

Arquitetura de software

Kleinner Silva Farias de Oliveira



Escola Politécnica

COLEÇÃO
FAD
EDITORA UNISINOS

ARQUITETURA DE SOFTWARE

KLEINNER SILVA FARIAS DE OLIVEIRA

Editora Unisinos, 2016

SUMÁRIO

Apresentação

Capítulo 1

Introdução

Capítulo 2

Camadas e modelo MVC

Capítulo 3

Cliente-servidor e *pipes and filters*

Capítulo 4

Componentes e linguagem de descrição arquitetural

Capítulo 5

Aspectos e web semântica

Capítulo 6

Web services e SOA

Referências

O autor

Informações técnicas

APRESENTAÇÃO

Caro leitor, este livro foi elaborado com o objetivo de servir como um guia para aspectos introdutórios da área de Arquitetura de Software. Desta forma, o mesmo deve ser visto como um ponto de partida para o estudo e entendimento de vários conceitos que estão presentes em vários tópicos relacionados à arquitetura de software. Exemplos destes tópicos seriam princípios e padrões de projetos, arquitetura em camada, arquitetura usando o modelo MVC (*Model-View-Controller*), arquitetura cliente-servidor, arquitetura usando *pipes and filters*, arquitetura baseada em componentes, linguagem de descrição arquitetural, arquitetura baseada em aspectos, web semântica, *web services* e arquitetura orientada a serviços.

Perante a grande diversidade dos tipos de arquiteturas de software disponíveis nos dias atuais, é fundamental um bom entendimento dos conceitos centrais que norteiam tais arquiteturas, visando potencializar boas escolhas e a elaboração de projetos de sistemas de software cada vez mais flexíveis, reutilizáveis, performáticos, modularizados, estáveis, com baixo grau de acoplamento e alto grau de reuso e coesão.

Além da diversidade dos tipos ou padrões arquiteturais disponíveis, percebem-se também desafios diante da necessidade de definir uma arquitetura para atender a um conjunto de requisitos, especificar esta arquitetura e, ao mesmo tempo, permitir que esta especificação possa ser compreendida por várias pessoas, as quais estão diretamente (ou indiretamente) utilizando os artefatos gerados para representar a arquitetura de um sistema.

Atualmente se observa também que o perfil técnico dos usuários que lidam com tais artefatos da arquitetura tem se

diversificado. Exemplos destes usuários: clientes, analistas de negócio, analistas de sistemas, desenvolvedores de software, testadores, engenheiros de software, gerentes, arquitetos de software, entre outros.

Diante do exposto, o livro busca apresentar aspectos teóricos e tecnológicos associados ao contexto de projeto de arquitetura de software. Além disso, apresentam-se em linhas gerais várias abordagens para a concepção, modelagem e implementação de arquiteturas de sistemas de software, visando potencializar o reuso dos seus componentes, evitar a sua degradação, bem como aumentar a sua estabilidade.

CAPÍTULO 1

Introdução

Este capítulo possui como objetivo apresentar uma visão geral da área de arquitetura de software. Para isso, aspectos gerais sobre projeto arquitetural serão discutidos, conceitos serão definidos e contextualizados. Serão apresentados conceitos gerais de princípios e padrões de projetos de software, destacando os seus benefícios.

Sistemas organizacionais estão cada vez mais complexos. Para viabilizar tais sistemas, eles são decompostos em subsistemas, os quais formam módulos (ou componentes arquiteturais) que cooperam para implementar os requisitos do sistema. De acordo com Sommerville (2007), projetar uma arquitetura consiste no processo de identificar esses subsistemas, estabelecer relações entre eles, bem como definir um arcabouço que estabeleça o controle e a comunicação entre tais subsistemas. De acordo com Bass (2003), a arquitetura de software pode ser definida como sendo a estrutura de um programa que compreende os seus componentes, as propriedades visíveis externamente e os relacionamentos entre tais componentes. A definição da arquitetura é vista como uma das etapas mais importantes, entre as etapas de engenharia de requisitos e do desenvolvimento propriamente dito. Pois é nesta etapa que serão determinadas e tomadas as principais decisões, principalmente aquelas relacionadas com o aspecto estrutural do sistema, bem como o comportamental.

Na sua essência, projetar uma arquitetura de software pode ser visto como um processo criativo, em que se busca estabelecer uma estrutura de sistema (seus componentes) que atenda aos requisitos do sistema, sejam eles funcionais ou não funcionais. A Figura 1 mostra uma ilustração que representa a arquitetura como sendo um artefato de transição entre os requisitos do sistema e a implementação. Além disso, a imagem apresenta a arquitetura como

uma solução de alto nível, entre o domínio do problema e o domínio da solução, bem como mostra onde o arquiteto de software deve atuar.

Por ser um processo criativo, a forma de executar as atividades do projeto, por parte dos arquitetos, difere sensivelmente de um sistema para outro, principalmente quando os requisitos são diferentes, os membros da equipe de desenvolvimento são diferentes, bem como a origem e o nível de experiência do arquiteto. Sommerville (2007) defende que é muito mais interessante entender o processo de projeto de arquitetura através de uma perspectiva de tomada de decisão, ao contrário de uma perspectiva de atividade. De fato, ao longo do processo de definição da arquitetura, os arquitetos do sistema devem tomar várias decisões importantes, as quais afetarão de forma significativa, não só o sistema como um todo, mas também o processo de desenvolvimento.

Com esse entendimento, Sommerville (2007) recomenda algumas questões importantes que arquitetos precisam responder para projetar uma arquitetura: Como o sistema será distribuído nos servidores? Quais estilos ou padrões arquiteturais são adequados para os requisitos definidos? Há alguma arquitetura genérica que possa servir como base para o projeto arquitetural? Como a arquitetura será especificada? Como o projeto da arquitetura será avaliado? Como será a decomposição do sistema em módulos?

Apesar de cada sistema a ser desenvolvido ser único, é comum que sistemas de um mesmo domínio de aplicação tenham projetos arquiteturais semelhantes, visto que essa semelhança possa refletir a similaridade entre os domínios. Dessa forma, ao responder às questões anteriores, é possível definir arquiteturas genéricas que possam atender grande quantidade de aplicações que possuem um domínio de aplicação similar. Por exemplo, linhas de produtos de aplicações Web podem ser criadas baseadas em um núcleo comum, bem como em algumas variações que estão diretamente relacionadas com particularidades do domínio e requisitos específicos.

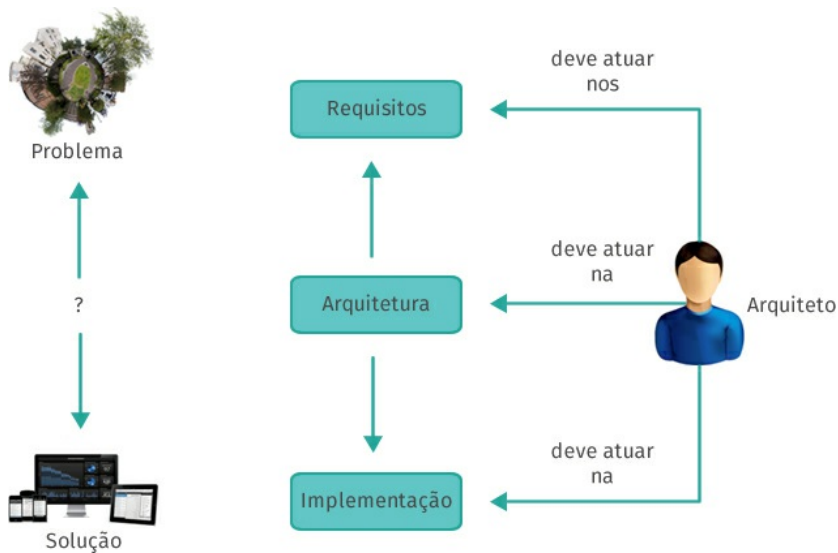


Figura 1 – Contextualização de arquitetura de software.
Fonte: elaborada pelo autor.

Uma vez que as questões sobre como projetar uma arquitetura tenham sido respondidas, bem como arquiteturas provenientes de domínios semelhantes tenham sido identificadas, um próximo passo, que merece atenção especial, seria documentar a arquitetura. Sommerville (2007) e Bass (2003) destacam a importância e as vantagens de se projetar e documentar a arquitetura criada. A primeira delas se refere aos ganhos relacionados com a comunicação com os usuários do sistema. Dado que a arquitetura representa uma visão abstrata do sistema, então ela pode ser utilizada para fomentar discussões entre os diferentes usuários do sistema, promover um melhor entendimento do sistema, bem como embasar as decisões arquiteturais. A segunda seria relacionada com a análise de sistema. Ao tornar a arquitetura do sistema explícita e representada de forma clara no estágio inicial do processo de desenvolvimento, isso traz boas contribuições para a análise do projeto. De fato, decisões arquiteturais podem causar efeitos severos, principalmente no que se refere à capacidade da arquitetura em atender aos requisitos solicitados pelos usuários. Isso se torna ainda mais evidente ao levar em consideração requisitos mais críticos

como, por exemplo, performance, escalabilidade, confiabilidade, reusabilidade, usabilidade, entre outros.

Uma terceira vantagem que merece destaque seria o reuso em larga escala. A especificação da arquitetura de um sistema descreve de forma compacta e administrável a maneira como o sistema foi organizado e como os componentes operam entre si. Ao especificar a arquitetura, é possível também verificar a similaridade entre diferentes arquiteturas de outros sistemas, ajudando a verificar se decisões tomadas em uma arquitetura podem ser também válidas para outras arquiteturas. Nota-se que arquiteturas tendem a ser similares, quando as mesmas passam a atender requisitos similares. Sendo assim, tendo a especificação de requisitos, bem como a modelagem da arquitetura, é possível promover reuso em largar escala.

Ao reconhecer a importância e as vantagens de uma arquitetura, entender os fatores que podem influenciá-la é algo também primordial. De acordo com Bass (2003), um projeto arquitetural é influenciado pelos usuários do sistema, os quais são responsáveis diretos pela definição dos requisitos do sistema. Além disso, fatores técnicos e organizacionais também podem afetar o projeto arquitetural, incluindo aspectos de orçamento, *time-to-market*, cultura da empresa, compatibilidade com sistemas legados, manutenção de procedimentos antigos, alinhamento de processos, funcionalidades cada vez mais complexas, constante evolução dos requisitos, entre outros. Outro fator muito importante é o *background* do arquiteto, visto que toda solução apontada pelo arquiteto é baseada no seu *background* acadêmico e/ou prático. Logo, se o arquiteto tem boas experiências, então elas poderão ajudar na tomada de decisões durante a elaboração da arquitetura.

Associadas a este último fator, encontram-se algumas competências que um bom arquiteto deve ter. Exemplos dessas competências: (1) habilidades interpessoais, incluindo a capacidade de negociar com os usuários, promover trabalho colaborativo e prezar pela camaradagem, e não pela competitividade entre os pares; (2) habilidades técnicas, incluindo a fluência com tecnologias atuais, entendimento dos princípios de projetos de software, bom prospectador de requisitos não especificados, saber prever cenários de evolução, prezar pelos atributos de qualidade; (3) habilidades de comunicação, tais como saber comunicar de forma clara as decisões

arquiteturais, saber convencer pessoas e entender diferentes pontos de vista.

Além disso, outro importante questionamento que frequentemente precisa ser realizado é sobre o que a arquitetura pode influenciar em um ambiente organizacional. A arquitetura pode influenciar (BASS, 2003) (1) a estrutura organizacional, ao afetar a estruturação da organização para fazer frente à implementação da arquitetura, na formação dos times de desenvolvimento, bem como na contratação de pessoas para trabalhar na implementação de módulos específicos da arquitetura; (2) a formação dos times de desenvolvimento, ao considerar a complexidade dos módulos arquiteturais e a formação da equipe, podendo exigir ou não a contratação de pessoas com novas competências; (3) os requisitos dos clientes, os clientes podem decidir adicionar requisitos ao ver a arquitetura, bem como remover requisitos também; (4) a experiência dos arquitetos, arquitetos passam a adquirir novas experiências; (5) os times de desenvolvimento, adaptados de acordo com as exigências para implementar os módulos arquiteturais; (6) orçamento do projeto, as características da arquitetura podem causar ajustes no orçamento.

1.1 Princípios de projeto de software

Esta seção tem como objetivo apresentar alguns sintomas de problemas na arquitetura, discutir possíveis causas para tais problemas, e apresentar um conjunto de princípios de projetos de software, os quais podem ser utilizados para mitigar os problemas arquiteturais.

Começando pelos sintomas de problemas na arquitetura, de acordo com Martin (2002), quatro sintomas são comuns: *rigidez*, tendência do software se tornar difícil de ser alterado mesmo para mudanças simples; *fragilidade*, tendência do software quebrar em muitos lugares, sempre que é alterado; *imobilidade*, inabilidade de reusar código de outros projetos, ou reusar partes do código do próprio projeto; e *viscosidade*, devido à necessidade de mudança, desenvolvedores costumam encontrar mais de uma maneira de fazer a mudança. Algumas das formas preservam a arquitetura, outros não.

Exemplo de consequência diante da presença de tais sintomas seria o aumento do acoplamento de um módulo do sistema que foi projetado para ser reusado. Esse cenário acontece, por exemplo,

quando um *framework* de um sistema passa a ter alto grau de dependência em relação às aplicações instanciadas a partir dele. Como consequência direta, tem-se o baixo grau de reuso do *framework*, menor produtividade da equipe, e maior custo de desenvolvimento. Além disso, é possível também destacar que as mudanças na arquitetura, por menores que sejam, passam a ter consequências imprevisíveis. Logo, não é possível mensurar o impacto das mudanças. Uma das consequências mais críticas é o desalinhamento entre a arquitetura projetada e a arquitetura implementada. Esse desalinhamento torna a manutenção propensa a erros e exige alto esforço para refatorar o código (isto é, melhorar aspectos estruturais do código, preservando o seu comportamento).

Uma das causas para o surgimento de problemas relacionados à degradação da arquitetura seria a implementação de forma inadequada de mudanças não antecipadas. De fato, a especificação de requisitos é um dos artefatos de software mais voláteis atualmente, e tipicamente a arquitetura proposta inicialmente não suporta as modificações solicitadas. Isso também é causado pela falta de familiaridade, por parte de desenvolvedores, com a cultura da empresa, o estilo arquitetural utilizado, ou os *frameworks* e linguagens utilizadas. Outro motivo seria a falta de uma gerência eficiente das dependências entre os módulos da arquitetura. Devido à dificuldade de visualizar e entender o impacto das mudanças, arquiteto e desenvolvedores passam a ter dificuldades para controlar as alterações arquiteturais, bem como mensurar seus efeitos e propagações. Para mitigar essa problemática, recomenda-se o uso de princípios e padrões de projetos de software, pois eles podem tornar a arquitetura flexível, o que permitirá uma acomodação menos onerosa das modificações na arquitetura. Exemplos de princípios de projetos de software (MARTIN, 2002):

- Princípio aberto-fechado (do inglês, *Open Closed Principle*). Um módulo deve estar aberto para extensão, porém fechado para modificação. Isto é, os módulos (ou componentes arquiteturais) devem ser projetados de tal forma que as suas modificações sejam dominadas por adição de novos conteúdos, e nunca por alteração.
- Princípio de responsabilidade única (do inglês, *Single Responsibility Principle*). Não deve existir mais que uma razão para um módulo ser alterado. Inicialmente proposto como

Coesão por DeMarco (1979) e Jones (1988), esse princípio leva ao entendimento de que cada responsabilidade pode ser um eixo de mudanças para um módulo ou componente arquitetural. Por exemplo, se uma classe assume mais de uma responsabilidade, então existe mais de uma razão para modificá-la. Além disso, é importante destacar que, se uma classe tem mais de uma responsabilidade, então as responsabilidades se tornam acopladas.

- Princípio de inversão de dependência (do inglês, *Dependency Inversion Principle*). Módulos de mais alto nível não devem depender de módulos de mais baixo nível. Ambos devem depender de abstrações. Em outras palavras, abstrações não devem depender de detalhes. Esse princípio é essencial para projetar, por exemplo, *frameworks*, onde as classes dos *frameworks* (consideradas abstratas) não dependem das classes que instanciam o *framework*. O conceito de abstração utilizado aqui se refere à baixa probabilidade de mudança. Os módulos de mais alto nível seriam aqueles que implementam regras de negócio, suas características identificam a aplicação, enquanto módulos de mais baixo nível são os responsáveis por detalhes de implementação.
- Princípio de substituição de Liskov (do inglês, *Liskov Substitution Principle*). Esse princípio define que uma subclasse deve ser capaz de substituir a sua superclasse.
- Princípio de dependências estáveis (do inglês, *Stable Dependencies Principle*). As dependências entre pacotes em um projeto devem ser na direção da estabilidade dos pacotes. Sendo assim, cada pacote deve depender de pacotes que sejam mais estáveis que ele.
- Princípio de abstrações estáveis (do inglês, *Stable Abstractions Principle*). Os pacotes que são estáveis devem ser abstratos. Por outro lado, os pacotes que são instáveis devem ser concretos. É importante destacar que a abstração de um pacote deve ser na proporção do seu grau de estabilidade. Quanto mais estável, mais abstrato.
- Princípio do reuso comum (do inglês, *Common Reuse Principle*). Classes que são alteradas juntas, devem permanecer

juntas. Em outras palavras, sempre que modificações em uma classe provocarem também modificações em outra classe, elas devem permanecer juntas em um pacote.

- Princípio do fechamento comum (do inglês, *Common Closure Principle*). Classes que não são reusadas juntas não devem permanecer juntas. Ou seja, ao reusar um pacote (ou componente arquitetural) com um conjunto de classes, se algumas classes não são utilizadas, tais classes não devem permanecer no pacote.
- Princípio de dependências acíclicas (do inglês, *Acyclic Dependencies Principle*). Dependências entre pacotes não devem formar ciclos.

De acordo com Martin (2002), o cálculo da instabilidade de pacotes pode ser feito da seguinte forma: $Ce/(Ca+Ce)$. Ca é o acoplamento aferente, o número de classes fora desse pacote que depende de classes dentro do pacote. Ce é o acoplamento eferente, o número de classes dentro do pacote que depende de classes fora do pacote. Esse cálculo de estabilidade pode assumir um valor de 0 a 1, onde 0 indica um pacote com o menor índice de instabilidade, e 1 indica o maior índice de instabilidade.

1.2 Padrões de projeto de software

O conceito de padrões de projeto é baseado nos trabalhos de Christopher Alexander (da área de Construção Civil), o qual defende que “cada padrão descreve um problema que ocorre repetidas vezes no ambiente, e então descreve o núcleo da solução para aquele problema, de tal maneira que se possa usar essa solução milhares de vezes, sem nunca fazê-la da mesma forma duas vezes”. De acordo com Gamma (1994), padrões de projetos podem ser definidos como soluções gerais e reutilizáveis para um conjunto de problemas recorrentes, dentro de um determinado contexto no projeto de software. Mais especificamente, os padrões de projeto podem também ser vistos como descrições de objetos que se comunicam e classes que são customizadas, para resolver um problema genérico de projeto em um contexto específico. Essas definições deixam claro, portanto, que padrões de projeto não devem ser vistos como uma solução final para um determinado problema, mas sim como um arcabouço para chegar à solução desejada.

O uso de padrões de projetos de software permite o desenvolvimento de arquiteturas flexíveis, modularizadas e com alto grau de reuso de seus componentes. Por isso, têm sido amplamente utilizados em projetos de arquitetura de software. Durante a definição dos padrões de projetos, alguns elementos essenciais para a descrição dos padrões foram definidos para potencializar o bom entendimento dos mesmos, uma lista completa desses elementos pode ser encontrada em Gamma (1994):

- Nome do padrão. Ao dar nomes significativos aos padrões de projetos, alguns benefícios são percebidos: é um identificador para descrever um problema de projeto, suas soluções e consequências em uma ou duas palavras; aumenta imediatamente o vocabulário de projeto; permite projetar em um nível mais alto de abstração; favorece a comunicação em equipe e melhora a documentação de projetos; e, por fim, cria um vocabulário que facilita pensar e entender decisões de projeto.
- Problema. Esse elemento descreve o problema a que o padrão se destina a resolver. Desse modo, a criação do problema de cada padrão busca descrever em qual contexto o padrão deve ser aplicado, apresentar uma lista de condições que devem ser satisfeitas para que o padrão possa ser utilizado, descrever estruturas de classes que são sintomas de um projeto inflexível e, por fim, apresentar problemas específicos em um projeto de software.
- Solução. Descreve os elementos que compõem o padrão de projeto, incluindo as classes, seus relacionamentos, responsabilidades e colaborações. Não fornece uma solução concreta ou final, dado que um padrão é um *template* que pode ser utilizado para diferentes situações. Além disso, mostra como organizar classes e objetos para solucionar um problema de projeto.
- Consequências. Esse elemento foca em mostrar quais são os resultados e os *trade-offs* obtidos com o uso do padrão, os quais são fundamentais para avaliar alternativas de projeto e para a compreensão dos custos e benefícios da aplicação do padrão. Os resultados e os *trade-offs* ajudam a ponderar a aplicabilidade dos padrões e apresentam o impacto do uso do

padrão em atributos de qualidade, incluindo reusabilidade, flexibilidade, modularidade, portabilidade, entre outros.

Entender padrões de projetos é importante para reusar soluções já utilizadas e testadas, fazer uso da experiência dos outros, reconhecer problemas recorrentes de projeto e suas respectivas soluções, usar de forma sistemática os recursos de orientação a objetos, projetar e desenvolver software com uma melhor qualidade, melhorar a comunicação em equipes e a documentação de decisões de projeto, melhorar a compreensão de sistemas existentes, ajudar um novato a agir como um especialista e aumentar o nível de abstração da solução proposta para melhorar a comunicação.

Os padrões de projetos definidos por Gamma (1994) são os mais amplamente utilizados e difundidos, os quais são classificados em padrões de criação, associados à criação de objetos; padrões estruturais, tratam de como estruturar classes e objetos; e padrões comportamentais, definem como as interações e separação de responsabilidades entre as classes (e objetos) devem ser feitas. Esses padrões são também conhecidos como Padrões GoF (*Gang of Four*), os quais são descritos detalhadamente em Gamma (1994). A seguir, os cinco padrões de criação GoF são definidos.

- *Abstract Factory*: fornece uma interface para a criação de objetos dependentes e relacionados, sem especificar suas classes concretas.
- *Builder*: separa a construção de um objeto complexo de sua representação, a fim de que o mesmo processo de criação possa criar diferentes representações.
- *Factory Method*: define uma interface para criar um objeto, mas deixa para a subclasse decidir qual classe instanciar. Ele permite postergar a instanciação para uma subclasse.
- *Singleton*: garante a instância única de uma classe e fornece um ponto central de acesso a esta instância.
- *Prototype*: especifica os tipos de objetos que podem ser criados, e cria tais objetos usando um protótipo.

A seguir, os sete padrões estruturais GoF são também definidos.

- *Facade*: fornece uma interface única para um conjunto de interfaces e encapsula um subsistema. Este padrão define uma interface de mais alto nível que torna o subsistema mais fácil de ser utilizado e mantido.
- *Adapter*: converte a interface de uma classe em outra interface esperada por outra classe, permitindo que a incompatibilidade entre elas seja contornada.
- *Bridge*: desacopla uma abstração de sua implementação, visando que as duas possam variar independentemente.
- *Composite*: forma os objetos através de uma estrutura de árvore para representar um relacionamento parte-todo.
- *Decorator*: adiciona responsabilidades a um objeto dinamicamente. Ele fornece uma alternativa flexível para estender funcionalidades.
- *Flyweight*: descreve como compartilhar objetos para permitir seu uso com granularidade fina sem custos elevados.
- *Proxy*: aplicável sempre que haja a necessidade de uma referência mais versátil ou sofisticada para um objeto que um ponteiro simples.

Os onze padrões comportamentais GoF são definidos.

- *Strategy*: define uma família de algoritmos, encapsula cada algoritmo em uma classe. Ele permite que um algoritmo possa ser alterado independente do cliente que o utiliza.
- *Chain of Responsibility*: evita acoplamento entre quem envia uma requisição e quem recebe e trata a requisição. Ele permite que mais de um objeto tenha a oportunidade de tratar a requisição feita.
- *Observer*: define uma dependência 1:N entre objetos para permitir que, caso um objeto mude seu estado, todos os objetos dependentes sejam notificados e atualizados automaticamente.
- *Interpreter*: dada uma linguagem, define uma representação

para a sua gramática juntamente com um interpretador que, baseado na representação da gramática, interpreta sentenças da linguagem.

- *Command*: encapsula uma requisição como um objeto. Desacopla quem solicita uma requisição de quem implementa a requisição. Usado para encapsular comandos de interface.
- *Iterator*: fornece uma forma de acessar os elementos de um objeto composto de forma sequencial sem expor sua representação. Ele faz interações sobre um objeto composto para acessar as suas partes.
- *Mediator*: define um objeto que encapsula como objetos interagem. Ele permite variar suas interações independentemente.
- *Memento*: sem violar o encapsulamento, captura e disponibiliza o estado interno de um objeto, a fim de que o objeto possa retornar para estados anteriores.
- *State*: permite um objeto alterar seu comportamento quando seu estado interno for alterado.
- *Template Method*: define o esqueleto de um algoritmo, deixando parte da implementação para a subclasse. Ele permite que subclasses redefinam alguns passos de um algoritmo, sem alterar a estrutura do algoritmo.
- *Visitor*: representa uma operação a ser realizada sobre os elementos de uma estrutura de objeto. Ele permite a definição de uma nova operação, sem alterar as classes dos elementos sobre os quais opera.

CAPÍTULO 2

Camadas e modelo MVC

Neste capítulo serão discutidos conceitos teóricos fundamentais associados à elaboração de arquiteturas em camadas e usando o modelo MVC, bem como aspectos práticos necessários para o projeto de arquiteturas organizadas em camadas de abstração, tais como evolução das camadas, arquitetura em duas e três camadas, evolução da arquitetura de duas camadas para três camadas, elementos que formam o modelo MVC, bem como a dinâmica de execução do modelo MVC.

Como ponto inicial sobre a discussão sobre arquiteturas em camadas e o modelo MVC, é fundamental elencar alguns aspectos iniciais que contextualizam e motivam o uso de tais conceitos no projeto de arquitetura de software. Dado o ambiente de negócios cada vez mais turbulento e mutável nos dias atuais, são inevitáveis as exigências cada vez mais frequentes de mudanças nos requisitos dos sistemas. Isso tem levado as especificações de requisitos de software a serem um dos artefatos mais voláteis ao longo do processo de desenvolvimento.

É nesse cenário de mudanças frequentes e de requisitos voláteis que arquitetos de software devem trabalhar, com o objetivo de projetar e prover soluções arquiteturais que sejam capazes de suportar essa volatilidade dos requisitos, assim como outras restrições, por exemplo, prazos curtos e equipe de desenvolvimento com pouca experiência. Associada diretamente a esse desafio, encontra-se a necessidade sempre presente de trabalhar com um grande volume de dados, de implementar regras de negócios complexas e mutáveis, de dar suporte a diferentes formas de visualização e interação com os dados do sistema, bem como manter a interface do cliente sempre sincronizada com o estado corrente da aplicação. Perante esses desafios, é crítico projetar arquiteturas que,

não só suportem os requisitos do sistema, mas também garantam atributos de qualidade centrais como, por exemplo, flexibilidade, reusabilidade e modularidade.

Para isso, entende-se que a decomposição da arquitetura em camadas pode representar uma solução efetiva para tais desafios, visto que ela permitirá projetar a arquitetura em blocos ou módulos em um particular nível de abstração, com responsabilidades específicas, com alta coesão e baixo acoplamento. Além disso, ao atribuir papéis específicos e estabelecer regras que normatizam como tais camadas devem se relacionar, é possível mitigar os desafios de projetar arquiteturas flexíveis para sistemas com ciclo de desenvolvimento e manutenção cada vez mais curto.

2.1 Arquiteturas em camadas

Inicialmente proposta por Dijkstra em 1968, arquitetura em camadas surgiu como uma possível solução para modularizar sistemas complexos, de tal forma que seu entendimento, implementação e manutenção fossem gerenciáveis por desenvolvedores (FOWLER, 2006).

Ao projetar uma arquitetura em camadas, algumas características podem ser percebidas: (1) através da decomposição do sistema em camadas (ou módulos), cada camada passa a ter um particular nível de abstração, bem como responsabilidades específicas; (2) a divisão e a organização do sistema passam a ser de forma hierárquica, frequentemente em camadas superpostas, onde cada camada pode ser um subsistema com um propósito bem definido; (3) cada camada é projetada para ter uma alta coesão e baixo acoplamento. Além disso, busca-se que as camadas sejam estáveis ao longo do tempo. Isto é, recebam uma baixa quantidade de mudanças, à medida que a arquitetura precisa dar suporte a novos requisitos (FARIAS, 2013). Motivada, em parte, por tais características, arquitetura em camadas passou a ser amplamente utilizada como uma abordagem para viabilizar a decomposição de sistemas complexos de software em módulos menos complexos.

Além disso, é possível também destacar que, ao projetar um sistema usando uma arquitetura em camadas, usualmente as camadas serão superpostas formando camadas sucessivas, onde uma camada superior usa vários serviços da camada inferior, porém a camada

inferior ignora a existência da camada superior. Normalmente, cada camada esconde suas camadas inferiores das camadas superiores. Um dos desafios é, por exemplo, definir quantas camadas são necessárias, e quais serão suas respectivas responsabilidades. Um exemplo clássico de arquitetura em camadas é o modelo OSI (*Open Systems Interconnection*). Essa arquitetura é ilustrada na Figura 2, a qual é formada por sete camadas organizadas de forma hierárquica.

Cada camada da arquitetura OSI foi projetada de tal forma que fosse possível compreendê-la sem a necessidade de conhecer os detalhes das outras camadas. Por exemplo, é possível compreender a viabilidade da construção de um serviço FTP sobre TCP, sem conhecer os detalhes de como funciona *Ethernet*. Baseado nessa linha de raciocínio, cada camada inferior oferece um conjunto de serviços para a camada superior. Além disso, é possível também perceber a variabilidade de serviços que pode ser ofertada em cada camada. Por exemplo, na camada de transporte é possível utilizar os protocolos TCP ou UDP. Outras observações relevantes seriam que a dependência entre as camadas é minimizada, as camadas passam a ser bons lugares para o uso de padrões, bem como, uma vez que uma camada tenha sido projetada, ela poderá ser utilizada por vários serviços de uma camada superior.

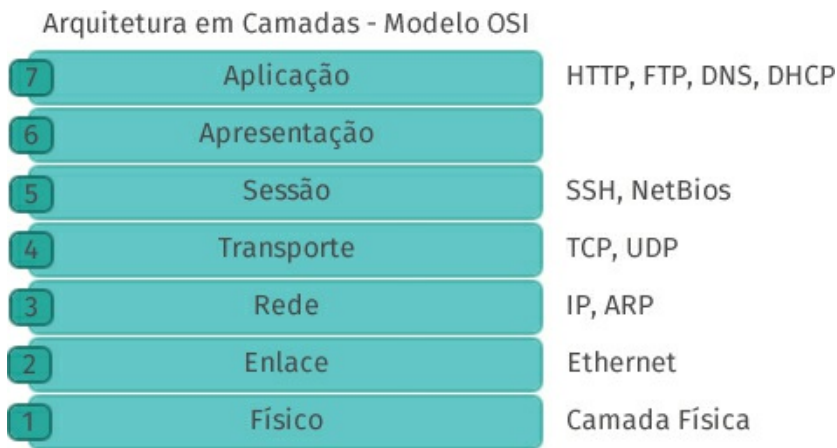


Figura 2 – Exemplo ilustrativo de uma arquitetura em camadas.
Fonte: elaborada pelo autor, com base em Fowler (2006).

Embora os benefícios de uma arquitetura em camadas fossem reconhecidos, esse tipo de arquitetura passou a ser amplamente utilizado por volta dos anos 1990, com o advento dos sistemas com arquitetura cliente-servidor (FOWLER, 2006). Seguindo uma arquitetura cliente-servidor, enquanto o cliente mantinha a interface com o usuário, o servidor era normalmente um banco de dados relacional. *Visual Basic*, *PowerBuilder* e *Delphi* são exemplos de tecnologias que rodavam nessa arquitetura. Mais especificamente, elas rodavam no lado do cliente, pois tornaram possível criar aplicações que manipulavam dados (no servidor) de forma fácil.

A arquitetura cliente-servidor funcionava muito bem enquanto o sistema em questão tivesse que exibir e fazer atualizações simples no banco de dados. A Figura 3 mostra uma arquitetura em duas camadas (cliente-servidor). O problema surgiu com a manutenção da lógica de domínio, envolvendo regras de negócio, validações, cálculos. Normalmente, esta lógica era inserida na própria interface do cliente e geralmente não era modularizada (FOWLER, 2006). Consequentemente, à medida que a lógica do domínio se tornava complexa, esse tipo de arquitetura em duas camadas (cliente-servidor) se tornava difícil de manter. Isso se justifica pela dificuldade de inserir lógica nas telas, sem ter que replicar código, o que significava que alterações simples resultavam em buscas por códigos replicados em várias telas do sistema. Uma saída foi inserir a lógica de domínio no banco de dados através de *stored procedure*, que também passaram a não suportar modificações constantes sem produzir código não modularizado.

Outra solução encontrada pela comunidade foi usar orientação a objetos, com uma arquitetura em três camadas: (1) apresentação, tratava da interface do usuário; (2) domínio, onde lógica de domínio era inserida; e (3) dados, onde as responsabilidades associadas à persistência dos dados eram tratadas. Desse modo, a lógica de domínio saiu da interface para a camada de domínio, sendo estruturada usando os recursos das linguagens orientadas a objetos.

Por fim, para finalizar essa discussão sobre a evolução das camadas, dois conceitos são apresentados: *layer* e *tier*. Às vezes, os dois são tratados como sinônimos, porém usá-los dessa forma não seria o mais adequado. Pois *tier* trata-se das camadas que representam uma separação física das partes do sistema. Por exemplo, sistemas com uma arquitetura cliente-servidor são chamados de sistemas com

duas camadas (*two-tier systems*). Enquanto a interface do usuário roda no cliente, a lógica do negócio é executada no servidor. Por outro lado, o conceito de *layer* traz uma visão de camada lógica, e não física, para os projetos de arquitetura em camadas. Por exemplo, as camadas do modelo MVC (*Model, View, Controller*), o qual será posteriormente apresentado, podem ser tratadas como camadas lógicas.

Ao projetar uma arquitetura é possível utilizar um número ilimitado de camadas. Em particular, serão introduzidas as arquiteturas projetadas em duas e três camadas. De acordo com Fowler (2006) e Sauv   (2015), a arquitetura em duas camadas surgiu com a finalidade de melhorar o aproveitamento dos computadores pessoais nas empresas, suportar sistemas com interfaces gr  ficas amig  veis, integrar a aplica  o com os dados corporativos, permitir uma maior escalabilidade dos sistemas de informa  o, entre outras raz  es. A Figura 3 apresenta um exemplo de arquitetura em duas camadas.

A primeira camada trata-se da camada cliente, a qual possui a l  gica de neg  cio e da interface do usu  rio. A segunda se refere    camada do servidor que tipicamente trata da manipula  o dos dados da aplica  o. De acordo com Sauv   (2015), algumas desvantagens podem ser consideradas ao utilizar essa arquitetura, tais como falta de escalabilidade, conex  es problem  ticas ao banco de dados, enormes problemas de manuten  o, mudan  as na l  gica de aplica  o for  am atualiza  es nas vers  es do sistema que se encontra na m  quina do usu  rio, e dificuldade de acessar fontes heterog  neas de dados.

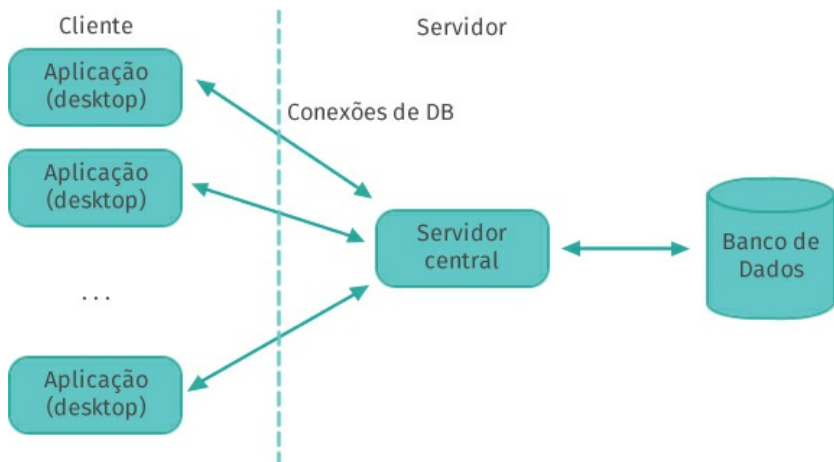


Figura 3 – Exemplo de arquitetura em duas camadas.
Fonte: elaborada pelo autor, com base em Sauv  (2015).

  medida que os problemas encontrados na arquitetura em duas camadas foram investigados, percebeu-se que, remodelando a arquitetura em duas camadas para tr s, alguns problemas cr ticos eram mitigados. Nessa nova configura  o, a arquitetura passou a ser organizada em camadas de apresenta  o, dom nio e de acesso a dados. Por exemplo, a camada de dom nio passou a esconder as responsabilidades inerentes   camada de acesso a dados da camada de apresenta  o. A Figura 4 apresenta uma ilustra  o sobre como as tr s camadas s o organizadas hierarquicamente. Por sua vez, a Figura 5 mostra a evolu  o da arquitetura de duas camadas para tr s.

Arquitetura em Três Camadas

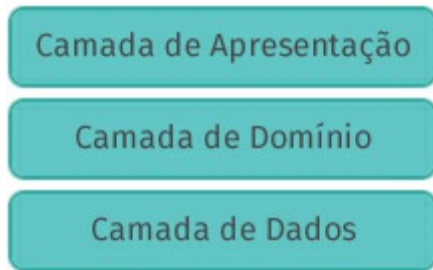


Figura 4 – Exemplo ilustrativo de arquitetura em três camadas.

Fonte: elaborada pelo autor.

Ao utilizar uma arquitetura em três camadas, recomenda-se que cada camada assuma um conjunto predefinido de responsabilidades, visando uma boa modularização do sistema. Sendo assim, as responsabilidades primárias da camada de apresentação são exibir informações para o usuário e traduzir comandos do usuário em ações sobre a camada de domínio e a camada de dados. A interação entre o usuário e essa camada pode ocorrer, por exemplo, através de linha de comando ou de interface gráfica. Por sua vez, a camada de domínio trata da lógica de domínio, com suas regras de negócio e validações, executa cálculos baseados nas entradas e em dados armazenados, valida dados provenientes da camada de apresentação, entre outras responsabilidades. Deve também ter a compreensão exata de quais rotinas na camada de dados devem ser executadas dependendo dos comandos recebidos da camada de apresentação. Por fim, a camada de dados trata da comunicação com outros sistemas que executam tarefas no interesse da aplicação. Tais sistemas podem ser, por exemplo, monitores de transações, outras aplicações, e sistemas de mensagens. Note que, para a maioria das aplicações corporativas, a camada de dados trata das responsabilidades associadas à persistência de dados.

Ao ser projetada com uma arquitetura em três camadas, uma aplicação pode ter subdivisões dentro de cada camada para atender as variabilidades dos requisitos e responsabilidades que elas devem atender. Ressalta-se também que, como uma boa prática, dependências originadas das camadas de domínio e de dados em direção à camada de apresentação não devem existir, ou mesmo ser

evitadas. Isso flexibilizará a arquitetura, ao permitir a troca da camada de apresentação, sem provocar impacto nas camadas inferiores (FOWLER, 2006).

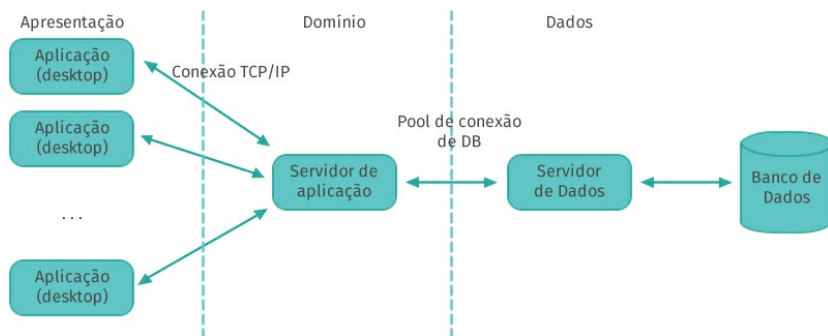


Figura 5 – Evolução da arquitetura de duas camadas para três camadas.

Fonte: elaborada pelo autor, com base em Sauvé (2015).

O próximo passo é descrever como projetar uma arquitetura em camadas. De acordo com Fowler (2006), os seguintes passos podem ser considerados para esse propósito: (1) definir critérios para abstrair as camadas; (2) identificar o número de níveis de abstração; (3) nomear as camadas e atribuir suas responsabilidades; (4) especificar os serviços de cada camada (interfaces); (5) definir os componentes que implementarão cada camada; e (6) definir uma estratégia para tratamento de erros.

Uma vez definida, as arquiteturas em camadas visam melhorar vários aspectos de qualidade, tais como reusabilidade, permite que módulos sejam reusados em outras aplicações; modularidade, permite estruturar a aplicação em módulos independentes; compreensibilidade, melhora o entendimento dos módulos; extensibilidade, permite que novas funcionalidades sejam adicionadas sem promover grande impacto na arquitetura.

2.2 Arquiteturas usando o modelo MVC

Sistemas organizacionais tipicamente possuem um grande volume de dados, os quais estão constantemente sendo criados e alterados. Nesse contexto de constante manipulação de dados, surge o desafio de projetar arquiteturas que suportem a apresentação das

informações de diferentes formas, bem como as exibam sempre atualizadas para os usuários. É possível também citar como outros desafios a necessidade de desenvolver aplicações com interfaces sensíveis às mudanças realizadas na base de dados (isto é, a interface do usuário sincronizada com a base de dados); e o suporte ao desenvolvimento de aplicações multi-plataformas, prezando pelo baixo acoplamento entre as regras de negócio e as interfaces do sistema. De fato, diferentes plataformas tendem a exigir diferentes formas de apresentação e interação com os usuários do sistema, ora o usuário pode interagir como o sistema via teclado, ora via mouse, por exemplo. É preciso garantir que o acoplamento entre interface do usuário e as regras de negócio é baixo ou inexistente, caso contrário a manutenção do sistema pode se tornar uma atividade custosa e propensa a erros.

O *Model-View-Controller* (MVC) foi introduzido por Trygve Reenskaug, um desenvolvedor *Smalltalk* na *Xerox Palo Alto Research Center* em 1979 (REENSKAUG, 1979). Em termos práticos, o MVC é um padrão arquitetural para implementar aplicações que precisam manter uma representação interna das informações, ter uma interface com o usuário e controlar a maneira como o sistema irá reagir diante das ações dos usuários. De acordo com Sommerville (2007), o MVC é uma maneira eficiente de viabilizar diferentes formas de apresentação de dados de aplicações, ao mesmo tempo que preza por atributos de qualidade, tais como reusabilidade, modularidade e estabilidade. O modelo MVC é formado por três conceitos centrais: *Model*, *View* e *Controller*. A Figura 6 apresenta uma ilustração do modelo MVC.

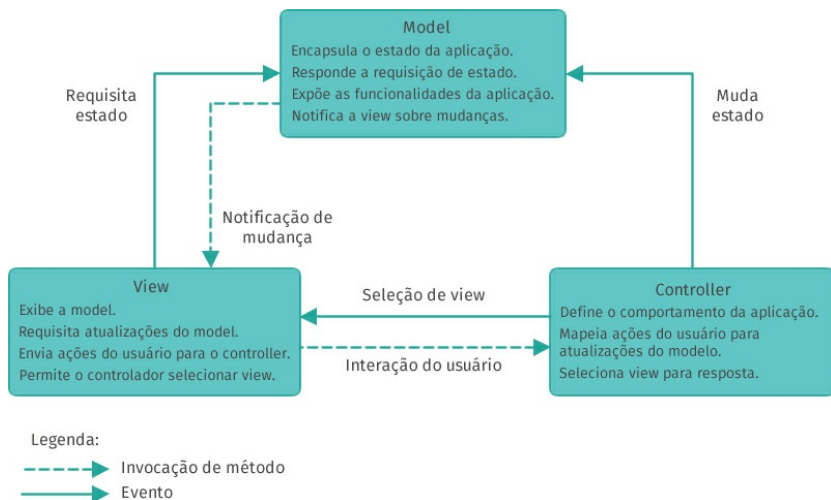


Figura 6 – Ilustração do modelo MVC.

Fonte: elaborada pelo autor, com base em Eckstein (2007).

Começando pelo *Model*, ele é responsável por encapsular o estado da aplicação e representar o estado da aplicação. Sendo assim, cada alteração do *Model*, também implica mudança do estado da aplicação; responder a requisições de solicitação de estado proveniente da *View*; expor as funcionalidades da aplicação; ao ter seu estado alterado, notificar a *View* sobre mudanças no seu estado.

As principais responsabilidades da *View* seriam exibir o estado do *Model*, o que pode significar exibir o resultado do processamento de regras de negócio; quando é notificada de mudança de estado do modelo, requisitar o estado corrente do *Model*; permitir a interação do usuário com o sistema; e gerar eventos provenientes da interação do usuário, os quais serão tratados pelo *Controller*.

Por fim, o *Controller* possui as seguintes responsabilidades: definir o comportamento da aplicação, isto é, orquestrar a aplicação; mapear ações do usuário para atualizações do modelo através da requisição de comportamentos do *Model*; ao receber eventos da *View*, pode atualizar a própria *View*, redirecionar para outra *View*, chamar outro *Controller*, ou executar algum comportamento do *Model*.

Fazendo uso do modelo MVC para projetar arquitetura, é possível perceber alguns benefícios, tais como suporte às múltiplas interfaces de usuário utilizando o mesmo *Model*; baixo acoplamento entre a implementação das regras de negócio e a interface com o usuário; flexibilidade para adicionar novas interfaces; alterações no estado do *Model* podem ser facilmente propagadas para todas as *Views*, entre outros. Por outro lado, nem sempre é facilmente compreendido por desenvolvedores inexperientes, exigindo um “maior tempo” para desenvolver uma aplicação.

CAPÍTULO 3

Cliente-servidor e *pipes and filters*

Neste capítulo serão apresentados e ilustrados conceitos teóricos e práticos sobre uma arquitetura clássica e amplamente utilizada, a arquitetura cliente-servidor. Além disso, outra arquitetura que também tem uma grande adoção em projetos arquiteturais é a arquitetura usando pipes and filters, as quais demandam uma sequência de processamentos a partir de uma entrada, visando gerar uma ou mais saídas. Para ambas as arquiteturas, serão discutidos os elementos que as formam, bem como exemplos mostrando cenários de uso.

Sistemas organizacionais cada vez mais precisam dar suporte a um conjunto de serviços, os quais serão utilizados por diversos tipos de clientes, incluindo *browser*, *smartphones*, entre outros. Esses serviços precisam ser disponibilizados em servidores, localizados dentro ou fora da organização, para que os clientes possam ter acesso. Na prática, é comum que tais serviços recebam um conjunto de dados, executem uma sequência de processamentos ou transformações desses dados, visando gerar uma saída desejada, a qual será dada como resposta aos clientes. Diante desse cenário, arquitetos precisam elaborar arquiteturas (a parte lógica) que implementem um modelo de computação distribuída, a fim de permitir que os módulos projetados da arquitetura, bem como suas cargas de processamento, possam ser executados em unidades físicas (os servidores ou clientes).

3.1 Arquitetura cliente-servidor

Esta seção apresenta aspectos gerais sobre a arquitetura cliente-servidor, a qual é amplamente utilizada, e originalmente foi proposta na Xerox PARC nos anos 1970. Em termos práticos, a arquitetura cliente-servidor trata-se, na sua essência, de uma arquitetura de

aplicação distribuída que visa alocar as tarefas e cargas de trabalho entre servidores (que fornecem recursos ou serviços), bem como definir clientes que usam os serviços e recursos disponibilizados. Em outras palavras, uma arquitetura cliente-servidor pode ser definida como um modelo em que o sistema é organizado como um conjunto de serviços, tipicamente alocados em servidores, e entidades que usam tais serviços, representando os clientes. A Figura 7 apresenta uma ilustração da arquitetura cliente-servidor.

De acordo com Sommerville (2007), os principais componentes desse modelo são os seguintes:

- Servidores que são responsáveis por oferecer um conjunto de serviços. Exemplos de servidores seriam os servidores de impressão, servidores de arquivos, servidor com *web services*, entre outros.
- Clientes que usam os serviços disponibilizados nos servidores. Tipicamente, são subsistemas que são executados independentemente. Precisam ser informados sobre os servidores disponíveis, não sabem, no entanto, da existência de outros clientes. É importante destacar que os processos dos clientes e dos servidores são separados. A Figura 8 apresenta uma ilustração dos processos de um sistema com arquitetura cliente-servidor. De acordo com Sommerville (2007), essa ilustração pode ser vista como um modelo lógico de uma arquitetura cliente-servidor distribuída.
- Uma rede que viabilize o acesso por parte dos clientes aos serviços disponibilizados nos servidores. Porém, quando o cliente e o servidor encontram-se em uma mesma máquina, esse componente não é necessário. Ressalta-se, no entanto, que sistemas que fazem uso de uma arquitetura cliente-servidor fazem uso de um sistema distribuído.

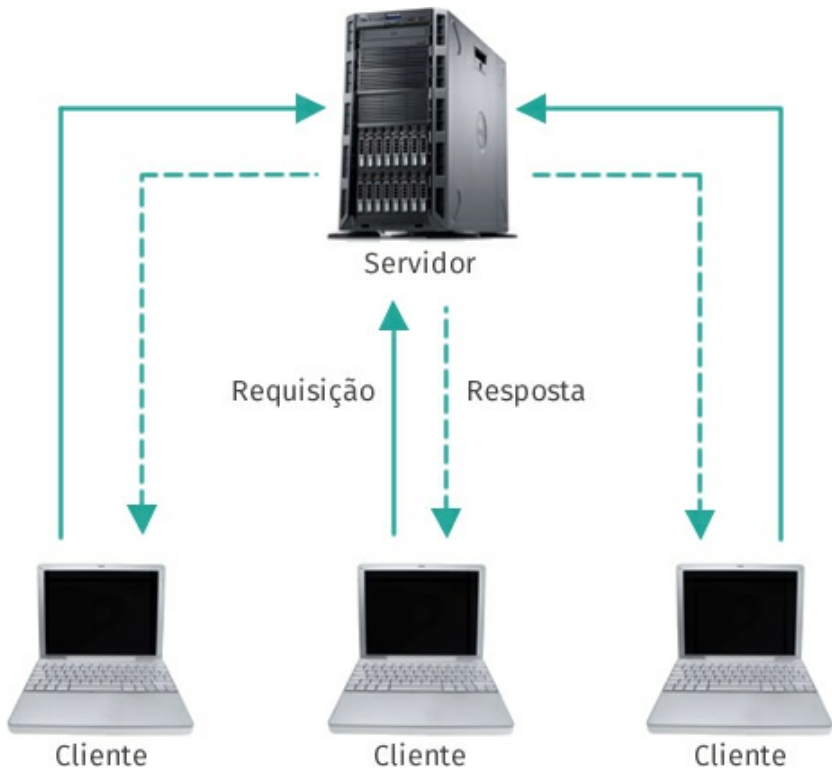
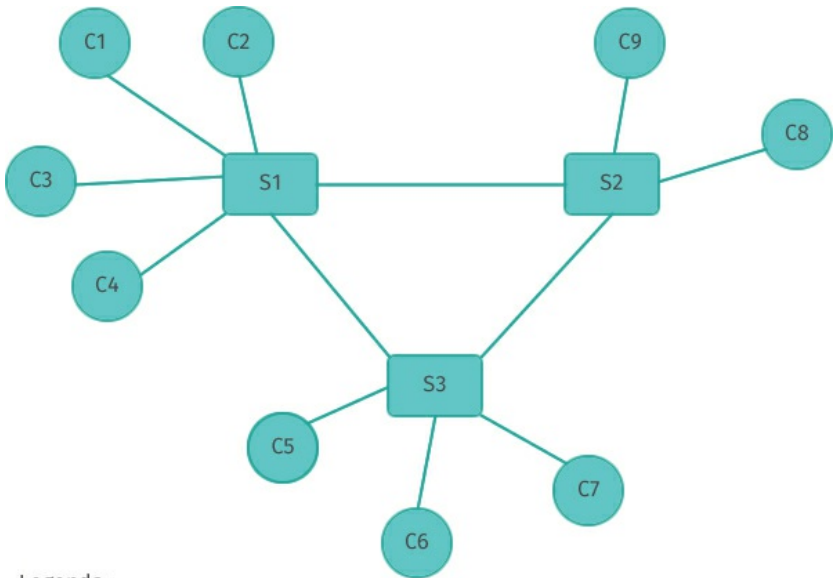


Figura 7 – Ilustração da arquitetura cliente-servidor.
Fonte: elaborada pelo autor.

A dinâmica das interações entre o cliente e o servidor pode ser representada da seguinte forma: como um primeiro passo, o cliente inicializa uma requisição ao servidor; após isso, ele espera por alguma resposta por parte do servidor; recebendo uma resposta, o ciclo de requisição-resposta é fechado; caso contrário, pode realizar uma nova requisição. O cliente pode se conectar a um (ou mais) servidor(es) para utilizar os recursos ou serviços oferecidos, como anteriormente mencionado. Por parte do servidor, ele espera por uma requisição de um cliente, ou até mesmo de outro servidor (o qual passa a desempenhar o papel de cliente), recebe as requisições feitas, processa as requisições, respondendo aos clientes

com os recursos/serviços solicitados. Além disso, o servidor é o responsável por estruturar e manter os serviços funcionando. Considerando o terceiro elemento, a rede, é através dela que toda a comunicação é realizada. Para isso, protocolos de rede são utilizados, tais como TCP (*Transmission Control Protocol*), UDP (*User Datagram Protocol*), entre outros. Revisitando a Figura 8, destaca-se que um servidor pode executar vários processos. Logo, não há necessariamente um mapeamento 1:1 entre processos, que podem representar um serviço, e quem executa esse processo, no caso o servidor.

O projeto de uma arquitetura cliente-servidor precisa refletir a estrutura lógica do sistema em desenvolvimento. Por exemplo, seguindo uma arquitetura em três camadas, como a ilustrada na Figura 4 e Figura 5, a camada de apresentação trata da apresentação das informações para o usuário, bem como de toda a interação com o mesmo. A camada de domínio trata da lógica de negócio, enquanto a camada de dados é responsável pelo gerenciamento dos dados. Essas camadas poderão ser executadas em uma única máquina, ao seguir um modelo centralizado. Porém, seguindo um sistema distribuído, é necessário determinar em qual máquina (servidor ou cliente) cada camada será executada. Usualmente, a camada de apresentação é executada no cliente, enquanto a camada de domínio e dados no servidor.



Legenda:


-  Processo cliente
-  Processo servidor

Figura 8 – Processos de um sistema cliente-servidor.

Fonte: elaborada pelo autor, com base em Sommerville (2007).

Sendo assim, tem-se uma arquitetura cliente-servidor de duas camadas físicas (cliente e servidor), tendo três camadas lógicas (apresentação, domínio e de dados). De acordo com Sommerville (2007), uma arquitetura cliente-servidor com duas camadas físicas pode ter duas configurações diferentes, ao alternar a localização das camadas lógicas. Uma primeira configuração seria ter a camada de apresentação sendo executada no cliente, enquanto as camadas de domínio e de dados sendo executadas no servidor. Por outro lado, uma segunda configuração seria ter o servidor tratando apenas da camada de dados, enquanto o cliente trata da camada de apresentação e de domínio.

Entre as duas configurações, a primeira seria a mais simples, ao ter a camada de domínio e de dados fisicamente localizada no servidor. Dessa forma, diversos tipos de cliente poderão rodar a camada de

apresentação, tais como *browser*, *smartphones*, computadores pessoais, entre outros. Essa configuração pode levar, porém, a uma grande carga de processamento sobre o servidor e a rede, visto que o servidor é responsável pela execução das funcionalidades da camada de domínio e de dados. Isso pode gerar um grande volume de dados trafegando pela rede. Por outro lado, a segunda configuração leva o servidor a ter somente a responsabilidade de gerenciar as transações com a base de dados, ao mesmo tempo que exigirá uma capacidade de processamento por parte do cliente, para executar a camada de apresentação e de domínio (SOMMERVILLE, 2007). Os sistemas de caixas eletrônicos de bancos seriam, um exemplo de uso dessa segunda configuração, nos quais o caixa é o cliente e o servidor frequentemente é um *mainframe* que executa operações sobre a base de dados dos clientes. Porém, é importante destacar que, independente da configuração, uma arquitetura cliente-servidor de duas camadas físicas e três lógicas apresenta alguns problemas, tais como escalabilidade, desempenho, gerenciamento de sistemas, entre outros.

Uma forma de mitigar essa problemática é através da distribuição das três camadas lógicas em três camadas físicas. Por exemplo, em uma aplicação Web, a camada de apresentação pode ser executada no cliente, a camada de domínio em um servidor de aplicação, e a camada de dados em servidor de gerenciamento de dados. Promover essa distribuição das camadas lógicas em um número maior de camadas físicas (isto é, em um número maior de servidores) permitirá ter uma arquitetura com melhor escalabilidade, quando comparada àquelas distribuídas em apenas duas camadas.

Por fim, alguns benefícios ao utilizar uma arquitetura cliente-servidor seriam a distribuição de papéis e responsabilidades entre os elementos da rede formada; facilidade na manutenção da estrutura; os recursos e serviços são mantidos no servidor, mitigando problemas de segurança no lado cliente; maior segurança dos dados em servidores com uma maior política de segurança; controle melhor dos recursos, permitindo, por exemplo, apenas os clientes com credenciais válidas terem acesso aos serviços.

Por outro lado, algumas desvantagens podem ser também observadas: se um servidor crítico falhar, os pedidos dos clientes não poderão ser fornecidos; pode acontecer a sobrecarga de um servidor, caso o número de solicitações seja maior que a sua

capacidade suportada; a centralização de recursos e serviços no servidor pode levar à indisponibilidade em algum momento; e, por fim, quanto mais clientes, mais informações serão solicitadas. Logo, problemas com escalabilidade podem existir.

3.2 Arquiteturas usando *pipes and filters*

Sistemas tipicamente têm a atribuição de receber um conjunto de dados e gerar uma saída. Exemplos desses sistemas seriam (EASTERBROOK, 2004) compiladores, recebem texto (código) e geram programas executáveis; e compactadores, recebem um conjunto de arquivos e geram saídas compactadas. Para gerar uma saída a partir de uma entrada, uma sequência de transformações é executada na maioria dos casos. No caso dos compiladores, por exemplo, para gerar programas executáveis, é preciso fazer uma análise léxica, executar um *parser*, analisar a semântica, gerar código, entre outros processamentos. Diante desse cenário de entrada, saída e uma sequência de processamentos, Ken Thomson propôs inicialmente a ideia de *pipes and filters* no sistema Unix. *Pipes and filters* trata-se, portanto, de um estilo arquitetural que se caracteriza pela execução de uma sequência de processamento por quarto elementos (ou componentes) arquiteturais com papéis bem definidos.

A Figura 9 apresenta uma ilustração dos elementos que formam a arquitetura usando *pipes and filters*, são eles *Pump*, *Pipe*, *Filter* e *Sink*. O primeiro elemento, *Pump*, possui a responsabilidade de gerar os dados que serão utilizados pelos elementos que formam a arquitetura. São esses dados que serão modificados, aplicando as cadeias sucessivas de transformações encontradas nos *Filters*. O segundo elemento, *Pipe*, é o responsável por transferir os dados de um *Pump* para um *Filter*, de um *Filter* para outro *Filter*, ou de um *Filter* para um *Sink*. Note que a seta na Figura 9 *Pump* para o *Filter* representa um fluxo de dados. De forma similar, esse fluxo também ocorre do *Filter* para *Filter*, bem como do *Filter* para o *Sink*. Dessa forma, esse conector viabiliza as entradas necessárias para o funcionamento do *Filter* e do *Sink*, bem como para permitir a saída dos dados do *Pump*. O terceiro elemento, *Filter*, por sua vez, desempenha um papel central na arquitetura, visto que implementa as transformações necessárias para realizar as modificações nos dados provenientes do *Pump*. Por fim, o *Sink* representa o alvo do processamento (isto é, trata-se do destino dos resultados dos processamentos realizados), podendo ser um arquivo,

banco de dados ou mesmo uma interface de usuário.

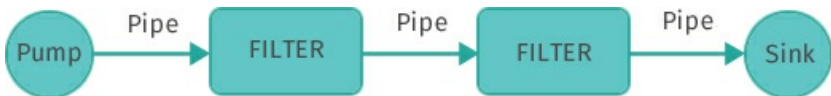


Figura 9 – Elementos da arquitetura baseada em *pipes and filters*.

Fonte: elaborada pelo autor, com base em Bergen (2016).

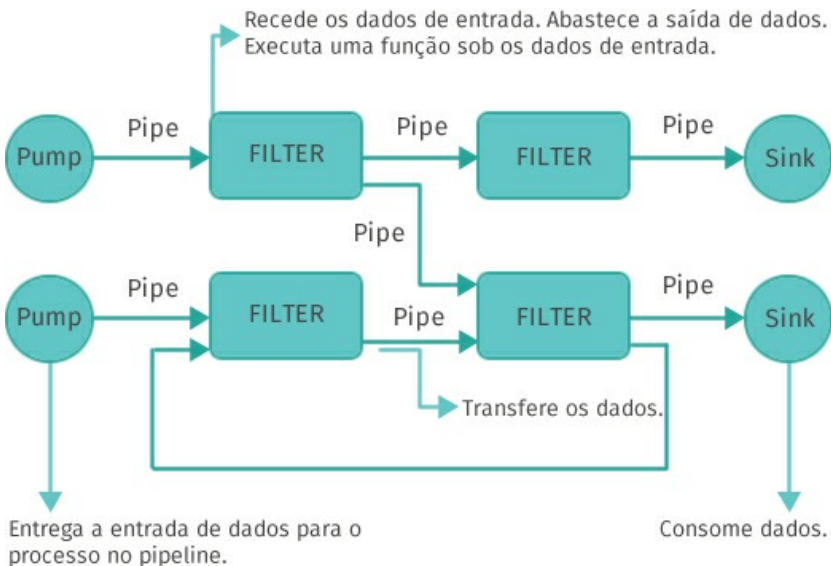


Figura 10 – Exemplo de arquitetura usando *pipes and filters*.

Fonte: elaborada pelo autor, com base em Bergen (2016).

Após apresentar os principais elementos que formam a arquitetura, um exemplo mais refinado é apresentado na Figura 10. Nesse exemplo é possível perceber uma combinação dos elementos básicos da arquitetura de uma forma mais elaborada. Nesse caso, alguns aspectos interessantes são observados: dois *Pumps* são utilizados, permitindo entradas de dados independentes; *Filters* podem ser executados em paralelo; *Filters* podem transferir dados para um mesmo *Filter*; dois *Sinks* são utilizados também, deixando claro que é possível ter mais de um interessado em consumir os dados produzidos ao longo da cadeia de transformações estruturada.

Diante do exposto, recomenda-se utilizar *pipes and filters* quando existe uma grande quantidade de transformações a serem feitas, bem como existe a necessidade de flexibilizar o uso de transformações (HOHPE, 2004).

O funcionamento de uma arquitetura usando *pipes and filters* pode ser basicamente descrito em três passos: aplicação, conecta todas as entradas e saídas dos *Filters* através de *Pipes*; cria um processo para cada *Filter*, o qual pode rodar em paralelo, ou não; se não existirem dados suficientes, o *Filter* pode esperar. Note que um *Filter* pode ter mais de uma entrada. Nesse caso, é preciso definir quando ele será executado, isto é, ele deve esperar que todos as entradas sejam satisfeitas, ou apenas um subconjunto delas.

Após explicar os aspectos gerais da arquitetura usando *pipes and filters*, descrendo seus elementos, bem como mostrando exemplos, o próximo passo é apresentar as etapas que podem ser seguidas para projetar uma arquitetura (BERGEN, 2016; FOWLER, 2006):

1. Dividir o processamento que o sistema precisa realizar em uma sequência de estágios ou etapas. A execução de cada etapa deve depender apenas do seu predecessor direto, caso contrário poderá gerar dependências indesejadas. Além disso, cada etapa deve estar conceitualmente conectada pelo fluxo de dados; caso contrário, ela não fará parte da arquitetura.
2. Definir o formato dos dados a serem passados ao longo de cada *Pipe*.
3. Decidir como implementar cada conexão *Pipe*, essa decisão influencia a forma como os *Filters* serão implementados.
4. Projetar e implementar os *Filter*. O projeto de um *Filter* é baseado tanto na tarefa que precisa executar, quanto nos *Pipes* adjacentes.
5. Projetar o tratamento de erro. Como os componentes da *pipeline* não compartilham estado, erros são difíceis de identificar. A infraestrutura da *pipeline* deve prover um bom mecanismo para tratamento de erros.
6. Ajustar a *pipeline* de processamento. Se o sistema executa um único tipo de tarefa, é possível fazer com que o programa

principal inicie a *pipeline* e, na sequência, a execução.

Ao projetar um sistema com uma arquitetura usando *pipes and filters*, é possível perceber algumas vantagens, tais como (1) um melhor encapsulamento das responsabilidades do sistema, alta coesão, recombinação e reuso dos dados, implicando alta reutilização; (2) diferentes fontes de entrada de dados existem, e pode-se apresentar e armazenar o resultado final de diferentes formas; e (3) o sistema pode ser facilmente estendido e modificado, facilitando a implementação em processadores paralelos. Por outro lado, algumas desvantagens também podem ser observadas: (1) devido ao processamento ocorrer em lotes, é difícil criar aplicações interativas; (2) pode existir a necessidade de utilizar um *buffer* de tamanho limitado que pode causar a *deadlock*; e (3) possível baixa performance.

CAPÍTULO 4

Componentes e linguagem de descrição arquitetural

Neste capítulo serão apresentados conceitos fundamentais para o entendimento de arquiteturas baseadas em componentes de software, os quais permitirão compreender as vantagens e desvantagens deste tipo de arquitetura, permitindo também um encaminhamento prático de ações nessa área. Além disso, serão discutidas algumas propostas de linguagem de descrição de arquitetura, sendo introduzidas as notações utilizadas, as restrições e configurações das linguagens.

A engenharia de software baseada em reuso está se tornando a principal abordagem de desenvolvimento de sistemas corporativos e comerciais. Para colocar isso em prática, é necessário um conjunto de iniciativas, tais como a definição, implementação, integração ou composição de componentes arquiteturais, os quais são usualmente projetados de forma independente e prezam pelo baixo grau de acoplamento e alta coesão. Um aspecto relevante neste contexto são as características cada vez mais marcantes que os componentes arquiteturais precisam ter, tais como padronização, flexibilidade e independência, capacidade de se integrar com outros componentes, facilidade de um componente ser implantado, bem como a exigência pela documentação atualizada para viabilizar compartilhamento e alinhamento de entendimento da equipe sobre a arquitetura utilizada. Nesse cenário, tanto a arquitetura baseada em componentes, quanto a linguagem de descrição de arquitetura, desempenham papéis fundamentais para viabilizar o desenvolvimento de software com arquiteturas flexíveis, com estrutura fortemente baseada em reuso, o qual pode ser viabilizado através da composição de componentes, por exemplo.

4.1 Arquiteturas baseadas em componentes

De acordo com Sommerville (2007), existe um consenso de que um componente de software é uma unidade de software independente que pode ser composta com outras unidades (ou componentes) para criar um sistema de software. Para tornar esta composição viável, os componentes definem os serviços de que eles precisam para funcionar através de suas interfaces requeridas, e os serviços que eles oferecem ao meio através das interfaces providas. A Figura 11 apresenta uma ilustração de um componente com duas interfaces requeridas e duas interfaces providas. É importante lembrar que tais interfaces são iguais àsquelas já amplamente utilizadas no diagrama de classes da UML (*Unified Modeling Language*) (OMG, 2011). Além disso, destaca-se que um componente de software deve ser uma unidade coesa e de baixo acoplamento que deve representar de forma simplificada os serviços produzidos.

Apesar deste entendimento, vários pesquisadores e profissionais da indústria propuseram outras definições. Em Council e Heineman (2001), componente é definido como um elemento de software que está em conformidade com um modelo de componente e pode ser implantado de forma independente e composto sem modificação de acordo com um padrão de composição. Nesta definição, o autor toma como base que um componente deve ser derivado de um modelo, o qual define um padrão. A definição apresentada por Szyperski (2002) foca em destacar as principais partes de um componente: um componente é uma unidade de composição com interfaces contratualmente especificadas e com dependências de contexto explícitas. Um componente de software pode ser implantado de forma independente e está sujeito à composição por terceiros.



Figura 11 – Exemplo de arquitetura usando *pipes and filters*.
Fonte: elaborada pelo autor, com base em Bergen (2016).

Os componentes arquiteturais devem ser definidos, implementados, mantidos e evoluídos por arquitetos e desenvolvedores de software,

sempre tendo suas principais características preservadas. Exemplos destas características essenciais para os componentes são padronização, independência, capacidade de interagir com outros componentes, alta capacidade de ser implantado, documentação atualizada e definida de forma clara.

Seguindo a linha de raciocínio de Council e Heineman (2001), por definição, um componente deve atender as exigências de um modelo. Por consequência, um componente necessariamente seguirá um padrão, algo de grande valor para viabilizar a composição dos componentes. Se os componentes não seguem uma padronização, então a integração entre eles pode ser comprometida, visto que o esforço necessário para contornar esta incompatibilidade não justificaria a reimplementação de tais componentes.

Observando a segunda característica, os componentes arquiteturais devem ser idealizados de forma independente, de tal forma que cada componente implemente uma responsabilidade (ou requisito) com o menor grau de dependência em relação aos demais componentes. Esta independência irá garantir um atributo de qualidade essencial para os componentes arquiteturais: a modularidade. Projetar componentes modularizados implica prezar pela alta coesão e o baixo acoplamento do componente em questão, em relação aos demais. Sendo assim, a arquitetura que dará suporte aos requisitos do sistema será alcançada através da integração dos componentes independentes. Desta forma, projetar uma arquitetura implica definir suas unidades funcionais (isto é, os componentes) de forma independente e garantir a compatibilidade entre os componentes.

Outra característica importante é a capacidade do componente de interagir com outros componentes. Esta interação é essencial para viabilizar a característica descrita anteriormente, bem como para colocar em prática o reuso de componentes previamente implementados, ou até mesmo componentes de terceiros. Priorizar e garantir que um componente tenha esta característica trata-se de algo crítico, pois sem ela não é possível criar uma arquitetura flexível, que tem a sua estrutura fortemente baseada em reuso e na composição das suas partes.

Alinhada diretamente com a independência e a composição dos componentes para formar estruturas arquiteturais mais complexas,

encontra-se a capacidade de ser implantado e executado, independente da plataforma. Para projetar componentes arquiteturais que tenham tais características, arquitetos devem focar na elaboração de componentes autocontidos e que não precisem ser compilados para que possam ser utilizados.

A quinta característica a ser destacada seria a documentação que todo componente deve necessariamente ter. Ao não ser documentado, pouco importará se o mesmo for padronizado, independente, passível de composição e implantável, pois arquitetos, analistas e desenvolvedores terão dificuldades de compreender a estrutura do componente, os serviços disponibilizados, as decisões de projeto que foram tomadas para garantir atributos de qualidade e assegurar restrições impostas por requisitos não funcionais. A Tabela 1 resume as cinco características de componente que se deve observar ao projetar uma arquitetura baseada em componentes.

Tabela 1 – Características de componente

Característica	Descrição
Padronizado	Padronizar um componente significa torná-lo aderente a algum modelo de componente. Este modelo definirá como as interfaces requeridas e providas devem ser definidas e especificadas, quais metadados devem ser gerados, como a documentação deve ser gerada, o que deve ser feito para viabilizar a composição, entre outras coisas.
Independente	Um componente arquitetural não deve estar acoplado a outros componentes de tal forma que comprometa a sua capacidade de composição e implantação. Pelo contrário, deve ser independente e ter alta capacidade de compor com outros componentes. Caso utilize algum serviço disponibilizado por outro componente, o mesmo deve fazer seguindo a especificação da interface requerida.
Passível de composição	Toda interação de um componente com outro deve ser feita através das interfaces definidas (providas e/ou requeridas).

Implantável	Para que possa ser implantado, o projeto de uma componente deve prezar pela capacidade de ser autocontido. Além disso, o componente deve ser capaz de operar de forma independente sobre a plataforma de componente que implementa o modelo de componente.
Documentado	A documentação dos componentes deve ser, sempre que possível, completa e atualizada, pois é através dela que os usuários decidirão se o componente atende ou não aos requisitos que devem ser implementados. Deve-se prezar por um detalhamento da sintaxe das interfaces providas e requeridas, bem como da semântica delas.

Fonte: elaborada pelo autor, com base em Sommerville (2007).

Após definir uma arquitetura baseada em componentes, recomenda-se representá-la com o objetivo de garantir que decisões arquiteturais tomadas possam ser devidamente documentadas. Por esta razão, várias notações têm sido propostas ao longo dos anos, sendo o diagrama de componentes da UML (OMG, 2011) a notação que tem sido amplamente utilizada. Esta notação é definida baseada em algumas definições de componentes encontradas na especificação da UML que corrobora com as definições apresentadas anteriormente. A especificação da UML pode ser encontrada em OMG (2011).

De acordo com OMG (2011), um componente arquitetural deve representar uma parte física e substituível de uma arquitetura de um sistema de software, que modulariza os serviços fornecidos e exigidos do meio pelo componente, ao realizar um conjunto de interfaces requeridas e providas, respectivamente. Além disso, entende-se que um componente de uma arquitetura trata-se de uma unidade autocontida que encapsula estados e comportamentos de um conjunto de classificadores. Na UML, um classificador é uma metaclassa que é utilizada para definir, por exemplo, uma classe, uma interface, uma classe abstrata, uma enumeração, ou mesmo um próprio componente.

É também claro o entendimento proposto pela OMG (2011) de que um componente arquitetural deve ser projetado para ser uma unidade substituível (em tempo de projeto ou execução) por outro

componente, o qual deve ser capaz de oferecer os serviços previamente disponibilizados. Para que isso seja de fato possível, o componente que irá substituir o componente atual (ou seja, aquele que se encontra em uso) deve respeitar a compatibilidade das interfaces. Desta forma, os contratos estabelecidos através das interfaces serão assegurados.

A Figura 12 apresenta um exemplo de diagrama de componentes da UML. Esse diagrama representa a arquitetura baseada em componentes de uma ferramenta de integração de modelos UML, a qual é chamada de MOCOTO (*Model Composition Tool*). O objetivo central neste exemplo é mostrar um uso prático dos principais elementos de um diagrama de componentes (tais como componente, interface requerida e provida). Essa ferramenta de integração de modelos UML visa permitir, por exemplo, a fusão de dois diagramas de classes da UML criados em paralelo por diferentes desenvolvedores.

Em desenvolvimento de software colaborativo, desenvolvedores podem trabalhar em paralelo, por exemplo, criando um diagrama de classes da UML, com o objetivo de permitir que eles se concentrem na elaboração das partes dos modelos mais relevantes para eles, geralmente aqueles que eles têm mais conhecimento. Porém, em um determinado momento é necessário unir diagramas de classes da UML criados em paralelo, visando criar um diagrama que contemple todas as decisões arquiteturais tomadas e inseridas nos diagramas. Neste momento, é crucial o uso de uma ferramenta que auxilie os desenvolvedores no processo de integração dos diagramas criados em paralelo.

A literatura atual (FARIAS, 2014) reforça a importância de uma ferramenta com esse propósito, visto que, para unir modelos criados em paralelo, os desenvolvedores precisam frequentemente resolver conflitos entre os elementos que formam os modelos. Sabe-se atualmente que a resolução de conflitos é uma atividade altamente propensa a erros e que tende a consumir bastante esforço.

Diante desse cenário, a ferramenta de composição de modelo visa dar suporte aos desenvolvedores na atividade de integração ao implementar cinco requisitos essenciais, os quais são citados brevemente no lado superior esquerdo da Figura 12. Entender esses requisitos é uma etapa fundamental para concepção de uma

arquitetura adequada. Compreende-se como uma arquitetura adequada aquela que atende aos requisitos do sistema, incluindo requisitos funcionais e não funcionais. Note que atributos de qualidade como, por exemplo, modularidade e flexibilidade, entre outros, podem ser vistos como requisitos não funcionais, pois os mesmos podem impor restrições à arquitetura, bem como ao projeto do sistema como um todo. Então, um desafio seria como projetar uma arquitetura baseada em componentes que seja capaz de suportar os seguintes requisitos.

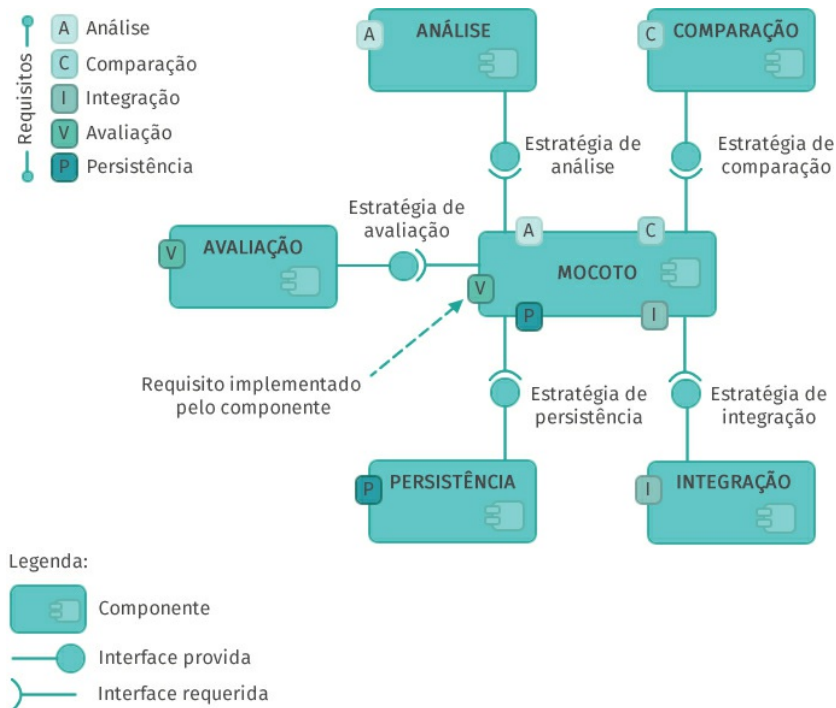


Figura 12 – Exemplo de diagrama de componentes da UML

Fonte: elaborada pelo autor.

- Análise de modelos. A ferramenta deve analisar dois modelos de entradas com objetivo de verificar inconsistências entre os mesmos, bem como checar se os dois modelos podem ser

integrados ou não.

- Comparação de modelos. Após passar pelo processo de análise, a ferramenta deve comparar os dois modelos, visando identificar as similaridades entre os elementos, classificar os elementos dos modelos que são iguais e aqueles que são diferentes, bem como aqueles que têm partes conflitantes. Ao identificar elementos conflitantes, tais elementos dos modelos serão recomendados para passarem por um processo de resolução de conflitos.
- Integração de modelos. Tendo identificados os elementos dos modelos equivalentes, os completamente diferentes e aqueles que possuem partes conflitantes, o próximo passo é realizar a integração. Sendo assim, a arquitetura da ferramenta deve ser capaz de permitir integrar os elementos equivalentes, acomodar os elementos diferentes no modelo integrado que será produzido, bem como recomendar e aplicar um processo de resolução de conflitos dos elementos que têm partes conflitantes.
- Avaliação do modelo. Após integrar os dois modelos de entrada, um modelo composto será produzido. Esse modelo pode ter problemas gerados, por exemplo, devido à resolução incorreta de conflitos. Um exemplo de resolução incorreta de conflito seria quando um método de uma classe no modelo composto que deveria ter um tipo de retorno *double*, porém é do tipo *String*. Isso pode acontecer porque durante a resolução de conflito para decidir entre *double* e *String*, o desenvolvedor optou incorretamente pelo tipo *String*. Desse modo, a arquitetura da ferramenta deve também ser capaz de avaliar os modelos gerados.
- Persistência dos modelos. Durante o processo de integração dos modelos, é necessário que a ferramenta manipule modelos. Para isso, deve-se persistir com diagramas UML em disco no formato XML, o qual é o padrão adotado pela OMG (2011).

Diante desses requisitos, a arquitetura baseada em componentes proposta é mostrada na Figura 12, como previamente citado. Essa arquitetura tenta ao máximo prezar por boas práticas de projeto, ao

respeitar princípios de projetos como, por exemplo, o princípio da responsabilidade única, bem como o princípio *open-close* (MARTIN, 2002).

Visando atender o princípio da responsabilidade única, buscou-se ao máximo delegar um requisito (ou responsabilidade) para um componente da arquitetura. Sendo assim, um componente será alterado se mudanças estiverem relacionadas com a responsabilidade que o mesmo possui. Seguindo esta linha de raciocínio, os cinco requisitos citados anteriormente foram modularizados em cinco componentes. Cada componente encontra-se diretamente relacionado a uma interface, que especifica os serviços disponibilizados por ele. Por exemplo, o componente de Análise de Modelos implementa a interface Estratégia de Análise, o componente de Comparação de Modelos implementa a interface Estratégia de Comparação, o componente de Integração de Modelos implementa a interface Estratégia de Integração, o componente de Persistência de Modelos implementa a interface Estratégia de Persistência e o componente de Avaliação de Modelos implementa a interface Estratégia de Avaliação. Ao centro do diagrama, tem-se o componente principal da arquitetura, chamado de MOCOTO. Esse componente foi idealizado como uma unidade orquestradora da aplicação, aquela que fará uso dos serviços disponibilizados pelos demais componentes.

Procurou-se também satisfazer o preconizado pelo princípio aberto-fechado (MARTIN, 2002), um software deve estar aberto para extensão, e não para modificação. Se novas estratégias de análise, comparação, integração, avaliação e persistência forem propostas, as mesmas serão implementadas e inseridas nos respectivos componentes. Sendo assim, a modificação será dominada por adição de novos elementos na arquitetura, não exigindo a alteração dos elementos existentes.

Após descrever o exemplo, destaca-se a possibilidade da criação de arquiteturas a partir da composição de componentes, visando ter ganhos significativos em produtividade. Criar uma arquitetura baseada na composição de componentes arquiteturais consiste, basicamente, no processo de combinar os componentes, considerando os serviços disponibilizados pelos mesmos, bem como suas compatibilidades. A Figura 12 apresenta um exemplo prático de composição de serviços disponibilizados pelos componentes Análise,

Comparação, Integração, Avaliação e Persistência. Neste caso, o componente MOCOTO compõe os serviços de tais componentes em uma unidade central. Não confundir o exemplo de integração de modelos UML com composição de serviços de componentes. Enquanto o primeiro une dois modelos, visando produzir um modelo único, o segundo visa integrar serviços para formar um componente composto, ou seja, aquele que pode ter acesso aos serviços disponibilizados por mais de um componente. A composição de serviços visa primordialmente criar uma arquitetura através do reuso de componentes, ao mesmo tempo que os requisitos do sistema são suportados.

Seguindo essa abordagem, é fundamental que componentes reusáveis estejam disponíveis *a priori*, visando promover o máximo de reuso, ao mesmo tempo que seja possível potencializar o ganho de produtividade da equipe envolvida com o projeto arquitetural. Porém, caso um determinado componente não esteja disponível, é possível também desenvolvê-lo, objetivando garantir o reuso dos serviços dos outros componentes, bem como dar suporte aos requisitos, até então, não suportados pelos componentes já desenvolvidos.

De acordo com Sommerville (2007), a composição dos componentes pode ser feita de três formas. A Figura 13 apresenta um exemplo das composições, as quais são descritas a seguir:

- Composição sequencial. Nesta modalidade de composição, os componentes que disponibilizam um conjunto de serviços através de suas interfaces providas são executados em sequência por um componente central (ou composto). Este componente central, por sua vez, ao ter acesso aos serviços poderá executá-los. Na Figura 13, é possível verificar que os Componentes A e B são os componentes que fornecem um conjunto de serviços ao componente composto, o qual realiza a composição ao executar de forma sequencial os serviços disponibilizados através das interfaces providas dos Componentes A e B.
- Composição hierárquica. Diferente da composição sequencial, esta modalidade de composição ocorre quando um componente chama diretamente os serviços disponibilizados por outro. Desta forma, a interface provida por uma

componente é combinada com a interface requerida por outro componente. Na Figura 13, verifica-se este tipo de composição quando o Componente A utiliza os serviços disponibilizados pelo Componente B de forma direta.

- Composição aditiva. Esta modalidade de composição ocorre quando duas ou mais interfaces providas de dois ou mais componentes são compostas para criar um novo componente. Desta forma, os serviços que serão oferecidos pelo componente composto serão constituídos pelos serviços originalmente oferecidos pelos componentes constituintes, removendo os serviços duplicados, caso eles existam. Na Figura 13, observa-se essa modalidade de composição ao ter dois componentes compostos formados pela composição de interfaces providas por outros componentes. Nesta ocasião, o Componente Composto (na parte inferior) faz uso dos serviços disponibilizados pelos Componentes A e B, os quais passam a oferecer um conjunto de serviços através de duas interfaces providas. Os serviços disponibilizados pelo componente composto podem ser os mesmos oferecidos pelos Componentes A e B, podem ser um subconjunto deles, ou mesmo podem oferecer um conjunto de serviços a mais que os disponibilizados previamente pelos Componentes A e B. Seguindo essa mesma linha de raciocínio, o Componente Composto (na parte superior) também fará uso dos serviços providos por outros componentes. Os serviços que ele precisa para funcionar são aqueles especificados nas suas duas interfaces requeridas, as quais na ilustração não estão sendo fornecidas por outros componentes. Uma vez que tais serviços venham a ser fornecidos, o Componente Composto poderá fornecer os serviços requisitados pelos Componentes A e B.

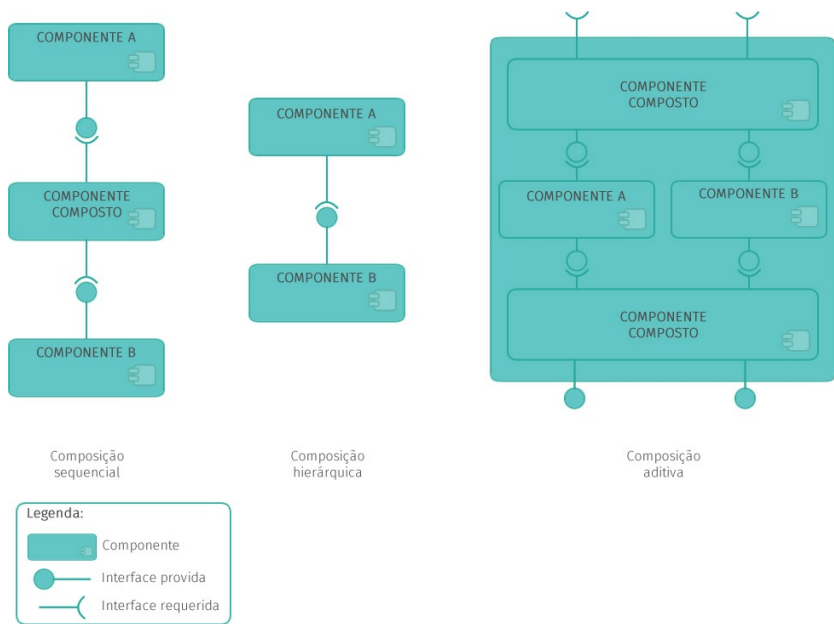


Figura 13 – Tipos de composição de componentes.
Fonte: elaborada pelo autor, com base em Sommerville (2007).

Ao projetar uma arquitetura, é possível utilizar todas as formas de composição de componentes, visando a criação de módulos flexíveis e o ganho de produtividade da equipe. Porém, as composições entre os componentes devem ser feitas com cuidado, visto que tais composições podem ser propensas a erros e consumir tempo (FARIAS, 2014). Essa propensão a erros deve-se à necessidade de adaptações para viabilizar a integração dos componentes.

Na Figura 13, por exemplo, para realizar a composição sequencial, o componente central que faz uso dos serviços dos Componentes A e B deve viabilizar a compatibilidade entre os serviços oferecidos pelos mesmos. Isso porque tipicamente as saídas (ou resultados) produzidas pelos serviços oferecidos pelo Componente A serão tratadas como as entradas para os serviços oferecidos pelo Componente B. Desta forma, é fundamental que uma saída produzida por um serviço pelo Componente A seja compatível com a

entrada esperada pelo serviço oferecido pelo Componente B. Caso contrário, adaptações serão necessárias. Tais incompatibilidades surgem devido às especificações nos serviços nas interfaces providas dos Componentes A e B serem similares, não iguais. Por exemplo, um serviço do Componente A pode calcular o valor de um imposto, produzindo como resultado um valor do tipo *double*. Por outro lado, outro serviço do Componente B usará o resultado do cálculo do imposto produzido, porém ele espera um valor do tipo inteiro, e não *double*. Desta forma, para que o serviço do Componente B possa usar o resultado do cálculo do imposto, o componente composto (central) precisará fazer algumas adaptações.

Em Sommerville (2007), o autor reforça a necessidade de declarações intermediárias que chamem os serviços do Componente A, colem os resultados, e então chamem os serviços do Componente B com as devidas adaptações. Na prática, tais adaptações são concretizadas através da manipulação dos parâmetros que serão passados para os serviços do Componente B. Exemplos de tais manipulações seriam a conversão de tipo, redução de número de casas decimais, entre outras.

Diante do exposto, recomenda-se que, ao projetar uma arquitetura fundamentada na composição de componentes, é primordial definir interfaces dos componentes arquiteturais de tal maneira que sejam compatíveis. No entanto, nem sempre é possível conseguir projetar uma arquitetura com uma completa compatibilidade entre as interfaces dos componentes. Neste caso, deve-se, ao máximo, minimizar as incompatibilidades. Note que, se o número de incompatibilidades for elevado, o reuso dos componentes pode ser questionável, devido ao esforço que será exigido pelas adaptações.

As incompatibilidades podem acontecer quando componentes são desenvolvidos de forma independente. Nestes casos, os componentes tipicamente não são projetados para trabalharem juntos, revelando, como citado anteriormente, contradições entre os serviços especificados nas interfaces. De acordo com Sommerville (2007), três tipos de incompatibilidades podem ocorrer, as quais são brevemente descritas a seguir:

- Incompatibilidade de parâmetro. Essa modalidade de incompatibilidade acontece quando as interfaces (isto é, a provida e a requerida) têm o mesmo nome, porém possuem

tipos de parâmetros diferentes ou o número de parâmetros é diferente.

- Incompatibilidade de operação. Os nomes dos serviços descritos nas interfaces provida e requerida são diferentes.
- Inconclusão de operação. Uma interface provida por um componente representa um subconjunto da interface requerida por outro componente, ou vice-versa.

Independente do tipo de incompatibilidade, a resolução do problema passa pelo projeto e desenvolvimento de um componente adaptador, o qual permitirá reconciliar os conflitos entre as duas interfaces (provida e requerida). O papel central deste adaptador é converter os serviços oferecidos, com o objetivo que os mesmos possam ser utilizados pelos serviços especificados na interface requerida. Note que a forma como a conversão será feita dependerá do tipo de incompatibilidade. Em muitos casos, o adaptador precisará apenas converter o tipo de resultado produzido por um serviço, em outro formato que possa ser utilizado como entrada para outro serviço.

A adoção de arquitetura baseada em componentes deve ser feita observando as vantagens e as desvantagens que a mesma pode proporcionar. Como desvantagem, é possível citar a necessidade de experiência para conseguir projetar corretamente uma boa arquitetura baseada em componentes, visto que a mesma exige conhecimento de princípios de projetos, de padrões de projetos e de alguma notação para conseguir especificá-la corretamente. Além disso, para que de fato os benefícios possam ser observados, exige-se que um investimento prévio tenha sido realizado; caso contrário, o reuso desejado com esta abordagem não será alcançado. Por outro lado, algumas das vantagens seriam as seguintes:

- Reutilização de componentes. Uma vez que a arquitetura tenha sido proposta, implementada e testada, é possível que, em um próximo ciclo de desenvolvimento, ocorra a redução do esforço de desenvolvimento, ao mesmo tempo que a qualidade do produto final seja também assegurada. De fato, isso acontecerá ao reaproveitar módulos já testados e validados.
- Desenvolvimento ágil. À medida que a arquitetura pode ser

definida considerando o reuso de componentes previamente definidos, especificados, implementados, validados, testados e documentados, é possível, de fato, promover um desenvolvimento ágil.

- Modularização. Ao projetar utilizando os princípios de projetos de software e o conceito de componentes, torna-se mais prático decompor responsabilidades em unidades mais modulares, priorizando um alto encapsulamento, uma alta coesão, ao mesmo tempo que também permite um baixo acoplamento. Uma consequência direta disso são os benefícios encontrados durante a manutenção e evolução da arquitetura.
- Gerenciamento de complexidade. Ao ter a arquitetura devidamente modularizada, torna-se possível acelerar o desenvolvimento ao paralelizar o desenvolvimento dos componentes; garantir uma conformidade das unidades da arquitetura, ao exigir que cada componente seguirá um modelo previamente estabelecido; e reduzir o *time-to-market* enquanto também garante a qualidade.

Por fim, alguns elementos centrais encontrados na arquitetura orientada a componentes são identificados e elencados a seguir (SOMMERVILLE, 2007):

- Arquitetura de software baseada em componentes trata-se de uma abordagem baseada em reuso para definição, implementação e composição de componentes independentes, com baixo grau de acoplamento e alta coesão.
- Um componente pode ser visto como uma unidade de central dentro de uma arquitetura de software, cujas responsabilidades e dependências podem ser definidas por um conjunto de interfaces públicas. Cada componente deve ser projetado de tal forma que ele possa ser combinado com outros componentes de forma flexível, garantindo sempre alta coesão e baixo acoplamento.
- Preconiza-se que cada componente possa ser projetado e implementado como uma unidade independente e executável. Quando necessário, será integrado com outros componentes considerando sempre os contratos estabelecidos pelas

interfaces definidas, e nunca fazendo referência direta para a implementação.

- Um projeto arquitetural pode fazer uso de padrões de componentes arquiteturais, os quais definem um modelo de interface, de uso e implementação. Na essência, cada componente deve fornecer um conjunto de serviços (ou comportamentos), os quais poderão ser utilizados por outros componentes da arquitetura.
- Ao projetar uma arquitetura baseada em componentes, é necessário definir de forma clara os requisitos que a arquitetura deve suportar, bem como estabelecer um mapeamento dos requisitos desejáveis com os serviços disponibilizados por componentes reutilizáveis (ou mesmo que serão implementados).
- O processo de composição (ou combinação) de componentes arquiteturais, visando colocar em prática o mapeamento requisito-serviço, tem como objetivo promover reuso e ganho de produtividade através de composições sequencial, hierárquica e aditiva. Todo o processo de composição deve respeitar as boas práticas e os princípios de projetos de software.
- Criar uma arquitetura através de um processo de combinação dos serviços oferecidos pelos componentes pode exigir adaptações das interfaces implementadas pelos componentes. Isso acontece devido às incompatibilidades que podem surgir entre as maneiras como os serviços foram especificados. Recomenda-se, desse modo, usar adaptadores como aqueles definidos, por exemplo, usando o padrão de projeto *Adapter* (GAMMA, 1994).
- Ao combinar os componentes, é importante levar em consideração os requisitos funcionais e não funcionais que a arquitetura deve contemplar, bem como aspectos relacionados à facilidade de troca dos componentes por outros.

4.2 Linguagem de descrição arquitetural

Linguagem de descrição de arquitetura (LDA) é utilizada para

representar a estrutura, os componentes e suas interconexões, ao mesmo tempo que propõe notações para melhorar a forma de especificar e comunicar decisões arquiteturais aos usuários da arquitetura. O uso de linguagem de descrição de arquitetura visa melhorar a comunicação entre os usuários, permitir a representação de decisões de projeto, e permitir a criação de abstrações transferíveis ou intercambiáveis entre sistemas. Exemplos de linguagens de descrição de arquitetura são ACME (The Acme Project) (ACME, 2016), AADL (*Architecture Analysis & Design Language*) (FEILER, 2012) e ABC (*Supporting Component Composition*) (MEI, 2002). Uma lista de linguagens e ferramentas de suporte para linguagens de descrição de arquitetura pode ser encontrada em Alt (2016).

Os autores da linguagem de descrição de arquiteturas defendem que, ao utilizar uma linguagem como, por exemplo, ACME (ACME, 2016), será possível, não só a modelagem de arquitetura em si, mas também permitir que outras atividades possam ser executadas, tais como a verificação se a arquitetura prevista está compatível com a arquitetura implementada, a representação da arquitetura através de diferentes perspectivas, visando atender as necessidades dos seus usuários, geração de código da aplicação a partir das especificações, e documentação da arquitetura.

O valor associado à linguagem de descrição de arquiteturas pode ser percebido ao vê-las como uma forma de representar formalmente uma arquitetura e uma maneira de representar estruturas de alto nível, ao contrário de detalhes de implementação. Além disso, ao projetar uma arquitetura usando uma linguagem de descrição de arquitetura, deve-se buscar a especificação da arquitetura dando ênfase na composição das partes da arquitetura, na abstração das partes, e na reusabilidade, sempre fazendo uso efetivo e sistemático do conceito de componentes e conectores.

A especificação gerada pela linguagem pode ser utilizada para entender a estrutura da arquitetura proposta, avaliar se a arquitetura, de fato, suportará os requisitos da aplicação, simular a arquitetura em questão, bem como os resultados dessa simulação podem ser usados para modificar a especificação da própria arquitetura, visando produzir a melhor arquitetura para um conjunto de aplicações.

Ao utilizar uma linguagem de descrição de arquitetura, um arquiteto

poderá abstrair a arquitetura, ao passo que poderá se concentrar na elaboração de uma visão geral dos componentes, destacando os protocolos de comunicação de alto nível e as responsabilidades de cada componente da arquitetura. Vale salientar também que os usuários de uma linguagem de descrição de arquitetura poderão ter um melhor apoio e controle durante o processo de desenvolvimento, bem como se beneficiar pela incorporação de conceitos específicos de domínio de uma arquitetura (ou mesmo comum a um conjunto de arquiteturas). Essa inserção de conceitos específicos da área de arquitetura pode permitir uma maior compreensibilidade, ao tornar as notações mais próximas da realidade encontrada ao especificar ou mesmo ler uma arquitetura. De fato, o estudo experimental reportado em Ricca (2010) mostra que a lacuna de conhecimento existente entre desenvolvedores inexperientes e experientes pode ser reduzida, ao inserir conceitos de domínio nas anotações das linguagens de modelagem utilizadas para representar a arquitetura de uma aplicação Web. Nesse estudo, o recurso utilizado para representar os conceitos de domínio foi o estereótipo da UML.

Relembrando que a arquitetura de um sistema de software deve guiar e restringir como a implementação deve ser feita e não deve surgir do ato de implementar. Ao usar uma linguagem de descrição de arquitetura, evidencia-se a necessidade de se definir estrategicamente os componentes, suas responsabilidades e interconexões.

As linguagens de descrição de arquitetura apresentam diferentes construtores, os quais representam os elementos que serão aplicados pelos usuários para representar os diferentes aspectos estruturais e comportamentais encontrados em uma arquitetura. Os principais elementos são citados a seguir:

- **Componente.** Assim como no diagrama de componentes da UML, o conceito de componente é comumente encontrado na maioria das linguagens. Sendo assim, é considerado como um elemento central na definição da estrutura da arquitetura em linguagem de descrição de arquitetura. É por meio dos relacionamentos entre os componentes da arquitetura que os requisitos são viabilizados. Tipicamente, os componentes das LDAs possuem interfaces bem definidas, devem ser projetados visando uma estabilidade, baixo acoplamento e alta coesão, bem como são vistos como uma unidade da arquitetura.

- Interface. Cada componente precisa especificar quais serviços eles fornecem e quais utilizam. Para isso, são utilizadas as interfaces, as quais são também amplamente encontradas em diagramas UML. É através da interface que usualmente as LDAs definem como cada componente da arquitetura irá interagir com o meio externo.
- Conector. Elemento é responsável por encapsular a comunicação, coordenação e as decisões de mediação de um componente.
- Configuração arquitetural. Como previamente já mencionado, toda arquitetura deve contemplar os requisitos funcionais e não funcionais do sistema de software para ela projetado. Com isso em mente, as LDAs trazem o conceito de configuração arquitetural como um recurso utilizado para representar os atributos e as propriedades que uma arquitetura deve atender. Esses atributos e propriedades tipicamente estão relacionados com atributos de qualidade, tais como rastreabilidade, flexibilidade, modularidade, escalabilidade, estabilidade, entre outros. Fazendo uso da configuração arquitetural, é possível também definir sob quais restrições a arquitetura deve operar. Por exemplo, o modelo MVC define algumas restrições sobre como os módulos do sistema devem operar estabelecendo papéis e como tais módulos, uma vez assumindo um papel, devem se relacionar com os demais. Considerando a heterogeneidade, é possível também especificar como deve ser o desenvolvimento de componentes e conectores heterogêneos do ponto de vista arquitetural.

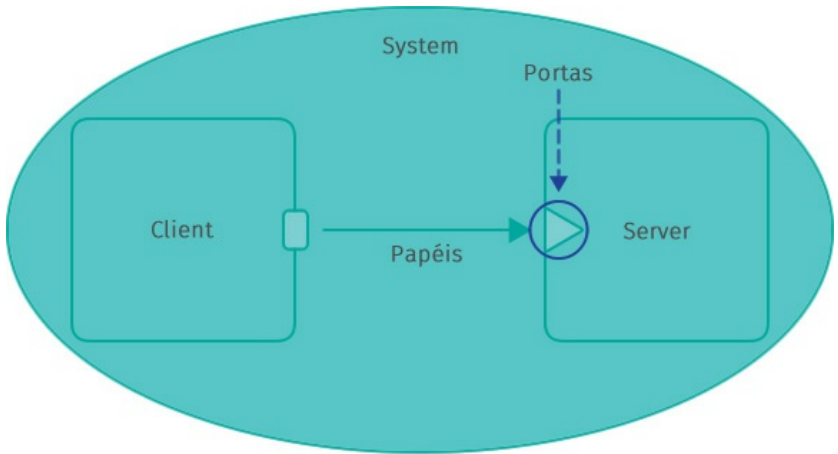
Um exemplo de linguagem de descrição de arquitetura é ACME. O projeto ACME (ACME, 2016) iniciou-se em 1995 com o objetivo de produzir uma linguagem comum que pudesse ser usada para suportar o intercâmbio de descrições arquiteturais entre ferramentas. Como resultado do projeto, foi produzida uma linguagem de descrição de arquitetura, nomeada de ACME, a qual foi projetada para ser uma linguagem de descrição simples e genérica. A linguagem ACME fornece uma infraestrutura extensível para descrever, representar, gerar e analisar descrições de arquitetura de software. Além disso, ela foi projetada para permitir a interoperabilidade entre ferramentas de projeto de arquitetura, viabilizar o desenvolvimento de novas ferramentas e técnicas de

análise de arquitetura, assim como servir como base para o desenvolvimento de novas ferramentas de análise e projeto arquitetural.

É necessário destacar que ela possui três capacidades fundamentais: (1) intercâmbio de arquitetura, ao fornecer um formato genérico para projeto arquitetural, ACME permite que desenvolvedores de ferramentas rapidamente integrem a linguagem a outras ferramentas; (2) extensível, com o objetivo de permitir diminuir os custos para o desenvolvimento de novas ferramentas; e (3) descrição de arquitetura, fornece um conjunto de construtores para descrever arquiteturas e seus estilos arquiteturais.

A Figura 14 apresenta um exemplo de uma arquitetura representada na linguagem ACME. Como pode ser visto, a linguagem tanto oferece uma representação diagramática, quanto textual, onde uma complementa a outra. Nesse exemplo, é possível observar o suporte a conceitos como componentes, papéis, portas, conectores e *attachments*. Em particular, esse exemplo representa uma arquitetura cliente-servidor.

Sendo isso, o sistema é nomeado como *simple_client_server* (linha 1) e dois componentes são definidos e nomeados de *Client* e *Server*, os quais possuem duas portas (linha 2 e 3). Enquanto a porta do *Client* é para enviar requisições (no código essa afirmação é representada pelo trecho *Port send-request*); a porta do *Server* é para receber requisições (sendo representada pelo trecho *Port receive-request*). Além disso, o diagrama também apresenta um conector (nomeado de RPC (chamada remota de método) na linha 4), o qual é utilizado para representar como as interações entre os componentes devem ser feitas. Para isso, dois papéis são definidos (linha 4), representado pelo trecho de código *Roles {caller, callee}*. Por fim, *attachment* é utilizado para anexar os papéis às portas previamente definidas e associadas aos componentes, o qual é representado pelo trecho de código entre as linhas 5-7.



```

System simple_client_server = {
  Component client = {Port send-request}
  Component server = {Port receive-request}
  Connector rpc = {Roles {caller, callee}}
  Attachments: {
    client.send-request to rpc.caller;
    server.receive-request to rpc.callee}
}

```

Figura 14 – Linguagem de descrição de arquitetura ACME.

Fonte: ACME (2016).

Diante do exposto, observa-se que LDAs apresentam algumas vantagens e desvantagens. Uma primeira vantagem que poderia ser citada trata-se da maneira legível e formal de representar a arquitetura. Sendo assim, é possível ter um maior detalhamento com as descrições arquiteturais geradas, facilitando a avaliação da arquitetura, no que se refere aos atributos de qualidades (como, por exemplo, completude, consistência, ambiguidade, entre outros). Uma segunda vantagem seria que arquitetos podem fazer uso dessas especificações para agilizar o desenvolvimento do sistema. Para isso, eles poderiam, por exemplo, aplicar cadeias sucessivas de transformações para gerar o código da aplicação.

Por outro lado, algumas desvantagens também podem ser

observadas ao utilizar uma LDA. Primeira, não há um consenso nem na academia, nem na indústria, sobre o que uma LDA deve, de fato, representar, sobretudo no que diz respeito ao comportamento da arquitetura. Segunda, as notações das linguagens atuais são relativamente difíceis e não são suportadas pelas ferramentas de modelagem comerciais. Uma terceira desvantagem seria que a propensão a erros e o custo para representar arquiteturas complexas, devido à ausência de ferramentas de suporte robustas que auxiliem na prática a representação dos elementos da arquitetura.

Por fim, pode-se concluir que uma linguagem de descrição arquitetural é mais uma alternativa para representar os elementos que formam uma arquitetura. Quando utilizada corretamente, torna a descrição de arquitetura mais formal e legível. Se uma LDA for utilizada com ferramentas de suporte inadequadas, pode tornar a especificação da arquitetura ainda mais complexa, comprometendo, por exemplo, a compreensibilidade das decisões arquiteturais tomadas. Embora algumas LDAs tenham sido propostas ao longo dos anos, nenhuma delas tem sido adotada como padrão, nem sido amplamente utilizada na indústria.

CAPÍTULO 5

Aspectos e web semântica

Este capítulo possui como objetivo apresentar uma visão geral sobre os conceitos encontrados em arquiteturas orientadas a aspectos. Para isso, os conceitos básicos de programação orientada a aspectos são introduzidos, tais como ponto de junção, ponto de corte, aspecto, adendo, entre outros. Destaca-se também a motivação para o uso de programação orientada a aspectos, bem como exemplos são utilizados para introduzir os conceitos. Além disso, são introduzidos alguns conceitos de Web semântica.

Como introdução aos temas propostos neste capítulo, destaca-se a limitação da programação orientada a objetos de modularizar requisitos funcionais e não funcionais em sistemas com arquiteturas complexas. Frequentemente esses requisitos são projetados e implementados de tal forma que seus códigos ficam entrelaçados e espalhados através de vários módulos do sistema. Consequentemente, tais sistemas, juntamente com suas respectivas arquiteturas, tornam-se difíceis de evoluir e manter. Diante desse cenário, alguns mecanismos de modularização são apresentados, a partir da perspectiva de programação orientada a aspectos. Outro tema que também será discutido neste Capítulo trata-se da Web semântica. Tópico amplamente discutido e visto como estratégico para viabilizar a transformação da Web atual (dos documentos) em uma Web semântica (dos dados).

5.1 Arquiteturas orientadas a aspectos

O conceito de Programação Orientada a Aspectos (POA) foi criado por Gregor Kiczales e a sua equipe na Xerox PARC, a divisão de pesquisa da Xerox. Em particular, ela surgiu a partir do trabalho Kiczales (1997), publicado em 1997 na conferência ECOOP. Kiczales propôs POA para lidar com uma preocupação central ao projetar e implementar

um software, bem como sua arquitetura: separação de interesse. De acordo com Sommerville (2007), a separação de interesse é um princípio de projeto que deve ser levado em consideração ao projetar uma arquitetura de software. Na essência, projetar uma arquitetura respeitando esse princípio significa organizá-la de modo que cada módulo possua uma única responsabilidade.

Em particular, um módulo pode ser um componente, classe, método, procedimento. O importante é que cada um, independente do nível de abstração, trata de uma única responsabilidade, considerando o seu nível de abstração. Isso facilita, entre outras coisas, a compreensão da arquitetura, pois é possível entender seus módulos apenas tendo conhecimento da responsabilidade do respectivo módulo, sem precisar entender outros elementos. Dessa forma, a manutenção do sistema também será favorecida, pois quando existe a necessidade de mudanças, elas serão realizadas em um número pequeno de unidades de código, bem como será evitada a propagação de alterações indesejadas.

O conceito de separação de interesse não é um tema recente, estando presente desde o início da história da computação. Por volta da década de 1950, sub-rotinas foram inventadas para modularizar unidade (ou parte) de uma funcionalidade. Sendo assim, uma funcionalidade poderia ser executada tendo acesso às sub-rotinas que a implementavam. Desde então, novos conceitos surgiram para melhorar a separação de interesses como, por exemplo, classes de objetos, métodos, componentes, entre outros.

Embora seja senso comum que a separação de interesse deve estar presente em todo projeto arquitetural, um desafio é definir claramente o que seria um interesse. Algumas definições adotam o conceito de interesse como sendo um elemento estritamente funcional do sistema. Outras adotam uma definição mais abrangente, entendendo um interesse como sendo um objetivo que o sistema deve atingir. De acordo com Sommerville (2007), todas as tentativas de associar interesse ao software serão tentativas erradas. De acordo com Jacobsen e Ng (2004), interesses são entendimentos sobre os requisitos que o sistema deve ter, bem como as restrições e as prioridades impostas pelos usuários do sistema.

Um entendimento coerente com as definições apresentadas seria um interesse como sendo um requisito funcional e não funcional do

sistema. Seguindo essa linha de raciocínio, um interesse, portanto, pode ser brevemente definido como sendo um “requisito de um (ou mais) usuário que deve ser atendido para satisfazer o objetivo geral do sistema”.

Em particular, a Programação Orientada a Objetos (POO) tem sido amplamente adotada e utilizada em sistemas simples e complexos. Porém, apesar dos seus benefícios, ela apresenta uma incapacidade de modularizar interesses transversais, aqueles que entrecortam vários módulos do sistema para que possam ser implementados. Se tais interesses não forem modularizados, eles se espalharão em vários módulos, para que a sua implementação seja viabilizada.

Diante de um contexto de desenvolvimento de sistemas de software cada vez mais complexos, que precisam ser projetados e desenvolvidos em um espaço de tempo curto e com baixo custo, o novo desafio seria como modularizar interesses transversais que acabam afetando atributos de qualidade. Exemplos desses atributos seriam confiabilidade, reusabilidade, flexibilidade, testabilidade, entre outros. Diante desse desafio, Kiczales (1997) propôs a programação orientada a aspectos para mitigar essa limitação de modularização de interesses transversais encontrada em programação orientada a objetos. Desse modo, a programação orientada a aspectos estende os conceitos de orientação a objetos para tornar possível a modularização de interesses transversais.

Para um sistema bancário, por exemplo, os interesses centrais do sistema são aqueles ligados diretamente ao seu propósito primário. Portanto, para um sistema bancário, os interesses centrais seriam (SOARES, 2010) gestão dos clientes, gerenciamento de empréstimos, gerenciamento de conta, acesso a dados, taxas administrativas, cobrança e relatórios, entre outros. Além desses interesses primários, sistemas bancários de grande porte podem ter funcionalidades secundárias, tais como *logging*, segurança, tratamento de exceções, autenticação, desempenho, concorrência, entre outros. Essas funcionalidades secundárias podem ser enquadradas como requisitos não funcionais, pois os mesmos podem impor restrições sobre como o sistema deve funcionar, entre outras coisas. O desafio é que, para implementar tais funcionalidades secundárias (ou requisitos não funcionais), tipicamente eles compartilham informações com as funcionalidades centrais do sistema. Logo, tem-se requisitos não funcionais que se

relacionam com requisitos funcionais.

Para ilustrar esse relacionamento, a Figura 15 apresenta um diagrama de classes abstrato com a manifestação de um interesse transversal. Isto é, um requisito não funcional entrecortando requisitos funcionais. Cada cor das classes representa um requisito que o sistema deve implementar. Por exemplo, as classes com a cor verde representam o módulo de gerenciamento de empréstimo, enquanto as classes com a cor azul representam o módulo de gerenciamento de conta de usuário. Além disso, percebe-se também uma classe com a cor azul clara, a qual faz parte de um módulo genérico de gerenciamento de transações bancárias. Nos três casos, percebe-se que esses interesses que estão projetados no diagrama de classes da UML estão modularizados. Logo, há uma grande probabilidade que tais interesses também assim permaneçam no código. No entanto, as classes com a cor vermelha representam o módulo responsável pela implementação do requisito de *logging*. Nesse momento, é importante que o leitor perceba a necessidade do espalhamento desse interesse nas classes do módulo de gerenciamento de empréstimo e nas classes do módulo de gerenciamento de conta de usuário. Esse espalhamento é um sintoma da presença de um interesse transversal, o qual entrecorta vários módulos do sistema para que possa ser projetado e implementado.

Esse exemplo mostra de forma ilustrativa a falta de abstrações em OO para projetar interesses transversais de uma maneira modularizada. Pelo contrário, usando os conceitos de orientação, objeto, arquitetos e desenvolvedores são forçados a espalhar o interesse de *logging* através de vários módulos, para que o mesmo possa ser implementado. Portanto, um interesse passa a se caracterizar como transversal quando pode influenciar ou afetar a implementação de outros interesses, sejam eles requisitos funcionais ou não funcionais.

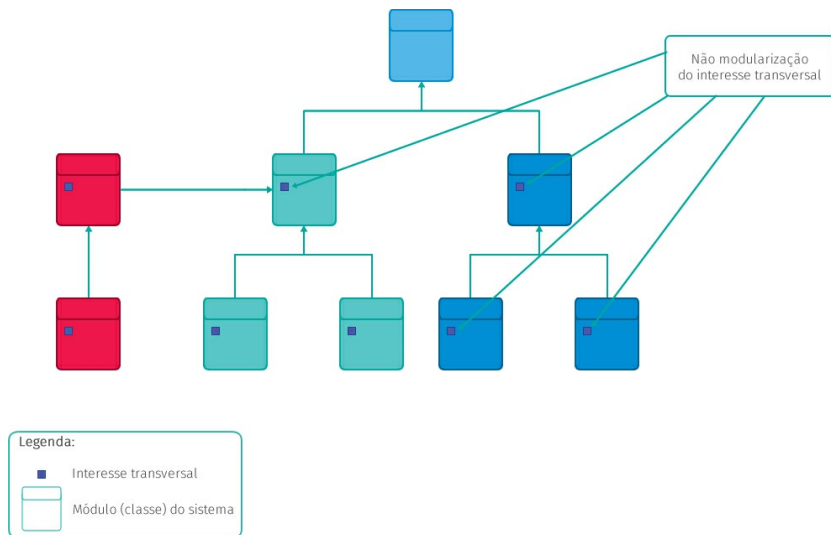


Figura 15 – Ilustração de um interesse transversal.

Fonte: adaptada pelo autor de Soares (2010).

De acordo com Sommerville (2007), os conceitos encontrados em programação orientada a objetos são normalmente efetivos para o projeto e a implementação de interesses não transversais. Porém, a implementação de requisitos funcionais exige também a implementação de requisitos não funcionais. Logo, código de requisitos funcionais passa a ter código de requisitos não funcionais. Isso conduz para dois fenômenos indesejáveis: entrelaçamento (do inglês, *tangling*) e espalhamento (do inglês, *scattering*). A Figura 16 apresenta um exemplo ilustrativo desses fenômenos.

O entrelaçamento acontece quando um módulo de um sistema passa a ter código que implementa diferentes requisitos dos sistemas. Tipicamente, esse módulo passa a não respeitar o princípio da responsabilidade única (MARTIN, 2002), visto que ele passa a implementar mais de uma responsabilidade. Observando a Figura 16 em questão, observa-se que três interesses são implementados por três módulos do sistema. Os interesses são de segurança (azul), lógica do negócio (verde) e gerenciamento de transação (amarelo). O módulo mais à frente na imagem possui código dos três interesses.

Logo, tem-se um exemplo de entrelaçamento de interesses. Sendo assim, ao ter uma manutenção no requisito de segurança neste módulo, um desenvolvedor precisará também entender sobre os requisitos de gerenciamento de transação e de lógica do negócio. Essa problemática também é percebida nos outros dois módulos do sistema.

Considerando o segundo fenômeno, o espalhamento ocorre quando a implementação de um interesse se espalha através de vários módulos do sistema. Revisitando a Figura 16, percebe-se que a implementação dos três requisitos está espalhada através dos três módulos do sistema. Os problemas com entrelaçamento e espalhamento acontecem quando os requisitos centrais do sistema mudam, ou mesmo quando os interesses transversais precisam passar por manutenção. As mudanças a serem feitas não estarão localizadas em um único lugar. Pelo contrário, será necessário varrer todos os módulos do sistema para verificar quais módulos precisam de modificações. Em sistemas complexos e de grande porte, essa busca pode ser onerosa e propensa a erros, pois há sempre a chance de esquecer de alterar algum módulo que deveria ser alterado. Sommerville (2007) também reforça que, quando várias mudanças precisam ser feitas, isso aumenta as chances de que alguma alteração seja feita de forma indesejada, introduzindo defeitos no módulo.

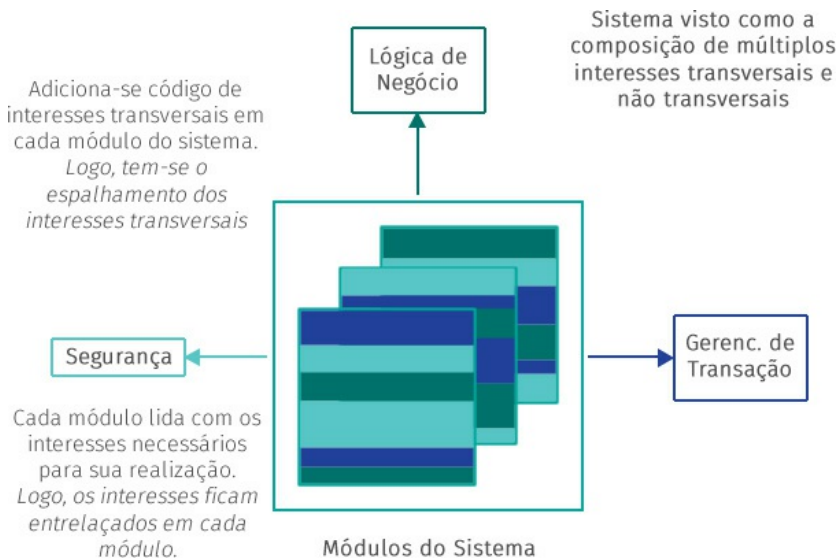


Figura 16 – Exemplo ilustrativo de entrelaçamento e espalhamento.
Fonte: elaborada pelo autor, com base em Laddad (2009).

Diante do exposto, programação orientada a aspectos visa a modularização de interesses transversais ao permitir a separação dos mesmos. Para isso, uma nova unidade de modularização e abstração é proposta: *aspecto*. Para que a modularização de interesses transversais se torne viável, outros conceitos também são propostos: ponto de corte (*point cut*), ponto de junção (*join point*), adendo (*advice*) e *inter-type declarations*.

Para ilustrar tais conceitos, a Figura 17 apresenta um exemplo de código, no qual se deseja modularizar o interesse transversal de *logging*. Suponha que o trecho de código em vermelho (referente ao *logging*), esteja presente em todas as classes do sistema bancário. O primeiro passo é identificar os pontos de interesse na execução do sistema, também conhecido como *ponto de junção*. Nesse exemplo, o ponto de interesse se restringirá à execução de crédito ou débito em qualquer conta do sistema bancário. Exemplos de pontos de junção seriam, por exemplo, chamadas de métodos, acesso a atributos (escrita e leitura), chamadas a construtores, entre outros. O interesse transversal de *logging* em questão acontece em todas as chamadas

para os métodos debitar e creditar. O objetivo é remover esses trechos de código de *logging* para modularizar tais trechos em um aspecto.

```
public class Conta {  
    private String numero;  
    private double saldo;  
    ...  
    public void debitar (double valor) {  
        if (this.getSaldo() >= valor){  
            this.setSaldo(this.getSaldo() -valor);  
            System.out.println("ocorreu um debito!");  
        }  
    }  
    public void creditar (double valor) {  
        this.setSaldo(this.getSaldo() +valor);  
        System.out.println("ocorreu um credito!");  
    }  
}
```

Figura 17 – Exemplo de código com interesse transversal: logging.

Fonte: Soares (2010).

Para agrupar esses pontos de interesse, programação orientada a aspectos faz uso de ponto de corte (*pointcut*), o qual possui os mecanismos de capturar a ativação desses pontos de interesse da execução do sistema. Sendo assim, um *pointcut* é capaz de definir um conjunto de pontos de junção.

Os conceitos de POA aqui apresentados serão exemplificados em AspectJ (2016) e Laddad (2009). Trata-se de uma linguagem que estende Java com os conceitos de programação orientada a aspectos. A Figura 18 apresenta um exemplo de ponto de corte (*pointcut*). Nesse exemplo, o termo *pointcut* é uma palavra reservada para denominar o ponto de corte. *LogCredito* é o nome do ponto de corte; *call* é um designador utilizado para capturar os pontos de interesse na execução do código. Esse *pointcut* captura todas as chamadas

para o método creditar com parâmetro do tipo *double*, com qualquer tipo de retorno (*), das classes que tenham o nome começando com a *string* Conta. O segundo *pointcut*, por sua vez, captura todas as chamadas para o método debitar com parâmetro do tipo *double*, com qualquer tipo de retorno (*), das classes que tenham o nome começando com a *string* Conta.

```
pointcut logCredito();  
call (* Conta*.creditar(double));
```

```
pointcut logCredito();  
call (* Conta*.debitar(double));
```

Figura 18 – Exemplo de *pointcut*.

Fonte: Soares (2010).

Uma vez definidos os *pointcuts*, o próximo passo é especificar o código que será executado quando os pontos de execução contemplados pelos *pointcuts* forem identificados. Para isso, programação orientada a aspectos propõe o conceito de adendo (*Advice*). Em outras palavras, trata-se do código que será executado quando um determinado *pointcut* for detectado. Ele terá os trechos de código que se espalham pelos módulos do sistema; dessa forma, permitindo a modularização dos mesmos. Considerando o exemplo, é preciso implementar um adendo que modularize os códigos de *logging* exibidos na Figura 17. A Figura 19 apresenta um exemplo de adendo. Inicialmente, é definido quando o adendo será incorporado na classe que é capturada pelo *pointcut* *logCredito()*. Para isso, o adendo é iniciado com a palavra reservada *after* para informar que o adendo deve ser incorporado depois da execução do ponto de interesse capturado pelo *pointcut* *logCredito*. Sendo assim, o trecho de código “System.out.println(“ocorreu um credito”)” será executado após a chamada do método *creditar*, com parâmetro do tipo *double* com qualquer tipo de retorno (*) das classes que comecem com a *string* Conta.

```
pointcut logCredito();  
    call (* Conta*.creditar(double));  
  
after(): logCredito(){  
    System.out.println("ocorreu um credito");  
}
```

Figura 19 – Exemplo de adendo (*advice*).

Fonte: Soares (2010).

Por fim, o aspecto responsável pela modularização do interesse transversal de *logging* encontrado na classe *Conta* é mostrado na Figura 20. Como pode ser observado, o aspecto *LogContas* possui dois pontos de corte (*logCredito* e *logDebito*) e dois adendos que utilizam esses dois pontos de corte.

```

public aspect LogContas {

    pointcut logCredito();
        call (* Conta.creditar(double));

    pointcut logDebito();
        call (* Conta.debitar(double));

    after(): logCredito(){
        System.out.println("ocorreu um credito");
    }

    after () returning: logDebito(){
        System.out.println("ocorreu um debito");
    }

}

```

Figura 20 – Código do aspecto utilizado para modularizar o interesse transversal da classe Conta.
Fonte: Soares (2010).

Outro exemplo é apresentado para descrever mais detalhes sobre os recursos que AspectJ possui. A Figura 21 apresenta os principais elementos encontrados em um aspecto, os quais são discutidos a seguir:

- Linha 1: aspecto projetado para capturar e manipular falhas nos objetos de uma classe *Server*. *FaultHandler* possui comportamento que deve ser disparado sempre que a chamada de um método público de um objeto do tipo *Server* for realizada. O servidor pode estar inabilitado para responder a uma requisição por causa de alguma falha. Logo, as

chamadas para os métodos são interessantes eventos para realizar o tratamento das falhas. Se um objeto da classe *Server* não estiver ativo para responder a uma requisição, o aspecto irá atuar.

- Linha 3: declara que cada *Server* tem uma variável do tipo *boolean* nomeada de *disabled*, inicializada com *false*. *Private* indica que apenas código no aspecto pode ter acesso a esta variável.
- Linha 5-11: são métodos iguais aos encontrados em classes Java.
- Linha 13: captura a execução do programa quando os objetos do tipo *Server* têm seus métodos públicos chamados. Permite a quem utilizá-lo acessar o objeto *Server*, cujo método está sendo chamado. O parâmetro do *pointcut* expõe o contexto: *Server s*. O *target(s)* indica o “alvo” do qual serão capturados os eventos. A *&&* indica composição, operado lógico.
- Linha 15-17: uma vez que o *pointcut services(s)* tenha sido ativado, é verificado se o servidor encontra-se desabilitado (linha 16). Caso positivo, é lançada uma exceção nomeada de *DisabledException*.
- Linha 19-22: uma vez que o *pointcut services(s)* e uma exceção foi lançada, então o servidor é habilitado (linha 20) e a falha é reportada (linha 21).

1	aspect FaultHandler {	
2		
3	private boolean Server.disabled = false;	Inter-type field
4		
5	private void reportFault() {	
6	System.out.println("Failure! Please fix it.");	Method
7	}	
8		
9	public static void fixServer(Server s) {	
10	s.disabled = false;	Method
11	}	
12		
13	pointcut services(Server s): target(s) && call(public *(..));	Pointcut
14		
15	before(Server s): service(s) {	
16	if (s.disabled) throw new DisabledException();	Advice
17	}	
18		
19	after(Server s) throwing (FaultException e): service(s) {	
20	s.disabled = true;	Advice
21	reportFault();	
22	}	
23	}	

Figura 21 – Anatomia de um aspecto.
 Fonte: adaptada pelo autor de AspectJ (2016).

Diante do exposto, os principais conceitos encontrados em programação orientada a aspectos são os seguintes: *adendo*, possui o código que será empacotado ou inserido nas classes onde o aspecto irá atuar; *aspecto*, a nova abstração introduzida para modularizar interesses transversais; *ponto de junção*, ponto de interesse na execução de um programa; e *ponto de corte*, declaração que define os pontos de junção do sistema que devem ser capturados. Por fim, algumas considerações finais (SOMMERVILLE, 2007):

- A separação de interesses pode ser considerada como o principal benefício da abordagem orientada a aspectos para projeto de arquitetura de software. Com a modularização de interesses transversais em aspectos, tais interesses podem ser compreendidos, reusados e modificados independentemente.
- O entrelaçamento dos interesses (ou responsabilidades dos módulos) acontece quando um módulo em um sistema possui código que implementa mais de uma responsabilidade, ou

interesse. Por outro lado, o espalhamento acontece quando a implementação de um interesse se espalha por vários módulos do sistema.

- Aspectos possuem *pointcut*, o qual representa uma declaração precisa sobre onde o aspecto deve atuar. O *pointcut* é representado através de uma linguagem de *pointcut* que expressa possíveis pontos de junção.
- Utilizando programação orientada a aspectos ao projetar uma arquitetura, é possível ter o suporte de separação de interesses transversais (no aspecto), bem como a implementação dos requisitos centrais ao sistema, feito pelo código da aplicação sem utilizar aspectos.
- A definição de requisitos pode ser feita através de uma especificação de casos de uso, a qual representa o primeiro artefato onde indícios de interesses transversais podem ser detectados. Uma vez detectado, os interesses transversais poderão ser representados em modelos UML, utilizando extensão para representar a semântica dos conceitos encontrados em programação orientada a aspectos.
- Problemas com uso de programação orientada a aspectos em projetos arquiteturais estão relacionados, por exemplo, com a inspeção e derivação de testes. Essa problemática ainda é uma barreira significativa para a adoção de programação orientada a aspectos em projetos arquiteturais de grande porte.

Por fim, um risco que se deve considerar ao utilizar programação orientada a aspectos, é o poder de expressão da linguagem de *pointcuts* encontrado em linguagens como AspectJ. Devido à sua alta expressividade, é possível ter um ponto de junção sendo capturado por mais de um *pointcut*. Logo, pode acontecer de dois ou mais aspectos atuarem ao mesmo tempo em um mesmo ponto de junção.

5.2 Web Semântica

O conceito de Web Semântica surgiu no artigo publicado por Tim Berners-Lee, James Hendler e Ora Lassila em Lee (2001). Os autores propõem e provocam uma nova compreensão da World Wide Web. Sendo assim, define-se Web Semântica como uma extensão da Web

atual, visando permitir que computadores e humanos trabalhem de forma cooperativa. Em termos práticos, ela visa atribuir semântica ao conteúdo disponibilizado na internet, de modo que seja compreendido tanto por humanos quanto por máquinas.

De acordo com W3C (2016), a Web Semântica pode ser vista como uma evolução da tradicional “Web de documentos” para a “Web dos dados”. O objetivo central em expandir para uma Web de dados é viabilizar que computadores possam realizar operações mais úteis, ao mesmo tempo que o desenvolvimento de sistemas possa oferecer suporte a interações na rede. Com esse entendimento, a Web Semântica tratará a Web como um grande repositório de dados, permitindo que pessoas sejam protagonistas nessa criação dos dados. Sendo assim, vocabulários e regras também serão criados para viabilizar a interoperabilidade entre os dados.

Alguns dos desafios enfrentados atualmente seriam o projeto e desenvolvimento de tecnologias, ferramentas e linguagens que permitam que os dados disponíveis na Web possam ser definidos por pessoas, ligados (ou associados), bem como compreendidos por máquina. Exemplos de tecnologias que têm sido desenvolvidas para superar tais desafios seriam XML, RDF, SPARQL, OWL, SKOS, Ontologias, agentes computacionais, entre outros.

O que se pretende, de fato, é criar uma plataforma de dados que viabilize estender a atual Web de tal forma que permita compartilhar os documentos e seus dados, bem como reutilizar os dados, processar os dados automaticamente tanto por ferramentas quanto manualmente, e revelar novos relacionamentos possíveis entre porções de dados. Isso permitirá o desenvolvimento de novas funcionalidades a serem oferecidas aos usuários, gerando novas experiências.

Para isso, busca-se integrar vários (muitas vezes heterogêneos) “depósitos de dados” numa plataforma coerente de dados ligados (ou linkados). Como mencionado anteriormente, tendo esses dados ligados, será possível gerar novas experiências para os usuários, ao disponibilizar melhores serviços e produtos provenientes das melhorias da representação do conteúdo.

Um questionamento pertinente seria como, de fato, isso irá acontecer, uma vez que os dados comecem a ser ligados (ou

linkados). De acordo com a visão de Lee (2001), “ilhas de conhecimento” serão formadas, representando conhecimentos associados às aplicações específicas. Essas ilhas serão integradas usando tecnologias de representação de conhecimento como, por exemplo, ontologias.

Uma das tecnologias já amplamente utilizada nesse contexto é a linguagem XML (*eXtensible Markup Language*), uma linguagem de marcação de dados usada para descrever dados estruturados. *Tags* são usadas para delimitar trechos de dados e atributos para caracterizar tais dados (“nome” = “valor”). A Figura 22 mostra um exemplo de código XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Figura 22 – Exemplo de código em XML

Fonte: W3S (2016).

Algumas observações interessantes sobre XML: (1) trata-se de um padrão recomendado pela W3C para armazenar e transportar dados; (2) o foco dos arquivos XML está na representação da informação, deixando para a aplicação a responsabilidade de interpretá-la; (3) embora tenha sido projetada para ter seus arquivos autodescritivos e autocontidos, eles são bastante verbosos, não sendo recomendados para leitura por humanos; e (4) busca simplificar o compartilhamento de dados, o transporte dos dados, a troca de dados entre aplicações em diferentes plataformas, promovendo a interoperabilidade e a disponibilização dos dados. Por outro lado, linguagens semelhantes, como, por exemplo, o HTML, permitem uma melhor leitura dos dados. Isto é, XML fornece uma representação estruturada dos dados, o que torna fácil o desenvolvimento de aplicações que a utilizem. Destaca-se que, enquanto os arquivos XML foram projetados para armazenar e transferir dados, os arquivos

HTML foram projetados para exibir tais dados, ou seja, o foco é na representação dos dados. A Figura 23 apresenta um exemplo de dados em código XML sendo exibidos em HTML.

XML	HTML
<pre><note> <to>Tove</to> <from>Jani</from> <heading>Reminder</heading> <body>Don't forget me this weekend!</body> </note></pre>	<div>Note</div> <div>To: Tove From: Jani</div> <div>Reminder</div> <div>Don't forget me this weekend!</div>

Figura 23 – Exemplo de dados em XML sendo exibidos em HTML
Fonte: adaptada pelo autor de W3S (2016).

O XML define sua estrutura através de um DTD (*Document Type Definitions*). Sendo assim, para que um arquivo XML possa ser considerado válido é necessário que o arquivo DTD referenciado seja respeitado. Se as regras no DTD não forem respeitadas, então o arquivo XML será considerado inválido. O uso de DTD permite que equipes independentes possam trabalhar em paralelo, ao mesmo tempo que um padrão de intercâmbio de dados seja estabelecido. Dessa forma, é possível, por exemplo, verificar se os dados recebidos de terceiros são válidos ou não. A Figura 24 mostra um exemplo de código XML e seu respectivo DTD. Os elementos encontrados no arquivo DTD são os seguintes:

- *!ELEMENT heading*: define o elemento *heading* sendo do tipo “#PCDATA”.
- *!ELEMENT body*: define o elemento *body* sendo do tipo “#PCDATA”.
- *#PCDATA*: significa que o mesmo pode ser manipulado por um *parse*.

Exemplo de XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

Exemplo de DTD

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

Figura 24 – Exemplo de código XML e DTD.
Fonte: adaptada pelo autor de W3S (2016).

Outra tecnologia utilizada é o RDF (*Resource Description Framework*), o qual se trata de um modelo padrão para o intercâmbio de dados na Web (W3C, 2016). RDF tem características que facilitam a integração de dados, mesmo se os esquemas sejam diferentes, e suporta a evolução dos esquemas, sem a necessidade que todos os dados dos clientes sejam alterados. Outra característica é a descrição de recursos através de declarações (RDF, 2016).

RDF propõe identificar “coisas” usando identificadores, e descreve recursos através da definição de “propriedade = valor”. Então, tem-se três conceitos centrais: recurso, propriedade e o valor da propriedade. Recurso é qualquer coisa que possa ter um URI, tal como <http://www.w3schools.com/rdf>. Propriedade é um recurso que possui um nome, tais como “autor”, “título” e “nota”. Valor de propriedade é o valor atribuído a uma propriedade definida como, por exemplo, “Machado de Assis” ou “<http://www.w3schools.com>”. Dessa forma, RDF pode ser utilizado, por exemplo, para descrever as propriedades dos itens de um *e-commerce*, tais como preço e disponibilidade, descrever os itens de uma agenda, descrever a informação de páginas web (data, conteúdo, autor, data de alteração etc.), bem como descrever conteúdo, de tal forma que máquinas de busca possam processar o conteúdo.

Outro conceito encontrado no RDF para viabilizar a representação de dados consiste no “*statement*”, o qual consiste da combinação de um recurso, de uma propriedade, e de um valor de uma propriedade. Também conhecido como *subject*, *predicate* e *object*. Exemplo de um RDF *statement* seria (RDFa, 2016): “O site da Unisinos é <http://www.unisinos.br>”, onde o *Subject* seria Unisinos, o *Predicate* seria site, e *Object*: <http://www.unisinos.br>.

No caso da próxima figura (Figura 25), observa-se que uma tabela contendo dados é apresentada, um elemento normalmente associado à representação e estruturação de dados. Nesse exemplo, em particular, tem-se uma tabela com seis colunas (*Title*, *Artist*, *Country*, *Company*, *Price* e *Year*) e duas linhas. Além dos dados referentes à descrição das colunas, ela apresenta os dados das linhas. Dessa forma, esses dados também devem ter alguma forma de representação. Ela está, portanto, ao mesmo tempo descrevendo características (nas colunas) dos dados e os respectivos valores associados a tais características. Esses elementos permitem que os dados sejam adequadamente representados. Dado que uma especificação RDF pode ser mal especificada, é disponibilizado um validador de RDF, o que pode ser encontrado em Prud (2016).

Title	Artist	Country	Company	Price	Year
Empire burlesque	Bob Dylan	USA	Columbia	10.90	1985
Hide your hearth	Bonnie Tyler	UK	CBS Records	9.90	1988

```
<? xml version = "1.0"?>
<rdf: RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:cd="http://www.recshop.fake/cd#">

<rdf:Description
rdf:about="http://www.recshop.fake/cd/Empire Burlesque">
  <cd:artist>Bob Dylan</cd:artist>
  <cd:country>USA</cd:country>
  <cd:company>Columbia</cd:company>
  <cd:price>10.90</cd:price>
  <cd:year>1985</cd:year>
</rdf:Description>

<rdf:Description
rdf:about="http://www.recshop.fake/cd/Hide your hearth">
  <cd:artist>Bonnie Tyler</cd:artist>
  <cd:country>UK</cd:country>
  <cd:company>CBS Records</cd:company>
  <cd:price>9.90</cd:price>
  <cd:year>1988</cd:year>
</rdf:Description>
</rdf:RDF>
```

Figura 25 – Exemplo de código XML referente ao RDF.
Fonte: adaptada pelo autor de RDFa (2016).

CAPÍTULO 6

Web services e SOA

Neste capítulo serão destacados conceitos gerais relacionados com arquiteturas orientadas a serviços e web services, bem como definições necessárias para o desenvolvimento de atividades práticas nessa área, tais como análise de aspectos gerais de web services, discussão sobre os SOAP e RESTful Web services, arquitetura conceitual de um sistema orientado a serviços, elementos de uma mensagem SOAP, e exemplo de arquitetura usando barramento de serviços.

Um dos aspectos iniciais a serem tratados refere-se à contextualização do surgimento da arquitetura orientada a serviços e de *web services*. Inicialmente por razões de segurança e interoperabilidade, as organizações adotavam arquiteturas e modelos computacionais em nível organizacional, para processar as transações dos sistemas da empresa. Em geral, no passado, uma organização tinha uma série de servidores, um conjunto de processamentos (ou transações) a serem realizados e tinha que, de forma efetiva e sistemática, distribuir a carga de processamento entre os servidores. Como os servidores estavam na própria organização, era possível fazer uso de determinados padrões, adotar boas práticas e executar processos operacionais localmente. À medida que os sistemas organizacionais migraram para a Web, essa realidade mudou sensivelmente. Nesse novo cenário, a interoperabilidade e a flexibilidade dos sistemas são cruciais, o que tem exigido uma migração das arquiteturas dos sistemas organizacionais para uma abordagem orientada a serviços, tendo os *web services* como um dos componentes centrais para viabilizar essa nova realidade.

6.1 Web services

Com o advento da internet, os sistemas organizacionais passaram a ter novas demandas e necessidades de processamento. Nessa nova realidade, eles passaram a utilizar modelos emergentes de computação distribuída, que permitiram um modelo computacional interorganizacional distribuído ao contrário de intraorganizacional. Nesse novo modelo, os sistemas organizacionais passaram a requisitar a execução de processos que não mais se encontravam dentro dos limites da empresa (isto é, nos seus servidores). Os sistemas tiveram que assumir o papel de clientes de processos agora executados em servidores localizados fora dos limites da empresa. Essa necessidade de romper a fronteira organizacional, a qual os sistemas estavam limitados, tornou-se evidente e inevitável com a disseminação e ampla adoção de sistemas baseados na Web. Como consequência, essa nova realidade levou ao surgimento de novas tecnologias, visando dar suporte às demandas geradas, ao mesmo tempo que permitissem não só realizar uma mudança do modelo intraorganizacional para interorganizacional de computação, mas também garantissem a segurança e a interoperabilidade dos sistemas.

O desenvolvimento de sistemas empresariais baseados na Web gerou uma nova realidade caracterizada pelo acesso de computadores clientes aos servidores fora dos limites organizacionais, como anteriormente mencionado. Por sua vez, o cliente de um sistema Web tinha acesso às informações organizacionais através de arquivos HTML. Esse acesso, contudo, era limitado, visto que o mesmo era realizado via um navegador Web, ao mesmo tempo que o acesso às informações contidas em repositórios por outros sistemas não era prático (SOMMERVILLE, 2007). Consequentemente, a interoperabilidade entre sistemas Web ficou comprometida, visto que a conexão, bem como a comunicação, entre os servidores ficou impossibilitada. Por exemplo, a simples consulta de um catálogo de produto de um sistema Web não era feita de forma prática. Para contornar essa problemática, a academia e a indústria propuseram o conceito de *web services*, bem como arquitetura baseada em serviços.

De acordo com Sommerville (2007), em termos gerais, um *web service* pode ser definido como sendo uma representação padronizada de alguns recursos computacionais e de informações que podem ser usados por outros sistemas. Com esse entendimento, é possível, por exemplo, desenvolver um sistema que disponibilize um formulário

de cadastro de despesas e rendimentos para os usuários da aplicação. Através do conceito de *web service*, pode-se ter a verificação automática dos campos preenchidos do formulário através de serviços disponibilizados por outros sistemas como, por exemplo, validação de CPF (Cadastro de Pessoa Física).

Ao longo dos últimos anos, várias definições de *web service* têm surgido. Algumas delas são citadas a seguir. De acordo com Booth (2016), um serviço Web pode ser definido como um sistema de software projetado para suportar interação máquina-a-máquina através de uma rede, visando dar suporte à interoperabilidade. Essa definição traz e ressalta a importância da interoperabilidade que é viabilizada através do uso de *web service*. Em Oracle (2016), apresenta-se de uma forma sucinta que *web services* são aplicações cliente-servidor que se comunicam através da Web usando, por exemplo, o protocolo HTTP. Essa segunda definição ressalta que a comunicação entre os serviços é suportada por uma arquitetura cliente-servidor, na qual o protocolo HTTP dita a comunicação sendo feita através de requisições e respostas.

Em Erl (2005), o autor ressalta outros aspectos interessantes em relação aos *web services* ao destacar que são caracterizados pela sua grande interoperabilidade e extensibilidade, bem como suas descrições processáveis por máquina, graças ao uso de XML, JSON (*JavaScript Object Notation*), entre outros. O autor também reforça que eles devem ter uma interface descrita em um formato processável por máquina (por exemplo, WSDL (*Web Services Description Language*)). Desse modo, outros sistemas interagem com o serviço de acordo com a descrição usando, por exemplo, mensagens SOAP.

Então, a essência de um serviço é de que o fornecimento dos serviços deve ser independente da aplicação que usa o serviço (TURNER, 2003) (SOMMERVILLE, 2007). Com isso em mente, um provedor de serviço pode definir, implementar e publicar serviços genéricos ou especializados para oferecer a um conjunto de sistemas de software de diferentes organizações. Seguindo esta lógica, sistemas de software organizacionais poderão ser construídos pela composição de serviços disponibilizados pelos provedores de serviços.

Atualmente há vários modelos de serviços disponíveis (SOMMERVILLE, 2007). Exemplos desses modelos seriam JINI (KUMARAN, 2001), *web*

services (STAL, 2002) e *grid services* (FOSTER, 2002). Todos esses modelos são, na sua essência, similares ao modelo conceito de *web service* proposto por Kreger (2001). Todos esses modelos funcionam de acordo com a ilustração encontrada na Figura 26. Essa ilustração representa uma arquitetura conceitual de um sistema orientado a serviços. Portanto, essa arquitetura pode ser vista como uma generalização do modelo conceitual de serviços proposto por Kreger (2001). Observando a Figura 26 é possível verificar os elementos centrais na arquitetura de um sistema orientado a *web services*:

- Provedor de serviços. Esse elemento representa a entidade responsável pela criação do *web service*, a qual o descreve em um formato padrão, tendo detalhes sobre como o mesmo deve ser utilizado pelo Solicitador (ou Consumidor) de serviços.
- Solicitador (ou consumidor) de serviços. Esse elemento trata-se de qualquer entidade capaz de consumir o serviço disponibilizado por um provedor. Para utilizar o serviço, faz-se necessário estabelecer uma conexão seguindo o padrão estabelecido pelo provedor.
- Registrador de serviços. Esse elemento representa o local onde o provedor de serviço registra os serviços e o local onde o consumidor poderá buscar serviços.

Complementando a descrição acima, o provedor de serviços define uma interface através da qual o serviço será oferecido, bem como a sua implementação. Desse modo, um solicitante de serviços poderá vincular serviços, por exemplo, a uma aplicação Web após entender as descrições dos serviços ofertados. Na prática, para que o solicitante do serviço tenha, de fato, acesso ao comportamento implementado por um serviço, o mesmo deve incluir códigos na aplicação cliente para chamar e processar os resultados da chamada ao serviço. No entanto, para que esse uso seja realizado, o provedor de serviços deve registrar o serviço, o que implica disponibilizar informações sobre a sua interface, bem como sobre o que ele faz (SOMMERVILLE, 2007).



Figura 26 – Arquitetura conceitual de um sistema orientado a serviços.

Fonte: elaborada pelo autor, com base em Sommerville (2007).

De acordo com Sommerville (2007), as principais diferenças entre o modelo de serviço e a abordagem de objetos distribuídos encontrados em arquiteturas de sistemas distribuídos são as seguintes:

- Os serviços podem ser oferecidos por qualquer provedor de serviço, esteja ele dentro ou fora da organização. Assume-se que esses serviços estão de acordo com um padrão, logo as organizações podem criar sistemas através da composição de serviços disponibilizados por vários provedores. Por exemplo, um sistema empresarial pode ter acesso aos serviços disponibilizados pelos seus sistemas de seus fornecedores, pela empresa de logística para rastrear mercadorias, fazer consultas de cotação de moedas, bem como de condições climáticas.
- As informações sobre os serviços são tornadas públicas pelos seus provedores, visando que usuários autorizados possam acessá-los. Seguindo essa linha de raciocínio, não é necessário ocorrer uma negociação entre o provedor e os usuários do serviço a respeito da finalidade do serviço disponibilizado. Simplesmente o usuário analisa se o serviço disponível satisfaz as suas necessidades (ou não). Caso positivo, o mesmo pode ser incorporado ao sistema.
- Aplicações podem atrasar a vinculação dos serviços até que

eles sejam implantados ou executados. Portanto, uma aplicação que usa um serviço de consulta de preço, por exemplo, pode mudar dinamicamente os provedores de serviços enquanto o sistema estiver em execução.

- A criação de novos serviços é possível e prática. Um provedor de serviços pode reconhecer novos serviços que podem ser criados pela vinculação de serviços existentes de maneira prática.
- Uma vez que um serviço estiver disponível, é possível pagar pelo uso do serviço. Isso gera uma nova realidade, pois sistemas Web podem fazer uso de serviços sem ter a necessidade de comprar o componente que fornece aquele serviço de terceiros. Sendo assim, ao contrário de comprar um componente caro que será utilizado algumas vezes, a equipe de desenvolvimento pode decidir pelo uso do serviço externo, o qual será pago quando utilizado (isto é, por demanda).
- É possível adaptar as aplicações à medida que o ambiente passe por alterações, dando origem a aplicações reativas e adaptadas ao ambiente.

Para que a comunicação entre *web services* seja viável, alguns padrões têm sido propostos (SOMMERVILLE, 2007):

- SOAP (*Simple Object Access Protocol*). Protocolo padrão de troca de mensagens que dá suporte a comunicação entre serviços.
- WSDL (*Web Service Description Language*). Linguagem padrão de definição das interfaces dos serviços. Então, um provedor utilizará WSDL para representar as interfaces dos serviços ofertados pelo mesmo.
- UDDI (*Universal Description, Discovery and Integration*). Trata-se de um padrão para descrição, descoberta e integração universal de *web services*, o qual define os elementos de uma especificação de serviços que podem ser usados para descobrir a existência de um serviço. Esses elementos incluem informações sobre o provedor de serviços, o serviço fornecido, a localização da descrição de serviços (usualmente representada em WSDL) e informações sobre relacionamentos

de negócio.

No nível conceitual, um serviço é um componente de software fornecido através de um *endpoint* acessível pela rede. O cliente e o provedor do serviço usam mensagens para trocar requisições e respostas na forma de documentos autocontidos, os quais não fazem qualquer suposição sobre as tecnologias utilizadas no cliente. No nível técnico, um *web service* pode ser implementado de diversas formas. Os tipos de *web services* discutidos serão os SOAP e RESTful *Web services*.

Os SOAP *Web services* são tipicamente serviços que se baseiam em mensagens no formato XML, usam o padrão SOAP, e usam uma linguagem que define o formato e a arquitetura das mensagens trocadas. Os RESTful *Web services* não se baseiam em mensagens no formato XML, usam HTTP, não exigem uma linguagem de descrição dos *web services*. SOAP foi inicialmente projetado como um protocolo de acesso a objetos em 1998 por Don Box, Bob Atkinson e Mohsen Al-Ghosein na Microsoft, visando permitir a troca de informação em um ambiente distribuído (GUDGIN, 2007; SOAP, 2016). Usa XML para representar as informações e se baseia em protocolos da camada de aplicação, tais como HTTP e SMTP, para viabilizar a transmissão de mensagens.

Possui três características centrais: *extensibilidade*, *segurança* e *WS-roteamento* estão entre as extensões em desenvolvimento; *neutralidade*, SOAP pode operar sobre qualquer protocolo de transporte como HTTP, SMTP, TCP, UDP, ou JMS; e *independência*, SOAP permite qualquer modelo de programação. Exemplo de operação do SOAP:

- Uma aplicação pode enviar solicitação SOAP para um servidor que tem um *web service* que calcula o valor de um imposto (ex.: ICMS) para uma dada localidade.
- O servidor recebe a especificação da localidade e então retorna a resposta SOAP, um arquivo no formato XML como sendo o resultado.
- Desde que a resposta será fornecida em um formato que pode ser facilmente processado, a aplicação que requisita o cálculo do imposto poderá integrá-la facilmente à aplicação

requisitante.

O modelo de processamento de *web service* baseado em SOAP pode ser brevemente representado desta forma (GUDGIN, 2007; SOAP, 2016):

- SOAP *sender*: nó SOAP que é o responsável por originar e transmitir as mensagens SOAP.
- SOAP *receiver*: nó SOAP que recebe uma mensagem SOAP.
- SOAP *message path*: conjunto de nós SOAP através dos quais uma mensagem SOAP irá passar.
- SOAP *intermediary*: atua como um SOAP *receiver* e um SOAP *sender*. Recebe uma mensagem, processa o *header block* e repassa a mensagem em direção do SOAP *receiver final*.
- SOAP *receiver* (final): trata-se do destino da mensagem SOAP, é responsável por processar o conteúdo da mensagem.

Esse tipo de *web service* pode utilizar um conjunto diversificado de protocolos para viabilizar a troca de mensagens, como anteriormente mencionado. Por exemplo, SOAP pode utilizar os protocolos da camada de aplicação, tais como HTTP, HTTPS e SMTP.

Uma mensagem SOAP é formada por alguns elementos que são primordiais para viabilizar a troca de mensagens através de *web services*. Pode-se destacar quatro elementos. O primeiro é o *Envelope*, o qual consiste no mecanismo utilizado para identificar o arquivo XML que será transferido como sendo uma mensagem SOAP. O segundo é o *Header*, que se trata da parte da mensagem onde se encontra o cabeçalho, porém não é obrigatório ter conteúdo nesta parte para que uma mensagem seja formada. O próximo é o *Body*, o qual representa o corpo da mensagem, local onde o conteúdo propriamente dito da mensagem será inserido. Dessa forma, como não se justifica ter uma mensagem sem conteúdo, logo essa parte torna-se obrigatória. A quarta parte é a *Fault*, local onde informações sobre possíveis erros serão inseridas. Assim como o *Header*, esse elemento é opcional. Esses elementos de uma mensagem SOAP são resumidos na Tabela 2. Dando continuidade à explicação de uma mensagem SOAP, a Figura 27 apresenta um exemplo de um envelope SOAP. Nesse exemplo, é possível perceber os elementos que compõem o envelope.

POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock/Surya">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Figura 27 – Exemplo de um envelope SOAP.

Fonte: SOAP (2016).

Tabela 2 – Elementos de uma mensagem SOAP

Elemento	Descrição	Exigido
<i>Envelope</i>	Identifica o arquivo XML como sendo mensagem SOAP.	Sim
<i>Header</i>	Possui as informações do cabeçalho.	Não
<i>Body</i>	Possui as informações de chamada e resposta.	Sim
<i>Fault</i>	Fornecer informações sobre erros que aconteceram durante o processamento da mensagem.	Não

Fonte: elaborada pelo autor, com base em Gudgin (2007) e SOAP (2016).

Após discutir sobre o envelope SOAP, o próximo passo será apresentar um exemplo de SOAP *Web service*. Para isso, será utilizada a tecnologia JAX-WS¹ (*Java API for XML Web Services*), que é uma API para construir *web services* e clientes que se comunicam usando arquivos no formato XML. A JAX-WS permite que desenvolvedores não só escrevam *web services* orientados a mensagem, mas também implementem como eles serão acessados. Além disso, ela não é

restritiva e tem a vantagem da independência de plataforma da linguagem Java. Um cliente pode acessar um serviço que não esteja rodando na plataforma Java, e vice-versa. Essa flexibilidade é possível porque JAX-WS usa tecnologias, tais como HTTP, SOAP e WSDL. O projeto de SOAP *Web service* deve incluir um contrato, o qual será estabelecido para descrever a interface que o *web service* fornece. Para isso, WSDL pode ser usada para descrever os detalhes do contrato. O exemplo que será apresentado utilizará a tecnologia JAX-WS, a qual se trata de uma tecnologia Java para dar suporte à construção de SOAP *web services*, como mencionado.

A Figura 28 apresenta um exemplo de *web service* implementado utilizando a API Java JAX-WS (JENDROCK, 2014). Nesse exemplo, é apresentado um serviço simples que é capaz de somar dois números. Voltando ao código, foi criado um pacote nomeado de *org.me.calculator*, no qual a classe *CalculatorWS* foi inserida. Como o exemplo utiliza a API JAX-WS, a definição de um *Web service endpoint* é feito através do uso da anotação `@WebService`. Em outras palavras, associar a anotação `@WebService` à classe, tornará essa classe um *web service*.

O próximo passo é definir como essa classe irá disponibilizar um serviço (ou comportamento), o qual poderá ser acessado pelos clientes desse serviço. Para isso, a API JAX-WS também é bastante prática e simples, ao permitir que isso seja realizado através do uso de anotações. No exemplo, a classe *CalculatorWS* declara um método nomeado de *add* com a anotação `@WebMethod`. Essa segunda anotação irá expor o método para os clientes. Basicamente, o serviço declarado receberá dois números inteiros, os quais são representados pelos parâmetros *int i* e *int j*, e retornará o resultado da soma dos parâmetros. Vale salientar que a classe que recebe a anotação `@WebService` deve ter o construtor padrão público (JENDROCK, 2014).

```
package org.me.calculator;
```

```
import javax.ws.WebMethod;
```

```
import javax.ws.WebParam;
```

```
import javax.ws.WebService;
```

```
@WebService()
```

```
public class CalculatorWS {
```

```
    @WebMethod(operationName = "add")
```

```
    public int add(@WebParam(name = "i") int i, @WebParam(name = "j") int j) {  
        return i + j;  
    }
```

```
}
```

Figura 28 – Exemplo de SOAP Web service utilizando a API Java JAX-WS.

Fonte: elaborada pelo autor.

Recomenda-se que o leitor implemente o serviço utilizando, por exemplo, a IDE Netbeans² ou Eclipse.³ Após definir o serviço, o próximo passo é testá-lo. Para isso, é necessário executar a classe *CalculatorWS* e executar o testador de *web service* como mostra a Figura 29. O leitor deve inserir dois números para que seja possível processar a soma dos dois números pelo *web service*. Feito isso, o próximo passo é clicar no botão nomeado de *add*.

CalculatorWSService Web Service Tester

This form will allow you to test your web service implementation ([WSDL File](#))

To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.

Methods :

public abstract int org.me.calculator.CalculatorWS.add(int,int)

add (1 , 1)

Figura 29 – Testando o serviço.

Fonte: elaborada pelo autor.

Ao clicar, será gerada uma requisição SOAP que será enviada ao *web service*. A Figura 30 ilustra essa requisição SOAP gerada. Note que nessa requisição encontra-se o envelope SOAP, com os elementos apresentados anteriormente.

```
1      <?xml version="1.0" encoding="UTF-8"?>
2      <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
3          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
4          <SOAP-ENV:Header/>
5          <S:Body>
6              <ns2:add xmlns:ns2="http://calculator.me.org/">
7                  <i>1</i>
8                  <j>1</j>
9              </ns2:add>
10         </S:Body>
11     </S:Envelope>
```

Figura 30 – Envelope SOAP gerado na requisição.

Fonte: elaborada pelo autor.

Dando continuidade, o *web service* receberá a requisição, processará e mostrará como resultado o valor "2". A Figura 31 exibe o resultado do processamento pelo *web service*.

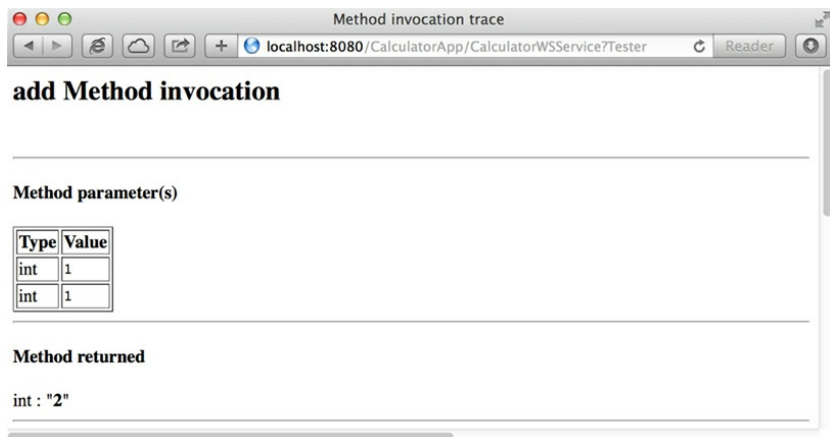


Figura 31 – Resultado retornado pelo serviço.

Fonte: elaborada pelo autor.

Assim como na geração da requisição, um envelope SOAP também é gerado no retorno do processamento do *web service*. Para o exemplo em questão, o envelope SOAP gerado é exibido na Figura 32. Outros exemplos e detalhes sobre a API Java JAX-WS podem ser encontrados em Jendrock (2014).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
3   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
4   <SOAP-ENV:Header/>
5   <S:Body>
6     <ns2:addResponse xmlns:ns2="http://calculator.me.org/"
7       <return>2</return>
8     </ns2:addResponse>
9   </S:Body>
10 </S:Envelope>
```

Figura 32 – Envelope SOAP gerado como resposta.

Fonte: elaborada pelo autor.

Após apresentar diferentes aspectos teóricos e práticos sobre *web service* usando SOAP, o próximo passo será discutir aspectos gerais sobre *web services* baseados em REST, também conhecidos como *RESTful Web services*.

REST (*Representational State Transfer*) trata-se de um estilo arquitetural proposto por Roy Fielding em 2000 na sua tese de doutorado na Universidade da Califórnia em Irvine (FIELDING, 2000). Na prática, fornece um estilo de aplicação cliente-servidor para a transferência de representações de recursos sobre o protocolo HTTP. Desse modo, os *RESTful Web services* são serviços construídos com base na arquitetura REST, a qual foi proposta por Fielding (2000). Um ponto central na arquitetura REST é tratar funcionalidades e dados como recursos, os quais serão tratados e disponibilizados através de um serviço. Um recurso pode ser qualquer informação que possa ser disponibilizada através de um serviço. Exemplos de recursos seriam o catálogo de produtos de uma empresa, dados pessoais, uma lista de tarefas de um projeto, entre outros.

Algumas características do estilo arquitetural REST são descritas a seguir. A primeira se refere a uma arquitetura baseada em cliente-servidor. REST adotou a abordagem cliente-servidor devido à

preocupação de viabilizar uma separação de interesses de forma sistemática, priorizando a modularização, o reuso e a flexibilidade. Ao separar as preocupações relacionadas à interface do usuário daquelas relacionadas ao processamento e armazenamento de dados, torna-se viável o aumento da portabilidade da interface do usuário através de múltiplas plataformas, bem como melhora a escalabilidade, ao simplificar os componentes do servidor. Recomenda-se que, ao projetar os componentes que serão responsáveis pela implementação de um determinado conjunto de serviços, deve-se dar prioridade à modularização dos mesmos, a fim de permitir que possam evoluir de forma independente.

Uma segunda característica interessante é que o estilo arquitetural REST adiciona uma restrição à interação cliente-servidor ao definir que a comunicação deve ser sem estado por natureza. Isto é, o servidor não deve manter o estado da aplicação. Dessa forma, cada requisição do cliente para o servidor deve conter todas as informações necessárias para que o servidor possa, de fato, compreender a solicitação feita. Além disso, ressalta-se que cada requisição não pode levar em conta qualquer informação de estado ou contexto armazenada no servidor. O estado da sessão é, portanto, mantido inteiramente no cliente. De acordo com Fielding (2000), essa restrição arquitetural tem como objetivo garantir três propriedades: visibilidade, confiabilidade e escalabilidade. Considerando a primeira propriedade, um sistema não terá que fazer referência para mais de um único ponto, a fim de determinar a natureza completa da requisição. A segunda, confiabilidade, também é melhorada por facilitar a tarefa de recuperação de falhas parciais. A escalabilidade também é melhorada, pois o estado entre as solicitações não precisa ser armazenado, o que permite que o servidor funcione de uma forma mais rápida. Além disso, o servidor não precisa gerenciar o uso de recursos ao longo das solicitações (FIELDING, 2000).

Esse estilo arquitetural sem manutenção do estado das requisições também apresenta desvantagens. Uma desvantagem pode estar relacionada ao desempenho da rede, visto que pode aumentar o número de dados repetitivos (isto é, sobrecarga por interação) enviados pela rede. Isso pode acontecer porque os dados não são armazenados no servidor em um contexto compartilhado. Ao recomendar que o estado da aplicação seja mantido no lado do cliente, e não no servidor, o estilo arquitetural REST busca reduzir o controle do servidor sobre a consistência do comportamento da

aplicação.

Outra característica importante do estilo arquitetural REST trata-se da recomendação do uso de *cache*. Com o objetivo de melhorar a eficiência da rede, adiciona-se o conceito de *cache* para formar um estilo arquitetural cliente-servidor, sem estado e com *cache*. Essa decisão exige que os dados de uma resposta a uma requisição sejam rotulados, implícita ou explicitamente, como *cacheable* ou não *cacheable* (FIELDING, 2000). Se os dados poderão ser armazenados, então os dados de resposta a uma requisição poderão ser reutilizados posteriormente, sempre que uma requisição equivalente for realizada

Fielding (2000) reforça que, ao adicionar *cache*, aumentam as chances de eliminar parcialmente ou completamente algumas interações, melhorar a eficiência, escalabilidade e desempenho. Isso pode ser percebido pelo usuário pela redução da latência média ao longo de uma série de interações (isto é, requisição e resposta). No entanto, o uso de *cache* pode também diminuir a confiabilidade, pois os dados em *cache* podem ser significativamente diferentes daqueles dados que poderiam ser obtidos via uma requisição direta ao servidor, e não à *cache*. Outros detalhes sobre o estilo arquitetural REST podem ser encontrados em Fielding (2000).

Baseado nessas características, os RESTful *Web services* são propostos e idealizados para fundamentalmente lidar com recursos disponibilizados em servidores. Sendo assim, permitem criar arquiteturas baseadas em serviços, flexíveis e com componentes reutilizáveis, ao mesmo tempo que permitem aos usuários acessar recursos, solicitar alterações e até mesmo solicitar a remoção dos mesmos. Para que isso seja de fato possível, várias tecnologias têm sido propostas que incorporam o estilo arquitetural REST. Um exemplo seria a API Java JAX-RS (JENDROCK, 2014), a qual é definida na JSR 339 com o objetivo de dar suporte aos desenvolvedores no momento da criação de RESTful *Web services* usando a linguagem de programação Java.

Como essa modalidade de *web services* é fortemente baseada no protocolo HTTP⁴ (*Hypertext Transfer Protocol*), o acesso aos recursos disponibilizados é realizado através de operações encontradas no protocolo HTTP, tais como GET, utilizada para buscar informações; PUT, utilizada para criar recursos e fazer alterações; POST, utilizada

para inserir alguma informação; e DELETE, utilizada para remover informações ou dados dos recursos. Ao utilizar essas operações, os usuários poderão manipular os recursos disponibilizados através de um serviço.

A próxima questão a ser discutida se refere à forma de representação desses recursos. Recordem que nos *SOAP Web services* os recursos eram predominantemente associados ao formato XML. No *RESTful Web services*, por sua vez, não há uma restrição associada à forma de representação do recurso. Esse é outro aspecto interessante nos *RESTful Web services* que merece destaque, porque os recursos disponibilizados através dos serviços não estão acoplados a uma forma de representação. Exemplos de formatos seriam JSON, XML, HTML, PDFs, arquivos JPG, entre outros.

Sendo assim, essa flexibilidade permite aos arquitetos e desenvolvedores escolherem o melhor formato de acordo com as restrições que eles tenham em mãos. Uma questão que poderia ser feita se refere às reais necessidades das diferentes formas de representação de um recurso. Na sua essência, isso permite facilitar a integração de diferentes tipos de clientes. Em outras palavras, é possível ter um recurso sendo utilizado por diferentes tipos de clientes. Exemplos de clientes seriam *web browsers*, *smartphones*, *tablets*, aplicativos *desktop*, entre outros. Seguindo essa linha de raciocínio, tem-se “um” recurso para “n” clientes. Portanto, um recurso pode ser representado em diferentes formatos.

Dito isso, o próximo passo é apresentar aspectos gerais de uma tecnologia amplamente utilizada para desenvolver *RESTful Web services*. Seguindo a linha de suporte da linguagem Java para *web services*, tem-se JAX-RS (JENDROCK, 2014), como mencionado anteriormente, trata-se da principal API Java para implementar *RESTful Web services*. Detalhes sobre a JAX-RS podem ser encontrados na documentação da plataforma Java *Enterprise Edition* em Jendrock (2014).

O principal objetivo da JAX-RS é simplificar o desenvolvimento de aplicações REST. Foi um dos primeiros *frameworks* a utilizarem classes POJO e anotações para publicar serviços RESTful. A versão JAX-RS 2.0 foi lançada com a especificação Java EE 7 baseada na JSR-339. Essa versão 2.0 traz algumas melhorias interessantes em relação às versões anteriores.

A primeira seria o suporte aprimorado à negociação de conteúdo. Essa melhoria trata exatamente do conceito trazido pela arquitetura REST de permitir a escolha da melhor forma de representação do conteúdo dos recursos disponibilizados, quando múltiplas formas de representação estão disponíveis. A segunda seria a *criação de uma API para o cliente*. Na versão JAX-RS 1.0, o cliente de um REST Web service era o responsável, por exemplo, por abrir conexões HTTP, sendo totalmente voltada para o lado do servidor. Já na JAX-RS 2.0, foram adicionados construtores para invocar um serviço a partir do cliente. Essa melhoria permite uma padronização na forma de acesso aos recursos.

Busca fornecer RESTful Web services com baixo acoplamento, leves que são particularmente adequados para a criação de APIs para diferentes tipos de clientes espalhados pela internet (JENDROCK, 2014). Ao se basear no estilo arquitetural REST, como já mencionado, JAX-RS provê uma arquitetura baseada cliente-servidor para a criação de serviço, o qual está centrado em torno da transferência de representações de recursos por meio de solicitações e respostas. Outro detalhe importante é que dados e funcionalidade são considerados como recursos e são acessados através de URI (*Uniform Resource Identifier*).

Os recursos são representados por documentos, os quais podem ser manipulados pelos clientes através de operações simples, bem definidas como, por exemplo, GET, PUT, POST e DELETE. Um exemplo de recurso REST seriam as condições meteorológicas atuais do tempo para uma cidade, ou mesmo o endereço completo associado a um CEP. Sendo assim, um cliente poderia acessar o recurso em diferentes formatos (por exemplo, XML, uma página HTML, um arquivo texto), modificar o recurso, atualizar seus dados ou mesmo excluir o recurso inteiramente. Note que esses detalhes de JAX-RS estão diretamente relacionados com as características do estilo arquitetural REST.

De acordo com Jendrock (2014), os seguintes princípios tornam as aplicações baseadas em JAX-RS simples, leves e rápidas:

- Identificação de recursos através de URI. Em RESTful Web service, os recursos são expostos e identificados através de URIs, os quais representam uma forma de endereçamento global para recursos e descoberta de serviços.

- Interface uniforme. Os recursos são manipulados usando um conjunto fixo de operações (GET, PUT, POST e DELETE). Enquanto PUT cria um novo recurso, DELETE permite, em seguida, excluir o recurso criado. GET recupera o estado atual de um recurso. POST transfere um novo estado para um recurso.
- Mensagens autodescritivas.⁵ Os recursos são dissociados da sua representação para que o seu conteúdo possa ser acessado em uma variedade de formatos. Além disso, metadados do recurso são também disponibilizados para permitir, por exemplo, o controle de *cache*, detectar erros de transmissão, negociar o formato de representação adequada e realizar a autenticação ou controle de acesso.
- Interações através de *links*. Cada interação com um recurso é sem a manutenção de estado e feita através de *links*. Ou seja, as mensagens de requisição e resposta são autocontidas. Para isso, várias técnicas para a troca de estado estão disponíveis. Dois exemplos seriam a URI *rewriting* e *cookies*.

Diante do exposto, uma comparação sobre os principais aspectos de SOAP e RESTful *Web services* é introduzida a seguir.

- Tamanho. Quanto ao tamanho, os REST *Web services* são considerados como leves, pois trafegam normalmente um conteúdo menor, quando comparados aos tradicionais envelopes SOAP.
- Tempo de transferência e processamento. Considerando o tempo de transferência e processamento, o tempo de transferência dos dados e a quantidade de processamento necessária para empacotar/desempacotar as mensagens são menores comparados ao SOAP.
- Acesso aos recursos. Em REST, é feito de forma simples, basta ter um *browser* ou uma API básica de uma linguagem de programação. Em SOAP, é necessário ter ferramentas ou algum *framework* (geralmente complexo) que devem atender todas as normas de especificação SOAP.
- *Cache* e formato. As respostas dos REST *Web services* podem utilizar *cache* e podem utilizar diversos formatos, sendo JSON

(*JavaScript Object Notation*) o formato mais utilizado, visto que o mesmo é menos verboso e normalmente menor em tamanho. Sendo assim, transferir JSON em comparação ao formato XML, que é mais verboso e possui tamanho maior, causa uma menor carga na rede.

Por fim, algumas considerações são feitas sobre as vantagens e as desvantagens de se utilizar *web services* (SOMMERVILLE, 2007): (1) reutilização, visa facilitar o reuso de serviços ao projetá-los respeitando boas práticas de projeto; (2) integração, viabiliza a integração de sistemas heterogêneos, aqueles implementados em diferentes linguagens; (3) interoperabilidade, disponibiliza serviços que poderão ser utilizados por diferentes aplicações, independentemente das suas respectivas linguagens/plataformas utilizadas; (4) testabilidade, muitas vezes não é trivial testar aplicações seguindo uma arquitetura orientada a serviços; (5) custo com infraestrutura, é necessário geralmente reformular a infraestrutura para dar suporte à nova estrutura exigida para disponibilizar os serviços; (6) curva de aprendizagem, desenvolver e manter aplicações seguindo uma arquitetura baseada em serviços exigem uma maior maturidade da equipe.

6.2 Arquiteturas orientadas a serviços

Nesta seção, são discutidos conceitos introdutórios e práticos sobre arquiteturas orientadas a serviços, também amplamente conhecidas pela sigla SOA (*Software Oriented Architecture*). Para começar, destaca-se que o conceito de SOA surgiu pela primeira vez em 1996 em Schulte e Natis (1996). Os autores definiram SOA como sendo uma abordagem arquitetural corporativa que permite a criação de serviços de negócio interoperáveis que podem facilmente ser reutilizados e compartilhados entre aplicações e empresas.

Essa definição evidencia dois aspectos relevantes que motivaram o surgimento de SOA. O primeiro é que SOA surgiu definitivamente da necessidade de grandes organizações de expandirem seus sistemas, além dos seus limites organizacionais. Como citado na seção anterior, essa afirmação ratifica os argumentos citados anteriormente que, para expandir seus negócios, as organizações precisaram expandir seus sistemas para atender novas demandas e necessidades de processamento. Logo, foram forçadas a experimentar modelos emergentes de computação distribuída que

permitissem computação interorganizacional distribuída ao contrário da intraorganizacional. O segundo aspecto interessante se refere à necessidade de promover reuso, ao mesmo tempo que garantir a interoperabilidade entre, principalmente, os novos sistemas das organizações e os sistemas legados.

Diante desse cenário, SOA passou a ter uma evidência muito significativa, principalmente devido à necessidade, cada vez mais frequente, de integrar sistemas e processos, bem como desenvolver sistemas de software organizacionais baseados na composição de serviços. Porém, a prática de integração de sistemas e o desenvolvimento de sistemas organizacionais fortemente baseados na composição de serviços não eram, até então, comuns. Como consequência, vários desafios surgiram. Primeiramente, porque os sistemas não eram previamente projetados de tal forma que a integração dos mesmos fosse realizada com baixo esforço e baixa propensão a erros. Segundo, faltava uma tecnologia que viabilizasse desenvolver software baseada na composição em projetos na indústria. Então, desde o surgimento das ideias propostas por Schulte e Natis (1996), o desafio foi como viabilizar a integração de sistemas organizacionais, os quais tipicamente são legados e heterogêneos, e como desenvolver os novos sistemas evitando tais problemas.

A solução proposta foi projetar sistemas de software com arquiteturas orientadas a serviços, sejam eles SOAP ou REST. Como mencionado na seção anterior, os serviços devem ser independentes e fracamente acoplados, devem operar sempre da mesma maneira, independente da sua plataforma. Nessa nova configuração, tem-se os provedores de serviços disponibilizando serviços especializados para a uma gama de usuários de serviços de diferentes organizações.

Diante da necessidade de ajudar na definição dos conceitos de SOA, alguns pesquisadores e profissionais da indústria propuseram um manifesto SOA em Booch (2015). Esse manifesto deixa clara a necessidade de dar prioridade a valor de negócio ao contrário de estratégia técnica; objetivos estratégicos ao contrário de benefícios específicos do projeto; interoperabilidade intrínseca ao contrário de integração personalizada; serviços compartilhados ao contrário de implementações de propósito específico; flexibilidade ao contrário de otimização; e refinamento evolutivo ao contrário de busca da

perfeição inicial.

Além desses direcionamentos, alguns princípios SOA são também propostos em Booch (2015). Alguns deles são citados a seguir: respeitar a estrutura social e de poder da organização; SOA pode ser realizada através de uma variedade de tecnologias e padrões; evoluir serviços e sua organização em resposta ao seu uso real; verificar se os serviços satisfazem os requisitos de negócio e objetivos; e maximizar o uso do serviço, considerando o atual e o futuro escopo de utilização.

Para colocar tanto o manifesto como os princípios SOA em prática, algumas exigências ou demandas podem surgir. A primeira delas seria a necessidade de ter uma clara modelagem do negócio e dos processos das organizações, bem como uma possível simulação de possíveis cenários. Para isso, arquitetos e desenvolvedores precisam usar tecnologias, tais como a BPMN 2.0 (*Business Process Model and Notation*) (OMGb, 2011), para especificar os processos organizacionais. Entendendo bem os processos organizacionais, isso potencializará a definição de serviços, como componentes de software. Essa visão que busca disponibilizar e modularizar os processos de negócio como um serviço, permite não só fomentar o reuso de módulos dos sistemas organizacionais, bem como facilita a integração de aplicações.

Outros exemplos de exigências seriam o gerenciamento de processos em servidores de processos e o monitoramento dos processos de negócio da organização. Em particular, destaca-se o desafio de manter os serviços atualizados e alinhados com os processos de negócios das organizações, bem como a orquestração dos mesmos. Esse desafio recebe um destaque devido à volatilidade dos processos de negócio nos dias atuais, os quais estão sempre recebendo modificações para se adequarem às novas exigências, ou mesmo para representar uma expansão da área de atuação da organização.

Para a implementação de uma arquitetura orientada a serviços é fundamental entender três conceitos: provedor de serviço, solicitante de serviço e registro de serviço. Para isso, recomenda-se que o leitor revise a Figura 26. Basicamente, a dinâmica entre tais conceitos seria a seguinte: (1) um provedor de serviços oferece um (ou mais) serviço através da definição de sua interface e pela implementação do serviço; (2) o provedor de serviço especifica os serviços usando

WSDL; (3) o provedor publica informações sobre esses serviços em um registro de acesso geral usando um padrão de publicação, chamado de UDDI (*Universal Description Discovery and Integration*); (4) um solicitante do serviço vincula esse serviço à sua aplicação. Isso significa que a aplicação inclui códigos para chamar o serviço e processar os resultados da chamada; (5) os solicitantes de serviços (os clientes) buscam o registro UDDI para descobrir a especificação desse serviço e para localizar um provedor de serviços; e (6) a comunicação entre o servidor e o cliente pode ser feita utilizando, por exemplo, SOAP.

Esses conceitos são inseridos na prática no contexto da arquitetura apresentada na Figura 33. Neste exemplo, em particular, deseja-se viabilizar a interoperabilidade entre duas aplicações A e B. Ambas as aplicações compartilham de uma base de dados, camada de dados, camada de negócio, e camada de apresentação. Mas, independente dessas camadas, a interoperabilidade é viabilizada através da adição da Camada de Integração em ambas as aplicações, através das quais a comunicação entre os serviços será realizada.

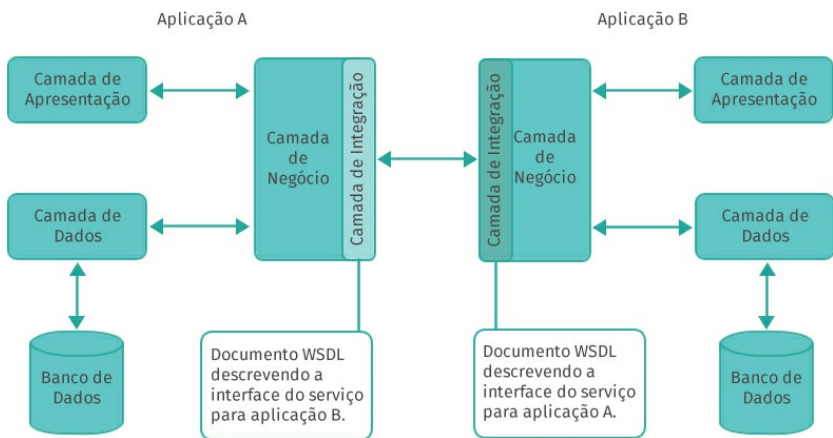


Figura 33 – Exemplo ilustrativo.
Fonte: elaborada pelo autor.

Outro conceito importante em SOA trata-se do barramento de serviços, o qual é utilizado como uma abordagem de centralização de serviços. Ao permitir a centralização dos serviços, o barramento

promete construir uma SOA de forma iterativa por integrar todos os tipos de aplicações isoladas, em uma infraestrutura descentralizada. De acordo com Breest (2015), o barramento disponibiliza uma camada de *middleware* através da qual um conjunto de serviços são largamente disponibilizados e compostos. O principal benefício é que evita a comunicação ponto a ponto entre aplicações, reduzindo um forte acoplamento entre as mesmas. A Figura 34 ilustra dois sistemas de duas empresas fortemente acoplados devido às dependências criadas entre eles, para viabilizar a integração dos sistemas. Através dessa ilustração disponível em Breest (2006), é possível perceber como a ausência de um barramento de serviços levou a uma arquitetura com aplicações fortemente acopladas, as quais estão conectadas através de conexões não confiáveis ponto a ponto. Nesse caso, o processo e a lógica do negócio são codificados nas aplicações, dificultando a manutenção e evolução quando mudanças não antecipadas surgem, e quando novas integrações precisam ser feitas.

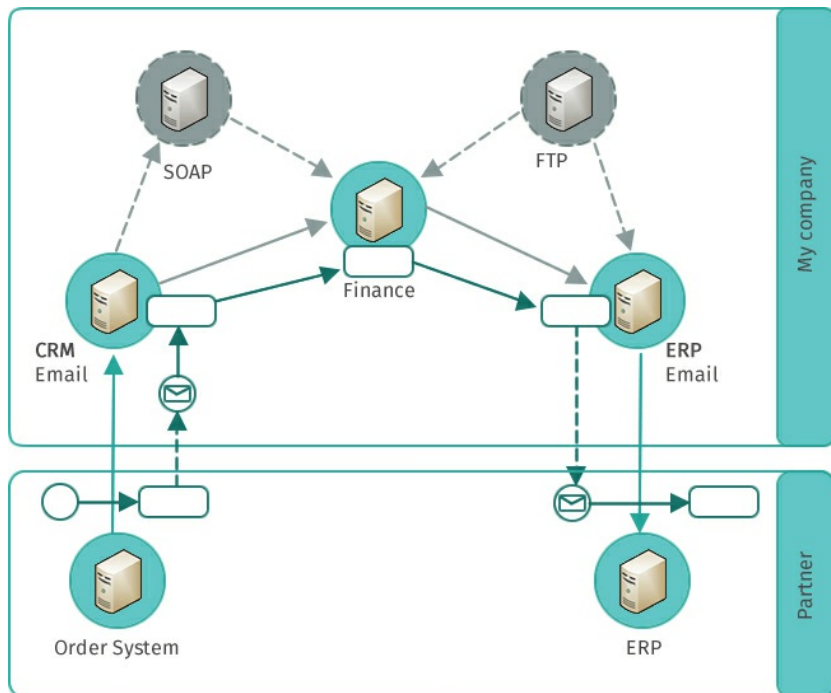


Figura 34 – Exemplo de sistemas fortemente acoplados.
Fonte: Breest (2006).

Para mudar essa realidade, busca-se desenvolver um barramento, visando trazer serviços empresariais reutilizáveis para o barramento. Agora os sistemas organizacionais são produzidos reutilizando e compondo os serviços disponibilizados no barramento. Em Breest (2006), o autor evolui o exemplo da Figura 34, ao introduzir o barramento de serviços.

A Figura 35 apresenta um exemplo de arquitetura em que todas as aplicações são fornecidas como serviços de negócio, bem como são conectadas via canais virtuais confiáveis, seguros e gerenciáveis. Uma consequência disso é que a orquestração de processos e a lógica de transformação de dados podem ser transferidas para o barramento, logo, as interações de processo podem ser realizadas de uma maneira controlada.

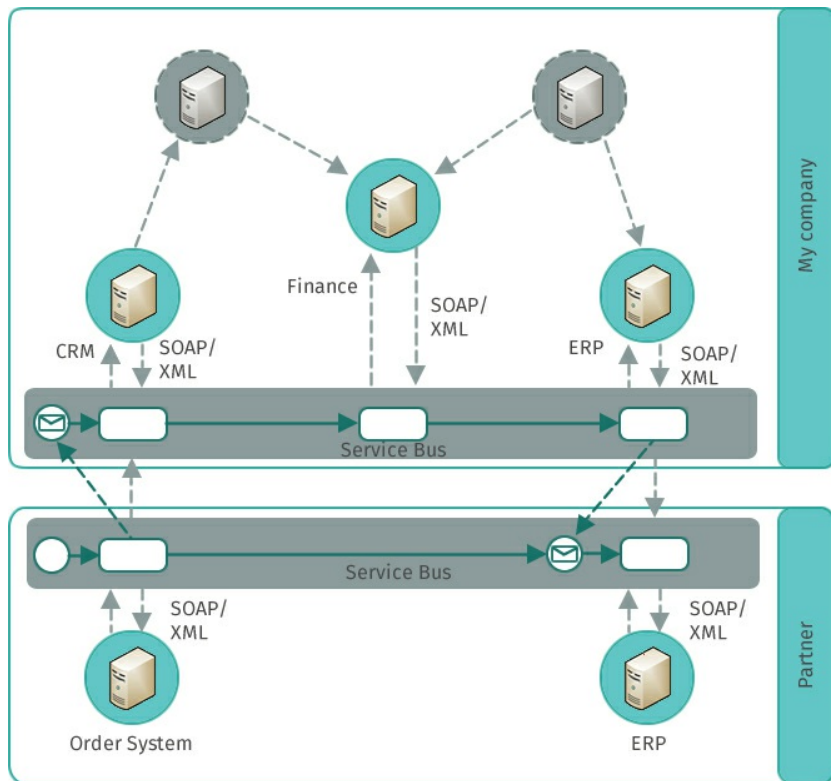


Figura 35 – Exemplo ilustrativo de um barramento de serviços.

Fonte: Breest (2006).

1. <https://docs.oracle.com/javaee/7/tutorial/jaxws.htm>

2. <https://netbeans.org/>

3. <http://www.eclipse.org/>

4. <https://www.w3.org/Protocols/>

5. Do inglês, *self-descriptive messages*.

REFERÊNCIAS

ACME. *The ACME Project*, 2016. Disponível em: <<http://www.di.univaq.it/malavolta/al/>>. Acesso em: 18 jun. 2016.

ASPECTJ, *AspectJ Programming Guide*. Palo Alto Research Center. Disponível em: <<http://eclipse.org/aspectj/doc/released/progguide/index>>. Acesso em: 18 jun. 2016.

BASS, L., CLEMENTS, P., KAZMAN, R. *Software architecture in practice*.

Boston: Addison-Wesley Professional. 2a edition, 2003.

BERGEN, P. V. *Pipes-And-Filters, Garfixia Software Architectures*. Disponível em: <http://www.dossier-andreas.net/software_architecture/pipe_and_filter.html>. Acesso em: 18 jun. 2016.

BREEST, M. *An Introduction to the Enterprise Service Bus*, Hasso-Plattner-Institute for IT Systems Engineering. University of Potsdam, 2006.

BOOCH, G. *et. al. The SOA Manifesto*. 2015. Disponível em: <<http://www.soa-manifesto.org/>>. Acesso em: 18 jun. 2016.

BOOTH, D.; HAAS, H.; MCCABE, F.; NEWCOMER, E.; CHAMPION, M.; FERRIS, C.; ORCHARD, D. *Web Services Architecture*. Disponível em: <<http://www.w3.org/TR/ws-arch/>>. Acesso em: 18 jun. 2016.

COUNCIL, W. T.; HEINEMAN, G. T. *Definition of a software component and its elements. Component-based software engineering*. Boston: Addison-Wesley, 2001.

DEMARCO, T. *Structured Analysis and System Specification*.

Yourdon Press Computing Series, 2a Edition, Boston: Prentice Hall PTR, 1979.

ERL, T. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Boston: Prentice Hall. ISBN: 0131858580. 760 pages, 2005.

ECKSTEIN, R. *Java SE Application Design With MVC*. 2007. Disponível em: <<https://goo.gl/Kos8dS>>. Acesso em: 18 de junho 2016.

FARIAS, K.; GARCIA, A.; LUCENA, C. *Effects of Stability on Model Composition Effort: an Exploratory Study*. *Journal on Software and Systems Modeling*. pages 1-22, 2013.

FARIAS, K.; GARCIA, A.; WHITTLE, J.; CHAVEZ, C.; LUCENA, C. *Evaluating the Effort of Composing Design Models: A Controlled Experiment*. *Journal on Software and Systems Modeling*, pages 1-17, 2014.

FEILER, P. H.; GLUCH, D. P. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. 1a Edição. New Jersey: Addison-Wesley Professional, 2012.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD Thesis. University of California, Irvine, 2000.

FOWLER, M. *Padrões de Arquitetura de Aplicações Corporativas*, 493 p., Bookman, 2006.

FOSTER, I.; KESSELMAN, C.; NICK, J. M.; TUECKE, S. *IEEE Computer*. Volume 35, Issue 6, Page 37-46, June 2002.

GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. New Jersey: Addison-Wesley, USA, 1994.

GUDGIN, M.; HADLEY, M.; MENDELSON, N.; MOREAU, J.; NIELSEN, H. F.; KARMARKAR, A.; LAFON, Y. *SOAP Specification (Part 1)*. *World Wide Web Consortium*. Version 1.2. 2a Edition. 2007. Disponível em: <<https://www.w3.org/TR/soap12/>>. Acesso

em: 18 jun. 2016.

HOHPE, G.; WOOLF, B. *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions*. 650 pages. New York: Addison-Wesley, 2004.

JACOBSEN, I.; NG, P.-W. *Aspect-oriented software development with use cases*. Boston: Addison Wesley, 2004.

JENDROCK, E.; NAVARRO, R. C.; EVANS, I.; HAASE, K.; MARKITO, W. *Java Platform, Enterprise Edition, The Java EE Tutorial*, Release 7, September 2014.

JONES, M. P. *The Practical Guide to Structured Systems Design*. 2ª Edition, Yourdon Press Computing Series, Boston: Prentice Hall PTR, 1988.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C. V.; LOINGTIER, J.-M.; IRWIN, J. *Aspect-Oriented Programming, European Conference on Object-Oriented Programming (ECOOP)*, pages 1-24, Springer-Verlag LNCS 1241, June 1997.

KREGER, H. *Web services conceptual architecture (WSCA 1.0)*. IBM Corporation. pages 1-42, 2001.

KUMARAN, S. I. *JINI technology: an overview*. Englewood Cliffs. New Jersey: Prentice Hall, 2001.

LADDAD, R. *AspectJ in Action*. 2ª Edition. Manning Publications, 2009.

OMG, *Object Management Group. Unified Modeling Language – Infrastructure*, Version 2.5. *Technical Report*, 2011. Disponível em: <<http://www.omg.org/spec/UML/2.5>>. Acesso em: 18 jun. 2016.

OMGb, *Object Management Group, Business Process Model and Notation*. Version 2.0. *Technical Report*. January 2011. Disponível em: <<http://www.omg.org/spec/BPMN/2.0/PDF/>>. Acesso em: 18 jun. 2016.

ORACLE, *Java Platform, Enterprise Edition: The Java EE Tutorial*. Disponível em:

<<https://docs.oracle.com/javaee/7/tutorial>>. Acesso em: 18 jun. 2016.

LEE, T. M.; HENDLER, J.; LASSILA, O. *The Semantic Web: a new form of Web content that is meaningful to computers will unleash a revolution of new possibilities*. Scientific American, May 2001.

MARTIN, R. C. *Agile Software Development, Principles, Patterns, and Practices*. 1a Edição. New Jersey: Pearson Education, 2002.

MEI, H.; CHEN, F.; WANG, Q.; FENG, Y. *ABC/ADL: An ADL Supporting Component Composition, Formal Methods and Software Engineering*. Volume 2495. LNCS. P. 38-47. October 2002.

PRUD, E. G. *Validation Service, World Wide Web Consortium (W3C)*. Disponível em: <<http://www.w3.org/RDF/Validator/>>. Acesso em: 18 jun. 2016.

RDF, *RDF Working Group. Resource Description Framework (RDF). W3C Semantic Web*. Disponível em: <<https://www.w3.org/RDF/>>. Acesso em: 18 de junho 2016.

RDFa, *XML RDF, W3 Schools*. Disponível em: <http://www.w3schools.com/xml/xml_rdf.asp>. Acesso em: 18 de junho 2016.

REENSKAUG, T. *MODELS-VIEWS-CONTROLLERS*, December 1979.

RICCA, F.; PENTA, M. D.; TORCHIANO, M.; TONELLA, P.; CECCATO, M. *How developers' experience and ability influence web application comprehension tasks supported by uml stereotypes: a series of four experiments*. *IEEE Transactions on Software Engineering*. Volume 36. Número 1. pp. 96-118, Janeiro/Fevereiro, 2010.

SAUVÉ, J. *Introdução e Motivação: Arquiteturas em n Camadas*. 2015. Disponível em: <<http://www.dsc.ufcg.edu.br/~jacques/cursos/j2ee/html/intro>> Acesso em: 18 jun. 2016.

SCHULTE, W. R.; NATIS, Y. V. *"Service Oriented" Architectures*.

Gartner. April, 1996.

SOAP, *Simple Object Access Protocol*. Disponível em: <<https://goo.gl/P0d3ZB>>. Acesso em: 18 jun. 2016.

SOARES, S. *Programação Orientada a Aspectos com AspectJ (Minicurso), Congresso Brasileiro de Software: Teoria e Prática. Simpósio Brasileiro de Engenharia de Software*. 2010.

SOMMERVILLE, I. *Engenharia de Software*. 8 ed. São Paulo: Pearson Addison-Wesley, 2007.

STAL, M. *Web services: beyond component-based computing. Communications of the ACM*. Volume 45. Issue 10. pages 71-76, October 2002.

EASTERBROOK, S. *Architectural Styles. Lecture 21: Software Architectures*. Department of Computer Science, University of Toronto, 2004.

SZYPERSKI, C. *Component software: beyond object-oriented programming*. 2. ed. Harlow: Addison-Wesley, 2002.

TURNER, M.; BUDGEN, D.; BRERETON, P. *Turning software into a service*. IEEE Computer. Volume 36. Number 10. pages 38-45, 2003.

W3C. *Web Semântica, W3C Brasil*, Disponível em: <<http://www.w3c.br/Padroes/WebSemantica>>. Acesso em: 18 jun. 2016.

W3S. *Web Semântica. W3Schools*. Disponível em: <<http://www.w3schools.com/xml/>>. Acesso em: 18 jun. 2016.

Sites recomendados:

Especificação da API Java para *web services* baseados em XML: <http://jcp.org/en/jsr/detail?id=224>

Site da API Java JAX-WS: <https://jax-ws.java.net/>

Simple Object Access Protocol (SOAP) 1.2 W3C Note:

<http://www.w3.org/TR/soap/>

Web Services Description Language (WSDL) 1.1 W3C Note:
<http://www.w3.org/TR/wsdl>

WS-I Basic Profile 1.2 and 2.0: <http://www.ws-i.org>

O AUTOR



Kleinner Silva Farias de Oliveira
Doutor em Informática

Doutor em Informática pela Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). Mestre em Ciência da Computação pela Pontifícia Universidade Católica do Rio Grande do Sul (PUC-RS). Bacharel em Ciência da Computação pela Universidade Federal de Alagoas (UFAL). Tecnólogo em Tecnologia da Informação pelo Instituto Federal de Alagoas (IFAL). Atua como professor assistente no Programa Interdisciplinar de Pós-Graduação em Computação Aplicada (PIPCA) e no curso de graduação em Análise e Desenvolvimento de Sistemas da Universidade do Vale do Rio dos Sinos (Unisinos) desde 2013. Tem experiência na área de Engenharia de Software, com ênfase nos seguintes temas: modelagem de software, desenvolvimento de software dirigido por modelos, engenharia de software experimental, desenvolvimento de software orientado a aspectos, engenharia reversa e arquitetura de software.

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

Reitor: Pe. Marcelo Fernandes de Aquino, SJ

Vice-reitor: Pe. José Ivo Follmann, SJ

Diretor da Editora Unisinos: Pe. Pedro Gilberto Gomes



Editora Unisinos

Avenida Unisinos, 950, 93022-000, São Leopoldo, Rio Grande do Sul, Brasil

editora@unisinos.br

www.edunisinos.com.br

© dos autores, 2016

2016 Direitos de publicação da versão eletrônica (em e-book) deste livro exclusivos da Editora Unisinos.

- O48a Oliveira, Kleinner Silva Farias de.
Arquitetura de software [recurso eletrônico] / Kleinner Silva Farias de Oliveira – São Leopoldo: Ed. UNISINOS, 2016.
1 recurso online – (EaD)

ISBN 978-85-7431-764-9

1. Arquitetura de software. 2. Componentes de software.
3. Software – Desenvolvimento. I. Título. II. Série.

CDD 005.3
CDU 004.273

Coleção EAD

Editor: Carlos Alberto Gianotti

Acompanhamento editorial: Jaqueline Fagundes Freitas

Revisão: Simone Ceré

Editoração: Guilherme Hockmüller

A reprodução, ainda que parcial, por qualquer meio, das páginas que compõem este livro, para uso não individual, mesmo para fins didáticos, sem autorização escrita do editor, é ilícita e constitui uma contrafação danosa à cultura. Foi feito depósito legal.

A coleção EaD, de que faz parte este livro, é uma produção da Universidade do Vale do Rio dos Sinos para apoiar os processos de ensino e aprendizagem dos seus cursos de graduação a distância. Entretanto, o uso dessa obra não fica restrito apenas a essa modalidade de ensino, uma vez que pode servir como orientador no estudo de qualquer acadêmico. Os exemplares foram elaborados a partir da experiência de professores de reconhecimento mérito acadêmico da Universidade e traduzem a excelência dos cursos de graduação ofertados na modalidade presencial e a distância da Instituição.

