas (pelo menos três) também podem estar presentes no padrão. Utilizações conhecidas ajudam a ver que o padrão realmente tem sido usado em uma situação como a sua.

16

RESUMO

Neste módulo vimos um conceito intimamente ligado ao do *framework*, o de ***design patterns***. Diferente dos *framework*s, os *design patterns* possibilitam o reuso de uma microarquitetura mais abstrata sem a necessidade de uma implementação.

Um padrão é um elemento de *design* considerado como uma solução para um projeto de *software* ou para um problema de codificação. Estes elementos são:

- Modelos reutilizáveis;
- Soluções comprovadas;
- Ferramentas educacionais;
- Guias do sistema;
- Vocabulário arquitetônico; e
- Base de conhecimento.

Vimos também que os padrões não podem ser confundidos com *framework*s, algoritmos, patentes, que não são restritos à Orientação a objeto e que não são solucionadores de problemas universais.

Por fim, vimos que os *Patterns* seguem um padrão de definição e que o Título, a descrição do problema, o contexto em que o problema se apresenta e a solução comprovada são os requisitos mínimos de informações para um padrão.

Algumas seções adicionais podem ser utilizadas para atender as necessidades do criador do padrão ou do público-alvo a que se destina este padrão. São elas: consequências, esboço, contexto do resultado, justificativa, implementação, código de exemplo e utilizações conhecidas.