PROJETO E ARQUITETURA DE SOFTWARE: UMA INTRODUÇÃO

KLEINNER SILVA FARIAS DE OLIVEIRA

São Leopoldo, 2016

SUMÁRIO

SOBRE O AUTOR

Kleinner Silva Farias de Oliveira. 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.

Currículo Lattes: http://lattes.cnpq.br/2582456631204400

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.

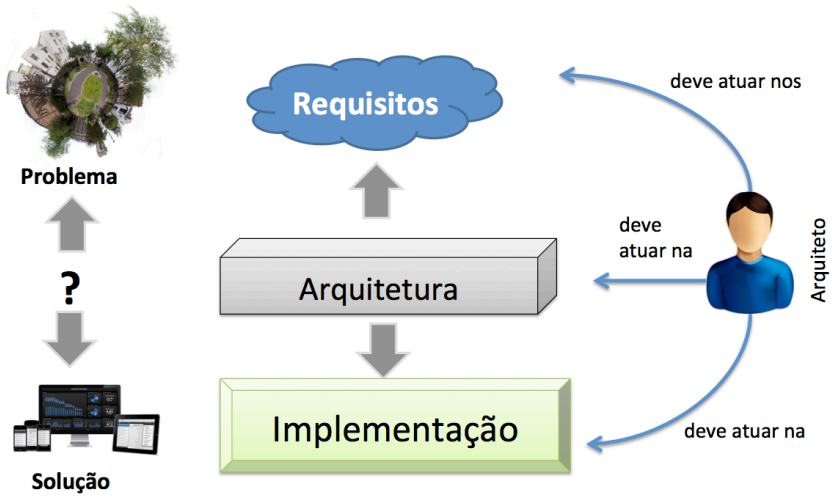


Figura 1 – Contextualização de arquitetura de software.

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 prospector 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.

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 caraterísticas 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; FARIAS, 2012). 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.

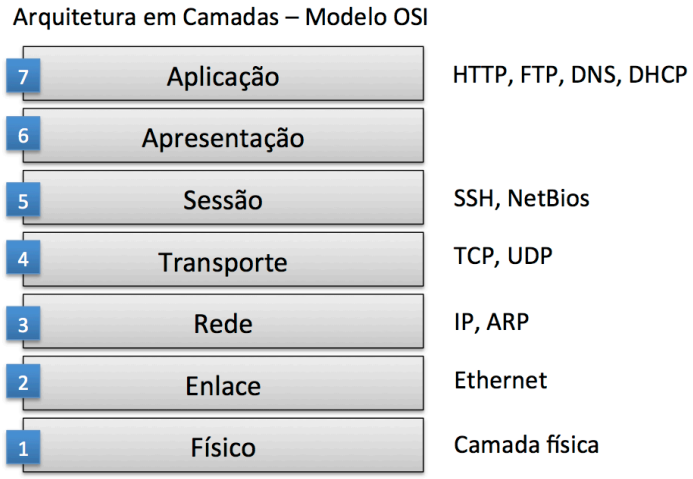


Figura 2 – Exemplo ilustrativo de uma arquitetura em camadas.

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 Sauvé (2015), a arquitetura em duas camadas surgiu com a finalidade de melhorar o aproveitamento dos computadores pessoais nas empresas, suportar sistemas com interfaces gráficas amigáveis, integrar a aplicação com os dados corporativos, permitir uma maior escalabilidade dos sistemas de informação, entre outras razões. A Figura 3 apresenta um exemplo de arquitetura em duas camadas.

A primeira camada trata-se da camada cliente, a qual possui a lógica de negócio e da interface do usuário. A segunda se refere à camada do servidor que tipicamente trata da manipulação dos dados da aplicação. De acordo com Sauvé (2015), algumas desvantagens podem ser consideradas ao utilizar essa arquitetura, tais como falta de escalabilidade, conexões problemáticas ao banco de dados, enormes problemas de manutenção, mudanças na lógica de aplicação forçam atualizações nas versões do sistema que se encontra na máquina do usuário, e dificuldade de acessar fontes heterogêneas de dados.

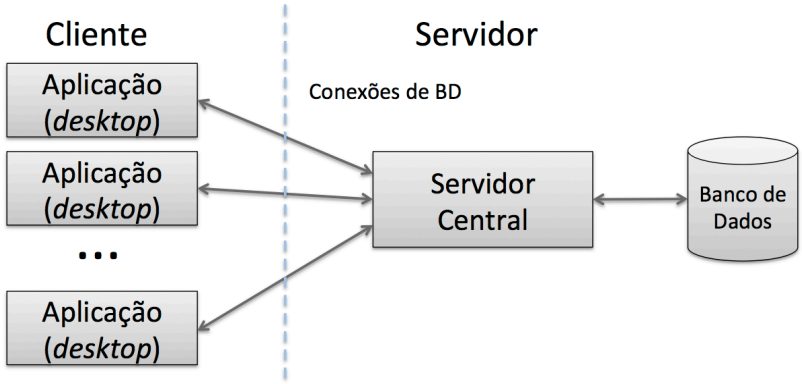


Figura 3 – Exemplo de arquitetura em duas camadas.

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

À medida que os problemas encontrados na arquitetura em duas camadas foram investigados, percebeu-se que, remodelando a arquitetura em duas camadas para três, alguns problemas críticos eram mitigados. Nessa nova configuração, a arquitetura passou a ser organizada em camadas de apresentação, domínio e de acesso a dados. Por exemplo, a camada de domínio passou a esconder as responsabilidades inerentes à camada de acesso a dados da camada de apresentação. A Figura 4 apresenta uma ilustração sobre como as três camadas são organizadas hierarquicamente. Por sua vez, a Figura 5 mostra a evolução da arquitetura de duas camadas para três.

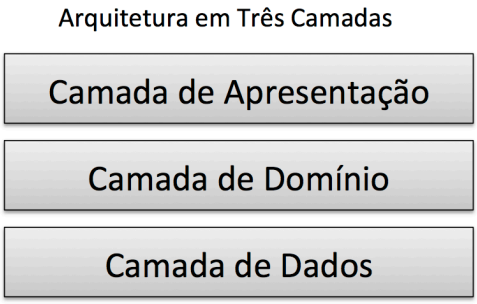


Figura 4 – Exemplo ilustrativo de arquitetura em três camadas.

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).

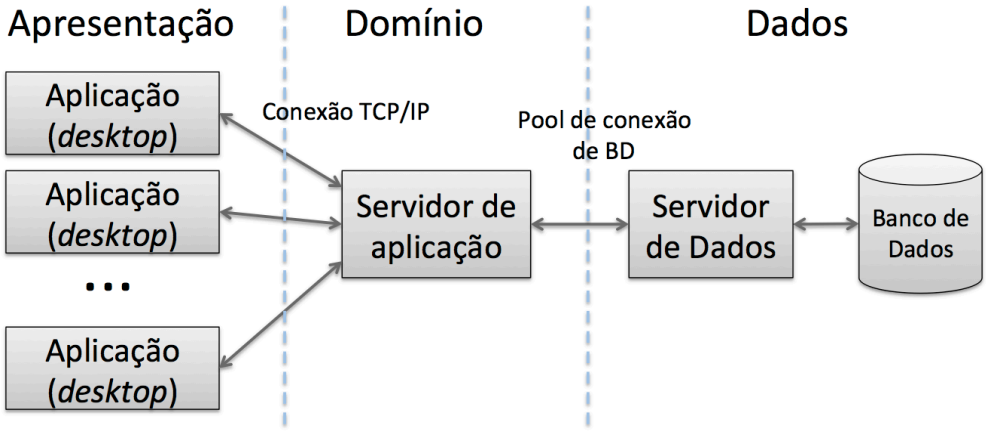


Figura 5 – Evolução da arquitetura de duas camadas para três camadas.

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.

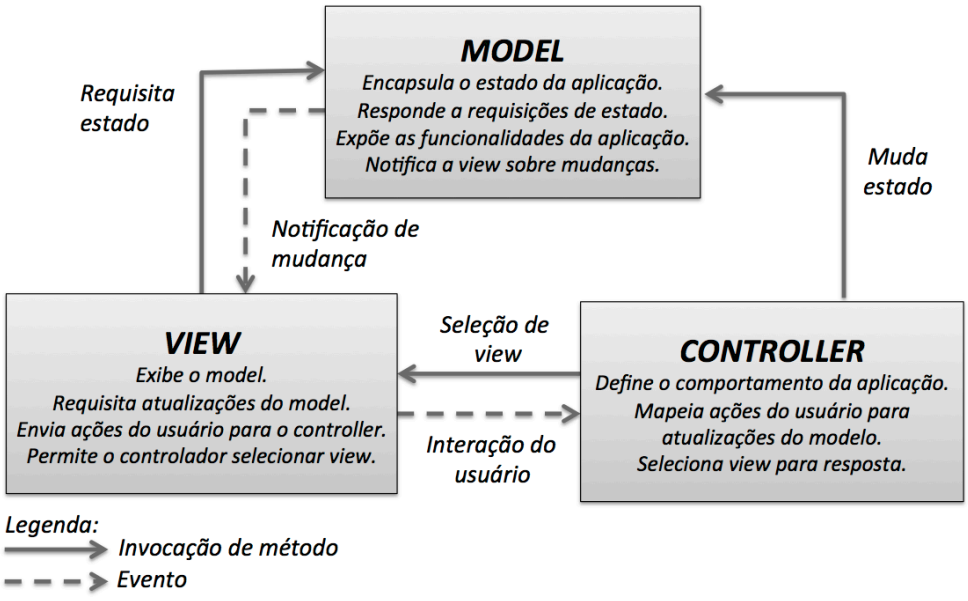


Figura 6 – Ilustração do modelo MVC.

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.

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.

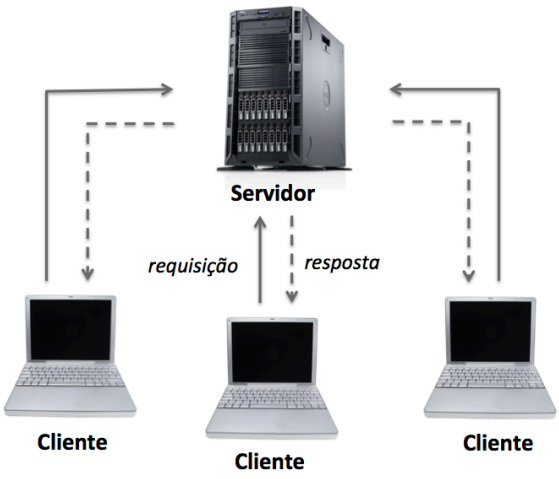


Figura 7 – Ilustração da arquitetura cliente-servidor.

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 Datagrama 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 um 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.

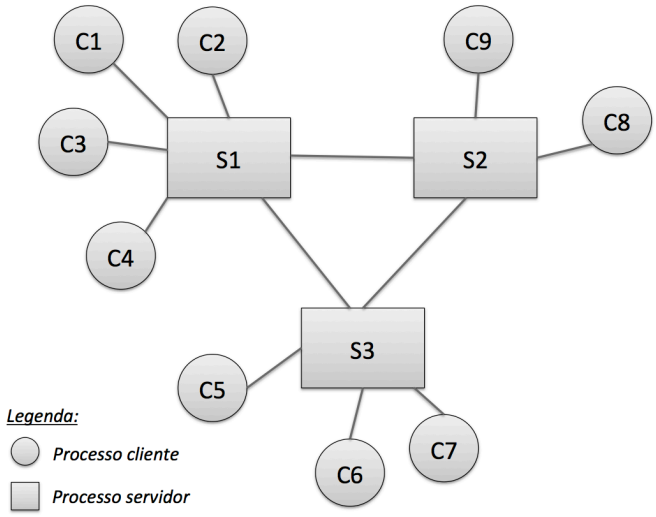


Figura 8 – Processos de um sistema cliente-servidor.

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 de *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.

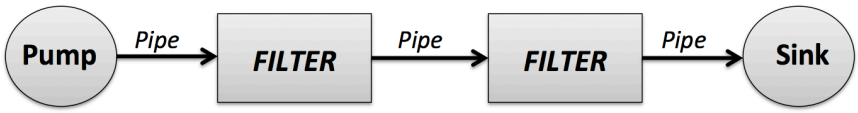


Figura 9 – Elementos da arquitetura baseada em *pipes and filters*.

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

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).

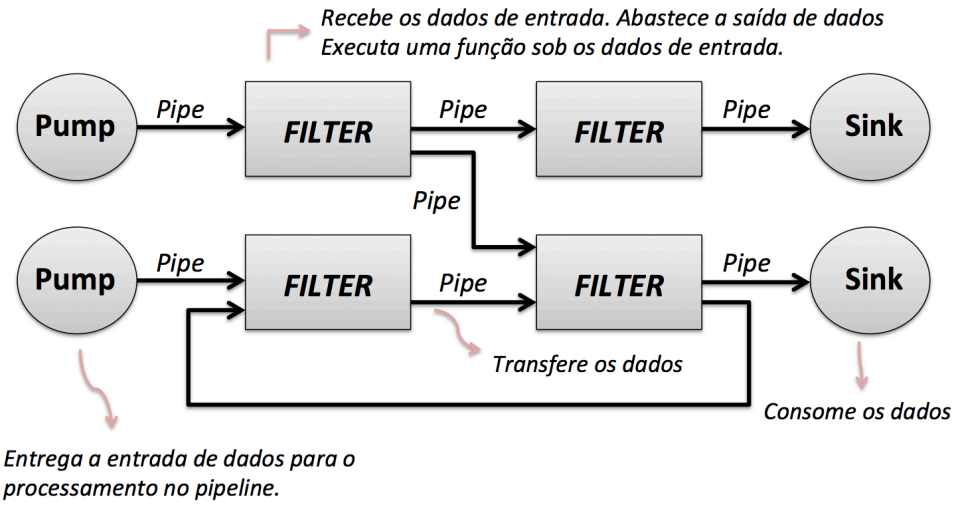


Figura 10 – Exemplo de arquitetura usando *pipes and filters* .

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

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.

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

àquelas 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.

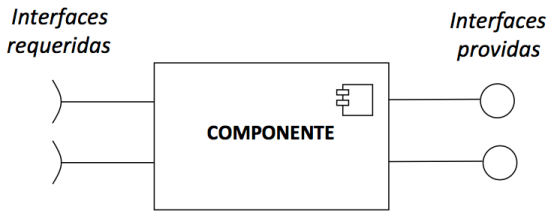


Figura 11 – Exemplo de componente com as suas interfaces requeridas e providas.

Fonte: elaborada pelo autor.

Os componentes arquiteturais devem ser definidos, implementados, mantidos e evoluídos por arquitetos e desenvolvedores de software, sempre tendo suas principais caraterísticas 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 caraterística, 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.

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.

Tabela 1 – Características de componente.

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 metaclasse 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*) (FARIAS, 2015). 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; FARIAS, 2013a; FARIAS, 2011) 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 (FARIAS, 2012b; FARIAS, 2014a), 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.

- 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 (GONÇALES, 2015; GONÇALES, 2015a).
- 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).

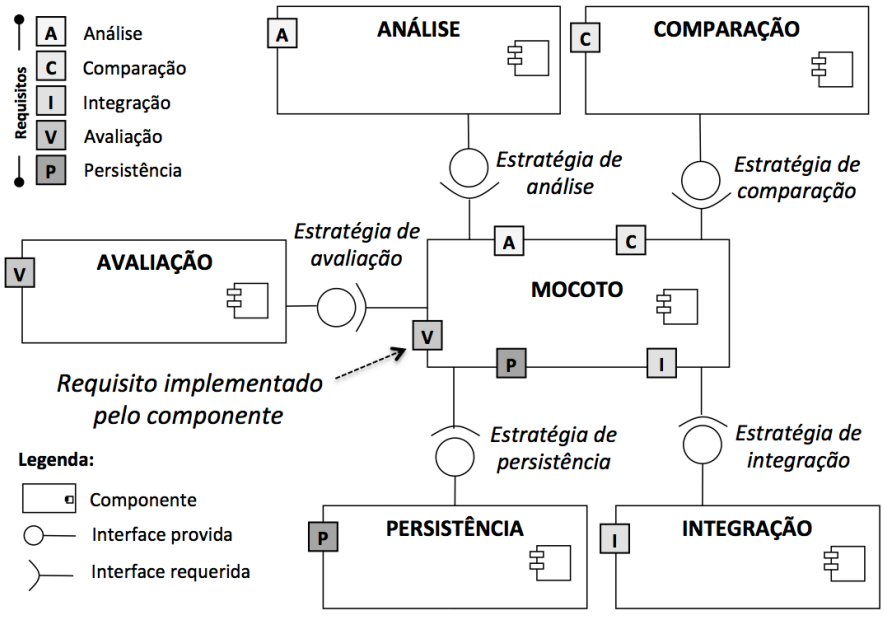


Figura 12 – Exemplo de diagrama de componentes da UML.

Fonte: elaborada pelo autor.

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 cincos requisitos citados anteriormente foram modularizados em cincos 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 13 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.

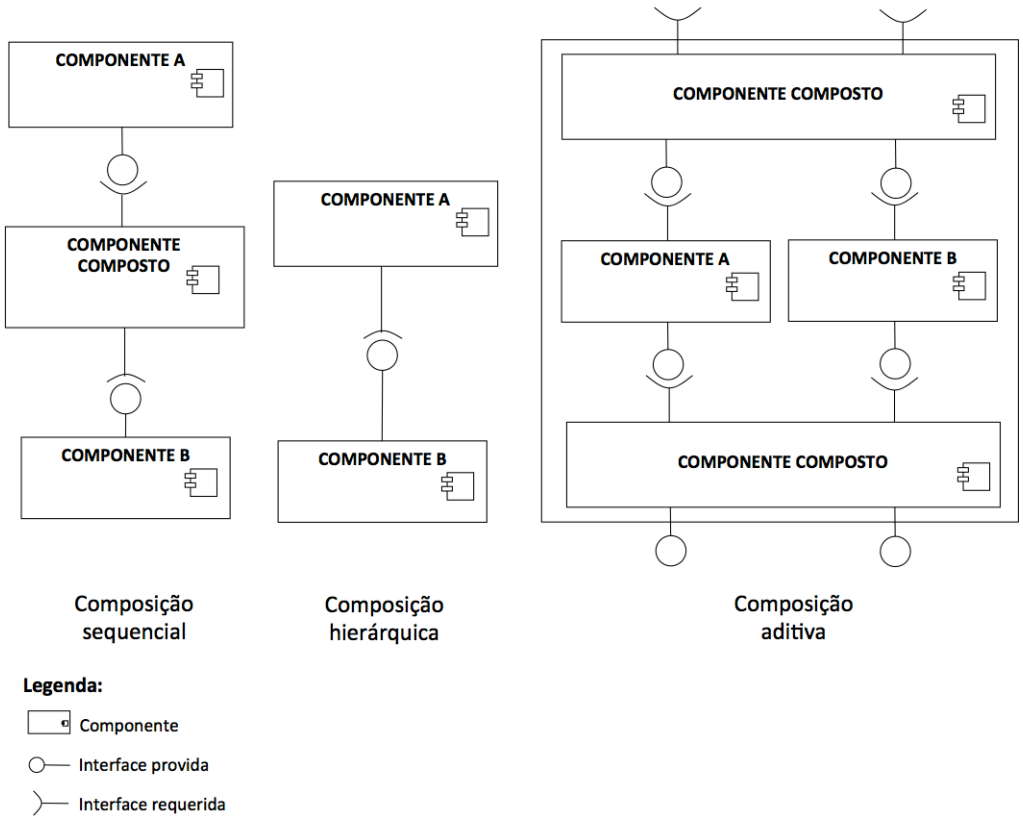


Figura 13 – Tipos de composição de componentes.

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; FARIAS, 2010). 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, coletem 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).

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.

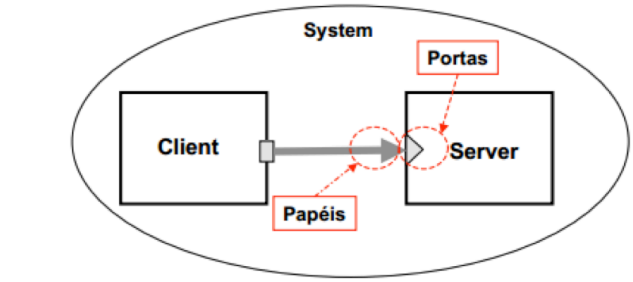
Um exemplo de linguagem de descrição de arquitetura é ACME. O projeto ACME (ACME, 2016) iniciou-se em 1995 com o objetivo de produzir uma linguagem comum que pudesse ser usada para suportar o intercâmbio de descrições arquiteturais entre ferramentas. Como resultado do projeto, foi produzida uma linguagem 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.

REFERÊNCIAS

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

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.

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

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

FARIAS, K. *Empirical Evaluation of Effort on Composing Design Models*, 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, pages 405-408, Cape Town, South Africa, 2010.

FARIAS, K., GARCIA, A., LUCENA, C. *Evaluating the Effects of Stability on Model Composition Effort: an Exploratory Study*, 8th Experimental Software Engineering Latin American Workshop (ESELAW'11), pages 77-86, Rio de Janeiro, Brazil, 2011.

FARIAS, K., GARCIA, A., WHITTLE, J., LUCENA, C. *Analyzing the Effort of Composing Design Models of Large-Scale Software in Industrial Case Studies*, 16th International Conference on Model-Driven Engineering Languages and Systems (MODELS), pages 639-655, Miami, FL, USA, 2013a.

FARIAS, K., GARCIA, A., LUCENA, C. *Evaluating the Impact of Aspects on Inconsistency Detection Effort: A Controlled Experiment*, 15th International Conference on Model-Driven Engineering Languages and Systems (MODELS'12), Vol. 7590, pages 219-234, Innsbruck, Austria, 2012.

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.

FARIAS, K. *Empirical Evaluation of Effort on Composing Design Models*, PhD Thesis, *Department of Informatics,* Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil, 2012b.

FARIAS, F., GARCIA, A., LUCENA, C., GONZAGA JR, L., COSTA, C. A., RIGHI, R. R., BASSO, F., OLIVEIRA, T. *Towards a Quality Model for Model Composition Effort*, 29th Annual ACM Symposium on Applied Computing (SAC.14), pages 1181-1183, Gyeongju, Korea, March, 2014a.

FARIAS, K., GONÇALES, L., SCHOLL, M., OLIVEIRA, T., VERONEZ, M., *Toward an Architecture for Model Composition Techniques*, 27th International Conference on Software Engineering and Knowledge Engineering (SEKE.15), pages 656-659, Pittsburgh, USA, 2015.

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.

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

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

GONÇALES, L., FARIAS, K., SCHOLL, M., VERONEZ, M., OLIVEIRA, T., *Comparison of Design Models: A Systematic Mapping Study*. International Journal of Software Engineering and Knowledge Engineering, 25(9-10): 1765-1770, 2015.

GONÇALES, L., FARIAS, K., SCHOLL, M., OLIVEIRA, T., VERONEZ, M. *Model Comparison: a Systematic Mapping Study*, 27th International Conference on Software Engineering and Knowledge Engineering (SEKE.15), pages 546-551, Pittsburgh, USA, 2015a.

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

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

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

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.

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/intro.htm>. Acesso em: 18 jun. 2016.

SOMMERVILLE, I. *Engenharia de Software*. 8 ed. São Paulo: Pearson Addison-Wesley, 2007.SZYPERSKI, C. *Component software: beyond object-oriented programming*. 2. ed. Harlow: Addison-Wesley, 2002.

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

Publish/Subscribe

1. Apache Kafka: http://kafka.apache.org/
2. Amazon Kinesis: https://aws.amazon.com/kinesis/streams/
3. Google Pub/Sub: https://cloud.google.com/pubsub/
4. Azure Event Hubs: https://azure.microsoft.com/en-us/services/event-hubs/
5. MapR Streams: https://www.mapr.com/products/mapr-streams
6. DistributedLog: http://distributedlog.incubator.apache.org/

Stream Processing

1. Apache Spark Streaming: http://spark.apache.org/streaming/
2. Apache Storm: http://storm.apache.org/

3. Apache Samza: http://samza.apache.org/
4. Heron: https://twitter.github.io/heron/
5. Amazon Kinesis: https://aws.amazon.com/kinesis/streams/
6. Google Dataflow: https://cloud.google.com/dataflow/
7. Azure Stream Analytics: https://azure.microsoft.com/en-us/services/stream-analytics/
8. Apache Apex: https://apex.apache.org/
9. Apache Flink: https://flink.apache.org/
10. Kafka Streams: http://kafka.apache.org/
11. Apache Flume: https://flume.apache.org/

Store/Analyze

1. Cloud Store/Analysis: AWS, Google, Azure
2. Apache Cassandra: http://cassandra.apache.org/
3. Apache HBase: https://hbase.apache.org/
4. Apache Druid: http://druid.io/
5. MemSQL: http://www.memsql.com/
6. VoltDB: https://www.voltdb.com/
7. Apache Solr: http://lucene.apache.org/solr/
8. ElasticSearch: https://www.elastic.co/
9. Mapr-DB: https://www.mapr.com/products/mapr-db-in-hadoop-nosql
10. Apache Kudu: https://kudu.apache.org/
11. FiloDB: https://github.com/tuplejump/FiloDB
12. Crate: https://crate.io/
13. Tachyon: http://www.alluxio.org/
14. Apache Spark: http://spark.apache.org/