ENGENHARIA DE SOFTWARE

AULA 3

Prof. Alex Mateus Porn

CONVERSA INICIAL

Olá! Seja bem-vindo(a)!

Nesta aula falaremos sobre projeto de arquitetura de software. O objetivo da aula é compreender como aplicar um ou mais modelos dos processos de software, que estudamos anteriormente, no desenvolvimento de todas as etapas da construção de um sistema, conceituando um único projeto. Para isso, iniciaremos esta aula conceituando o que são projetos e como são caracterizados no contexto do desenvolvimento de software.

Assim como aprendemos anteriormente vários modelos de processos de software, também estudaremos, nesta aula, diversos padrões de arquitetura, os quais podem ser aplicados mais de um padrão por projeto e executados por um ou mais modelos de processos.

Também será apresentada a ferramenta para gerenciamento de configuração e versionamento de projetos Git, em conjunto com a plataforma GitHub para hospedagem e compartilhamento dos arquivos de projetos.

A aula se encerra com a análise das leis de Lehman e Ramil, que apresentam as necessidades da evolução do software. Ao longo desta aula serão trabalhados os seguintes conteúdos:

TEMA 1 – PROJETO DE ARQUITETURA DE SOFTWARE

Anteriormente, estudamos vários modelos de processos prescritivos de software e também nos aprofundamos nos métodos ágeis. Pudemos observar que todos os modelos estudados abordam o planejamento e execução de cada uma das etapas do desenvolvimento do sistema. Porém, para construir um software completo, devemos partir do contexto geral desse sistema, para, então, separarmos os processos em cada fase de acordo com o modelo escolhido. Fazendo uma analogia com a construção civil, para construirmos uma casa, primeiro projetamos a planta dessa casa para, então, distribuirmos encanamentos, fiação elétrica, rede de dados, alarme, entre outros. Ou seja, partimos do projeto da casa como um todo e posteriormente analisamos cada etapa. Nesse contexto, é exatamente essa a tarefa de um projeto de arquitetura de software.

Antes de entrarmos nesse assunto, primeiramente precisamos compreender o conceito de projeto. O Guia PMBOK (PMI, 2017) define projeto como sendo “um esforço temporário empreendido para criar um produto, serviço ou resultado único”.

Seguindo essa perspectiva, na Metodologia de Gerenciamento de Projetos do SISP (Brasil, 2011), projeto é definido como “um empreendimento planejado, orientado a resultados, possuindo atividades com início e término, para atingir um objetivo claro e definido”.

De acordo com as definições de projeto apresentadas no Guia PMBOK e na Metodologia de Gerenciamento de Projetos do SISP, podemos perceber a similaridade com a execução das etapas dos modelos de processos prescritivos e ágeis que estudamos anteriormente.

Dado que um projeto é composto por atividades que tenham início e término (Brasil, 2011), que demandam um esforço temporário para criar um produto ou serviço (PMI, 2017), é a mesma concepção que temos ao executar um modelo de processos de software para o desenvolvimento de um sistema. Desse modo, a definição de projeto no contexto do desenvolvimento de sistemas computacionais, é dada por Pressman (2011) como “um processo em várias etapas em que as representações de dados e a estrutura do programa, as

características de interfaces e os detalhes procedurais são sintetizados com base nos requisitos de informação".

Na abordagem de Pressman, podemos ser um pouco mais específicos, e estabelecer que um projeto de software não é somente um processo, mas um ou vários processos em várias etapas, de acordo com a complexidade do sistema que será desenvolvido. Portanto, o projeto de software corresponde a um esquema preliminar por meio do qual o software é construído, considerando-se os domínios de dados, funcional e comportamental, os quais descrevem o projeto da arquitetura do software (Pressman, 2011).

Conforme Pressman (2011, p. 229):

> O projeto da arquitetura representa a estrutura de dados e os componentes de programa necessários para construir um sistema computacional. Ele considera o estilo de arquitetura que o sistema assumirá, a estrutura e as propriedades dos componentes que constituem o sistema, bem como as inter-relações que ocorrem entre todos os componentes da arquitetura de um sistema.

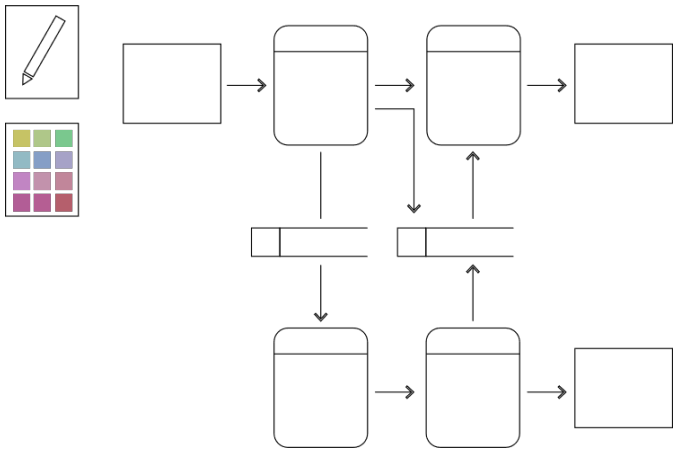
Parafraseando Pressman (2011), a arquitetura não é o software operacional, mas sim uma representação que nos permite analisar a efetividade do projeto no atendimento aos requisitos declarados, ela é a estrutura do sistema, que abrange os componentes de software, as propriedades externamente visíveis desses componentes e as relações entre eles.

De acordo com Pressman (2011), o projeto da arquitetura ocorre pelas seguintes etapas:

1. Projeto de dados;
2. Derivação da estrutura da arquitetura do sistema;
3. Análise de estilos ou padrões de arquitetura alternativos;
4. Implementação da arquitetura utilizando-se um modelo de processos.

Na primeira etapa, o projeto deve definir as entidades externas com as quais o software irá interagir, podendo ser adquiridas durante o levantamento de requisitos. Cabe destacar, aqui, como um exemplo de representação, o Diagrama Entidade-Relacionamento do banco de dados.

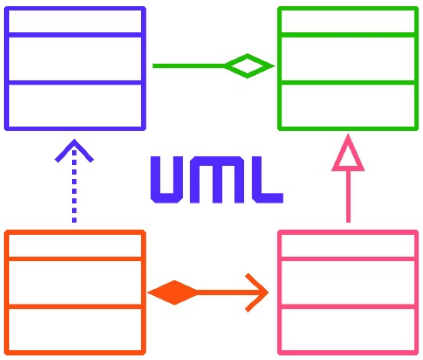
Figura 1 – Representação Diagrama



Crédito: Singkham/shutterstock.

Na segunda etapa, são identificados arquétipos arquiteturais, que representam abstrações de elementos do sistema, similares a classes em uma arquitetura orientada a objetos. Destaca-se, nessa etapa, como um exemplo de representação, o uso da linguagem UML e modelos como o Diagrama de Classes.

Figura 2 – Arquétipos arquiteturais



Crédito: laaisee/shutterstock.

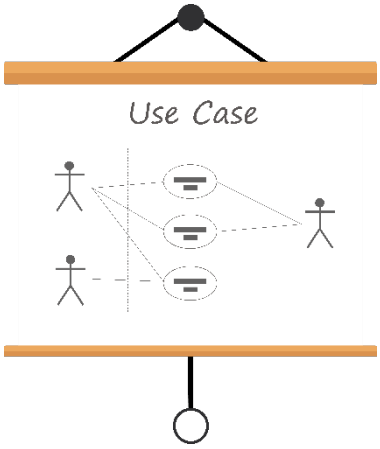
Na terceira etapa, define-se qual o padrão de arquitetura de software a ser implementado, os quais são abordados no Tema 2 desta aula, para finalmente a implementação dessa arquitetura conforme um modelo de processos escolhido.

Conforme destaca Pfleeger (2004), os clientes sabem o que o sistema deve fazer e os desenvolvedores devem saber como o sistema funcionará. Por

essa razão, Pfleeger destaca que o projeto é um processo iterativo composto pelo **projeto conceitual** e pelo **projeto técnico**.

- **Projeto conceitual**: apresenta ao cliente exatamente o que o sistema fará. Havendo aprovação pelo cliente, esse projeto é traduzido em um modelo mais detalhado, dando origem ao projeto técnico. Nessa fase, podemos destacar como um exemplo de um modelo alternativo a uma representação textual do sistema, a construção de diagramas de casos de uso na linguagem UML.

Figura 3 – Projeto conceitual



Crédito: fatmawati achmad zaenuri/shutterstock.

- **Projeto técnico**: possibilita aos desenvolvedores saber quais são o hardware e software necessários para resolver o problema do cliente. A função desse projeto é descrever a forma que o sistema terá.

A Figura 4 apresenta a diferença entre o projeto conceitual e o projeto técnico, elaborado com base em Pfleeger (2004, p. 160-161).

Figura 4 – Projetos conceitual e lógico (técnico)

Fonte: Elaborado com base em Pfleeger, 2004, p. 160-161.

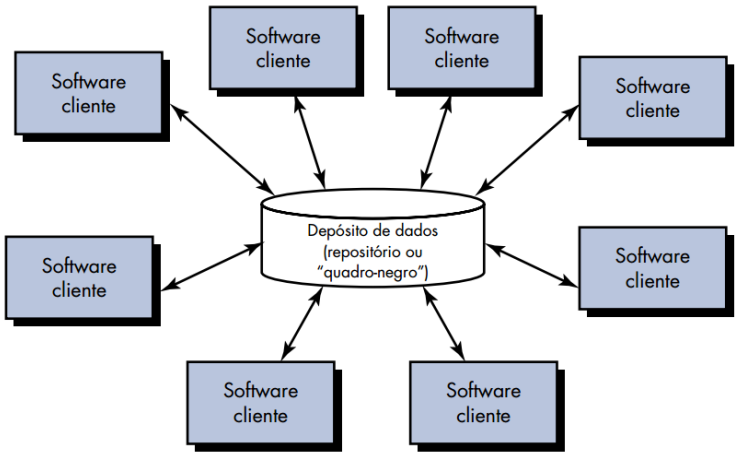
TEMA 2 – PADRÕES DE ARQUITETURA DE SOFTWARE

Parafraseando Sommerville (2018), um padrão de arquitetura é como uma descrição abstrata, estilizada, das práticas recomendadas que foram testadas e aprovadas em diferentes subsistemas e ambientes. Na sequência, veremos os principais padrões de arquitetura de software que podem ser utilizados no desenvolvimento de sistemas.

2.1 Arquitetura centralizada em dados

Essa arquitetura pode ser considerada como uma das mais aplicadas quando nos referimos ao desenvolvimento de sistemas de informação. Conforme Pressman (2011), um repositório de dados, tal como um banco de dados, reside no centro dessa arquitetura e, em geral, é acessado por outros componentes que atualizam, acrescentam, eliminam ou, de alguma forma, modificam dados contidos nesse repositório. A Figura 5 ilustra a representação dessa arquitetura.

Figura 5 – Arquitetura centralizada em dados



Fonte: Pressman, 2011, p. 236.

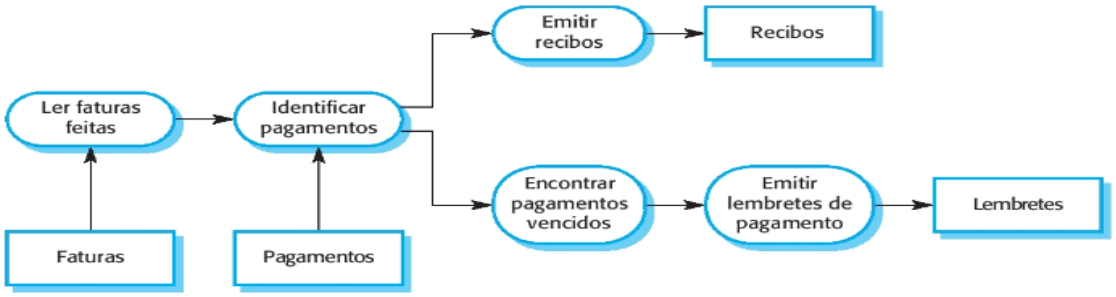
2.2 Arquitetura de fluxo de dados

Também conhecida como arquitetura de duto e filtro, essa arquitetura se aplica quando dados de entrada devem ser transformados por meio de uma série de componentes computacionais ou de manipulação em dados de saída (Pressman, 2011). Conforme Sommerville (2018), o processamento dos dados

está organizado de modo que cada componente de processamento (filtro) seja discreto e realize um tipo de transformação de dados.

Normalmente esse tipo de arquitetura é representada em sistemas desenvolvidos na metodologia de programação estruturada, por meio da representação de diagramas de atividades da linguagem UML. A Figura 6 ilustra um exemplo de um sistema de fluxo de dados para o processamento de faturas.

Figura 6 – Arquitetura de fluxo de dados para o processamento de faturas



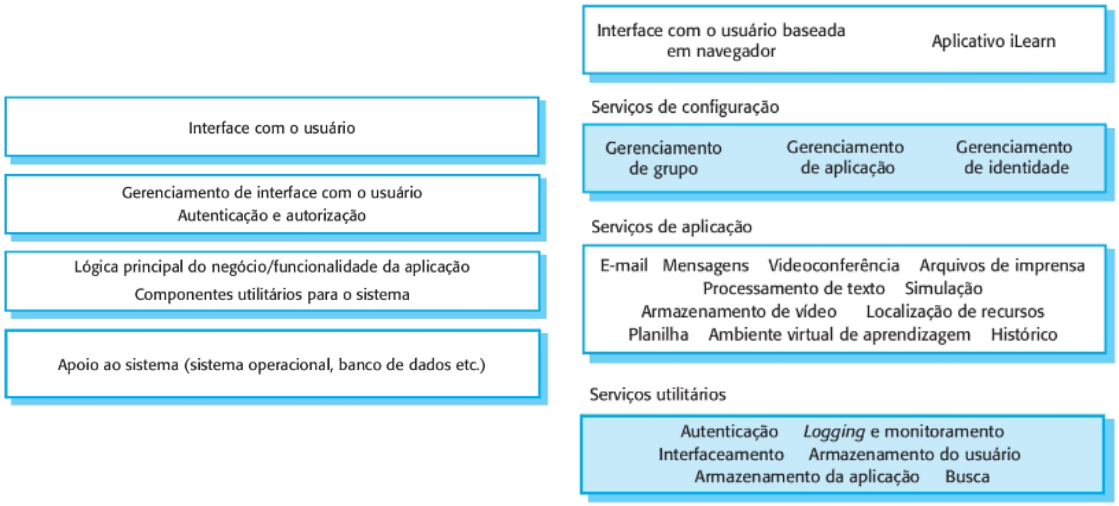
Fonte: Sommerville, 2018, p. 162.

2.3 Arquitetura em camadas

Nessa arquitetura, o sistema é organizado em camadas com a funcionalidade relacionada associada a cada camada (Sommerville, 2018). Conforme Pressman (2011), na camada mais externa, os componentes atendem operações de interface do usuário. Na camada mais interna, os componentes realizam a interface com o sistema operacional e, nas camadas intermediárias, são fornecidos serviços utilitários e funções de software de aplicação.

Como um exemplo de um sistema de arquitetura em camadas, podemos visualizar o exemplo proposto por Sommerville (2018, p. 157), de um sistema de aprendizagem digital para apoiar a aprendizagem de todas as disciplinas nas escolas, conforme é ilustrado na Figura 7.

Figura 7 – Arquitetura em camadas



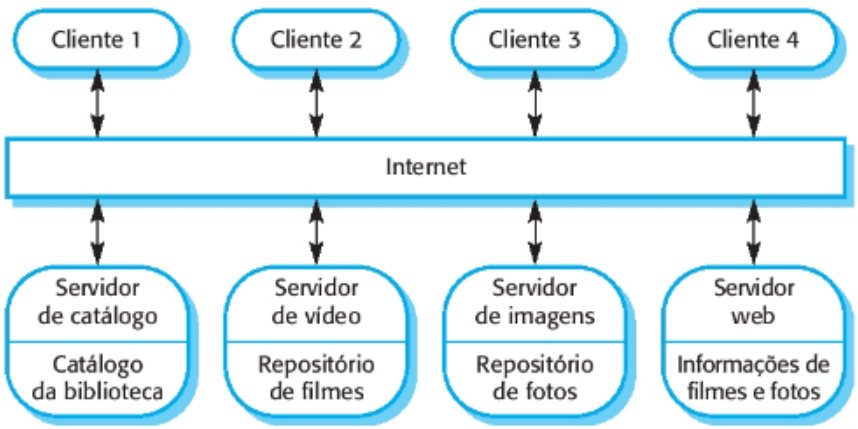
Fonte: Sommerville, 2018, p. 155-156.

2.4 Arquitetura cliente-servidor

A arquitetura cliente-servidor é muito utilizada para sistemas distribuídos e sistemas web. De acordo com Sommerville (2018), nessa arquitetura, o sistema é apresentado como um conjunto de serviços, e cada serviço é fornecido por um servidor separado. Os clientes são usuários desses serviços e acessam os servidores para usá-los.

Podemos considerar essa arquitetura como uma das mais utilizadas, pois todos os serviços que acessamos diariamente ao conectarmos na internet, são baseados na arquitetura cliente-servidor, e dentre eles podemos citar nosso e-mail, sites de *streamming* de vídeos, jogos online, entre outros. A Figura 8 apresenta a arquitetura cliente-servidor de um site de locação de filmes.

Figura 8 – Arquitetura cliente-servidor



Fonte: Sommerville, 2018, p. 161.

2.5 Arquitetura orientada a objetos

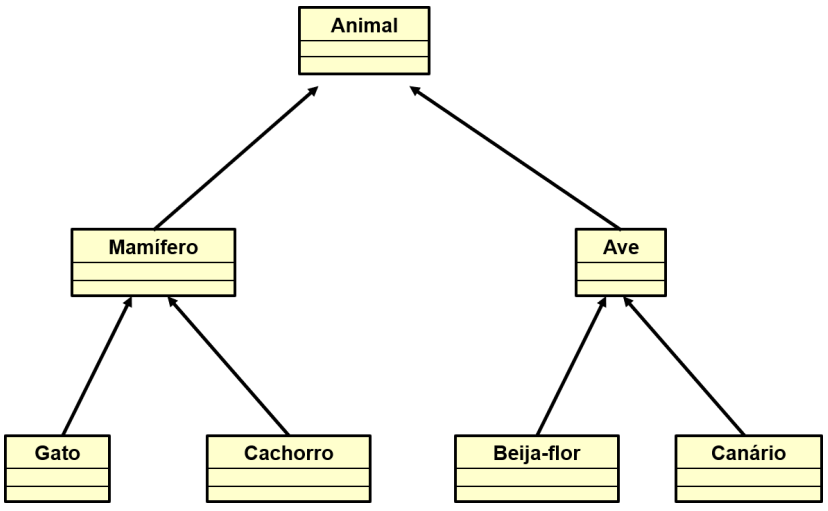
A arquitetura orientada a objetos pode ser considerada como uma das arquiteturas mais atuais. Conforme Pressman (2011), nessa arquitetura, os componentes de um sistema encapsulam dados e as operações que devem ser aplicadas para manipular os dados. A comunicação e a coordenação entre componentes são realizadas por meio da passagem de mensagens.

A arquitetura orientada a objetos fornece uma descrição abstrata do software, de modo que tem como intuito aproximar as estruturas de um programa das coisas do mundo real. Essa arquitetura se baseia em dois conceitos:

- **Classes**: é um conjunto de características e comportamentos que definem o conjunto de objetos pertencentes à classe;
- **Objetos**: uma ou várias instâncias de uma classe que correspondem as suas características e comportamentos.

Podemos representar um exemplo de arquitetura orientada a objetos, de um sistema para registro de animais em uma clínica veterinária. Sabendo-se que essa clínica atende vários tipos de animais e que diversos animais possuem várias características em comum, tal como gato e cachorro possuem quatro patas, um rabo e ambos são mamíferos e beija-flor e canário possuem duas patas, um bico e ambos são aves, poderíamos representar duas classes abstratas Mamífero e Ave, cada uma com duas subclasses para representar os respectivos animais. A Figura 9 apresenta uma ilustração da arquitetura orientada a objetos da clínica veterinária.

Figura 9 – Arquitetura orientada a objetos



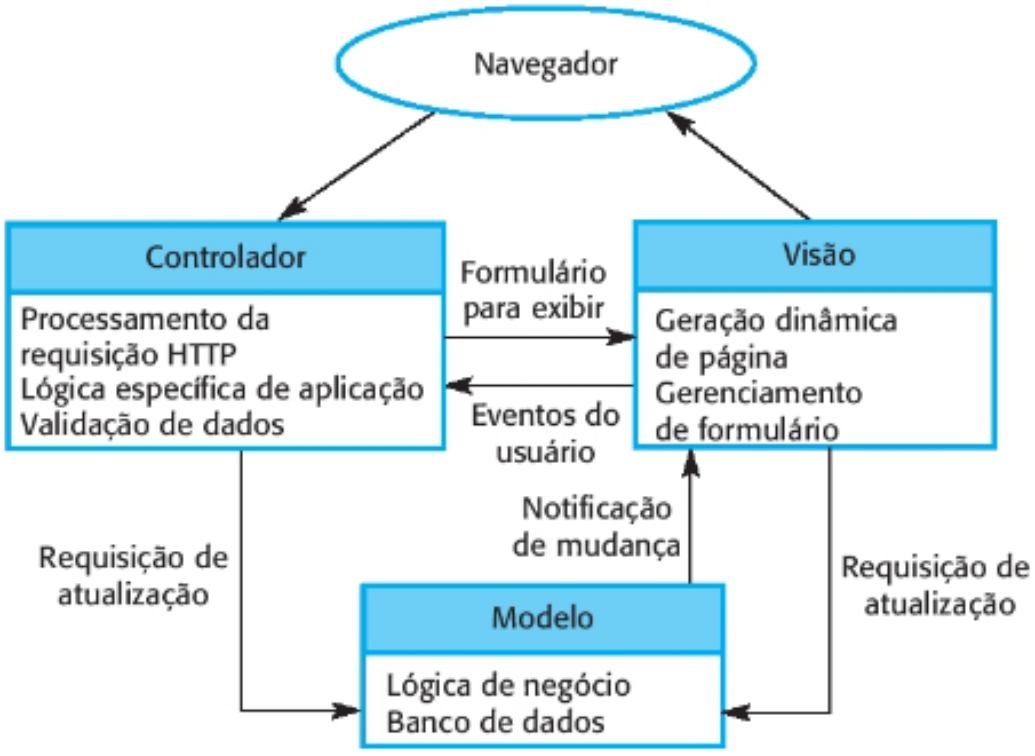
2.6 Arquitetura Modelo, Visão e Controlador – MVC

Essa arquitetura tem como foco separar a apresentação e a interação dos dados do sistema. Segundo Sommerville (2018), o sistema é estruturado em três componentes lógicos que interagem entre si, modelo, visão e controlador:

- **Modelo**: gerencia o sistema de dados e as operações associadas a esses dados;
- **Visão**: define e gerencia como os dados são apresentados ao usuário.;
- **Controlador**: gerencia a interação do usuário e passa essa interação para a Visão e o Modelo.

A Figura 10 apresenta a arquitetura MVC de uma aplicação web.

Figura 10 – Arquitetura MVC



Fonte: Sommerville, 2018, p. 156.

Embora cada sistema de software seja único, é muito comum identificarmos projetos de sistemas que são desenvolvidos em conformidade com mais de uma arquitetura. Olhando mais de perto o exemplo de arquitetura orientada a objetos de um sistema de uma clínica veterinária, apresentado na

Figura 6, podemos identificar perfeitamente que esse sistema também poderia ser baseado na arquitetura centralizada em dados, tendo como ponto central um banco de dados para o registro dos dados referentes aos animais atendidos. Este mesmo sistema poderia ser desenvolvido em uma versão web e ser apoiado pela arquitetura cliente-servidor, consequentemente poderia ser fundamentado pela arquitetura MVC, nos mesmos moldes apresentados no exemplo da Figura 10.

TEMA 3 – LEVANTAMENTO DE REQUISITOS

Anteriormente, estudamos os modelos de processos prescritivos e métodos ágeis para o desenvolvimento de software, que são aplicados de acordo com a escolha do engenheiro de software, em um projeto baseado em uma ou mais arquiteturas de software, conforme as arquiteturas que estudamos no Tema 2, desta aula. Para uma definição clara e objetiva na escolha da arquitetura e modelos de processos a serem empregados em um projeto de software, a primeira etapa consiste no levantamento de requisitos.

Conforme exposto por Sommerville (2018, p. 85):

> Os requisitos de um sistema são as descrições do que o sistema deve fazer, os serviços que oferecem e as restrições a seu funcionamento. Esses requisitos refletem as necessidades dos clientes para um sistema que serve a uma finalidade determinada, como controlar um dispositivo, colocar um pedido ou encontrar informações.

Os requisitos de software são classificados como requisitos funcionais e requisitos não funcionais que, de acordo com Sommerville (2018, p. 88-89), representam:

- **Requisitos funcionais**: São declarações dos serviços que o sistema deve fornecer, do modo como o sistema deve agir a determinadas entradas e de como deve se comportar em determinadas situações. Em alguns casos, os requisitos funcionais também podem declarar explicitamente o que o sistema não deve fazer;
- **Requisitos não funcionais**: São restrições sobre os serviços ou funções oferecidas pelo sistema. Eles incluem restrições de tempo, restrições sobre o processo de desenvolvimento e restrições impostas por padrões. Os requisitos não funcionais se aplicam, frequentemente, ao sistema como um todo, em vez de às características individuais ou aos serviços.

3.1 Técnicas para elicitação de requisitos

Na fase de elicitação dos requisitos, de acordo com Sommerville (2018, p. 96), os objetivos do processo de elicitação de requisitos são compreender o trabalho que as partes envolvidas no processo realizam e entender como usariam o novo sistema para apoiar seus trabalhos. Nessa fase, os engenheiros de software trabalham em conjunto com os clientes e usuários finais do sistema para obter informações sobre o domínio da aplicação, os serviços que o sistema deve oferecer, o desempenho dos sistemas, restrições de hardware, entre outros.

Conforme Sommerville (2018), a descoberta de requisitos é o processo de reunir informações sobre o sistema requerido e os sistemas existentes e separar dessas informações os requisitos de usuário e de sistema. As fontes de informação incluem documentações, usuários do sistema, especificações de sistemas similares, entre outros.

Sommerville (2011) apresenta as seguintes técnicas de levantamento de requisitos:

- **Entrevistas**: formais ou informais com usuários e demais partes envolvidas no sistema. A equipe de levantamento de requisitos questiona as partes envolvidas sobre o sistema que usam atualmente e sobre o sistema que será desenvolvido. Os requisitos surgem a partir das respostas a essas perguntas.
- **Cenários**: os cenários podem ser escritos como texto, suplementados por diagramas, telas, entre outros. Cada cenário geralmente cobre um pequeno número de iterações possíveis. Diferentes cenários são desenvolvidos e oferecem diversos tipos de informação em variados níveis de detalhamento sobre o sistema.
- **Casos de uso**: pode ser considerado como uma abordagem mais estruturada de cenários. Um caso de uso identifica os atores envolvidos em uma iteração e dá nome ao tipo de iteração.
- **Etnografia**: técnica de observação que pode ser usada para compreender os processos operacionais e ajudar a extrair os requisitos de apoio para esses processos. Um analista faz uma imersão no ambiente de trabalho em que o sistema será usado. O trabalho do dia a dia é

observado e são feitas anotações sobre as tarefas reais em que os participantes estão envolvidos.

Na Figura 11 podemos observar um exemplo de um cenário em que um paciente é atendido em uma clínica médica.

Figura 11 – Exemplo de cenário

Suposição inicial:
O paciente é atendido em uma clínica médica por uma recepcionista; ela gera um registro no sistema e coleta suas informações pessoais (nome, endereço, idade etc.). Uma enfermeira é conectada ao sistema e coleta o histórico médico do paciente.

Normal:
A enfermeira busca o paciente pelo sobrenome. Se houver mais de um paciente com o mesmo sobrenome, o nome e a data de nascimento são usados para identificar o paciente.
A enfermeira escolhe a opção do menu para adicionar o histórico médico.
A enfermeira segue, então, uma série de prompts do sistema para inserir informações sobre consultas em outros locais, os problemas de saúde mental (entrada de texto livre), condições médicas (enfermeira seleciona condições do menu), medicação atual (selecionado no menu), alergias (texto livre) e informações da vida doméstica (formulário).

O que pode dar errado:
O prontuário do paciente não existe ou não pôde ser encontrado. A enfermeira deve criar um novo registro e registrar as informações pessoais.
As condições do paciente ou a medicação em uso não estão inscritas no menu. A enfermeira deve escolher a opção "outros" e inserir texto livre com descrição da condição/medicação.
O paciente não pode/não fornecerá informações sobre seu histórico médico. A enfermeira deve inserir um texto livre registrando a incapacidade/ relutância do paciente em fornecer as informações. O sistema deve imprimir o formulário-padrão de exclusão afirmando que a falta de informação pode significar que o tratamento será limitado ou postergado. Este deverá ser assinado e entregue ao paciente.

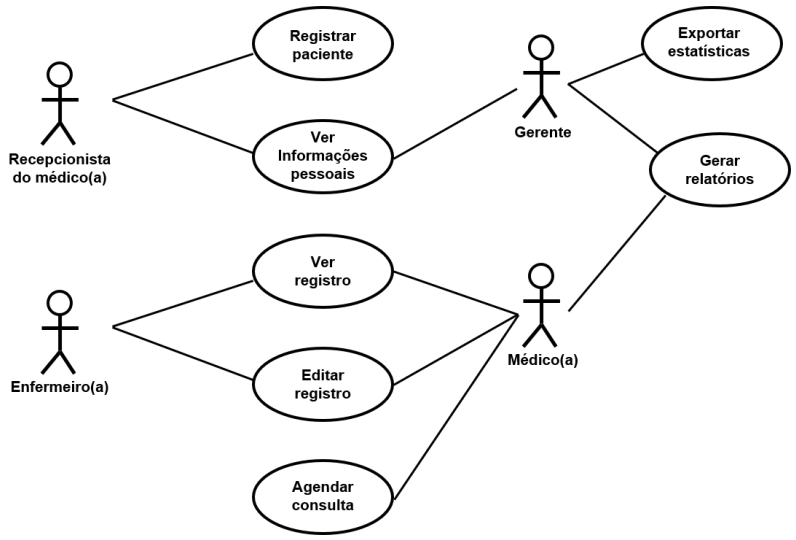
Outras atividades:
Enquanto a informação está sendo inserida, o registro pode ser consultado, mas não editado por outros agentes.

Estado do sistema na conclusão:
O usuário está conectado. O prontuário do paciente, incluindo seu histórico médico, é inserido no banco de dados e um registro é adicionado ao log do sistema, mostrando o tempo de início e fim da sessão e a enfermeira envolvida.

Fonte: Sommerville, 2011, p. 74.

Na Figura 12 podemos observar em um modo mais estruturado o cenário da Figura 11 representado em um modelo de casos de uso.

Figura 12 – Exemplo de casos de uso



Fonte: Sommerville, 2011, p. 75.

3.2 Especificação de requisitos

A especificação de requisitos refere-se ao processo de escrever os requisitos de usuário e de sistema em um documento de requisitos. No Quadro 1 podemos observar os quatro principais métodos de especificação de requisitos apresentados por Sommerville (2018, p. 104).

Quadro 1 – Especificação de requisitos

Notação	Descrição
Sentenças em linguagem natural	Os requisitos são escritos usando frases numeradas em linguagem natural. Cada frase deve expressar um requisito.
Linguagem natural estruturada	Os requisitos são escritos em linguagem natural em um formulário ou *template*. Cada campo fornece informações sobre um aspecto do requisito.
Notações gráficas	Modelos gráficos suplementados por anotações em texto, são utilizados para definir os requisitos funcionais do sistema. São utilizados com frequência os diagramas de casos de uso e de sequência da UML.
Especificações matemáticas	Essas notações se baseiam em conceitos matemáticos como as máquinas de estados finitos ou conjuntos. Embora essas especificações possam reduzir a ambiguidade em um documento de requisitos, a maioria dos clientes não compreende uma representação formal.

Fonte: Sommerville, 2018, p. 104.

TEMA 4 – GERENCIAMENTO DE CONFIGURAÇÃO E MUDANÇA

Os sistemas de software estão em constante evolução, acarretando em frequentes mudanças e, portanto, sendo necessário gerenciá-las. Parafraseando Pressman (2011, p. 514), a gestão de configuração de software é um conjunto

de atividades destinadas a gerenciar as alterações identificando os artefatos que precisam ser alterados, estabelecendo relações entre eles, definindo mecanismos para gerenciar diferentes versões desses artefatos, controlando as alterações impostas e auditando e relatando alterações feitas.

Conforme Sommerville (2018, p. 694), o gerenciamento de configuração de um produto de sistema de software envolve quatro atividades intimamente relacionadas:

- **Controle de versão**: envolve manter o controle das várias versões dos componentes do sistema e garantir que as mudanças feitas em componentes por diferentes desenvolvedores não interfiram com as outras.
- **Construção de sistema**: processo de reunir componentes, dados e bibliotecas do programa, compilando-os e ligando-os para criar um sistema executável.
- **Gerenciamento de mudanças**: envolve manter o controle das solicitações de mudança de clientes e desenvolvedores no software já entregue, elaborar os custos e o impacto de fazer essas mudanças e decidir se e quando as alterações devem ser implementadas.
- **Gerenciamento de lançamentos (releases)**: envolve a preparação de software para o lançamento externo e o acompanhamento das versões de sistema que foram lançadas para uso do cliente.

Neste tema desta aula, abordaremos em detalhes as ferramentas **Git** e **Github**, que possibilitam, juntas, atender as quatro atividades do gerenciamento de configuração de software apresentadas por Sommerville.

4.1 Sistema de controle de versão Git

O Git é um sistema de controle de versão de arquivos que, de acordo com Schmitz (2015), possibilita o desenvolvimento de projetos em que diversos desenvolvedores podem contribuir simultaneamente no mesmo projeto, editando e criando novos arquivos e permitindo que esses arquivos possam existir sem o risco de suas alterações serem sobrescritas.

Para instalar o Git, primeiramente devemos acessar a página de downloads da aplicação no endereço: /://git-scm.com/downloads (Acesso em: 29 jan. 2021). Em seguida escolher o sistema operacional desejado e baixar o

arquivo de instalação. Caso o sistema operacional utilizado seja Linux, basta digitar no Terminal o seguinte comando:

sudo apt-get install git

Desse ponto em diante, indiferente do sistema operacional utilizado, trabalharemos somente com a Interface de Linha de Comando (CLI) do respectivo sistema operacional em execução, por exemplo: Prompt de comando no Windows ou Terminal no Linux.

Na sequência estudaremos o processo de configuração e versionamento do Git. Abra a interface de linha de comando do seu sistema operacional e siga os passos a seguir para configurar o Git:

- **Configurar o nome do usuário, e-mail e definição de cores**
  - git config --global user.name “Alex Mateus”
  - git config --global user.email “alex@mateus.com.br”
  - git config --global color.ui true

Crie uma pasta para simular o diretório onde ficará o projeto em desenvolvimento, acesse essa pasta pela linha de comando e digite o seguinte comando para inicializar o versionamento:

- **Criar o repositório Git**
  - git init **(este comando cria uma pasta oculta .git)**

Até este ponto nossa aplicação Git para versionamento e gerenciamento da configuração dos nossos projetos já está pronta para uso, os próximos passos referem-se ao gerenciamento de projetos.

- **Criando o versionamento do projeto**
  - git status **(mostra o status dos arquivos)**
  - git add arquivo **(deixa o arquivo pronto para ser gravado)**
  - git add . **(deixa todos os arquivos do diretório prontos para commit)**
  - git commit **(cria o controle do versionamento)**
  - git commit –m “descrição” **(permite descrever cada commit)**
  - git commit –a –m “descrição” **(cria o commit sem add)**

Após a gravação dos novos arquivos ou dos que foram editados, para visualizar o log das gravações, utilize os seguintes comandos:

- **Visualizar logs do Git**
    - git log **(mostra o log de commits)**
    - git log –p **(mostra todos os commits e alterações)**
    - git log –p -2 **(mostra somente os dois últimos commits)**
    - git log –stat **(mostra estatísticas)**
    - git log --pretty=online **(ajusta a apresentação do log em uma linha)**
    - git log --pretty=format “%h - %a, %ar : %s” **(formatação do log)**
    - git log --since=2.days **(mostra o log dos dois últimos dias)**

Para remover arquivos que estão prontos para commit, mas foram adicionados incorretamente, utilize o seguinte comando:

- **Remover arquivos prontos para commit**
    - arquivo .gitignore

Para remover arquivos adicionados para commit, que possivelmente foram adicionados incorretamente ou que necessitem de novas alterações antes do commit, utilize o seguinte comando:

- **Retirar arquivos do add**
    - git reset HEAD arquivo

Caso tenha realizado um commit indevidamente e queira retorná-lo, basta utilizar o seguinte comando:

- **Retornar os commits realizados**
    - git reset HEAD~1 **(volta um commit)**
    - git reset HEAD~1 --soft **(remove o commit mas mantém as alterações)**
    - git reset HEAD~1 --hard **(exclui o último commit)**

Para voltar em alguma versão anterior do projeto, devemos anotar o código do commit desejado no log de commits, e informá-lo no seguinte comando:

- **Retornar em uma versão anterior do projeto**
    - git checkout “código do commit”

O Git possibilita que sejam criadas várias ramificações do projeto, de modo que cada desenvolvedor pode trabalhar em uma ou mais ramificações.

Para apresentar a ramificação atual, que por padrão será a ramificação master, por ainda não termos criado nenhuma, utilize o seguinte comando:

- **Apresentar a ramificação atual**
  - git branch

Para criar novas ramificações, use o seguinte comando:

- **Criar uma nova ramificação**
  - git checkout –b “nome da ramificação”

Ao criarmos uma nova ramificação, automaticamente o Git nos coloca para trabalhar nela. Para voltar à ramificação principal (master) ou trocar para outras ramificações, utilize os seguintes comandos:

- **Voltar à ramificação principal**
  - git checkout master
  - git checkout “nome da ramificação **(trocar de ramificação)**

Ao criarmos novas ramificações, precisamos associá-las a ramificação master para que não fiquem isoladas no projeto. Para isso utilize o seguinte comando:

- **Associar uma ramificação ao master**
  - git merge “nome da ramificação”
  - git release “nome da ramificação” **(associar a ramificação na ordem em que ocorreram os commits)**

4.2 Plataforma de hospedagem de controle de versão GitHub

O GitHub é uma plataforma para hospedagem dos projetos gerenciados com controle de versão usando o Git. Essa plataforma possibilita que os desenvolvedores possam contribuir nos projetos de qualquer lugar que estejam, podendo fazer o download dos projetos com o Git, contribuir com novas funcionalidades, gravar as contribuições (commits) e enviar novamente o projeto com o Git ao GitHub, para que outros colaboradores possam visualizar todas as novas contribuições adicionadas.

Na sequência veremos como associar nossa aplicação Git a plataforma de hospedagem GitHub. O primeiro passo necessário é realizar o cadastro gratuito acessando o endereço https://github.com/ (Acesso em: 30 jan. 2021).

Após termos nossa conta criada no GitHub, voltamos a nossa interface de linha de comando para gerar uma chave de criptografia pública para enviar nossos projetos ao repositório Github.

- **Gerar a chave pública para o GitHub**
    - ssh-keygen
    - cd ~/.ssh/ **(abre a pasta das chaves)**

O conteúdo do arquivo sra.pub, criado automaticamente ao gerar a chave pública, deve ser copiado e colado na opção “Chaves SSH” nas configurações do GitHub, em seguida criamos um novo repositório para o projeto. Nosso próximo passo é gerar um repositório remoto desse novo repositório do GitHub em nosso Git local.

- **Criar um repositório remoto do GitHub no Git**
    - git remote add origin “endereço do repositório no GitHub”
    - .git/config **(arquivo de configuração do git)**

Em seguida, enviamos os arquivos do nosso projeto do Git local para o GitHub com o seguinte comando:

- **Enviar os arquivos do Git para o GitHub**
    - git push origin máster

Para copiar (clonar) os arquivos de um repositório do GitHub para nosso Git local, utilize o seguinte comando:

- **Clonar um repositório do GitHub para o Git**
    - git clone “endereço do repositório no GitHub” **(clona ramificação master)**

Para clonar demais ou todas as ramificações de um projeto do GitHub para o Git, utilize os seguintes comandos:

- **Clonar demais ramificações do GitHub para o Git**
    - git branch –a **(mostra os branches remotos)**
    - git pull **(verifica se todos os arquivos estão atualizados)**
    - git checkout –b “nome da ramificação” origin/”nome da ramificação” **(cria a ramificação remota no git local)**
    - git pull origin master **(pega as atualizações)**

Para indicar as versões em que cada projeto se encontra, utilizamos o comando tag da seguinte maneira:

- **Criar tag das versões**
    - git tag versão (exemplo de versão? 1.0.1)
    - git tag –l (mostra todas as tags)

Para criar uma release pronta para distribuição, utilize o seguinte comando:

- git push origin master --tags

TEMA 5 – MANUTENÇÃO E EVOLUÇÃO DE SOFTWARE

Conforme abordado por Wazlawick (2013, p. 317), a manutenção de software é como se denomina, em geral, o processo de adaptação e otimização de um software já desenvolvido, bem como a correção de defeitos que ele possa ter. A manutenção é necessária para que um produto de software preserve sua qualidade ao longo do tempo, pois se isso não for feito haverá uma deterioração do valor percebido desse software e, portanto, de sua qualidade.

Nesse sentido, Wazlawick (2013, p. 318) ainda aponta que uma vez desenvolvido, um software terá um valor necessariamente decrescente com o passar do tempo. Isso ocorre porque:

- Falhas são descobertas;
- Requisitos mudam;
- Produtos menos complexos, mais eficientes ou tecnologicamente mais avançados são disponibilizados.

Tendo essa perspectiva, torna-se imperativo que, simetricamente, para manter o valor percebido de um sistema:

- Falhas sejam corrigidas;
- Novos requisitos sejam acomodados;
- Sejam buscadas simplicidade, eficiência e atualização tecnológica.

Parafraseando Pressman (2011, p. 662), independentemente do domínio de aplicação, tamanho ou complexidade, o software continuará a evoluir com o tempo. No âmbito do software, ocorrem alterações quando:

a) São corrigidos erros;

b) Quando há adaptação a um novo ambiente;
c) Quando o cliente solicita novas características ou funções e;
d) Quando a aplicação passa por um processo de reengenharia para proporcionar benefício em um contexto moderno.

Com base em análises detalhadas de softwares industriais e sistemas para desenvolver a teoria unificada para evolução do software, Lehamn (1980) e Lehman e Ramil (1997) propuseram oito leis para explicar a necessidade da evolução do software, conforme são apresentadas por Pressman (2011) e Wazlawick (2013).

1. **Lei da Mudança Contínua (1974)**: afirma que um sistema que é efetivamente usado deve ser continuamente melhorado, caso contrário torna-se cada vez menos útil, pois seu contexto de uso evolui. Se o programa não evoluir, terá cada vez menos valor até que se chegue à conclusão de que vale a pena substituí-lo por outro.
2. **Lei da complexidade crescente (1974)**: expressa que à medida que um programa evolui, sua complexidade inerente aumenta, porque as correções feitas podem deteriorar sua organização interna.
3. **Lei da autorregulação (1974)**: O processo de evolução do sistema é autorregulado com distribuição do produto e medidas de processo próximo do normal.
4. **Lei da conservação da estabilidade organizacional (1980)**: expressa que a taxa média efetiva de trabalho global em um sistema em evolução é invariante no tempo, isto é, ela não aumenta nem diminui.
5. **Lei da conservação da familiaridade (1980)**: conforme um sistema evolui, tudo o que está associado a ele, por exemplo, desenvolvedores, pessoal de vendas, usuários, deve manter o domínio de seu conteúdo e comportamento para uma evolução satisfatória.
6. **Lei do crescimento contínuo (1980)**: estabelece que o conteúdo funcional de um sistema deve crescer continuamente para manter a satisfação do usuário.
7. **Lei da qualidade em declínio (1996)**: expressa que a qualidade de um sistema vai parecer diminuir com o tempo, a não ser que medidas rigorosas sejam tomadas para mantê-lo e adaptá-lo.
8. **Lei do sistema de realimentação (1996)**: estabelece que a evolução de sistemas é um processo multinível, multilaço e multiagente de

realimentação, devendo ser encarado dessa forma para que se obtenham melhorias significativas em uma base razoável.

Conforme apresentado em Sommerville (2018, p. 246), a manutenção de software é o processo geral de mudança em um sistema depois que ele é liberado para uso, existindo três diferentes tipos de manutenção de software:

1. **Correção de defeitos;**
2. **Adaptação ambiental;**
3. **Adição de funcionalidade.**

De acordo com Sommerville (2018, p. 246-247):

> Na prática não há uma distinção nítida entre esses tipos de manutenção, quando se adapta um sistema a um novo ambiente, é possível adicionar funcionalidades para tirar vantagens das novas características do ambiente. Os defeitos de software são expostos frequentemente porque os usuários utilizam o sistema de maneiras imprevistas, e a melhor forma de corrigir esses defeitos é mudar o sistema a fim de acomodar a forma como eles trabalham.

CONSIDERAÇÕES FINAIS

Nesta aula, abordamos como tema principal a arquitetura de software. Iniciamos o conteúdo caracterizando o conceito de projeto, conforme definições do Guia PMBOK e do Guia de Gerenciamento de Projetos do SISP, dado nos aprofundamos na definição da elaboração de projetos para o desenvolvimento de softwares.

Conforme observamos ao final do Tema 1 desta aula, o projeto pode ser elaborado em dois formatos, o projeto conceitual, definido conforme a visão do cliente, normalmente de um modo descritivo e explicativo, e o projeto técnico, elaborado a partir das definições do projeto conceitual com foco nas definições técnicas para o desenvolvimento.

Aprendemos, também, a partir do Tema 2, diversos padrões de arquitetura de software que podem ser empregados no desenvolvimento de sistemas de acordo com as especificações. Também pudemos compreender que, dependendo da complexidade de um sistema, muitas vezes vários padrões de arquitetura de software necessitam serem utilizados no mesmo projeto. Essa necessidade da aplicação de mais de um padrão de arquitetura, normalmente é identificada a partir do levantamento e especificação dos requisitos, conforme vimos no Tema 3.

Na sequência aprendemos sobre o uso da ferramenta Git para automatizar o gerenciamento e versionamento do projeto de software, como também proporcionar o trabalho de forma colaborativa. Em conjunto estudamos o uso da plataforma GitHub para disponibilizar os arquivos do projeto.

Finalizamos esta aula com a análise das 8 leis de Lehman e Ramil, que demonstram a necessidade da evolução do software. Posteriormente, aprenderemos sobre estimativas de esforço para o desenvolvimento do software, focando em estimativas para a definição de prazos, número de desenvolvedores e custos do projeto de software.

REFERÊNCIAS

BRASIL. Ministério do Planejamento, Orçamento e Gestão. Secretaria de Logística e Tecnologia da Informação. **Metodologia de Gerenciamento de Projetos do SISP**. Brasília: MP, 2011.

LEHMAN, M. M. Programs, Life Cycles, and Laws of Software Evolution. **Proceedings of the IEEE**, 68(9), p. 1.060-76, 1980.

LEHMAN, M. M.; J. F. RAMIL, P. D. **Metrics and Laws of Software Evolution** – The Nineties View. Proc. 4th International Software Metrics Symposium. Albuquerque, NM, USA, 1997, p. 20-32.

PMI. **Um guia do conhecimento em gerenciamento de projetos**. Guia PMBOK. 6. ed. EUA: Project Management Institute, 2017.

PRESSMAN, R. S. **Engenharia de Software**: uma abordagem profissional. 7. ed. Porto Alegre: AMGH, 2011.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011.

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

PFLEEGER, S. L. **Engenharia de Software**: Teoria e Prática. 2. ed. São Paulo: Prentice Hall, 2004.

WAZLAWICK, R. S. **Engenharia de Software**: Conceitos e Práticas. São Paulo: Elsevier, 2013.