WBA0598_V2.0

ARQUITETURA DE SOFTWARE

Anderson da Silva Marcolino

ARQUITETURA DE SOFTWARE

1ª edição

São Paulo
Platos Soluções Educacionais S.A
2022

Head de Platos Soluções Educacionais S.A
Silvia Rodrigues Cima Bizatto

Conselho Acadêmico
Alessandra Cristina Fahl
Ana Carolina Gulelmo Staut
Camila Braga de Oliveira Higa
Camila Turchetti Bacan Gabiatti
Giani Vendramel de Oliveira
Gislaine Denisale Ferreira
Henrique Salustiano Silva
Mariana Gerardi Mello
Nirse Ruscheinsky Breternitz
Priscila Pereira Silva

Coordenador
Henrique Salustiano Silva

Revisor
Marco Ikuro Hisatomi

Editorial
Beatriz Meloni Montefusco
Carolina Yaly
Márcia Regina Silva
Paola Andressa Machado Leal

Dados Internacionais de Catalogação na Publicação (CIP)

M321a

Marcolino, Anderson da Silva
Arquitetura de software / Anderson da Silva Marcolino. – São Paulo: Platos Soluções Educacionais S.A., 2022.
32 p.

ISBN 978-65-5356-418-3

1. Arquitetura. 2. Software. 3. Tecnologia. I. Título.

CDD 004

Evelyn Moraes – CRB: 010289/O

2022
Platos Soluções Educacionais S.A
Alameda Santos, n° 960 – Cerqueira César
CEP: 01418-002— São Paulo — SP
Homepage: https://www.platosedu.com.br/

ARQUITETURA DE SOFTWARE

SUMÁRIO

Apresentação da disciplina

Nesta disciplina, poderemos compreender como projetamos softwares modernos, sendo que tais projetos são expressos por meio do que chamamos Arquiteturas de Software. Além disso, poderemos compreender ainda a importância de tais arquiteturas para o desenvolvimento de bons produtos de software.

A disciplina está dividida em quatro temas. No primeiro, veremos os conceitos fundamentais das arquiteturas e compreenderemos os principais estilos e padrões arquiteturais que permitem estabelecermos arquiteturas corretas e com qualidade.

No segundo tema, veremos sobre conceitos provenientes da Engenharia de Software, em especial as etapas de análise e projeto, que, inseridas processo de desenvolvimento de software, são fases essenciais, pois projetamos as arquiteturas de software nelas.

No terceiro tema, estudaremos e construiremos modelos arquiteturais de software, considerando os padrões mais recentes do mercado, entre eles, os orientados a serviços e microsserviços.

No último e quarto tema, veremos sobre os modelos de qualidade e como podemos integrar e mensurar características de qualidade em arquiteturas e produtos de software.

Dessa forma, conseguiremos formar uma base sólida para atuarmos como arquitetos de software e poder conceber modelos arquiteturais que facilitarão as atividades dos desenvolvedores, testadores e, posteriormente, na evolução ou refatoração do produto já

construindo, sempre com um olhar voltado a qualidade do produto e suas especificidades, de acordo com o estilo e padrão arquitetural selecionado.

Bom estudo!

Conceitos Fundamentais sobre a Arquitetura de Software

Autoria: Anderson da Silva Marcolino
Leitura crítica: Marco Ikuro Hisatomi

Objetivos

- Contextualizar os alunos sobre os conceitos essenciais envolvidos nas Arquiteturas de Software.
- Orientar os alunos na identificação dos componentes de uma arquitetura, considerando os diferentes elementos envolvidos.
- Apresentar os principais estilos e padrões arquiteturais adotados atualmente.

1. Introdução à Arquitetura de Software

Ao pensarmos em um projeto de uma casa, prédio ou qualquer outra construção rural ou urbana, pensamos em sua arquitetura. No contexto da construção civil, é o modo pelo qual os diferentes elementos de uma construção são considerados e integrados (SOMMERVILLE, 2018; PRESSMAN, 2021). Para isso, empregam-se normas, técnicas, entre outros, para criar tais construções.

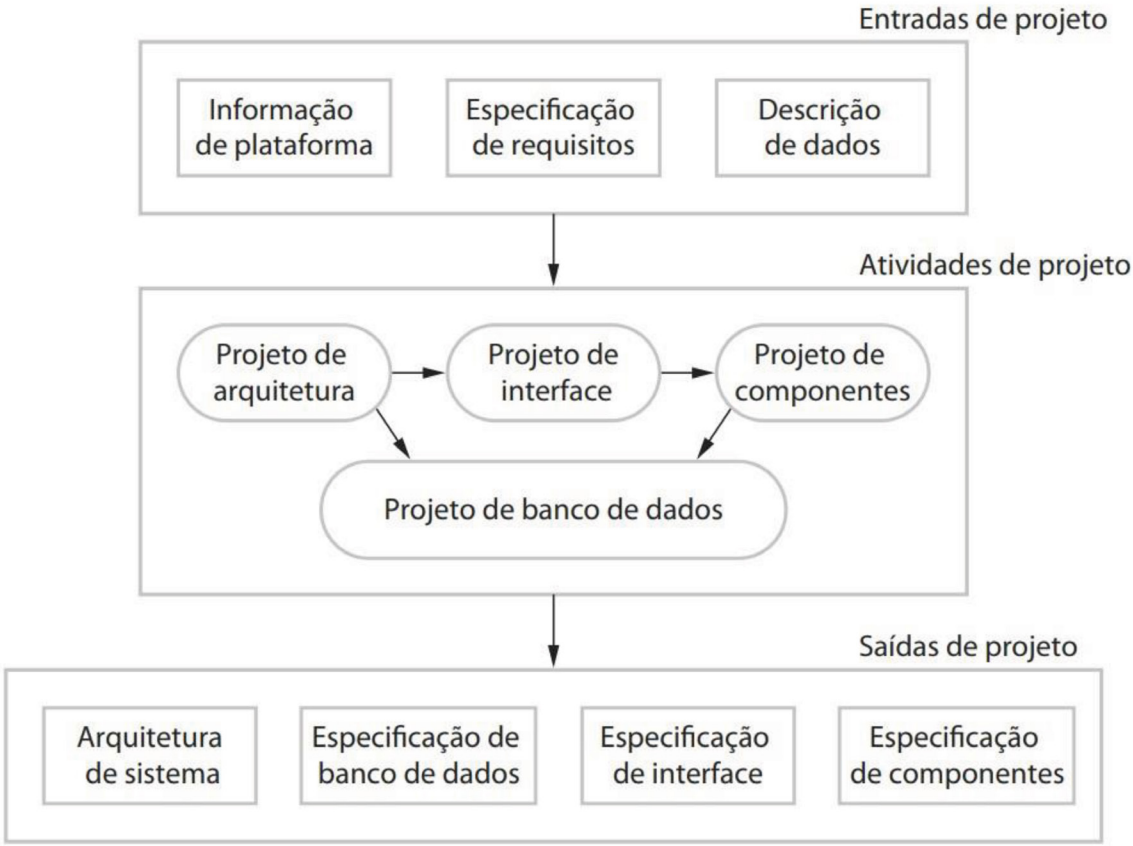
Analogicamente, pensando em um projeto arquitetura da construção civil, temos vários artefatos que constituem o projeto final: projeto hidráulico, de distribuição de energia, estrutural. Já no contexto de uma arquitetura para o software, também temos a especificação de diferentes representações que vão além do código-fonte, abrangendo armazenamento de dados, meios de conexão e acesso, comunicação entre os componentes ou elementos definidos por meio de metodologias específicas, como os diferentes artefatos se integram (suas interfaces) e o projeto das interfaces gráficas de usuário. Ao integrarmos todos eles, pode-se disponibilizar uma solução completa e de qualidade para uso (SOMMERVILLE, 2018; PRESSMAN, 2021).

Nessa perspectiva, percebe-se que temos diversos elementos que são necessários para entregar um software com a solução desejada. Dependendo de sua necessidade, os elementos envolvidos serão diferentes e devem estar organizados ao que se propõe no produto. As características de qualidade envolvidas para isso também variam e, deste modo, a complexidade em se definir um projeto de arquitetura de software é ampliado. Entender tais elementos e conseguir analisar para propor arquiteturas robustas, é imprescindível, e são estes elementos que serão tratados no decorrer deste texto.

1.1 A Arquitetura de Software e seus componentes

A arquitetura de um software ou sistema é o resultado de diferentes interações entre diversos artefatos analisados, projetados e estabelecidos em várias etapas especificadas pela Engenharia de Software. Logo, a arquitetura de software é uma subárea da Engenharia de Software. A Figura 1 apresenta um modelo geral do processo de projeto que concebe a arquitetura de software e outros artefatos.

Figura 1 – Modelo Genérico dos Processos para Projeto de um Software e sua Arquitetura



Fonte: adaptada de Sommerville (2018).

Como entradas de um projeto de software, temos as informações da plataforma. Essa informação especificará questões de hardware que serão necessárias para que o software possa ser utilizado. Existe, ainda,

a especificação de requisitos, que refletem o que o cliente deseja, bem como as restrições e itens de qualidade que devem ser considerados.

Os requisitos se dividem em requisitos funcionais, que, resumidamente, correspondem àquilo que será visível ao usuário final; os requisitos não funcionais, que estão relacionados a elementos de qualidade do software, como segurança, portabilidade, acessibilidade, facilidade de uso, desempenho, disponibilidade, entre outros, e os requisitos de domínio que especificam funcionalidades ou restrições e regras relacionadas ao foco o qual o produto atende.

Finalmente, entrada correspondente à descrição de dados, apresenta todos os dados e seus tipos que serão considerados na criação do projeto (SOMMERVILLE, 2018; PRESSMAN, 2021).

Nas atividades de projeto, temos a criação de cada um dos artefatos, que, juntos, formam de fato o projeto do sistema. O da interface gráfica de usuário, que considera as especificações das telas que serão criadas para a interação do usuário. O projeto de componentes são as partes, que, muitas vezes, são reutilizadas de outros projetos ou que, criadas para o projeto atual, serão reutilizadas posteriormente, cada componente desempenha funcionalidade específica e bem definida, a integração destes e como esta ocorre definem diferentes especificidades do software e como se comunicarão ao final.

O projeto de banco de dados especifica modelos de representação, como o diagrama de entidade relacionamento e modelo de entidade relacionamento, enquanto o primeiro expressa as entidades (tabelas) e seus relacionamentos, o segundo apresenta a especificação de cada entidade, com seus respectivos atributos e cardinalidades.

Finalmente, o projeto de arquitetura interligado ao projeto de interfaces e de banco de dados, contempla também os componentes. É nele, a parte do projeto em que se colocará no lugar todos os elementos que

devem ser desenvolvidos ou tecnologias que serão adotadas para se criar, de fato, o software final. Segundo Sommerville (2018, p. 42), o projeto de arquitetura é o projeto “no qual você pode identificar a estrutura geral do sistema, os componentes principais (algumas vezes, chamados subsistemas ou módulos), seus relacionamentos e como eles são distribuídos”.

Desse modo, é na fase de análise e projeto que a arquitetura ganha vida. A arquitetura de software garante o entendimento das partes necessárias para criar um software, como linguagens, plataformas, ambientes computacionais. Sua criação é imprescindível, pois garante melhor qualidade e otimização aos projetos de interface, de banco de dados e ao desenvolvimento, garantindo ainda que a solução esteja de acordo com padrões necessários para seu funcionamento adequado. Assim, projetos arquiteturais de softwares desenvolvidos pelos arquitetos de software reduzem riscos associados ao sistema. Adicionalmente, a arquitetura abrange outros elementos, como (SOMMERVILLE, 2018 PRESSMAN, 2021):

- Estrutura do sistema.
- Gerenciamento e controle.
- Protocolos para comunicação, sincronização e acesso aos dados.
- Designação de funcionalidades.
- Distribuição física de partes e sua escalabilidade.
- Atributos de qualidade.

Qualquer desenvolvedor que venha a lidar com decisões de alto nível necessita de um forte conhecimento técnico. Logo, o sucesso da arquitetura de um sistema está fortemente ligado com o design e a implementação. Design é uma forma de interpretar a implementação,

analisando e compreendendo as interações entre as partes do sistema, considerando os impactos que possíveis mudanças podem causar em outro componente que o acessa direta ou indiretamente (SILVEIRA et al., 2011; SOMMERVILLE, 2018).

Já a implementação, não se remete apenas a escrita de código, mas também às escolhas de ferramentas, versões no ambiente de desenvolvimento, homologação e produção, linguagem de programação adotada, legibilidade do código, entre outras. Tais itens modelam a implementação de um sistema, assim como sua qualidade (SILVEIRA *et al*., 2011; PRESSMAN, 2021). É importante destacar que diversos modelos são necessários para compor e representar uma arquitetura.

Arquitetura é um outro modo de interpretar a implementação, analisando e compreendendo como uma mudança em um ponto do sistema, afeta a solução por inteiro e de todas as maneiras possíveis. Por exemplo, caso um ponto da aplicação passe a ter acesso remoto ao invés de acesso local, isso pode modificar uma outra parte da aplicação, reduzindo seu desempenho e deixando-a mais lenta. Esse tipo de análise não costuma ser feita quando se pensa no design, já que o design tem uma preocupação mais local. Desse modo, essa visão mais global do sistema, é dado o nome de arquitetura (SILVEIRA *et al*., 2011; PRESSMAN, 2021).

1.2 Principais estilos e padrões Arquiteturais de Software

A escolha de uma arquitetura resultará no atendimento à aspectos como a performance, qualidade, facilidade de manutenção e escalabilidade. Portanto, essa decisão resulta em impacto no sucesso ou insucesso do projeto, especialmente a longo prazo (SILVEIRA *et al*., 2011). Cada decisão e solução a ser criada requer um determinado tipo de arquitetura, para auxiliar nas escolhas, alguns estilos e padrões arquiteturais foram,

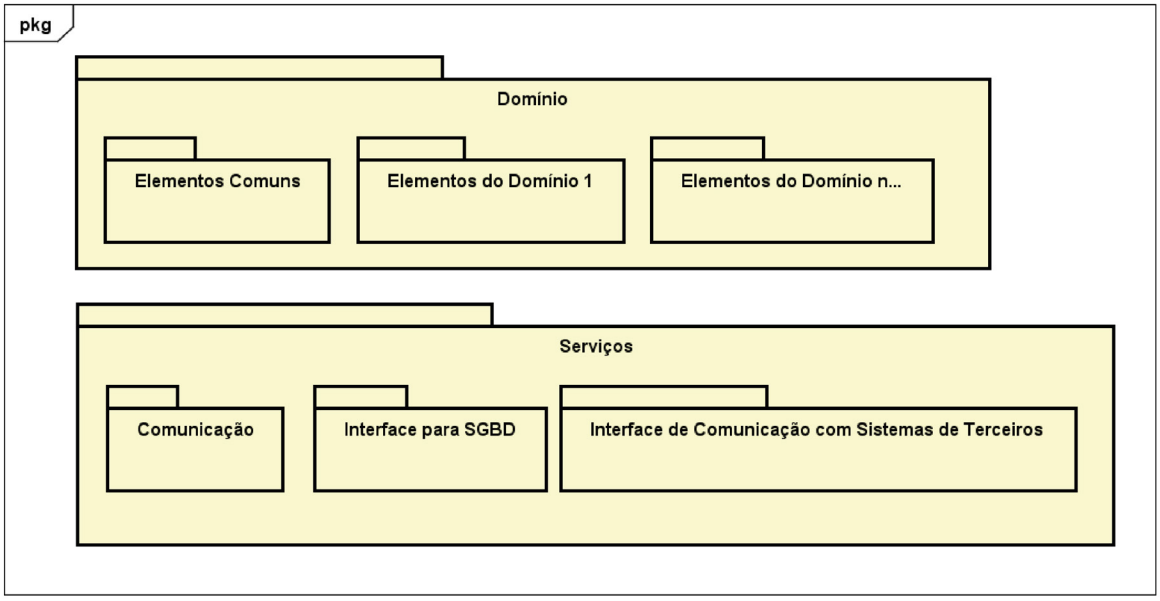
com base em sua utilização na indústria e academia, estabelecidos. A seguir tais padrões são apresentados e discutidos (SILVEIRA *et al*., 2011; PRESSMAN, 2021, SOMMERVILLE, 2018).

Arquitetura em Camadas *(Layers)*

Considerando um software, que será entrega via navegador (web), este tipo de arquitetura em camadas estrutura os elementos a serem integrados no produto final em camadas, que se interligam por meio de interfaces bem definidas, permitindo a modificação e adaptação das camadas, ao invés de uma manutenção inteira do software. Logo, este padrão favorece a criação de soluções como *market places* on-line, arquitetura de máquinas virtuais, interfaces de programação de aplicação (APIs).

Cada uma das camadas deverá ser criada estruturalmente, pensado em conjuntos específicos de funcionalidades. Geralmente, tais funcionalidades terão domínios e comportamentos que se assemelham, para que possam estar integrados em uma camada específica. O estabelecimento de tais camadas deve seguir a descrição dos dados, especificações de requisitos e outros elementos estabelecidos na análise e projeto para favorecer camadas com baixo acoplamento entre si, mas com forte acoplamento internamente, em cada camada, além de com boa coesão. A Figura 2 apresenta um modelo de arquitetura de software em camadas.

Figura 2 – Modelo de Arquitetura em Camadas



Fonte: elaborada pelo autor.

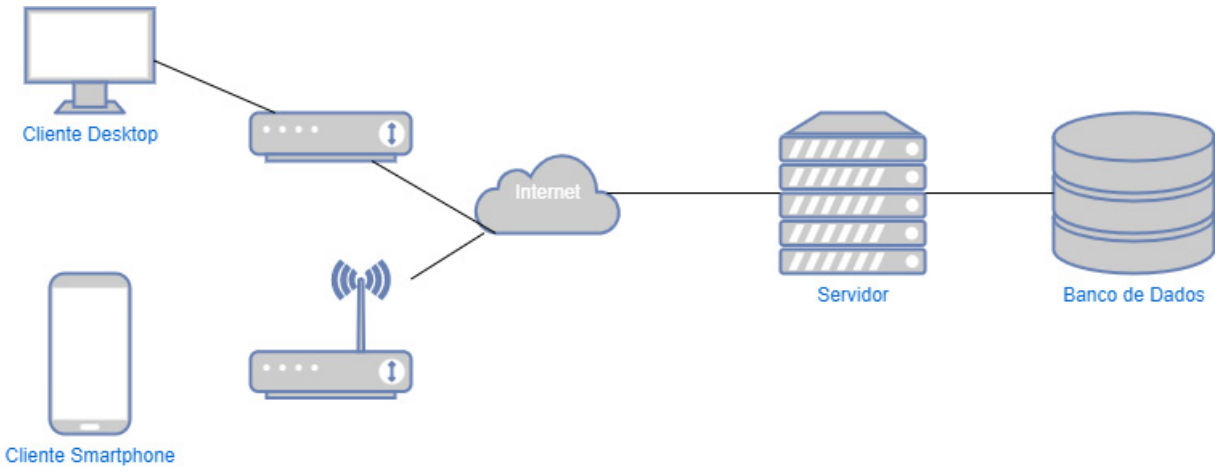
Há duas camadas na Figura 2, uma de domínio e outra de serviço. Cada uma delas possui seus pacotes internos, que representam funcionalidades a serem consideradas e integradas na aplicação. Logo, temos camadas verticais e as partições horizontais. Estas últimas são internas às camadas. Apesar da divisão, ambas as camadas mantêm comunicação e integração, a compor o software final.

Arquitetura cliente-servidor *(client-server)*

É um dos estilos arquiteturais mais conhecidos. Nesta arquitetura, temos a parte cliente, que é responsável por fornecer a informação ao usuário e a parte servidor, que é responsável por tratar as requisições vindas do cliente, manipular os dados de acordo com as requisições, atendendo-as. É um estilo comumente utilizado em aplicações e aplicativos para bancos, sistemas de gerenciamento de e-mails, e sistemas de informação web (aplicações web em geral). A Figura 3 apresenta um esquema de uma arquitetura básica de modelo cliente-servidor.

O processamento de informações e funcionalidades estão divididos em módulos que integram o produto final, esses módulos, em arquiteturas modernas, acabam sendo considerados componentes. Considerando soluções web ou móveis, a interface é executada no cliente, enquanto as funcionalidades que enviam ou recebem informações são executadas no servidor, que pode, por sua vez, estar dividido em mais de um servidor ou máquina virtual, dependendo da demanda.

Figura 3 – Modelo de Arquitetura cliente-servidor



Fonte: elaborada pelo autor.

Arquitetura Modelo-Visão-Controle *(Model-view-controller*–MVC)

O padrão MVC, como o nome sugere, divide o software em três camadas com responsabilidades bem definidas e independentes, sendo: o modelo, que permite o tratamento com a lógica de dados (busca de dados, validação para inserção/alteração/exclusão no banco de dados etc.); a visão, que se refere à interface gráfica disponibilizada ao usuário final (telas, botões, imagens, *layouts*); e o controle ou controlador, que trata do fluxo da aplicação, de onde e para onde as informações irão, provendo facilidades para a refatoração ou manutenção do código.

O tratamento em camadas e seus papéis isolados propiciam a reutilização de componentes de cada um dos conjuntos. É um modelo que tem sido menos utilizado, mas que ainda é adotado, em especial,

para o desenvolvimento web. Seu declínio se deve a modelos mais atuais, como os de serviços e microsserviços.

Arquitetura Orientada a Serviços (SOA)

As arquiteturas orientadas a serviço foram as que deram à luz às de microsserviços. Os serviços, na SOA são de maior granularidade e concentra um número maior de funcionalidades. Contudo, estas funcionalidades estão relacionadas e integradas a fornecer funcionalidades também modulares e coesas, porém, de maior granularidade que os microsserviços.

Empresas como Amazon, Google e outras grandes empresas da tecnologia, utilizam esse modelo em seus serviços, bem como microsserviços. Desse modo, podemos ter arquiteturas híbridas, em especial, nas que estão migrando de serviços para microsserviços.

A grade vantagem de tal modelo está na facilidade em se gerir e disponibilizar os serviços é maior do que de microsserviços, que precisam ser coordenados ou orquestrados por recursos específicos para enviar ou receber dados ou requisições.

Arquitetura de Microsserviços *(Microservices)*

O modelo arquitetural se baseia nos serviços, porém, com funcionalidades menos acopladas e mais específicas. Os microsserviços são componentes que formam uma estrutura modular, sendo um dos padrões mais utilizados na atualidade, já que permite melhor escalabilidade e baixo acoplamento, sendo integrado e mantido em infraestrutura de computação em nuvem.

Adicionalmente, a utilização deste padrão permite a atualização de suas funcionalidades de modo ubíquo, ou seja, o usuário final não percebe a atualização. Isso se dá também devido a utilização de ferramentas provenientes do paradigma cultural, mais do que uma metodologia,

conhecida como DevOps. As ferramentas utilizadas no DevOps proveem a entrega contínua (CD) e integração contínua (CI) por meio de *pipelines*, que, criados com o us$ENT = CR + ONT - V$os automatizados, permitem a correção de problemas ou implantação de melhorias mesmo quanto o software esteja em uso. Plataformas de versionamento, como Github e GitLab, permitem a prática e criação de projetos menores que possam integrar CI/CD, servindo como plataformas iniciais interessantes para aprender tais práticas da indústria.

A maior dificuldade em uma arquitetura de microsserviços é organizá-los e gerenciá-los para que funcionem em sinergia, necessitando de recursos como orquestração ou coreografia.

Arquitetura *Pipes-and-filters* (PF)

A tradução literal corresponde a canos e filtros. Nessa arquitetura, algoritmos são utilizados para filtrar e tratar dados que são transformados. A transformação ocorre para tornar os dados que ali trafegam, compatíveis com os componentes do sistema computacional que os transmite.

Isso significa que trata-se de uma arquitetura no estilo linear, que, para permitir o tráfego de informações, trata para que possam ser transportadas por diferentes componentes de um sistema computacional. Esta arquitetura é amplamente utilizada no contexto de softwares, que reproduzem mídias em formatos variados e no Shell dos sistemas operacionais Linux.

Arquitetura *Peer-to-Peer* (P2P)

A arquitetura ponto a ponto, ou pessoa a pessoa, transforma computadores conectados por meio de um software em servidores. Muito similar ao conceito de *clusters* ou *grids* para processamento,

porém, o contexto é tornar cada cliente uma máquina que possa prover serviços, sem a necessidade de uma infraestrutura centralizada.

Esta arquitetura é utilizada em programas de compartilhamento de arquivos intitulado *torrent.*

Arquitetura *Publish-Subscribe* **(Pub/Sub)**

Utilizada em diversas redes sociais, a arquitetura *publicar-inscrever-se* torna possível a comunicação entre os produtores de conteúdo e os consumidores. Ao disponibilizar novos conteúdos os consumidores, que ,de algum modo, estão relacionados ao publicador, recebem notificações quando um novo conteúdo é publicado, podendo, assim, consumir tais conteúdo.

Ests modelo arquitetural requer controle entre o fluxo de seus componentes e camadas de segurança aptas a gerir um fluxo grande de usuários, demandas e consumo de informações.

A implementação e adoção de um estilo ou padrão arquitetural resulta em vários benefícios, mas, em especial, podemos destacar (SILVEIRA *et al*., 2011; PRESSMAN, 2021):

Desempenho ou performance: para estabelecer uma arquitetura que atenda às demandas vindas dos usuários e a todos eles, simultaneamente, é necessário considerar o desempenho desta, permitindo que sejam utilizadas estratégias que possam flexibilizar a disponibilização de recursos de hardware ou criar mecanismos que consumam menos recursos no processo de conexão ou requisições. Desse modo, a especificação de demandas no momento do estabelecimento de arquiteturas que precisam ser performáticas é essencial e auxiliam na redução de problemas futuros, nesta perspectiva.

Escalabilidade: garantir que um software possa ser escalável de modo rápido e preciso, facilita sua utilização e evita atrasos nos cronogramas e possíveis problemas de implementação, caso modificações não previstas precisem ser realizadas.

Flexibilidade: garantir que o software possa ser modificado de modo flexível é primordial. Assim, garantir uma arquitetura que possibilite meios de se obter tal nível de flexibilização é outra característica, que é otimizada por meio de uma arquitetura bem estabelecida.

É importante destacar que as arquiteturas de software estão sempre em evolução e novos padrões, e estilos surgem na medida em que novas tecnologias são criadas ou evoluídas. A proposta é conseguir integrar diferentes elementos, atendendo restrições de hardware e de software que se integram, a compor um software robusto e que permita manutenção e evolução facilitada, além de outros requisitos de qualidade.

Desse modo, pudemos ter uma visão geral dos diversos estilos e padrões arquiteturais e alguns de seus benefícios. O estudo destes materiais e a integração de leituras das referências utilizadas são sugeridos como meio para ampliar ainda mais o conhecimento na temática! Bons estudos!

Referências

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de software-9**. McGraw Hill Brasil, [s. l.], 2021.

SILVEIRA, P. et al. **Introdução à Arquitetura e Design de Software**: uma visão sobre a plataforma Java. São Paulo: Campus–Elsevier, 2011.

SOMMERVILLE, I. **Engenharia de Software**. 10. ed. São Paulo: Pearson, 2018.

Análise e Projeto de Arquiteturas de Software

Autoria: Anderson da Silva Marcolino
Leitura crítica: Marco Ikuro Hisatomi

Objetivos

- Contextualizar os alunos sobre os conceitos essenciais de concepção das arquiteturas de software.
- Orientar os alunos na definição de metas e escopo para projetos de arquiteturas de software.
- Aprofundar nos padrões arquiteturais, considerando domínios específicos.

1. Introdução à análise e projeto de Arquitetura de Software

Ao considerarmos as atividades envolvidas e definidas na Engenharia de Software para a concepção de produtos de software, temos a necessidade de conduzir diferentes atividades, geralmente, integradas em cinco processos ou etapas (SOMMERVILLE, 2018; PRESSMAN, 2021):

1. Análise: corresponde a analisar o modelo de negócios e os requisitos (funcionais, não funcionais e de domínio) para permitir o entendimento e alinhamento com os *stakeholders* (as partes envolvidas), permitindo, ainda, a criação de artefatos iniciais para o projeto, como diagramas de casos de uso, de classes em uma visão macro, que estão na borda de transição entre análise e projeto. Todos estes modelos serão fundamentais para criar a arquitetura, na próxima etapa.
2. Projeto: aqui, temos a transformação dos artefatos criados na fase anterior, em especial, o documento de requisitos e diagramas iniciais que permitirão o estabelecimento de diferentes diagramas, como os de classes, de interação, de modelagem de dados, entre outros que, ao final, refletirão a arquitetura de software da solução a ser desenvolvida. Enquanto isso, na análise, temos uma limitação quanto às tecnologias adotadas e, nesta etapa, as tecnologias são definidas porque influenciam significativamente na estrutura e padrões arquiteturais adotados. Finalizada esta etapa, a etapa de desenvolvimento, pode ser conduzida.
3. Desenvolvimento: nesta etapa, desenvolvedores e desenvolvedoras, programadores e programadoras, guiados por engenheiros e engenheiras de software, trarão à luz os softwares, implementando-o com base nas especificações do projeto, ou seja, com base na Arquitetura de Software definida. A especificação garante o atendimento aos requisitos, melhor qualidade do software, pois o desenvolvimento se dirige ao que o cliente quer

de fato, comunicação nivelada, padrões e boas práticas para garantir que a equipe possa trabalhar de modo homogêneo e, assim, também resultar em melhor qualidade do produto e documentação, que será extremamente importante para a evolução do software.

Ao implementar componentes, unidades ou módulos, de acordo com a técnica de desenvolvimento adotada e devidamente utilizada na especificação da arquitetura, segue-se, então, para os testes. A interação entre a fase de desenvolvimento e testes é contínua e ocorre, muitas vezes, durante o processo de desenvolvimento, pois que garante a verificação e validação do que está sendo criado.

4. Testes: nos testes, temos a verificação e validação do que está sendo desenvolvido. A verificação garantirá a consistência, completude e corretude do produto em cada fase e entre as fases consecutivas do ciclo de vida do software. Apesar de não fazer parte da arquitetura, utiliza para embasar em que componentes ou elementos estruturais os testes unitários e de integração ocorrerão. É também durante a verificação que se confere se os métodos e processos de desenvolvimento foram aplicados adequadamente. Comumente perguntamos, nesta etapa, se estamos construindo certo o produto. Já na validação, assegura-se que o produto final atenda aos requisitos do usuário, em outras palavras, se faz aquilo que foi solicitado a fazer. Desse modo, como pergunta-chave para esta atividade, temos: o produto atende ao que foi solicitado pelo cliente. Integrado às atividades de verificação e validação, temos, então, a análise estática, que não considera a execução do software em si, e a análise dinâmica, que envolve a execução do produto e sua análise em termos de código e de produto funcional, sendo que ambas são complementares. Finalmente, após a execução das diferentes tarefas, o software atinge nível aceitável e pode, então, ser implementado no ambiente de produção para uso, e, neste

contexto, chega-se à fase de manutenção, que também é chamada de evolução.

5. Manutenção: nesta etapa, o software já está em uso pelo cliente. Consequentemente, problemas não detectados durante as fases de desenvolvimento e testes, ou simplesmente a evolução natural do software, é necessário para atender novas demandas, ainda, a etapa em que entra a refatoração do código e a integração de novas funcionalidades e rotinas. Esta fase, como mencionado anteriormente, terá um sucesso maior se puder contar com uma documentação de software bem-feita, incluindo comentários no código e um projeto arquitetural que reflita, de fato, tudo que foi implementado e como foi testado, já que os testes deverão ser realizados novamente para verificar se qualquer nova alteração não gerou algum problema ao que já foi desenvolvido e está em uso pelo cliente. Quando em uso, o que o cliente quer é a menor ocorrência de interrupções ou novos problemas decorrentes de novas implementações e isso depende, em grande parcela, de como o software foi produzido e documentado até aqui.

Agora que temos uma visão geral das fases de concepção de um software e onde a arquitetura é concebida, bem como seu papel nas demais etapas, aprofundaremos um pouco mais na análise e projeto e em domínios específicos, metas e escopos, no contexto arquitetural.

1.1 Análise e projeto de Arquiteturas de Software

A análise integra atividades que permitirão definir elementos relevantes do domínio do problema para o qual o software será criado. Neste contexto, analisar os requisitos e verificar suas especificidades, identificando se podem ser atendidos, e também se fazem sentido no contexto do domínio, são atividades executadas nessa fase.

O refinamento dos requisitos permitirá que o projeto a ser criado, as tecnologias e os elementos que comporão a arquitetura do software, sejam mais precisos, garantindo a redução de refatoração ou alterações tardias. Uma alteração ocasionará em maiores custos e necessitará de mais esforços, se ocorrerem em etapa posterior às etapas de desenvolvimento e testes, por isso, a análise é fundamental.

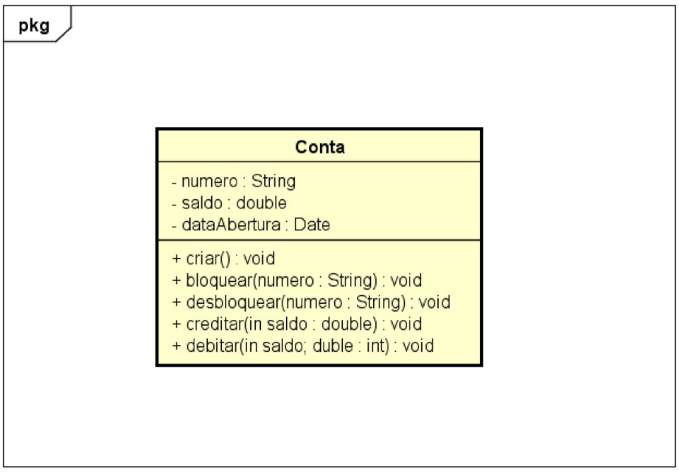
De posse dos requisitos detalhados e precisos, parte para a criação de visões estruturais que são apresentadas em diagramas da Linguagem de Modelagem Unificada (UML). A UML foi criada para representar elementos de software orientados a objetos. Como o paradigma de programação orientada a objetos favorece, em especial, a reutilização, e se torna essencial no contexto do desenvolvimento de software na atualidade. No contexto da etapa de análise, a UML é empregada para se apresentar visões de análise, sendo (FILHO, 2019):

- Estrutural: representa as classes, seus relacionamentos, atributos, operações, relacionamentos e diagramas, com suas descrições estáticas.

- Comportamental: representa como as classes colaboram entre si para realizar as funcionalidades do software. Está relacionado com as funcionalidades que são expressas também nos diagramas de casos de uso, criados em especial, para o refinamento de definição das interações dos usuários com o sistema, permitindo, assim, o estabelecimento mais detalhado dos requisitos.

Como a análise provê a modelagem do domínio do problema, ao utilizar da UML, temos a representação das entidades envolvidas, em um diagrama de classes. A Figura 1 apresenta uma classe, seus atributos e métodos.

Figura 1 – Diagrama de classe



Fonte: elaborada pelo autor.

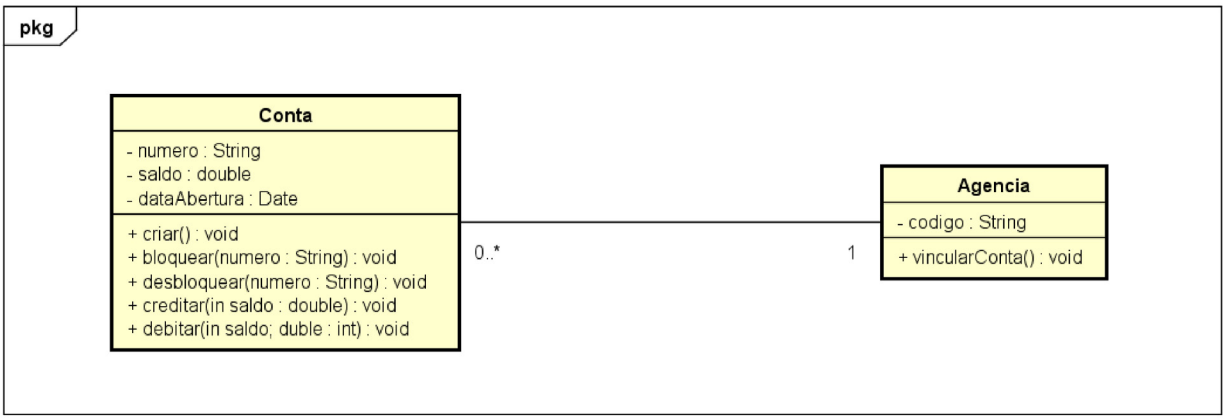
Adicionalmente, ainda na Figura 1, temos um outro conceito relacionado à orientação a objetos: o encapsulamento, que, como um dos pilares da orientação a objetos, provê a segurança e visibilidade de atributos e métodos, que serão depois, traduzidos na linguagem de programação. Na Figura 1, os atributos estão com o símbolo “-” na frente e os métodos, com o símbolo “+”. O primeiro indica que os atributos só são vistos na própria classe, ou seja, são privados (*private*), enquanto os métodos, podem ser vistos em todo o software sendo, portanto, públicos (*public*). Ao final, o encapsulamento garante que os valores de um objeto não possam ser modificados de qualquer modo, sem um controle previamente estabelecido e programado nos métodos.

Considerando a fase de análise, o diagrama de classes pode estar em estágios preliminares e, em algumas literaturas, são implementados completamente somente na fase de projeto. Para facilitar a concepção e identificação das classes, indica-se criar uma tabela com nome da classe, suas responsabilidades e colaborações. A este processo, dá-se o nome de cartões CRC, que corresponde a Classes, Responsabilidades e Colaborações.

Quando de posse das classes mais importantes, já começamos a nos deparar com o escopo do software se formando. Temos maior noção,

em termos de projeto, do que deverá ser implementado, podendo começar a estimar o tamanho do time a trabalhar no desenvolvimento. Adicionalmente, começamos a identificar os relacionamentos entre as classes, e assim como a cardinalidade no banco de dados, nas associações entre as tabelas, temos a multiplicidade nas associações entre as classes. A Figura 2 apresenta um exemplo de associação e multiplicidade entre classes. Estes relacionamentos ajudarão a definir o estágio comportamental, já que precisamos colocar as classes e, posteriormente, os objetos instanciados, para realizar a troca de mensagens.

Figura 2 – Diagrama de classe com associação



Fonte: elaborada pelo autor.

Ainda temos outras relações de multiplicidade: 1 que indica apenas um; 0..*, que indica zero ou muitos. 1..* que indica um ou muitos; 0..1 que indica zero ou um e 1..1s, que indicam intervalos específicos. Tais regras permitirão ao desenvolvedor estabelecer a relação, via linguagem de programação, dos objetos que serão instanciados.

Estabelecidos tais diagramas, podemos seguir então para a fase de projeto, que também pode ser chamada de design. É importante especificar que alguns autores traduzem o termo design para desenho, podendo levar a certa confusão na hora de seu entendimento. Desse modo, por padronização e facilidade do

entendimento do termo, este texto adota o termo projeto ao invés de design.

Na fase de projeto, há como objetivo criar uma estrutura que seja implementável para o produto de software. Tal estrutura deve estar fundamentada nos artefatos providos pela fase de análise e deve considerar (FILHO, 2019):

- Atendimento aos requisitos funcionais, não funcionais e de domínio e as inter-relações. Não é possível estabelecer o que será implementado, se não estiver alinhado especificamente com o que se deseja. Na inobservância dos requisitos, não há como validar o produto na fase de testes.

- Modelagem de diagramas de classes e outros elementos com detalhes para o desenvolvimento, na fase posterior.

- Decomposição do produto em componentes, de modo que a equipe de desenvolvimento possa desenvolver cada um deles, preferencialmente em paralelo e que estes possam ser testados de modo também independente (teste de unidade). Ao final, o produto é submetido ao teste de integração que permitirá a entrega ou não do produto para uso (produção), pelo cliente.

- Especificação das interfaces e integração destas com os componentes que estas inferem. Nesta perspectiva, os diferentes componentes estabelecidos deverão ter as especificações de quais interfaces estão relacionados, possibilitando que executem o comportamento esperado e previsto no contexto de executarem suas funções esperadas.

- Criação da documentação que será repassada nas demais etapas e será mantida durante todo o ciclo de vida do software.

- Análise, seleção e reutilização de componentes já existentes. Nesta etapa, considerando que estamos falando de uma

empresa de software já consolidada, o reuso é obrigatório para reduzir custos, prazos e esforços. Assim, é nela que componentes ou outros artefatos são analisados, selecionados e integrados ao projeto e sua documentação.

- Na condução do projeto de modo mais automatizado e preconizado, com a evolução dos métodos de desenvolvimento e testes de software, como a cultura DevOps, que prevê a utilização de diferentes ferramentas para permitir a entrega contínua e melhoria contínua, é também na fase de projeto que se identificam e empregam métodos e ferramentas para a geração semiautomática de código. Ferramentas intituladas *Computer Aided Software Engineering* (CASE), mais robustas, são capazes de realizar, por meio da engenharia reversa, a criação de estruturas de código de modo semiautomático. Por exemplo, o diagrama da Figura 2 poderia ser exportando, gerando o código das classes que representa, considerando seus métodos e atributos.

Desse modo, no projeto parte-se para a criação de modelos de solução, que são derivados do modelo do problema. Ambos são parte integrante, ao final, da Arquitetura de Software.

Enquanto no modelo de problema temos a especificação mais textual e não técnica, no modelo da solução precisamos, obrigatoriamente, tratar a solução no contexto tecnológico que será implementado, garantindo descrição técnica, modelo físicos para implementação, diagramas específicos para a implementação, diagramas e modelos diversos (componentes, de redes, telas, banco de dados etc.), pacotes lógicos especificados e distribuídos em camadas, sendo todos em nível mais formal e detalhado.

Se na análise temos uma representação mais macro, aqui, temos uma visão mais micro do problema, logo, conseguimos observar e identificar elementos que são fundamentais e servirão massivamente

para a criação da arquitetura do software e, consequentemente, a concretude do projeto em um produto funcional.

Desse modo, é com a união das diferentes representações que teremos de fato a arquitetura do software. Logo, a representação de uma arquitetura não consegue ser abrangente e única, por meio de um artefato apenas. As diferentes especificidades técnicas para descrever sobre os hardwares, as redes, os softwares de terceiros, os sistemas operacionais, entre outros, obrigatoriamente demanda por um tratamento específico uma representação adequada para cada uma delas. Assim, várias perspectivas são criadas para aumentar a compreensão aos envolvidos no processo de desenvolvimento e testes.

É como fazer uma analogia a um manual de montagem de um modelo com peças de montar. Se tivermos apenas imagens, ainda faltarão detalhes que possam permitir a montagem. Se tivermos apenas textos, também continuaremos com dificuldades para criar o modelo final. Entretanto, com as duas representações, gráfica e textual, teremos sucesso na montagem.

Dessa forma, no contexto de uma arquitetura de software e sua representação, cada perspectiva refletirá em diferentes partes para permitir a criação concisa do software. Ao final, temos o modelo de solução, que delimitam e definem a fronteira entre o sistema e seu ambiente, gerando a especificação de duas vertentes de projeto ou desenho (MARTIN, 2019; FILHO, 2019):

1. Projeto externo: tratará das interfaces que o sistema oferece ao ambiente, não se restringindo a interfaces gráficas para os usuários, mas a comunicação com outros sistemas. Um exemplo clássico, são as interfaces de programação de aplicação (APIs) dos correios para obter o Código de Endereço Postal (CEP) ou até mesmo as que o GitHub, ou plataformas

de desenvolvimento, oferecem aos seus usuários, para que possam praticar o desenvolvimento.

2. Projeto interno: tratará da organização interna do produto, seus componentes que estarão em colaboração para que possam realizar o comportamento esperado por eles, inclusive, com os de diferentes interfaces existentes.

Nesse contexto, temos, ainda, no projeto a representação, considerando o nível de abstração e como os elementos do produto de software estão agregados (MARTIN, 2019; FILHO, 2019):

- Nível estratégico ou desenho arquitetônico, que trata da representação da arquitetura do sistema, representa os conceitos ou propriedades de um sistema em seu ambiente, representados e explicitados em seus elementos, relacionamentos e associações, considerando princípios de projeto e evolução.

- Nível tático, que se refere ao projeto lógico, tratando da aplicação e decisões arquitetônicas na organização de componentes e reutilização de componentes envolvidos, alinhando as funcionalidades às funções, estrutura e comportamentos esperados pelo software.

- Nível operacional ou projeto detalhado, que tratará especificamente de cada componente e de sua interface. É concebido ao passo em que a documentação para os usuários também o é, no contexto do projeto externo, e em nível de código, considerando o projeto externo.

Diante de tais modelos e subdivisões para representação, pode-se definir o escopo final do projeto e traçar a equipe e sua estratégia para o desenvolvimento, de fato, do software almejado. Contudo, é importante mencionar que, dependendo do domínio, temos foco em mais elementos do ambiente do que internamente ao produto.

Por exemplo, em um sistema de *e-commerce*, temos que dar ênfase à segurança com a transação com sistemas de terceiros (bancos), a segurança de tais dados e transações, bem como a restrição de acesso a elementos, por usuários. Diferentemente de um sistema voltado a disponibilização de videoaulas.

A diferenciação dos domínios leva a um outro aspecto importante, que deve ser considerado, entre outros vários, na análise e projeto: o princípio de abstrações estáveis (*Stable Abstractions Principels*–SAP*),* essencial quando se trabalha de componentes (MARTIN, 2019).

Por meio de tal princípio, o escopo, em termos de componentes, será melhor definido. Tal princípio fornece métrica para estabelecer uma relação entre estabilidade e abstração, pois todo produto de software terá partes de alto nível, que sofrerão poucas alterações na medida que são desenvolvidas ou depois, na manutenção, modificadas e, também, as partes de baixo nível, consideradas as partes menos estáveis e que sofrerão mudanças com maior rapidez.

O SAP basicamente indica que um componente estável deve também ser abstrato para que essa estabilidade não impeça sua extensão. Por outro lado, afirma que um componente instável deve ser concreto, já que sua instabilidade permite que o código concreto dentro dele seja facilmente modificado (MARTIN, 2019). Desse modo, para que um componente seja estável, deve consistir de interfaces e classes abstratas de modo que possa ser estendido. Componentes estáveis e extensíveis são flexíveis e não restringirão à arquitetura, permitindo, assim, sua manutenção e evolução de um modo melhor. Logo, tal princípio é fundamental e deve ser empregado como meio de estabelecer métricas, em especial, na fase de projeto.

Finalizamos, dessa forma, a Leitura Digital, que deve ser complementada com mais leituras sobre o tema, em especial, considerando as referências utilizadas! Bons estudos!

Referências

FILHO, W. de P. P. **Engenharia de Software – Produtos**. 4. ed. Rio de Janeiro: LTC, 2019.

MARTIN, R. C. **Arquitetura Limpa:** o guia do artesão para estrutura e design de software. Rio de Janeiro: Alta Books, 2019.

PRESSMAN, R. S.; MAXIM, B. R**. Engenharia de software**. McGraw Hill Brasil, [s. l.], 2021.

SOMMERVILLE, I. **Engenharia de Software**. 10. ed. São Paulo: Pearson, 2018.

Modelagem de Arquiteturas

Autoria: Anderson da Silva Marcolino
Leitura crítica: Marco Ikuro Hisatomi

Objetivos

- Contextualizar os alunos sobre os conceitos essenciais da modelagem de arquiteturas considerando diferentes representações.
- Orientar os alunos para a remoção de ambiguidades, de modo a garantir maior precisão nos modelos, atingindo nível de formalidade suficiente.
- Aprofundar-se nos *frameworks* e automação para a concepção de modelos arquiteturas de sucesso.

1. Iniciando a modelagem de Arquiteturas de Software

As etapas de análise e de projeto de software são fundamentais para a concepção dos modelos de uma arquitetura de software. Os artefatos analisados e, inicialmente, projetados na etapa de análise são levados a um nível mais detalhado na fase de projeto, permitindo a criação dos modelos especializados que representarão a arquitetura e, assim, permitir aos desenvolvedores a criarem e posteriormente testá-la.

Para começarmos a modelar uma arquitetura, precisamos ter em mente qual seu foco principal. Dificilmente uma arquitetura de um sistema seria representado por um único modelo de diagrama, pois a visão de elementos de hardware, de domínio, de funcionalidades e das interações só poderão ser representadas por meio de diagramas distintos. A seleção de um foco principal levará a seleção de diagramas que fazem sentidos serem modelados, ao passo que outros diagramas serão excluídos, por não estarem relacionados com o foco da solução a ser criada (SOMMERVILLE, 2018; PRESSMAN, 2021).

Entre as categorias, temos a de comunicação, implantação (*deployment*), domínio e estrutural. É nessas categorias que temos vinculados os diversos estilos ou padrões arquiteturais. Para exemplificarmos a concepção dos modelos arquiteturais, considerando a diversidade de estilos e finalidades, utilizaremos uma das categorias que tem sido mais destacadas no contexto das indústrias de desenvolvimento de software: a categoria de comunicação e o estilo arquitetural orientado a serviços e microsserviços. A solução a ser modelada será uma aplicação web para o registro de filmes e séries que um usuário tenha assistido.

1.1 Definindo aspectos de precisão e estabelecendo formalidades

Apesar dos diagramas da arquitetura de software serem partes importantes para a leitura e desenvolvimento do produto, aspectos textuais e técnicos, como as estratégias de se organizar a arquitetura são fundamentais. Muitas vezes, após esses aspectos textuais serem definidos, são representados por meio de desenhos ou diagramas.

Nessa perspectiva, pontos importantes antes da modelagem devem ser considerados. Esses pontos podem ser apelidados de princípios da arquitetura, apoiando a remoção de ambiguidades, ou seja, a repetição ou interpretação dupla de especificações; o que resulta em maior precisão. Adicionalmente, tornam-se as representações mais formais, ou seja, deixam de seguir aspectos de uma pessoa ou pequeno grupo para ser representada considerando técnicas pré-estabelecidas e bem difundidas, técnicas comuns entre a comunidade de desenvolvimento.

Ao adotar uma ou mais técnicas, que, geralmente, tais técnicas não são prescritivas, o desenvolvedor é capaz de interpretar com maior facilidade os diagramas e textos, garantindo a implementação do produto e em consonância com os requisitos (aquilo que os *stakeholders* esperam).

Como princípios arquiteturais que devem ser considerados na hora das decisões técnicas e posteriores modelagens, podemos citar (SOMMERVILLE, 2018; PRESSMAN, 2021, MARTIN, 2019):

- Criação de estruturas flexíveis, que podem ser modificadas facilmente.

- Redução de riscos, por meio do uso de recursos tecnológicos que removam ambiguidades e dinamizem o processo de modelagem e verificação de consistência de tais modelos, como exemplo a Linguagem de Modelagem Unificada (UML).

- Utilizar modelos e representações gráficas como meio de facilitar a comunicação e colaboração entre as equipes. Se todos conhecem os modelos e diagramas, todos falam a mesma língua.

- Crie uma arquitetura de modo incremental e iterativa, ou seja, faça aos poucos (incrementos) e a expanda, a cada nova iteração. Essa prática pode ser agregada não somente à concepção da arquitetura, mas permite a identificação de erros, análises mais detalhadas e o descobrimento de problemas mais cedo, possibilitando menos esforços para tornar a arquitetura precisa e com nível de formalidade esperado.

Existem, ainda, aspectos de qualidade que devem ser considerados, que baseados em modelos de qualidade, permitem integrar e tratar aspectos de cunho técnico para garantir a redução de custos, o atendimento aos requisitos, a criação de uma arquitetura capaz de ser extensível e que garanta os princípios de usabilidade, desempenho e manutebilidade.

1.2 Modelando nossa arquitetura

A aplicação que modelaremos é, em sua essência, simples, mas seus aspectos permitirão entender e visualizar a aplicação de princípios arquiteturais e, também, princípios inerentes ao processo de projetar tais arquiteturas. Nessa perspectiva, o objetivo é desenvolver uma aplicação capaz de criar um catálogo de filmes e séries.

A aplicação de catálogo deverá registrar filmes e séries em uma lista do que foi assistido ou do que se pretende assistir. Tais registros estarão vinculados à um usuário que deve efetuar *login* de acesso na aplicação, para poder visualizar, incluir, alterar ou excluir registros em seu catálogo. Usuários não poderão visualizar os registros de outros usuários.

A interface gráfica de usuário será híbrida, ou seja, responsiva, que se adaptará ao tipo de dispositivo que está usando (e.g. *smartphones*, computadores, *tablets*). Em termos de qualidade, deve garantir fácil utilização, acesso, que mantenha as informações em segurança e que esteja sempre disponível. Um outro requisito importante é que a aplicação seja leve para ser utilizada com o menor consumo de banda da Internet, ou pacotes móveis.

É importante indicar que podemos seguir modos específicos de representar uma arquitetura, sendo: i) a UML; ii) modelo de visão da arquitetura; e iii) linguagem de descrição de arquitetura (ADL). A UML é o meio mais conhecido, mas temos a utilização do modelo de visão de arquitetura, em especial o intitulado 4+1, que representa as visões lógicas, de processo, de desenvolvimento, física e a visão "+1" de cenários específicos e de alto nível, para discussão com os *stakeholders*.

Finalmente, uma ADL é uma linguagem que provida de elementos sintáticos e semânticos permite representar uma arquitetura de modo completo e permite, ainda, por meio de engenharia reversa, extrair e criar estruturas de código, como as classes, por exemplo, para o desenvolvimento. A grande limitação se deve ao fato de que uma ADL, geralmente, é integrada a um software, mas destaca-se por permitir a aplicação de métricas e automatização de validações arquiteturais oferecidas pelas próprias ferramentas.

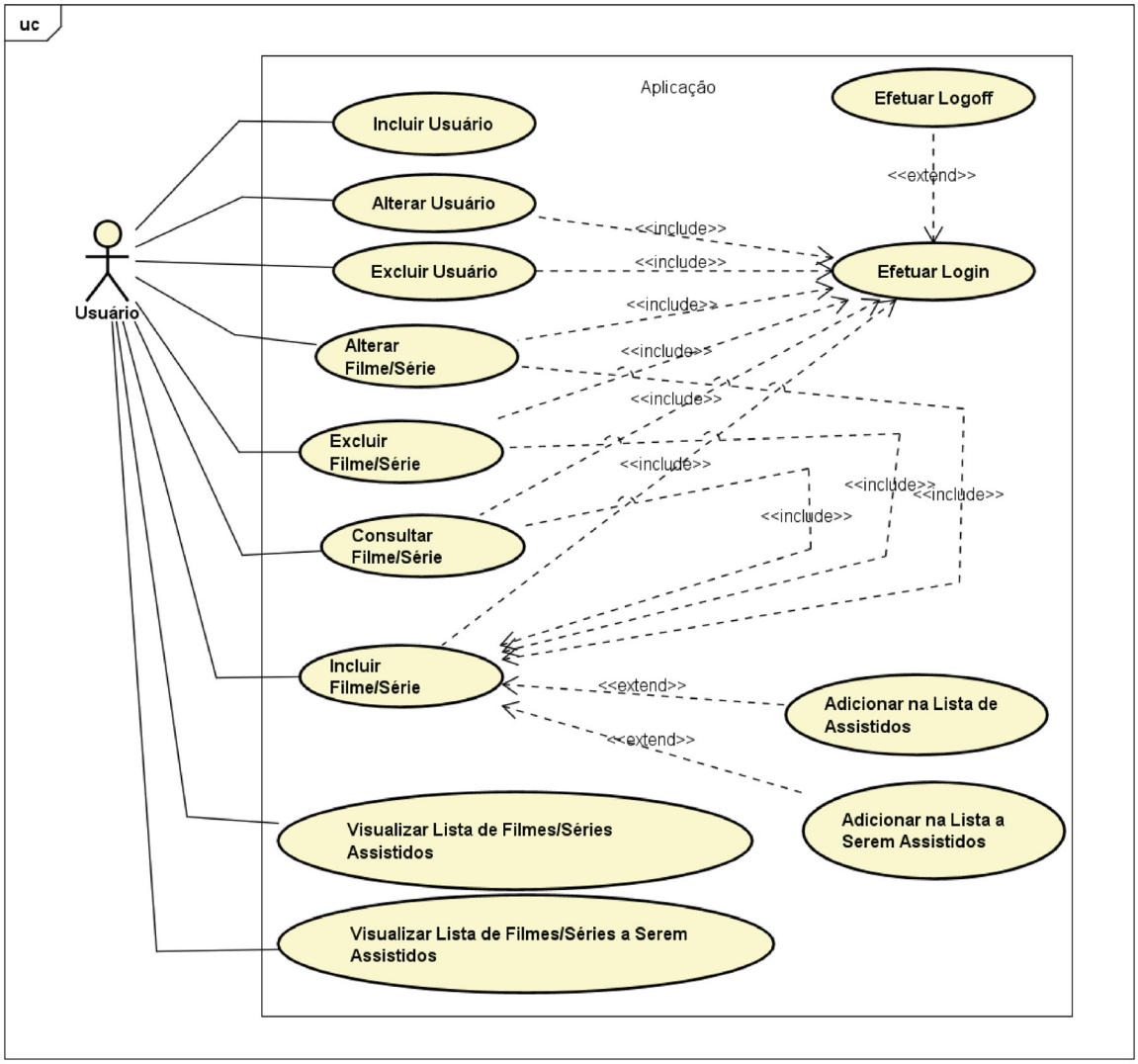
Destaca-se, ainda, a Linguagem de restrição de objeto (OCL), do termo em inglês *Object Constraint Language*, que integra e se aplica aos modelos da UML e também possibilitam a verificação automatizada de regras de derivação e especificação dos diagramas, porém, não é considerada uma ADL, por suas características. Além disso, ferramentas robustas de modelagem UML permitem o processo de engenharia reversa, permitindo a geração de código para ser utilizado no desenvolvimento.

Dessa forma, considerando a especificação da aplicação, considera-se que a etapa de análise dos requisitos está concluída, permitindo seguir

para a modelagem do diagrama de casos de uso, que apresenta uma das primeiras visões comportamentais da aplicação, em que as relações das funcionalidades do sistema recebem interações de atores externos.

Como o diagrama de caso de uso não expressa decisões técnicas, mas permite verificar as interações de usuários, representados por atores, com as diferentes funcionalidades do sistema, temos a possibilidade de um refinamento junto aos *stakeholders* dos requisitos e das funcionalidades que cada um dos casos de uso representa. A Figura 1 apresenta o digrama de caso de uso geral da aplicação.

Figura 1 – Diagrama de casos de uso geral: aplicação de catálogo de filmes e séries



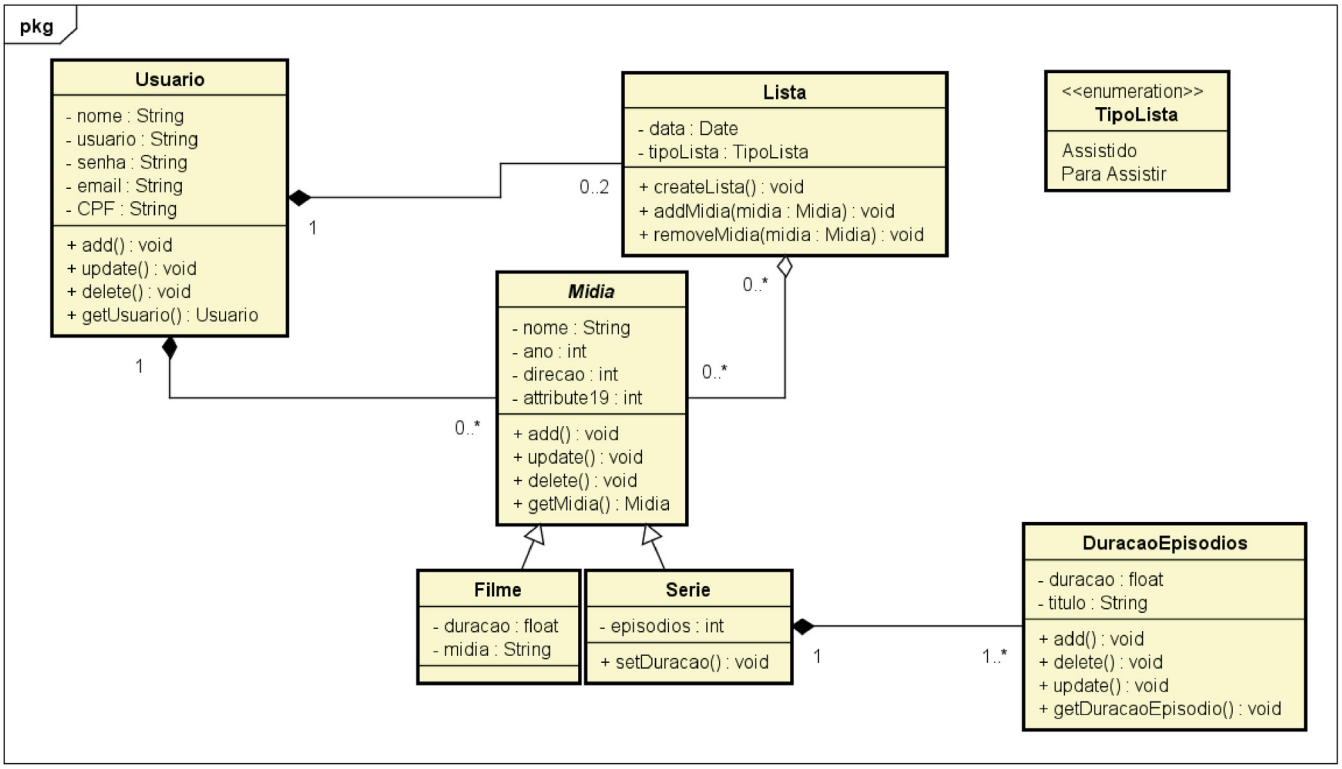
Fonte: elaborada pelo autor.

Um diagrama de casos de uso pode ser apresentado em uma visão geral ou em visões específicas. Cada diagrama possui uma descrição de caso de uso que serve para complementar e apresentar o fluxo normal, alternativo ou de exceção em que ocorrem as interações dos atores com as funcionalidades do sistema. O ator, na Figura 1, é representado pelo boneco de nome Usuário, e apresenta quem faz a interação com a aplicação.

Na Figura 1, há a representação de treze diagramas de casos de uso, sendo que alguns deles levam, obrigatoriamente, a interação com outros casos de uso, por meio de um relacionamento estereotipado com <<include>>, enquanto outros casos de uso levam a outros casos de uso, de modo opcional, é o caso do caso de uso *Efetuar login*, que pode estender para efetuar ou não o log off, visto que o usuário pode simplesmente fechar a aplicação, sem de fato pressionar o botão para *Efetuar Logoff*. Um relacionamento opcional é estereotipado por <<extend>>.

Com base nos outros métodos para representação de arquiteturas, o diagrama de caso de uso é equivalente a visão de cenários do Modelo de Visão 4+1, visto que se aproxima muito mais de uma representação direcionada ao cliente, do que ao desenvolvedor. Ainda que o diagrama de casos de uso apresente uma visão técnica, suas definições auxiliam a definir o grupo principal de classes. Desse modo, segue para a representação de um dos primeiros diagramas estruturais, o diagrama de classes, apresentado na Figura 2.

Figura 2 – Diagrama de classes – Aplicação de catálogo de filmes e séries



Fonte: elaborada pelo autor.

O diagrama de classes apresenta as classes que permitirão a instanciação dos objetos do sistema. Há a presença de relacionamentos de composição (linha com losango preenchido na cor preta em uma das pontas). A composição indica que a classe relacionada é parte de um todo e que, se o todo deixar de existir a parte também não existirá. Nesse contexto, sem um usuário não é possível existir a lista e as mídias.

Temos, ainda na Figura 2, a presença de uma enumeração, com valores a serem preenchidos no atributo *tipoLista*, da classe Lista, e a herança entre as classes *Filme* e Serie em relação à superclasse *Midia* que é abstrata. Para cada classe, temos os métodos que refletirão na troca de mensagens das classes. Note que classes referentes a aplicação, como menus de acesso ou transações com banco de dados que foram mantidas fora do diagrama. Considerando as etapas de concepção

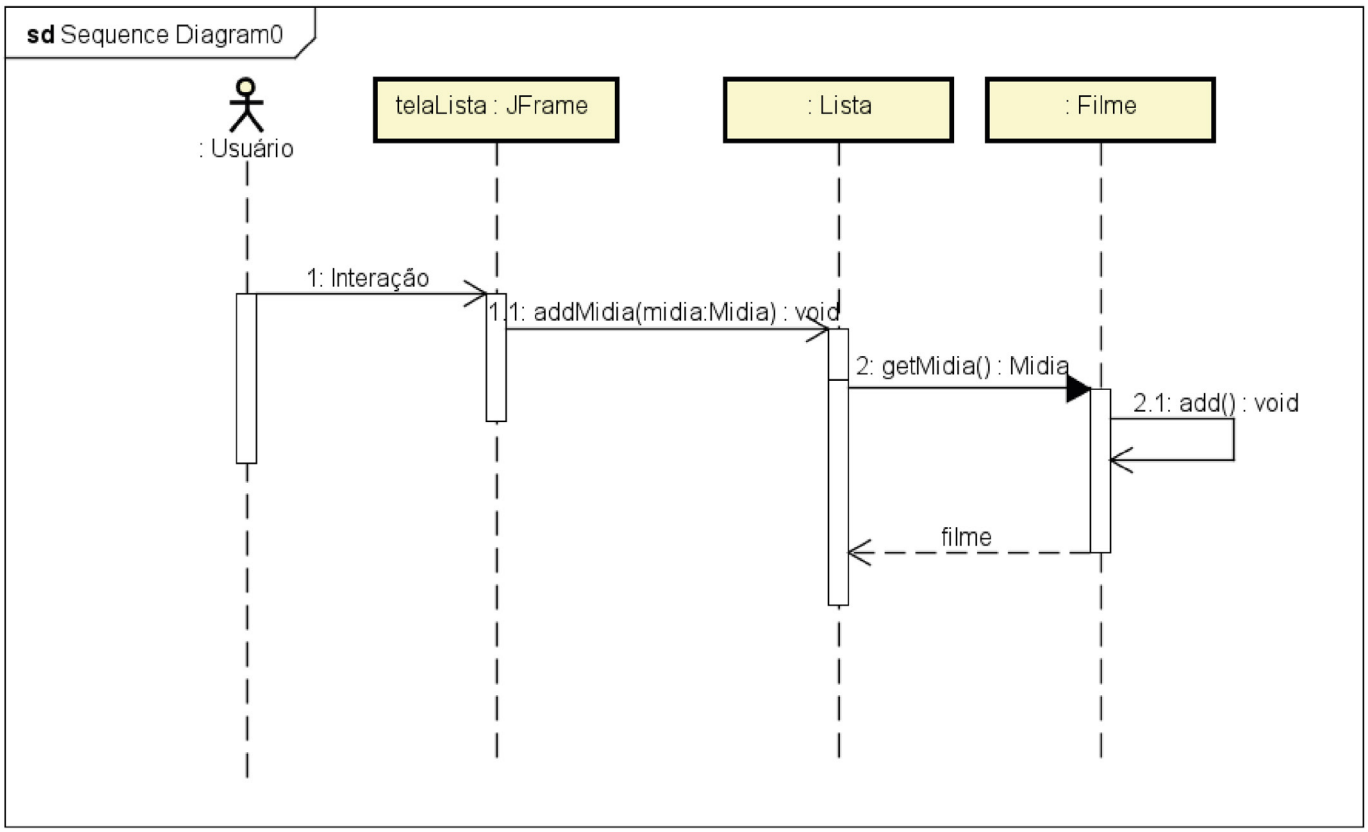
de uma arquitetura, estamos seguindo as definições temporais e específicas para cada fase.

Considerando os diagramas estruturais existentes, a próxima representação seria a de objetos. Contudo, tal representação acaba por ser similar à de classes, que se diferencia apenas por mostrar, ao invés de classes, objetos instanciados e o preenchimento de valores de seus atributos.

Assim, os próximos diagramas a serem considerados são os diagramas de atividades e o de estados, que está incluído no conjunto de diagramas comportamentais, junto com o de casos de uso. Contudo, estes diagramas comumente não trazem contribuições tão efetivas para o projeto da arquitetura de software, podendo ser representados, somente quando estritamente necessário ou quando os demais diagramas não conseguem representar especificidades arquiteturais. Logo, o diagrama de comportamento a ser modelado passa a ser o diagrama de sequência.

Os diagramas de sequência apresentam as chamadas do sistema para executar funcionalidades específicas. Logo, para uma representação completa das sequências de ações que as classes executam, seria necessário um diagrama para cada operação das classes apresentadas na Figura 2. Por questões didáticas, apresentamos apenas um diagrama, que permite o entendimento completo do que tais diagramas representam, como pode ser visto na Figura 3.

Figura 3 – Diagrama de sequência – Adicionar filme em lista



Fonte: elaborada pelo autor.

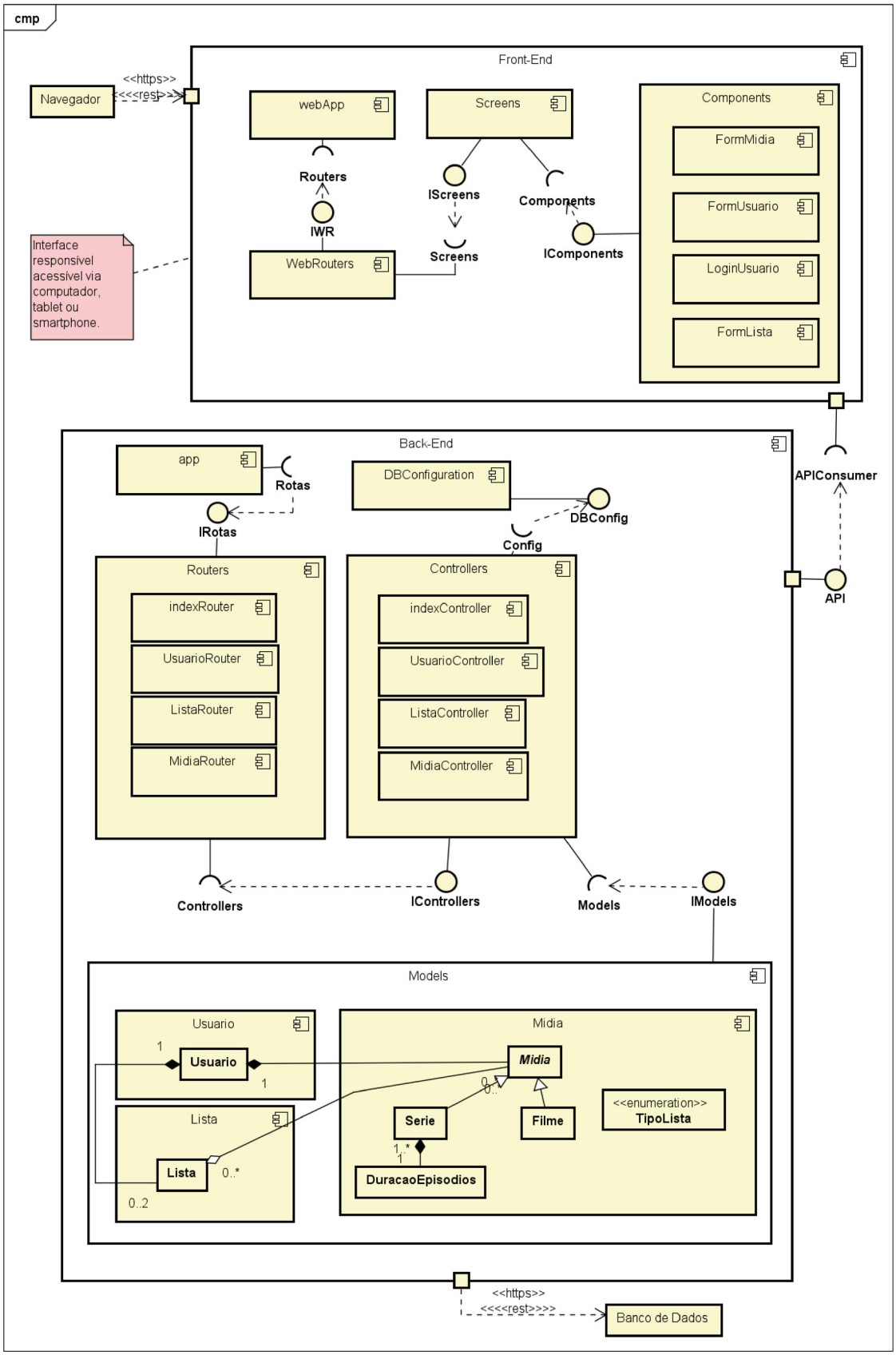
Quando há troca de mensagens síncronas, o retorno é obrigatório, ocorrendo apenas no método “getMidia()” e sinalizado pela seta preenchida na cor preta, já as mensagens assíncronas, que não possuem retorno, são representadas graficamente pelas setas vazadas. Logo, conseguimos observar os retornos e, quando necessário, as mensagens de retorno podem ser inseridas exatamente de acordo com o modo como devem ser apresentadas na interface gráfica do usuário.

A representação dos diagramas de sequência são, portanto, essenciais no contexto de uma arquitetura, pois permitirá identificar e depois a programação das interações como esperado pelo sistema e em acordo com os requisitos, representando, também, as restrições quando necessário.

No mesmo contexto e grupo dos diagramas de sequência, temos também os diagramas de comunicação, de visão geral de interação e sequência temporal. Considerando a representação do diagrama de sequência, os demais diagramas sequenciais não são modelados. Contudo, é importante mencionar que, dependendo da especificidade da solução de software a ser produzida, por exemplo, uma aplicação que considere interações temporais essenciais, o digrama de sequência temporal, do inglês *time sequence* é essencial.

De posse de tais diagramas, voltamos para os modelos estruturais. O diagrama de componentes é apresentado na Figura 4.

Figura 4 – Diagrama de componentes – Aplicação



Fonte: elaborada pelo autor.

No contexto de *front-end*, temos uma estrutura específica para aplicações Web que utilizam tecnologias Web, em especial a biblioteca JavaScript React[1]. Considerando que as decisões tecnológicas são embasadas também no conhecimento e tecnologias que a empresa ou indústria de desenvolvimento de software possuem, a decisão das tecnologias será direcionada àquela que é a mais familiar à equipe de desenvolvedores.

Para o *back-end*, as decisões de tecnologia também foram direcionadas ao conhecimento da equipe de desenvolvimento, tendo sido optado pelo uso de Node.js[2] e JavaScript. Contudo, é possível implementar o *back-end* com outras tecnologias similares, como as da Linguagem Python e Java. Ainda assim, a estrutura pode ser a mesma, com pequenas adaptações.

Independente da tecnologia, a estrutura de rotas (*routers*), controles *(controllers*) e modelos (*models)* podem ser divididas em microsserviços. A diferença entre serviços e microsserviços será a quantidade de funcionalidades, coesão e atomicidade destes. Na representação na Figura 4, temos componentes que possuem mais de uma funcionalidade. Já um microsserviço, terá apresentará funcionalidades específicas, enquanto em um serviço tenho várias funcionalidades, um microsserviço é especializado em uma das funcionalidades que antes eram agrupadas em um serviço. A grande vantagem dos microsserviços está na capacidade de se atualizar e modificar funcionalidades de modo ubíquo, ou seja, o usuário final não perceberá a atualização.

Considerando a integração contínua e entrega contínua[3] (dos termos, em inglês, *coninuous integration/continuous delivery* -CI/CD), uma arquitetura orientada a microsserviços favorece a entrega ininterrupta de novas funcionalidades e correções.

---

[1] O React é uma biblioteca front-end JavaScript de código aberto que permite criar interfaces gráficas de usuário para páginas ou aplicações web.

[2] Node.js é um software de código aberto que integra o interpretador V8 do Google, permitindo a execução de código JavaScript externamente a um navegador.

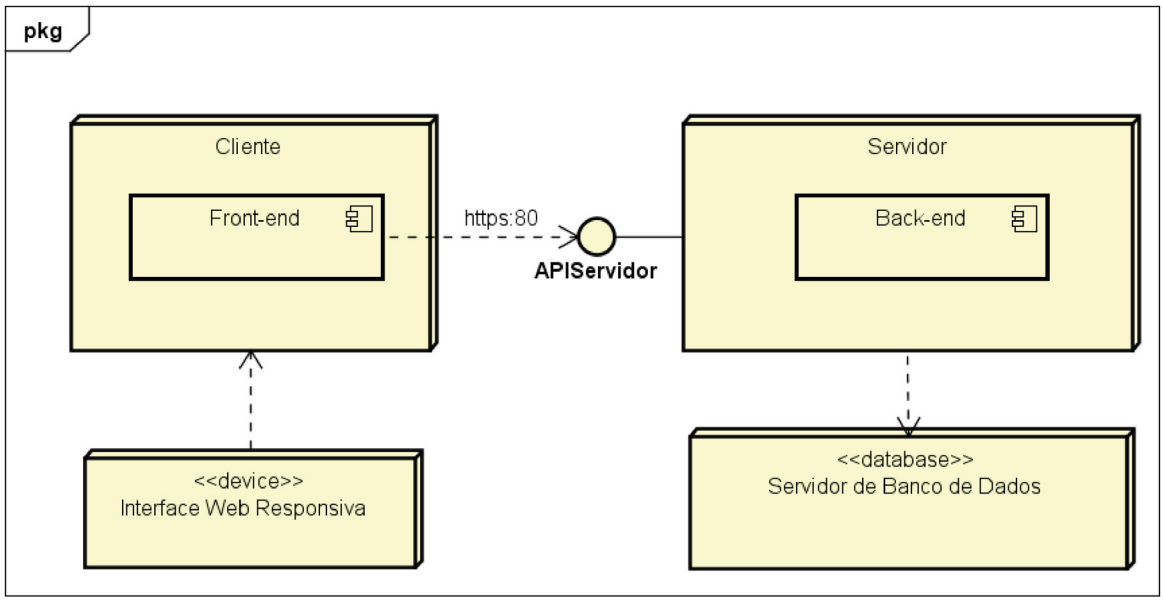
[3] CI/CD refere-se à integração contínua e entrega contínua, complementando as atividades entre equipes de desenvolvimento e operação, por meio da automação na compilação, teste e implantação de aplicativos.

Entretanto, nem só benefícios são aplicados a esse tipo de padrão ou estilo arquitetural. A adoção de uma arquitetura de microsserviços acarreta dificuldades de se controlar ou coordenar os microsserviços, já que precisam estar em constante comunicação e recebendo corretamente as requisições dos clientes. Essa integração requer mecanismos como os já mencionados: orquestração ou coreografia ou mesmo uma abordagem híbrida.

Na orquestração, temos um centralizador que permite tomar decisões de execução e integração entre os microsserviços ou serviços.
Na coreografia, temos uma técnica em que os microsserviços ou serviços estão distribuídos e descentralizados, não há um elemento centralizador, como na orquestração, já que cada serviço, no contexto da coreografia, sabe o que fazer e como fazer, colaborando com os demais microsserviços e serviços. Em uma abordagem híbrida, identificam-se grupos de microsserviços e serviços que requerem maior controle sendo, portanto, orquestrados e os microsserviços e serviços autônomos, serem coreografados.

A partir do diagrama de componentes, podemos seguir para o último modelo que integra a arquitetura de software. Aqui, considerando que o modelo de componentes apresentou mais detalhadamente a composição de cada componente (visão micro), o diagrama de implantação apresenta os nós de estrutura física, onde os componentes serão instalados. Adicionalmente, pode-se incluir especificidades técnicas das conexões. A Figura 5 apresenta o diagrama de implantação da aplicação.

Figura 5 – Diagrama de implantação da aplicação



Fonte: elaborada pelo autor.

Assim, completamos a representação de nossa arquitetura, que será possível ser interpretado e programado. É importante mencionar que, mesmo com a representação de uma técnica ou metodologia específica, que, no caso, foi adotado a UML, podemos utilizar representações informais de camadas, como um fluxograma ou mesmo a *Business Process Model and Notation (BPMN).*

Desse modo, recomenda-se o estudo de outras metodologias, além das apresentadas aqui, para ampliar ainda mais os conhecimentos sobre arquiteturas de software. Bons estudos!

Referências

MARTIN, R. C. **Arquitetura Limpa:** o guia do artesão para estrutura e design de software. Rio de Janeiro: Alta Books, 2019. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788550816043/. Acesso em: 30 nov. 2022.

PAULA FILHO, W. de P. **Engenharia de Software – Produtos**. 4. ed. Rio de Janeiro: LTC, 2019. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788521636724/. Acesso em: 30 nov. 2022.

PRESSMAN, R. S.; MAXIM, B. R**. Engenharia de software**. Porto Alegre: AMGH, 2021. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9786558040118/. Acesso em: 30 nov. 2022.

SOMMERVILLE, I. **Engenharia de Software**. 10. ed. São Paulo: Pearson, 2018. Disponível em: https://plataforma.bvirtual.com.br/Acervo/Publicacao/168127 Acesso em: 30 nov. 2022.

Estilos e Projetos Arquiteturais integrados a Requisitos de Qualidade

Autoria: Anderson da Silva Marcolino
Leitura crítica: Marco Ikuro Hisatomi

Objetivos

- Contextualizar os alunos sobre qualidade e os requisitos relacionados, e como permeiam e interferem nos estilos e projetos arquiteturais.
- Orientar os alunos como implementar, implantar e criar projetos arquiteturais, em especial, de sistemas distribuídos como os orientados a serviço ou de camadas, considerando fatores de qualidade.
- Aprofundar-se nas diferentes exigências de qualidade, dependendo da demanda para a concepção de um software e, consequentemente, sua arquitetura.

1. Introdução à qualidade de software e qualidade nas Arquiteturas de Software

A qualidade que um cliente, ou *stakeholder* espera em um produto de software é descrita por meio dos requisitos funcionais e não funcionais. Como a qualidade deve estar presente em toda a solução de software criada, permite que avaliemos a solução como um todo, para identificar se alcança o que se é esperado. Indo além, a qualidade já deve ser prevista e pensada já na criação dos modelos que estão incluídos na arquitetura de software.

Muitas vezes, entendemos que qualidade é apenas solucionar o problema do cliente, fazer aquilo para o que o software foi construído para fazer. Entretanto, a qualidade vai além e envolve questões que os usuários finais não enxergam nas interfaces gráficas, mas que sentem, como lentidão, indisponibilidade, dificuldade de uso, dificuldade ne instalação, entre outros.

A qualidade, no contexto de software, é padronizada por modelos. Existem os modelos de McCall e Matsumoto (1980), o de Deissenboeck (2011) e o mais famoso deles, o modelo ISO/IEC 25010:2011 (ISO/IEC, 2016), que substitui e simplifica o modelo predecessor, a ISO 9126. Todos permitem a criação de modelos de qualidade que integram a arquitetura de software.

O padrão de Avaliação de Qualidade de Requisitos de Sistemas e Software (SQuaRE) é composto por três conjuntos de padrões, com finalidades diferenciadas:

- ISO/IEC 25040:2011: define o modelo de processos de avaliação.
- ISO/IEC 25010:2011: define o modelo de qualidade de produto de software, com uma vasta gama de características e subcaracterísticas.

- ISO/IEC 25023:2016: define um conjunto de cálculos de métricas em um modelo de qualidade.

Concentraremos atenção na ISO/IEC 25010:2011 e em seu conjunto de características e subcaracterísticas. Cada característica representa um conjunto de elementos que possuem relação direta entre si, no que se refere ao requisito de qualidade. Vejamos as oito categorias, com um resumo de cada, e suas subcaracterísticas.

O entendimento da ISO/IEC 25010 é necessário para que, na sequência, possamos entender quais requisitos de qualidade são os mais apropriados para alguns tipos de arquitetura de software e para que se possa identificar métricas aderentes para a avaliação delas.

1.1 Características de qualidade da ISO/IEC 25010:2011

O conjunto de características e subcaracterísticas são aplicados no contexto da qualidade no uso e qualidade de produto da ISO/IEC 25010. Considerando esta última perspectiva, a seguir, temos a lista e descrição de cada características e subcaracterísticas que dão origem ao modelo de qualidade, integrado à arquitetura de software.

- **Adequação funcional:** esta característica está intimamente ligada ao propósito do produto, com o objetivo de permitir identificar a qualidade quanto à sua integridade, correção e adequação, baseada nas necessidades particulares do produto e de seus solicitantes.
  Desse modo, torna-se um item de qualidade relativamente simples de se identificar, se o analista e métricas estiverem bem estabelecidas e conhecidas. Um modo de facilitar ainda mais a garantia de tal qualidade é envolver *stakeholders* e usuários finais na avaliação de tal característica. Consequentemente, os requisitos funcionais também deverão ser considerados, já que relatam as funcionalidades a serem disponibilizadas e utilizadas no produto final.

- **Subcaracterísticas:** integridade funcional, correção funcional, adequação funcional.

- **Eficiência de desempenho:** esta característica visa identificar o desempenho da solução, considerando o ambiente do usuário, o que dificulta a aplicação de métricas e medição, visto que a solução só poderá ser avaliada considerando todo o contexto do usuário e a infraestrutura para que este considerará na utilização do sistema. O que ocorre são estimativas e cálculos que consideram comportamento simulado para tentar uma aproximação da realidade.
  Uma medida expressiva, dependendo do item de qualidade no contexto de desempenho a ser atingido, é considerar ambientes de computação em nuvem para disponibilização da solução e, assim, garantir uma estimativa mais real e aproximada, além de permitir com que recursos possam ser alocados de modo imperceptível ao usuário final (elasticidade). Consequentemente, esta solução se enquadra para arquiteturas cliente servidor, de serviços, microsserviços, entre outros, disponibilizadas on-line.

  - **Subcaracterísticas:** tempo-comportamento; utilização de recursos, capacidade.

- **Compatibilidade:** característica que integra elementos importantes para uma solução, como capacidade de interoperabilidade, ou seja, comunicação e integração com outras aplicações, incluindo sistemas operacionais, possibilitando a troca de informações e utilização integrada, permitindo a coexistência, que permite analisar o impacto do uso da solução de software desenvolvida com demais produtos que compartilham algo em comum com ela. Essa característica de qualidade considera, especialmente, como as funcionalidades foram implementadas, podendo as considerar como algo maior do que apenas funcionalidades, como módulos ou serviços.
  A subcaracterística de coexistência se torna fator de qualidade

decisivo para arquiteturas de microsserviços ou serviços, já que esses são autônomos, com baixa coesão e acoplamento. Logo, podem ser considerados como partes que precisam coexistir com o todo. Adicionalmente, a coexistência ultrapassa a barreira das funcionalidades criadas e permeiam e consideram elementos de outras aplicações, que integradas, também conversam com a solução arquiteturada.

- **Subcaracterísticas:** coexistência, interoperabilidade, funcionalidade.

- **Usabilidade:** corresponde ao contexto de utilização da solução e envolve todas as subcaracterísticas. A capacidade de reconhecimento, por exemplo, se refere ao grau de reconhecimento que o produto ou sistema é apropriado para suas necessidades, as disponibilidades de treinamentos de uso são específicas do fornecedor da solução, logo a usabilidade é resultado do produto, e não de extras que o fornecedor ou desenvolvedor do produto oferece.
  Logo, tornar a arquitetura, em especial, as interfaces da solução facilitadas para uso e que permitam um aprendizado prazeroso, rápido, com facilidade de operabilidade permitirá garantir qualidade no que tange essa característica de qualidade.

  - **Subcaracterísticas:** adequação, capacidade de reconhecimento, capacidade de aprendizado, operabilidade, erros de uso, proteção, interface de usuário, acessibilidade, estética.

- **Confiabilidade:** esta característica de qualidade não é dependente somente da aplicação, uma vez que os usuários serão as pontes entre identificar possíveis falhas e o nível de tolerância, a disponibilidade e sua recuperação, e reportar para o desenvolvimento, para que, assim, seja conduzida manutenção e solução de tais problemas.
  Contudo, é obrigatório que essas características sejam primadas

na solução, garantindo mensagens amigáveis, indicações de problemas pelo próprio produto e meios de permitir que o produto de software seja capaz de se recuperar das falhas, além de estar disponível para uso sempre que um usuário necessitar. Logo, as especificidades técnicas da arquitetura ditarão como será construído tais recursos e onde a solução será mantida para que esteja sempre disponível e seja capaz de se recuperar, mediante problemas inesperados.
Considerando soluções on-line, problemas de conexão com a Internet e outros recursos em rede, especificam um grupo preciso de funcionalidades a serem consideradas como, por exemplo, bases de dados temporárias para o envio das informações quando uma conexão for reestabelecida; possibilidade de conexão de múltiplos meios, entre outras medidas.

  - **Subcaracterísticas:** maturidade, disponibilidade, tolerância a falhas, recuperabilidade.

- **Segurança:** esta categoria integra os elementos que provêm segurança à solução. Muitas delas estão presentes na maioria das aplicações disponibilizadas, em especial, via web, como é o caso de mecanismos de acesso com usuário e senha, ou seja, controle de acesso e autenticidade. Além disso, mecanismos para garantir a integridade dos dados, no contexto de não serem modificados ao serem transmitidos on-line são funcionalidades requeridas. Bem como o bloqueio de possíveis interceptações de dados.
  Apesar de serem questões relativamente conhecidas, a implicação da reutilização de componentes e bibliotecas ou pacotes desenvolvidos por terceiros, gera maior cuidado no estabelecimento da arquitetura e na construção do produto, uma vez que não se sabe, sem uma análise aprofundada, o que está presente em tais pacotes. Outro ponto é a atualização das soluções, quanto desenvolvidas com tecnologias web. Novas atualizações para tratar vulnerabilidades podem gerar transtorno, no que se refere a características de manutenibilidade. Logo,

devem ser tratadas com cautela.
No que tange as responsabilidades e funcionalidades de segurança, em arquitetura em camadas, há a criação de camada exclusiva para manter tais elementos e seus componentes.

- **Subcaracterísticas:** confidencialidade, integridade, não repúdio, responsabilidade, autenticidade.

- **Manutenibilidade:** esta característica se refere ao grau de efetividade e eficiência em que um produto pode ser modificado. Tais modificações podem ser resultantes de novas demandas ou para ajustes. Inclui-se ainda a reutilização, com foco em reuso de artefatos de software que possam reduzir os esforços no desenvolvimento e que, consequentemente, reduzam os custos. A analisabilidade, ou capacidade de análise, corresponde à avaliação do impacto de mudanças, considerando um diagnóstico de possíveis problemas que possam vir a ocorrer, além de identificar as partes a serem modificadas. Já a testabilidade corresponde a execução de testes, tanto de baixo nível ou testes unitários, que testará cada componente de modo isolado, quanto testes de alto nível, como o de integração, que buscam identificar se o software como um todo mantem seu nível de funcionalidade; e o de regressão, que busca averiguar se o que estava desenvolvido, anteriormente, permanece funcional.
  A testabildiade influenciará no contexto da coesão e acoplamento dos componentes, desse modo, influenciará diretamente na arquitetura de tais componentes se estarão relacionados e se comunicarão com o software como um todo.

  - **Subcaracterísticas:** modularidade, reutilização, analisabilidade, modificabilidade, testabilidade.

- **Portabilidade:** característica que envolve preocupações relacionadas ao código fonte, recompilação, refatoração, entre outros, geralmente, sendo foco dos desenvolvedores e como, após

entregues em um produto de software, serão direcionados os esforços para migrar, substituir ou transportar a solução e demais itens relacionados, como base de dados, para um novo ambiente. Assim, a portabilidade, quando considerada na arquitetura de software deve ser capaz de permitir que um software, desenvolvido a princípio para um ambiente possa ser utilizado em outros ambientes. Logo, a especificação de plataformas para uma solução ou o atendimento à multiplataformas dependerá do tipo de solução.
Sendo algo disponibilizado on-line, a adaptabilidade ao ambiente, telas (com responsividade) e outras especificações, será algo mais tangível, que para softwares desktop, mas prever e garantir a utilização em diversos ambientes é uma característica obrigatória. Uma outra característica é a capacidade de substituição. Em um ambiente controlado e com integração contínua, a atualização de um produto de software será ubíquo, porém, nem sempre essa ubiquidade será garantida, dependendo da tecnologia e plataforma utilizada. Neste item, preocupa-se ainda com a capacidade de migração de dados de um repositório para outro, com minimização de mudanças referentes às interfaces dos usuários.

- **Subcaracterísticas:** adaptabilidade, instalabilidade, capacidade de substituição.

É importante destacar que as categorias e subcategorias permanecem em constante alteração, na medida em que novas tecnologias, estilos arquiteturais e metodologias de desenvolvimento de software surjam, logo, o estudo constante de possíveis atualizações são sempre necessárias.

Dessa forma, partindo do entendimento geral sobre requisitos de qualidade, vamos abordá-los considerando alguns dos principais padrões arquiteturais de software e estudar algumas das métricas

que podem ser utilizadas para permitir a criação de arquiteturas mais precisas.

1.2 Requisitos de qualidade e métricas nas Arquiteturas de Software atuais

Os padrões arquiteturais que têm se destacado no mercado acompanham tendências tecnológicas (tecnologias web para nuvem) e metodologias ou mudanças culturais (integração contínua), no contexto do uso de ferramentas para o apoio no processo de desenvolvimento, como é o caso do DevOps.

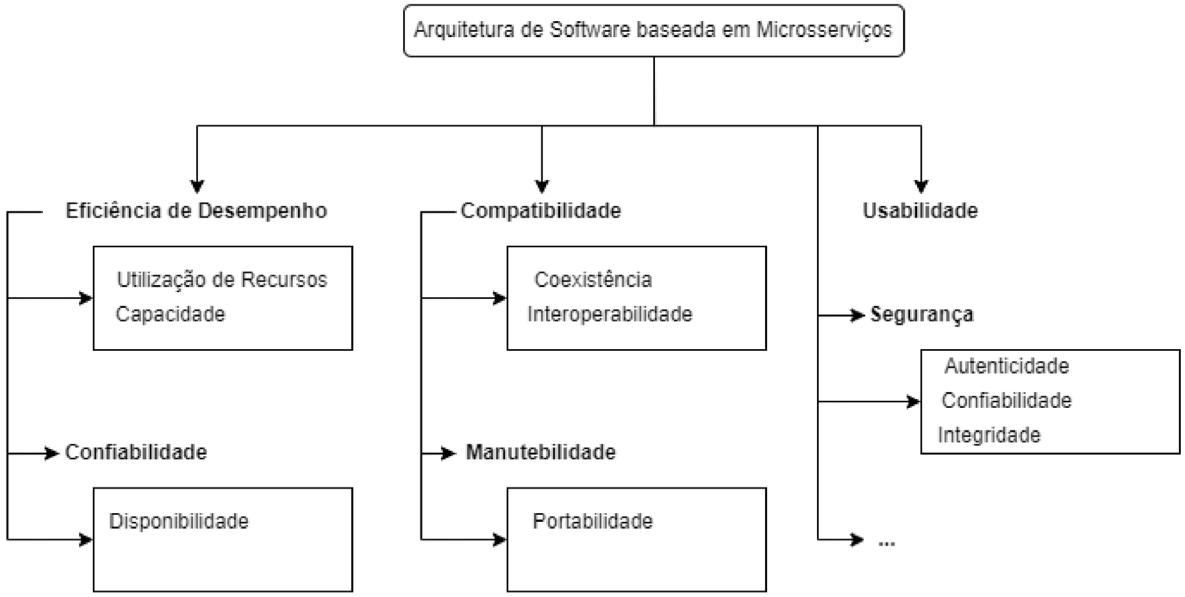
Nessa perspectiva, e considerando que os requisitos de qualidade, por si só, apresentam características que dificultam sua mensuração e precisão, diversos tipos de métricas e métodos para buscar avaliar a qualidade de uma arquitetura e produto de software são adotados.

Um levantamento dos métodos para mensurar qualidade de software apresentou a adoção de diferentes técnicas para permitir uma melhor qualidade dos produtos (WAHYUNINGRUM, 2017). Entre os métodos identificados, o fator humano é parte fundamental para que um resultado seja obtido. Tais métodos incluem: 1) método analítico por meio de questionários com usuários, desenvolvedores e outros membros da equipe de análise e projeto da arquitetura; 2) método empírico, por meio da observação de funcionalidades e sua integração à elementos de qualidade esperados; 3) métodos analíticos com grupos de avaliação, que formam equipes de análise de fatores de qualidade, comparando o produto a ser criado ou já pronto, incluindo seu modelo arquitetural, com modelos de qualidade e suas características e subcaracterísticas; e 4) a integração dos métodos analíticos e empíricos, com observação.

Os métodos descritos acima são, então, aplicados considerando um modelo de qualidade. As características e subcaracterísticas

apresentadas anteriormente devem ser selecionadas de acordo com o produto a ser especificado. A Figura 1 apresenta um possível modelo de qualidade para arquiteturas de software baseada em microsserviços.

Figura 1 – Modelo de qualidade para Arquiteturas de Software baseada em microsserviços.



Fonte: elaborada pelo autor.

O modelo apresentado na Figura 1 traça rotas e ordena as principais características da ISO/IEC 25010:2011 e incluí as subcaracterísticas ou subcategorias mais necessárias, considerando o tipo de arquitetura e produto de software a ser desenvolvido.

Cada elemento apresentado no modelo de qualidade deve ser selecionado, considerando a especificação dos requisitos funcionais. Ainda que o foco seja identificar pontos de interesse no contexto de qualidade, a qualidade só poderá ser obtida com base nos requisitos que forem desenvolvidos de fato, e refletem funcionalidades. Logo, é necessário, no contexto das etapas de análise e projeto identificar o que será entregável, as atividades de desenvolvimento e os pontos de revisão.

A partir do estabelecimento do modelo, temos, então, a especificação de métricas, ou seja, o que será usado para mensurar, em termos quantitativos, principalmente, se o requisito de qualidade é ou não atendido. É nesta perspectiva que o modelo apresentado na Figura 1, que pode ser criado em formatos de tabela, outros tipos de fluxograma, ou mesmo diagramas de atividades da linguagem de modelagem unificada entra.

O modelo de qualidade servirá como norteador para identificar os possíveis meios de se mensurar a qualidade. Considerando alguns deles, como, por exemplo, a usabilidade, será necessária a adoção de métodos híbridos (empírico e analítico), com observação. Considerando elementos que serão desenvolvidos e refletirão em unidades de código, como componentes, métodos, variáveis, entre outros, pode-se realizar a especificação e classificação deles em unidades ou elementos de dados quantificáveis. Inclui-se, nesta quantificação, também os valores de atributos, que podem ser obtidos por mensuração direta ou definição de alvos ou estimativas.

Para a utilização de estimativas, é necessário a coleta de dados, por meio de projetos anteriores, que possam servir de linha de base para apoiar comparações e análises. Modelos de predição, aprendizagem de máquina e *Data Science*, têm sido utilizados para criar modelos preditivos e, assim, auxiliar a equipe no contexto da criação de uma base para se tomar decisões e avaliar a qualidade do software e suas arquiteturas.

De posse dos mecanismos de mensuração e definição de seus métodos de coleta e análise, os submodelos ou características, como os apresentados na Figura 1, são então integrados aos métodos e métricas em um documento específico. Instruções de como cada decisão para mensurar a qualidade foi selecionada são essenciais. Elas ajudarão a traçar planos e executar as métricas, além de servirem como apoio na criação de novos modelos de qualidades para outras arquiteturas.

Feito isso, os objetos do projeto, e, aqui, entram os diversos diagramas da arquitetura de software, devem ser relacionados aos atributos que serão mensurados. Por exemplo, no contexto da arquitetura de microsserviços, a utilização de recursos, em especial, quantidade de requisições por unidade de tempo, como minutos, estarão ligados ao diagrama de componentes ou de implementação (WAZLAWICK, 2019). Além disso, em especificação textual, deverá integrar as especificações de software adotadas para que a relação (recursos de infraestrutura versus quantidade de requisições) eunidade de tempo possa ser calculado e então, termos uma mensuração dos recursos necessários. Esta previsão permitirá alocar mais ou menos recursos e permitirá atender ao requisito de qualidade de eficiência de desempenho, de modo direto, e o requisito de confiabilidade e sua subcaracterística de disponibilidade.

Finalmente, deve-se realizar a atribuição de cada subcategoria do modelo interno de qualidade com um objeto ou artefato do projeto, relacionados com regras e unidades de mensuração quantitativa de cada atributo, gerando um modelo condizente com os elementos arquiteturais e requisitos de qualidade.

Algumas métricas, que podem ser consideradas, com base na arquitetura de microsserviços da Figura 1, que também serve de base para uma arquitetura orientada a serviços e disponibilizada na nuvem, são: unidades de infraestrutura (unidades centrais de processamento, cache, memória RAM, estabilidade e tipo de conexão à rede interna e externa (Intranet e Internet) e suas capacidades; quantidades de requisições por unidade de tempo e a relação desta com os serviços de nuvem contratados; capacidade de direcionar ou manter requisições com outros sistemas ou interfaces de programação de aplicação (API), no caso da compatibilidade; a testagem do software no contexto da portabilidade em diferentes sistemas operacionais, considerando em especial contêineres ou mesmo máquinas virtuais de testes, que permitirão simular, inclusive, os recursos de hardware das máquinas dos

usuários; heurísticas de avaliação de usabilidade que requerem, em sua maioria, a participação de humanos para análise; testes de autenticação, transmissão de informação e uso de ferramentas de software para verificar as informações trafegadas, quanto a sua integridade e confiabilidade, no contexto de segurança, e muitas outras.

Observando os diferentes elementos relacionados ao contexto de qualidade, na perspectiva das arquiteturas, podemos identificar que é uma vasta área a ser explorada. Entretanto, é necessário destacar que não é possível garantir qualidade a níveis perfeitos e o aumento do nível de capacidade de se mensurar, criando e utilizando modelos de qualidade virá com o tempo, com base em muito estudo e participação em projetos. Recomenda-se, neste contexto, a busca por materiais e artigos que apresentem estudos de casos reais, que possam ampliar ainda mais os conhecimentos em relação a tais tópicos! Bons estudos!

Referências

DEISSENBOECK, F. *et al*. The quamoco tool chain for quality modeling and assessment. *In:* 33rd International Conference on Software Engineering (ICSE), p. 1007-1009. **IEEE**, [s. l.], 2011. Disponível em https://dl.acm.org/doi/abs/10.1145/1985793.1985977. Acesso em: 30 nov. 2022.

ISO/IEC. **Systems and Software Engineering**: Systems and Software Quality Requirements and Evaluation (SQuaRE): measurement of system and software product quality. Switzerland: ISO, 2016. Disponível em https://webstore.iec.ch/publication/25171. Acesso em: 30 nov. 2022.

MCCALL, J. A.; MATSUMOTO, M. T**. Software Quality Metrics Enhancements**. v. 1 General Electric CO Sunnyvale CA, [s. l.], 1980.

WAHYUNINGRUM, T.; MUSTOFA, K. A systematic mapping review of software quality measurement: research trends, model, and method. **International Journal of Electrical and Computer Engineering**, v. 7, n. 5, p. 2847, 2017. Disponível em https://ijece.iaescore.com/index.php/IJECE/article/view/8241/0 Acesso em: 30 nov. 2022.

WAZLAWICK, R. **Engenharia de software:** conceitos e práticas**.** São Paulo: Elsevier, 2019.

BONS ESTUDOS!