UNIVERSIDADE DO CONTESTADO – UnC

CURSO DE ENGENHARIA DE SOFTWARE

DEBORA CRISTINA FONTANELLA

**PROTÓTIPO DE SOFTWARE PARA BUSCA E DIVULGAÇÃO DE PRODUTORES LOCAIS DE ALIMENTOS**

CONCÓRDIA

2020

DEBORA CRISTINA FONTANELLA

**PROTÓTIPO DE SOFTWARE PARA BUSCA E DIVULGAÇÃO DE PRODUTORES LOCAIS DE ALIMENTOS**

Trabalho de Conclusão de curso, apresentado como exigência para obtenção do título de Bacharel em Engenharia de Software, pela Universidade do Contestado - UNC, Campus Concórdia, sob Orientação Específica do Professor Geordano Dalmédico e Orientação Metodológica da Professora Gisleine Merib Kichel.

CONCÓRDIA

2020

**PROTÓTIPO DE SOFTWARE PARA BUSCA E DIVULGAÇÃO DE PRODUTORES LOCAIS DE ALIMENTOS**

DEBORA CRISTINA FONTANELLA

Este trabalho de Conclusão de Curso foi submetido ao processo de avaliação pela Banca Examinadora para a obtenção do Título de:

**Bacharel em Engenharia de Software**

E aprovada(o) na sua versão final em XX de XXXXXXX de XXXX, atendendo às normas da legislação vigente da Universidade do Contestado e Coordenação do Curso de XXXXXXXXX.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. XXXXXXX

Coordenador do Curso de XXXXXXXX

**BANCA EXAMINADORA:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Profº. XXXXXXXXXX

(Orientador)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. XXXXXXXXXXX

(Avaliador)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. XXXXXXXXXX

(Avaliador)

*DEDICATÓRIA (Opcional)*

*Fonte Arial 12. Não é necessário escrever a palavra Dedicatória como título.*

**AGRADECIMENTOS**

Descrever aqui os agradecimentos. Texto justificado, fonte Arial 12.

**Epígrafe (opcional)**

*Texto em que o autor apresenta uma citação, seguida de indicação de autoria, relacionada com a matéria tratada no corpo do trabalho, não deve ser colocado o título Epígrafe.*

**RESUMO**

Na língua do texto (vernácula). Obrigatório. O termo **RESUMO** em letras maiúsculas, em negrito centralizado. O texto do resumo deve conter de 150 a 500 palavras, fonte Arial 12, espaço entre linhas simples. Parágrafo único, sem recuo da margem.

**Palavras-Chave**: Acrescentar termos ou palavras que representam o assunto abordado no trabalho, fonte Arial 12, espaçamento simples. O número mínimo de palavras-chave é 3 e o máximo 6. Separar as palavras com **ponto**. Evitar repetir palavras do título.

**ABSTRACT**

Resumo em língua estrangeira (inglês). Obrigatório. O texto do resumo deve conter de 150 a 500 palavras, fonte Arial 12, espaço entre linhas simples. Parágrafo único, sem recuo da margem.

**Keywords**: Acrescentar termos ou palavras que representam o assunto abordado no trabalho em língua estrangeira. Fonte Arial 12, espaçamento simples. Mínimo de 3 e máximo de 6 palavras. Separar as palavras com **ponto**.

**LISTA DE FIGURAS**

[Figura 1: Camadas da engenharia de software 26](#__RefHeading___Toc15752_1320429428)

[Figura 2: Representação da metodologia do processo de software 28](#__RefHeading___Toc3053_1634033230)

[Figura 3: Ciclo de vida clássico 31](#__RefHeading___Toc15756_1320429428)

[Figura 4: Paradigma de prototipação 32](#__RefHeading___Toc15758_1320429428)

[Figura 5: Modelo de processo orientado a cronograma 34](#__RefHeading___Toc15760_1320429428)

[Figura 6: Representação gráfica UML de Ator, Caso de Uso e Relacionamento de comunicação 39](#__RefHeading___Toc15762_1320429428)

[Figura 7: Exemplo de relacionamento de inclusão 39](#__RefHeading___Toc15764_1320429428)

[Figura 8: Exemplo de relacionamento de extensão 40](#__RefHeading___Toc15766_1320429428)

[Figura 9: Exemplo de relacionamento de generalização entre atores 41](#__RefHeading___Toc15768_1320429428)

[Figura 10: Exemplo de relacionamento de generalização entre casos de uso 41](#__RefHeading___Toc3177_1634033230)

[Figura 11: Exemplo de caso de uso com representação de fronteira de sistema 42](#__RefHeading___Toc15770_1320429428)

[Figura 12: Representação de uma Classe em UML 43](#__RefHeading___Toc15772_1320429428)

[Figura 13: Exemplos de associações entre classes 44](#__RefHeading___Toc15774_1320429428)

[Figura 14: Exemplos de associação com uso de símbolos de multiplicidade 45](#__RefHeading___Toc15776_1320429428)

[Figura 15: Exemplo de utilização dos recursos UML nome da associação, direção de leitura e papéis 46](#__RefHeading___Toc15778_1320429428)

[Figura 16: Exemplo de representação de uma classe associativa 47](#__RefHeading___Toc15780_1320429428)

[Figura 17: Exemplo de notação de relações de agregação e composição 47](#__RefHeading___Toc15782_1320429428)

[Figura 18: Exemplo de notação de relação de herança. 48](#__RefHeading___Toc15784_1320429428)

[Figura 19: Exemplos de notação UML para objetos em um diagrama de sequência 49](#__RefHeading___Toc15786_1320429428)

[Figura 20: Notação UML para linha de vida 50](#__RefHeading___Toc15788_1320429428)

[Figura 21: Notação UML para mensagens reflexivas 51](#__RefHeading___Toc15790_1320429428)

[Figura 22: Exemplo de notação de ocorrência de execução 52](#__RefHeading___Toc15792_1320429428)

[Figura 23: Exemplo de arquitetura cliente-servidor para uma biblioteca de filmes 54](#__RefHeading___Toc15794_1320429428)

[Figura 24: Padrão MVC 56](#__RefHeading___Toc21974_1320429428)

[Figura 25: Diagrama de classes de uma aplicação implementada no padrão Transaction Script 57](#__RefHeading___Toc21976_1320429428)

[Figura 26: Exemplo de modelo de entidade-relacionamento 61](#__RefHeading___Toc21978_1320429428)

[Figura 27: Representação de um sistema de banco de dados 62](#__RefHeading___Toc21980_1320429428)

[Figura 28: Implementação cliente-servidor de uma aplicação Web 66](#__RefHeading___Toc21982_1320429428)

[Figura 29: Camadas de uma aplicação web 67](#__RefHeading___Toc21984_1320429428)

[Figura 30: Estrutura de um elemento HTML 68](#__RefHeading___Toc21986_1320429428)

[Figura 31: estrutura de um conjunto de regras CSS 69](#__RefHeading___Toc21988_1320429428)

**LISTA DE QUADROS**

Quadro 1: Atividades de apoio típicas 24

Quadro 2: Simbologia para representar multiplicidades 42

Quadro 3: Conectividades em relação às multiplicidades 43

Quadro 4: Notação gráfica para tipos de mensagem em um diagrama de sequência 48

**LISTA DE TABELAS (opcional)**

Tabela 1 – Xxxxxxxxxxxxxxxxxxxxx............................................................................26

A tabela possui somente as primeiras linhas que indicam as variantes e a última linha. As linhas do meio não existem e as laterais (esquerda e direita) também não.

**LISTA DE GRÁFICOS (opcional)**

Gráfico 1 – Xxxxxxxxxxxxxxxxxxxxxx ........................................................................36

**LISTA DE ABREVIATURAS E/OU SIGLAS**

App – Aplicativo

Ajax – *Asynchronus JavaScript and XML*

CSS *– Cascade Style Sheet*

DCU – Diagrama de Caso de Uso

DDL – *Data-Definition Language*

DML – *Data Manipulation Language*

HTML *– Hyper Text Markup Language*

HTTP – *Hyper Text Transfer Protocol*

HTTPS *– Hyper Text Transfer Protocol Secure*

JSON – *JavaScript Object Notation*

MCU – Modelo de Caso de Uso

MVC – *Model View controller*

PWA *– Progresive Web App*

REST – *Representational State Transfer*

SGBD – Sistema Gerenciador de Banco de Dados

SIBC – Sistema de informação baseado em computador

SOAP – *Simple Object Access Protocol*

SQL – *Structured Query Language*

UML – *Unified Modeling Language*

**SUMÁRIO**

[1 INTRODUÇÃO 18](#__RefHeading___Toc22146_1320429428)

[1.1 OBJETIVOS 19](#__RefHeading___Toc7218_1320429428)

[1.1.1 Objetivo Geral 19](#__RefHeading___Toc7220_1320429428)

[1.1.2 Objetivos Específicos 19](#__RefHeading___Toc7222_1320429428)

[2 REFERENCIAL TEÓRICO 20](#__RefHeading___Toc7224_1320429428)

[2.1 LOCAVORISMO 20](#__RefHeading___Toc7226_1320429428)

[2.1.1 Alimento Local 21](#__RefHeading___Toc7228_1320429428)

[2.2 SOFTWARE E SISTEMAS DE INFORMAÇÃO 22](#__RefHeading___Toc7230_1320429428)

[2.3 ENGENHARIA DE SOFTWARE 25](#__RefHeading___Toc7232_1320429428)

[2.4 PROCESSO DE SOFTWARE 26](#__RefHeading___Toc7234_1320429428)

[2.5 MODELOS DE PROCESSO DE SOFTWARE 30](#__RefHeading___Toc7236_1320429428)

[2.5.1 Modelo Cascata 30](#__RefHeading___Toc7238_1320429428)

[2.5.2 Modelos de Processo Evolucionários 31](#__RefHeading___Toc7240_1320429428)

[2.5.2.1 Prototipação 32](#__RefHeading___Toc7242_1320429428)

[2.5.2.2 Modelo orientado a cronograma 33](#__RefHeading___Toc7244_1320429428)

[2.6 REQUISITOS DE SOFTWARE 34](#__RefHeading___Toc7246_1320429428)

[2.6.1 Requisitos Funcionais e Não Funcionais 35](#__RefHeading___Toc7248_1320429428)

[2.6.2 Engenharia de Requisitos 36](#__RefHeading___Toc7250_1320429428)

[2.7 UML 37](#__RefHeading___Toc7252_1320429428)

[2.7.1 Diagrama de Caso de Uso 37](#__RefHeading___Toc7254_1320429428)

[2.7.1.1 Caso de uso 38](#__RefHeading___Toc7256_1320429428)

[2.7.1.2 Atores 38](#__RefHeading___Toc7258_1320429428)

[2.7.1.3 Relacionamentos 38](#__RefHeading___Toc7260_1320429428)

[2.7.1.3.1 Relacionamento de comunicação 38](#__RefHeading___Toc7262_1320429428)

[2.7.1.3.2 relacionamento de inclusão 39](#__RefHeading___Toc7264_1320429428)

[2.7.1.3.3 Relacionamento de extensão 40](#__RefHeading___Toc7266_1320429428)

[2.7.1.3.4 Relacionamento de generalização 41](#__RefHeading___Toc7268_1320429428)

[2.7.1.4 Fronteira do sistema 42](#__RefHeading___Toc7270_1320429428)

[2.7.2 Diagrama de Classes 42](#__RefHeading___Toc7272_1320429428)

[2.7.2.1 Classes 43](#__RefHeading___Toc7274_1320429428)

[2.7.2.2 Associações 44](#__RefHeading___Toc7276_1320429428)

[2.7.2.3 Multiplicidades 45](#__RefHeading___Toc7278_1320429428)

[2.7.2.4 Participações 46](#__RefHeading___Toc7280_1320429428)

[2.7.2.5 Nome de associação, direção de leitura e papéis 46](#__RefHeading___Toc7282_1320429428)

[2.7.2.6 Classes associativas 47](#__RefHeading___Toc7284_1320429428)

[2.7.2.7 Agregações e composições 47](#__RefHeading___Toc7286_1320429428)

[2.7.2.8 Generalizações e especializações 48](#__RefHeading___Toc7288_1320429428)

[2.7.3 Diagrama de Sequência 49](#__RefHeading___Toc7290_1320429428)

[2.7.3.1 Atores 49](#__RefHeading___Toc7292_1320429428)

[2.7.3.2 Objetos 49](#__RefHeading___Toc7294_1320429428)

[2.7.3.3 Linha de vida 50](#__RefHeading___Toc7296_1320429428)

[2.7.3.4 Mensagens 51](#__RefHeading___Toc7298_1320429428)

[2.7.3.5 Ocorrências de execução 52](#__RefHeading___Toc7300_1320429428)

[2.7.3.6 Criação e destruição de objetos 53](#__RefHeading___Toc7302_1320429428)

[2.8 ARQUITETURA DE SOFTWARE 53](#__RefHeading___Toc7304_1320429428)

[2.8.1 Arquitetura Cliente-Servidor 54](#__RefHeading___Toc7306_1320429428)

[2.8.2 Arquitetura MVC 56](#__RefHeading___Toc7308_1320429428)

[2.8.3 Arquitetura de Transação (Transaction Script) 57](#__RefHeading___Toc7310_1320429428)

[2.9 TESTES 59](#__RefHeading___Toc7312_1320429428)

[2.10 BANCO DE DADOS 61](#__RefHeading___Toc7314_1320429428)

[2.10.1 Modelos de Dados 61](#__RefHeading___Toc7316_1320429428)

[2.10.1.1 Modelo de entidade-relacionamento 62](#__RefHeading___Toc7318_1320429428)

[2.10.2 Sistema Gerenciador de Banco de Dados 62](#__RefHeading___Toc7320_1320429428)

[2.10.2.1 MySQL e MariaDB 63](#__RefHeading___Toc7322_1320429428)

[2.11 PROGRESSIVE WEB APPS 64](#__RefHeading___Toc7324_1320429428)

[2.11.1 Requisitos 66](#__RefHeading___Toc7326_1320429428)

[2.12 APLICAÇÕES WEB 67](#__RefHeading___Toc7328_1320429428)

[2.12.1 Construção 68](#__RefHeading___Toc22129_1320429428)

[2.12.1.1 HTML 69](#__RefHeading___Toc22131_1320429428)

[2.12.1.2 CSS 69](#__RefHeading___Toc22133_1320429428)

[2.12.1.3 JavaScript 70](#__RefHeading___Toc22135_1320429428)

[2.12.1.4 Frameworks 71](#__RefHeading___Toc22137_1320429428)

[2.12.1.4.1 Express 71](#__RefHeading___Toc22139_1320429428)

[2.12.1.4.1 VueJS 71](#__RefHeading___Toc22141_1320429428)

[2.13 RESTFul Web Services 72](#__RefHeading___Toc22143_1320429428)

[3 MATERIAL E MÉTODOS 74](#__RefHeading___Toc7336_1320429428)

[3.1 PESQUISA BIBLIOGRÁFICA 74](#__RefHeading___Toc7338_1320429428)

[3.2 IDENTIFICAÇÃO DE STAKEHOLDERS 74](#__RefHeading___Toc7340_1320429428)

[3.3 ANÁLISE DE REQUISITOS 74](#__RefHeading___Toc7342_1320429428)

[3.4 PLANEJAMENTO 75](#__RefHeading___Toc7344_1320429428)

[3.5 MODELAGEM 75](#__RefHeading___Toc7346_1320429428)

[3.5.1 Diagrama de Casos de Uso 75](#__RefHeading___Toc7348_1320429428)

[3.5.2 Diagrama de Classes 76](#__RefHeading___Toc7350_1320429428)

[3.5.3 Diagrama de Sequência 76](#__RefHeading___Toc7352_1320429428)

[3.6 CONSTRUÇÃO 76](#__RefHeading___Toc7354_1320429428)

[3.6.1 Documentação dos Casos de Teste 76](#__RefHeading___Toc7356_1320429428)

[3.6.2 Codificação 77](#__RefHeading___Toc7358_1320429428)

[3.6.3 Execução dos Casos de Teste 78](#__RefHeading___Toc7360_1320429428)

[3.7 ENTREGA 78](#__RefHeading___Toc7362_1320429428)

[4 RESULTADOS E DISCUSSÕES 79](#__RefHeading___Toc7364_1320429428)

[5 CONCLUSÃO 80](#__RefHeading___Toc7366_1320429428)

[REFERÊNCIAS 81](#__RefHeading___Toc7368_1320429428)

# 1 INTRODUÇÃO

O cotidiano cada vez mais acelerado que vem sendo levado, principalmente em grandes centros, cria a necessidade de otimização do tempo, a fim de criar espaço para a realização de mais atividades ao longo do dia. Implicando, inclusive na alimentação, onde o preparo e apreciação de refeições vem sendo substituído por refeições rápidas e, em sua maioria, baseada em alimentos industrializados e altamente processados. Contribuindo, junto de outros hábitos, para a redução da qualidade de vida. Tomando ciência dos impactos negativos desse estilo de vida, as pessoas vem buscando minimizá-lo, através da adoção de hábitos alimentares mais saudáveis, por exemplo.

Isso inclui a busca por refeições compostas de alimentos menos processados, cultivados com menos agrotóxicos e mais frescos, os quais são frequentemente encontrados em pequenos produtores locais. Há uma tendência alimentar emergente, que incorpora esses hábitos, baseando-se na busca por uma alimentação mais saudável e sustentável, consequentemente sugerindo um modelo baseado no consumo de alimentos locais, denominada locavorismo.

O locavorismo opõe-se às cadeias produtivas globalizadas de alimentos e busca o estabelecimento de um hábito alimentar mais natural e menos prejudicial ao meio ambiente, sugerindo que o consumidor conheça as origens por trás do alimento que está adquirindo e, consequentemente, busque adquirir diretamente do pequeno produtor local. Os impactos positivos dessa proposta são o fortalecimento e valorização da agricultura familiar e da economia local. Além disso, reduz-se os impactos ambientais causados por fazendas de monocultura e criação de animais, além de dispensar o transporte por longas distâncias.

Uma dos principais desafios para viabilização e popularização do locavorismo é a dificuldade do consumidor conhecer e ter acesso a produtores locais. Atualmente, as formas mais comuns de comercialização de alimentos locais são através de feiras ou do fornecimento para lojas e supermercados. Para o produtor, o problema da comercialização em feiras é que o comércio sofre limitação de espaço físico e também temporal, uma vez que as mesmas não acontecem diariamente, por exemplo. Tratando-se do fornecimento para lojas e supermercados, a margem de lucro dos produtores acaba sendo baixa e até mesmo injusta, se comparado ao valor do produto pago pelo consumidor final e a margem de lucro do mercado ou loja dos quais são fornecedores.

Tendo em vista a crescente tendência da computação móvel e de desenvolvimento de aplicativos para as mais distintas finalidades, bem como a popularização dos dispositivos móveis, a existência de um aplicativo em que pequenos produtores locais de alimentos pudessem manter um perfil, fornecendo informações sobre si e seus produtos, para serem apresentados em lista aos consumidores interessados nas categorias de produtos fornecidos, permitiria o maior alcance de vendas em potencial, contribuindo para a viabilização do locavorismo e garantindo assim os diversos benefícios socioambientais citados acima. Dessa forma, que características um protótipo de aplicativo deveria ter para permitir a busca e divulgação de produtores de orgânicos, artesanais e coloniais?

## 1.1 OBJETIVOS

### 1.1.1Objetivo Geral

Desenvolver um protótipo de aplicativo que possibilite a busca e divulgação de produtores locais de alimentos.

### 1.1.2 Objetivos Específicos

* Revisar bibliografia acerca do tema abordado e especificações técnicas relacionadas à construção do protótipo;
* Realizar levantamento de requisitos do projeto a ser desenvolvido;
* Com base nos requisitos, modelar o protótipo;
* Implementar o modelo, desenvolvendo um protótipo de aplicativo para plataforma Android;
* Disponibilizar protótipo desenvolvido para testes.

# 2 REFERENCIAL TEÓRICO

## 2.1 LOCAVORISMO

Segundo Rudy (2012), locavorismo é um movimento de ativismo alimentar que defende o consumo baseado em alimentos locais e produzidos de forma sustentável, como melhor modelo para a saúde humana e do meio ambiente. O movimento une pessoas que, infelizes com alimentos industrializados, originários de fazendas de monocultura, cultivados com agrotóxicos, que são oferecidos atualmente em supermercados, buscam uma forma diferente de se alimentar. A palavra “locavorismo” deriva do termo “*locavore”,* cunhado por Jessica Pretience em 2006, que designa a pessoa interessada no consumo de alimentos locais. Em 2007, *locavore* foi escolhida pelo *New Oxford American Dictionary* como palavra do ano e desde então, o movimento e suas ideias centrais passaram a ganhar popularidade nos Estados Unidos.

Algumas definições de locavorismo sugerem uma alimentação baseada apenas em alimentos produzidos no raio de até 100 milhas, equivalente a aproximadamente 160,93 km, outras sugerem o consumo de alimentos produzidos somente dentro do estado ou região. Apesar dessas definições não estarem incorretas, a questão central vai além de simples limites geográficos, trata principalmente sobre o desenvolvimento de uma relação alimentar diferente, onde as origens do alimento são conhecidas, isto é, a fazenda e o agricultor que o produziu. Atualmente, nos EUA, ser um locavorista é também conhecer os métodos e produtos envolvidos no processo produtivo e o impacto ambiental causado durante a produção do alimento. Implicitamente, sugere-se que sejam consumidos mais vegetais de época e também que se reduza o consumo de carne, tendo em vista os impactos ambientais da indústria agroalimentar. (RUDY, 2012).

Azevedo (2015) apresenta que, na percepção do locavorista, o alimento local é mais saboroso e fresco, sua produção favorece a agricultura familiar e promove a agricultura urbana, sistemas agroalimentares sustentáveis e o bem-estar animal, além de estimular a economia local através da venda direta ao consumidor. Um estudo realizado por Painter (2008, apud AZEVEDO, 2015) realizado com *chefs* e donos de restaurante, indica novamente a visão do alimento local como mais fresco e superior em qualidade, além de criar um diferencial para o negócio e apoiar agricultores locais.

### 2.1.1 Alimento Local

Não há consenso sobre a definição de alimento local. Diferentes estudos sugerem limites de distância entre o local de produção e consumo a fim de conceder ao alimento a característica de “local”, entretanto, não apresentam consenso entre si. Outros, consideram limites geográficos como estados, regiões, ou até mesmo um conjunto de estados. Uma definição aparentemente mais adequada é apresentada pelo Departamento de Agricultura dos Estados Unidos (USDA), ao assumir mercados alimentares locais e regionais como aqueles onde o produto é adquirido diretamente do produtor, relacionando-o, frequentemente a pequenas propriedades rurais localizadas próximo a regiões metropolitanas (JOHNSON; AUSSENBERG; COWAN, 2013; MARTINEZ et al.; 2010, apud AZEVEDO, 2015).

Thompon, Harper e Kraus (2008), por sua vez, conceituam alimento local tendo em vista a forma como a produção, processamento, transporte e consumo afetam a saúde e o meio ambiente. Para os autores, a proximidade o local de produção do alimento indica uma maior probabilidade de superioridade qualitativa, além de um processo de transporte com menor uso de energia e emissão de poluentes, características certamente buscadas por locavoristas ao consumir alimentos locais. Essa é uma das partes da definição de alimento local, a outra considera aspectos de sustentabilidade aplicadas no processo produtivo do alimento como capazes caracterizá-los “locais” em relação a alimentos de origens convencionais. Métodos sustentáveis incluem a eliminação ou redução do uso de fertilizantes sintéticos ou derivados do petróleo, bem como de práticas que prejudiquem o solo, provoquem incêndios ou poluam o ar e a água. Para alguns consumidores, a sustentabilidade está também ligada a práticas trabalhistas justas e ao bem-estar animal. Por fim, estendem o conceito de produtor local também àqueles responsáveis pelo processamento do alimento, mas que incorporam essas noções ao processo produtivo, preocupando-se com a história por trás da comida.

Para Thompon, Harper e Kraus (2008), além dos benefícios ambientais associados à adoção de práticas sustentáveis na produção de alimentos locais, evitar o uso de pesticidas e demais químicos podem beneficiar a saúde dos consumidores. Halweil (2003) apresenta outros benefícios, como a redução de queima de combustíveis fósseis (pesquisas apontam que uma refeição preparada com ingredientes importados pode facilmente consumir energia e emitir gases de efeito-estufa quatro vezes mais do que uma refeição preparada com ingredientes locais); preservação propriedades agrícolas e agricultores locais (agricultores incluídos no comércio local estão menos propensos à extinção); sabor superior (uma das razões pelas quais o locavorismo atraiu a atenção de chefs, críticos de comidas e demais apreciadores); redução de riscos de segurança alimentar (viajar por longas distâncias e passar por inúmeras mãos expõe o alimento a maiores chances de contaminação).

Os empecilhos identificados para a expansão do locavorismo incluem:

[...] restrições de produção para pequenas propriedades; falta de sistemas de distribuição para levar os alimentos locais para os mercados convencionais; pesquisa limitada; falta de treinamento e informação do consumidor para promover a comercialização; e as incertezas relacionadas aos requisitos padronizados de segurança sanitária das agências reguladoras que podem afetar a produção local de alimentos (JOHNSON; AUSSENBERG; COWAN, 2013; MARTINEZ et al., 2010, apud AZEVEDO, 2015, p. 84).

Nesse sentido, medidas como a criação de canais de comercialização alternativos, apoio institucional, como a criação de programas governamentais ou políticas de subsídios econômicos para agricultores, vêm contribuindo para que produtores locais se mantenham no mercado (JOHNSON, AUSSENBER; COWAN , 2013 apud AZEVEDO, 2015).

## 2.2 SOFTWARE E SISTEMAS DE INFORMAÇÃO

Basicamente, podemos definir sistema como um conjunto de elementos inter-relacionados com um objetivo comum. Em uma definição mais elaborada, sistema seria um conjunto de elementos interconectados harmonicamente, de modo a formar um todo organizado. Um sistema artificial, isto é, um sistema produzido, que não existe naturalmente, é intencionalmente criado como um conjunto de partes, elementos ou componentes inter-relacionados (os subsistemas) que visa a realização de determinado objetivo, no ambiente em que está inserido. Para alcançar os objetivos do sistema, cada subsistema tem seu objetivo próprio e se acha conectado a outro subsistema, do qual recebe os insumos ou entradas (SBROCCO e MACEDO, 2012).

Um sistema de informação, por sua vez, é um sistema artificial que coleta, processa, armazena, analisa e dissemina informação para um propósito específico. O propósito dos sistemas de informação tem sido definido como obter a informação correta para as pessoas certas, no tempo certo, na quantidade certa e no formato certo. Um sistema de informação não necessariamente é um sistema computadorizado, embora a maioria o seja. Por isso, frequentemente o termo “sistema de informação” é usado como sinônimo de “sistema de informação baseado em computador” (RAINER JR. e CEGIELSKI, 2016).

Ainda, de acordo com Rainer Jr. e Cegielski (2016), um sistema de informação baseado em computador (SIBC), é um sistema de informação que usa a tecnologia de computador para realizar algumas ou todas as tarefas pretendidas pelo sistema. Sendo composto por:

* Hardware: dispositivos físicos, responsáveis pela recepção de dados de entrada (teclado, mouse), processamento (processador) e exibição (monitor, impressora);
* Software: um programa ou conjunto de programas que definem como o processamento de dados deverá ocorrer;
* Banco de dados: conjunto de arquivos ou tabelas responsável pelo armazenamento de dados;
* Rede: sistema de conexão (com ou sem fio) que permitem aos computadores compartilhar recursos;
* Procedimentos: instruções sobre como combinar todos os componentes para processar informações e gerar a saída desejada;
* Pessoas (ou peopleware): indivíduos que utilizam o hardware e software.

Os recursos mais importantes de SIBCs, segundo Rainer Jr. e Cegielski (2016) são: realizar cálculos numéricos de alta velocidade e alto volume; fornecer comunicação e colaboração rápidas e precisas dentro da organização e entre organizações; armazenar enormes quantidades de informação em um espaço fácil de acessar, ainda que pequeno; permitir acesso rápido e barato a enormes quantidades de informação em todo o mundo; interpretar grandes quantidades de dados de modo rápido e eficiente; automatizar tanto processos de negócio semiautomáticos quanto tarefas manuais. A parte do sistema de informação projetada para permitir a realização de tarefas como as citadas é o software.

Para Carvalho e Lorena (2017), um software é “nada mais que uma sequência de passos ou instruções descritos por um algoritmo, que, quando executados, fazem com que o computador realize uma tarefa”. O algoritmo é uma sequência de instruções, escritas com uma linguagem de programação, que coordenam o hardware para a execução de um determinado conjunto de operações para obtenção do resultado pretendido. O funcionamento de um software pode ser resumido, de forma básica, no seguinte processo: o software recebe dados ou valores como entrada, em seguida realiza o conjunto de operações descritos pelo algoritmo, e então retornam um resultado que pode ser novos dados ou uma sequência de ações.

Pressman (2016) classifica softwares em sete grandes categorias, sendo elas:

* Software de sistema: Conjunto de programas feito para atender a outros programas, por exemplo: sistemas operacionais, drivers, compiladores, gerenciadores de arquivos.
* Software de aplicação: Programas independentes criados para solucionar uma necessidade específica de negócio. A exemplo: processamento de dados para facilitar tomadas de decisão ou operações comerciais.
* Software de engenharia/científico: são programas utilizados no suporte a atividades científicas ou de engenharia, através do fornecimento de um modelo de simulação, ou realização de grande volume de cálculos complexos que levariam muito mais tempo caso fossem executados por um humano, por exemplo.
* Software embarcado: é um programa residente em um produto ou sistema, que implementa funções limitadas a fim de permitir o controle ou utilização de recursos de seu hospedeiro. Exemplos: controle do painel de um forno microondas, painéis digitais de automóveis.
* Software para linha de produtos: Trata-se de um conjunto de softwares que compartilham características em comum a fim de solucionar problemas específicos, para que possam ser reutilizados na construção de softwares que busquem solucionar tais problemas.
* Aplicações Web/aplicativos móveis: trata-se de uma categoria de software voltada a redes que compreende uma grande variedade de aplicações. Sua principal característica é a mobilidade e a possibilidade de utilização através de navegadores (aplicações web) ou instalação em dispositivos móveis (aplicativos móveis).

Para Pressman (2016), software é, atualmente, uma tecnologia indispensável para negócios, ciência e engenharia e se incorpora a sistemas das mais diversas áreas. É responsável pela viabilização de novas tecnologias, como engenharia genética e nanotecnologia; extensão de tecnologias existentes, como telecomunicações; e mudança radical em tecnologias antigas, como a mídia. Foi também a força motriz por trás da revolução do computador pessoal e, atualmente pode ser acessado no mundo todo através de um navegador web, utilizado em aplicações para dispositivos móveis, além de permear as mais diversas atividades humanas, como relacionamentos, compras, pesquisas bibliográficas, entre outras.

## 2.3 ENGENHARIA DE SOFTWARE

Schach (2010) define engenharia de software como uma disciplina cujo objetivo é produzir software isento de falhas, entregue dentro de prazo e orçamento previstos, atendendo também às necessidades do cliente. Além disso, o software deve ser fácil de ser modificado quando as necessidades do usuário mudarem.

Em uma definição formal, de acordo com a IEEE ( [?] *apud*. PRESSMAN 2016, p. 15), engenharia de software é: “(1) A aplicação de uma abordagem sistemática, disciplinada e quantificável no desenvolvimento, na operação e na manutenção de software; isto é, a aplicação de engenharia ao software. (2) O estudo de abordagens como definido em (1)”. Porém, segundo Pressman (2016), essa abordagem deve também apresentar adaptabilidade e agilidade, para que seja aplicável aos diversos cenários e equipes de desenvolvimento.

De acordo com Sommerville (2011), a engenharia de software preocupa-se com todos os aspectos da produção do software, desde os estágios iniciais de especificação, até a manutenção, incluindo não apenas aspectos técnicos de desenvolvimento, mas também atividades como gerenciamento de projeto e desenvolvimento de ferramentas, métodos e teorias para apoiar a produção de software. Além disso, enquanto uma disciplina de engenharia, tem como objetivo aplicar teorias, métodos e ferramentas adequados ao cenário e suas restrições.

Ainda, segundo o autor, a importância da disciplina tem dois motivos. O primeiro é que cada vez mais, indivíduos e sociedade dependem de softwares avançados e portanto deve-se viabilizar sua construção econômica e rapidamente.. Em segundo, para a maior parte dos sistemas, o maior custo está relacionado a mudanças no software depois que já está sendo utilizado e por isso a aplicação de métodos e técnicas de engenharia de software torna o desenvolvimento mais barato a longo prazo.

Pressman (2016) apresenta engenharia de software como uma tecnologia em camadas, conforme ilustrado pela Figura 1.

Figura 1: Camadas da engenharia de software

|  |
| --- |
|  |

Fonte: Pressman (2016, p. 16)

Na base está o foco na qualidade, que, conforme defendido pelo autor, deve ser a base de qualquer disciplina de engenharia. O comprometimento organizacional com a qualidade, aliado a uma cultura de aperfeiçoamento contínuo de processos leva ao desenvolvimento de abordagens cada vez mais eficazes de engenharia de software.

A base da engenharia de software é a camada de processos. O processo de software constitui a base para o controle do gerenciamento de projetos de software e estabelece os contextos para aplicação de métodos, produção de artefatos e estabelecimento de marcos, garantindo a qualidade e gestão apropriada de mudanças. Esse conceito será aprofundado no capítulo seguinte.

Os métodos de software envolvem uma ampla variedade de tarefas, como comunicação, análise de requisitos, modelagem de projeto, construção de programa, testes e suporte, por exemplo. São eles que fornecem as informações técnicas para o desenvolvimento de software, isto é, como as tarefas devem ser executadas.

Por fim, as ferramentas fornecem suporte automatizado ou semi automatizado para processos e métodos. Quando integradas de forma que as informações geradas por uma ferramenta alimentam outra, um sistema para o suporte ao desenvolvimento de software é estabelecido, denominado engenharia de software com o auxílio do computador (PRESSMAN, 2016).

## 2.4 PROCESSO DE SOFTWARE

Processo de software é um roteiro composto por uma sequência de passos previsíveis que deve ser seguido para a criação ordenada dos artefatos necessários para o desenvolvimento de um software. Esses passos podem ser atividades metodológicas, ações de engenharia de software e tarefas, as quais são organizadas dentro de uma metodologia ou modelo que determina sua relação com o processo e também entre si. A metodologia engloba também um conjunto de atividades de apoio aplicáveis a todo o processo de software (PRESSMAN, 2016).

Segundo Pressman (2016), uma atividade metodológica possui um objetivo amplo. Para atingi-lo, um conjunto de ações de engenharia de software deve ser executado. Cada ação é definida por um conjunto de tarefas, o qual identifica as tarefas de trabalho a ser completadas, os artefatos de software (programas, documentos ou dados) que serão produzidos, os fatores de garantia da qualidade que serão exigidos e os marcos utilizados para indicar o progresso.

Ainda de acordo com o autor, as atividades metodológicas são complementadas por diversas atividades de apoio, que auxiliam a gerenciar e controlar o andamento, a qualidade, as alterações e os riscos. O Quadro 1 apresenta atividades de apoio típicas.

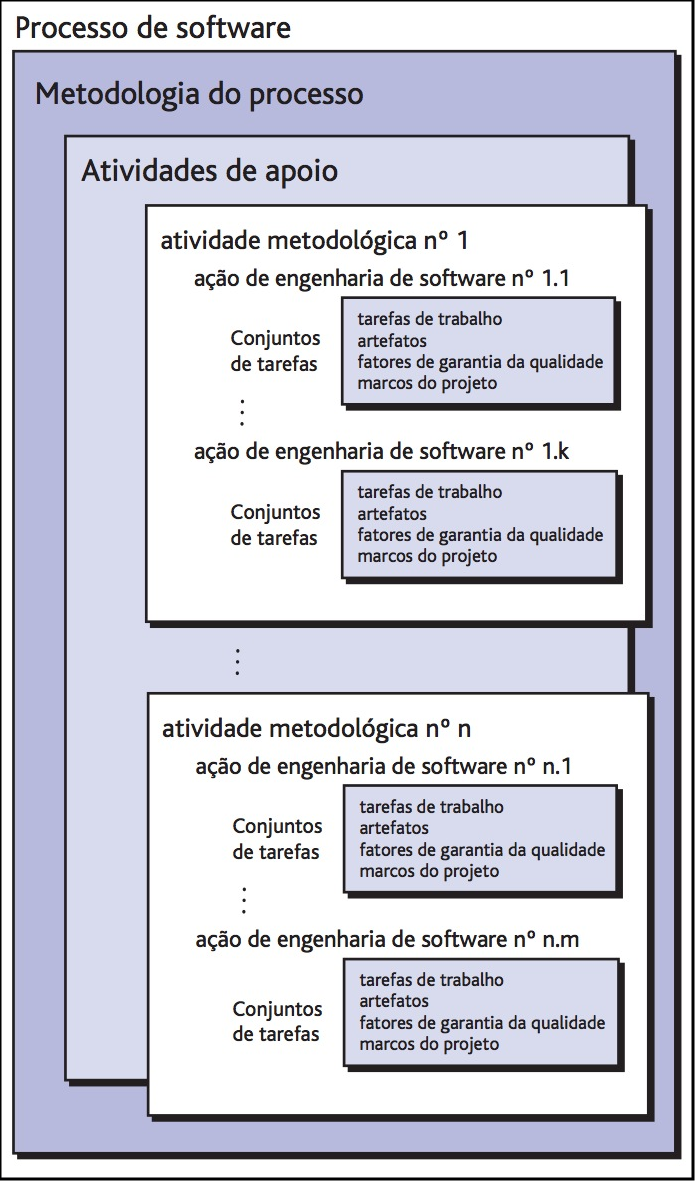
Quadro 1: Atividades de apoio típicas

|  |  |
| --- | --- |
| **Atividade de apoio** | **Descrição** |
| Controle e acompanhamento do projeto | Possibilita que a equipe avalie o progresso em relação ao plano do projeto e tome as medidas necessárias para cumprir o cronograma |
| Administração de riscos | Avalia riscos que possam afetar o resultado ou a qualidade do produto/projeto |
| Garantia da qualidade de software | Define e conduz as atividades que garantem a qualidade do software |
| Revisões técnicas | Avaliam artefatos da engenharia de software, tentando identificar e eliminar erros antes que se propaguem para a atividade seguinte |
| Medição | Define e coleta medidas (do processo, do projeto e do produto). Auxilia na entrega do software de acordo com os requisitos; pode ser usada com as demais atividades (metodológicas e de apoio) |
| Gerenciamento da configuração de software | Gerencia os efeitos das mudanças ao longo do processo |
| Gerenciamento da capacidade de reutilização | Define critérios para a reutilização de artefatos (inclusive componentes de software) e estabelece mecanismos para a obtenção de componentes reutilizáveis |
| Preparo e produção de artefatos de software | Engloba as atividades necessárias para criar artefatos como, por exemplo, modelos, documentos, logs, formulários e listas |

Fonte: Adaptado de Pressman (2016, p. 18)

A Figura 2 representa como atividades metodológicas e atividades de apoio são organizadas dentro de uma metodologia de processo de software.

Figura 2: Representação da metodologia do processo de software



Fonte: Pressman (2016, p. 32)

De acordo com Pressman (2016), uma metodologia genérica de processo de software é definida por cinco atividades metodológicas:

* Comunicação. O objetivo dessa atividade é entender os objetivos do cliente e dos envolvidos no projeto e identificar requisitos.
* Planejamento. A atividade de planejamento tem como o objetivo o desenvolvimento de um “mapa” para o projeto de software - denominado plano de projeto de software. O plano de projeto de software define o trabalho de engenharia de software, descrevendo tarefas técnicas, riscos de projeto, recursos necessários, produtos resultantes e o cronograma de trabalho.
* Modelagem. Nesta etapa, são desenvolvidos modelos que auxiliam a compreender melhor as necessidades do software e do projeto que irá atendê-las. Os modelos gerados fornecem uma visão do todo, apresentando características como arquitetura e funcionalidades desejadas, por exemplo.
* Construção. A atividade de construção combina a geração de código para a construção do modelo, aliada a testes para a identificação de erros na codificação.
* Entrega. Por fim, o software ou um incremento parcial é entregue ao cliente, que avalia o produto e fornece um feedback.

Independente do porte do software a ser desenvolvido, essas atividades permanecerão as mesmas. Nesse sentido, é importante destacar que em engenharia de software, uma metodologia de processo não é uma prescrição rígida de como desenvolver um software, muito pelo contrário: trata-se de uma abordagem flexível que permite a escolha das ações, tarefas e atividades de apoio adequadas às especificações de cada projeto e equipe. Isso significa dizer que, apesar das atividades metodológicas propostas serem as mesmas para todos os projetos, os detalhes do processo podem e devem ser adaptados às particularidades de cada cenário. (PRESSMAN, 2016).

Outro aspecto importante do processo de software é o chamado fluxo de processo. Um fluxo de processo descreve como são organizadas as atividades metodológicas, bem como a sequência e os momentos em que as ações e tarefas ocorrem dentro de cada atividade. Um fluxo de processo pode ser linear, onde cada uma das atividades metodológicas é executada em sequência, do início ao fim; iterativo, onde uma ou mais atividades se repetem antes de seguir para a próxima; evolucionário, onde o processo linear é executado de forma cíclica, incrementando o software a cada entrega; ou paralelo, em que uma ou mais atividades são executadas paralelamente (PRESSMAN, 2016).

## 2.5 MODELOS DE PROCESSO DE SOFTWARE

Segundo Pressman (2016), um modelo de processo de software[[1]](#footnote-2) fornece uma sequência de passos para a realização de um trabalho de engenharia de software disciplinado. Para isso, cada modelo acomoda atividades metodológicas abordadas com diferentes ênfases, invocadas em um fluxo de processo definido. Além disso, define também as ações e tarefas, o grau de iteração, os artefatos e a organização do trabalho a ser feito.

Segundo Sommerville (2011), os processos de software podem ainda ser classificados como dirigidos[[2]](#footnote-3) ou ágeis. Em um processo de software dirigido, todas as atividades são planejadas com antecedência, e o progresso é avaliado por comparação com o planejamento inicial. Um processo ágil, o planejamento é gradativo, tornando o processo mais flexível e adaptável a mudanças de requisitos.

Ainda de acordo com Sommerville (2011), um modelo de processo de software não é uma prescrição definitiva de uma metodologia, mas sim uma abstração, que pode ser adaptada e combinada com outros modelos de processo, a fim de determinar um processo específico adequado às particularidades de cada projeto.

### 2.5.1 Modelo Cascata

De acordo com Pressman (2016), o modelo cascata, também chamado de “ciclo de vida clássico” é o paradigma mais antigo da engenharia de software. O modelo acomoda as atividades de comunicação, planejamento, modelagem, construção e entrega, em um fluxo linear, conforme representado na Figura 3.

Figura 3: Ciclo de vida clássico

|  |
| --- |
|  |

Fonte: Pressman (2016, p. 42)

Pode ser uma abordagem adequada quando têm-se requisitos estáveis e bem definidos, porém, a aplicação do modelo frequentemente enfrenta alguns problemas. Os principais problemas, destacados por Pressman (2016) e Ellis (2010, apud WAZLAWICK, 2013, p. 26) são:

* Dificuldade de definir requisitos completos antes da codificação;
* Dificuldade para identificar todos os requisitos do cliente, que provavelmente só serão notados depois da entrega do produto;
* Projetos reais raramente seguem exatamente o fluxo estabelecido pelo processo, podendo provocar confusões no progresso do projeto, devido a mudanças;
* Falta de flexibilidade de requisitos, sendo difícil voltar atrás para corrigir requisitos mal definidos;
* Não produz resultados tangíveis até a etapa de codificação, exceto para pessoas familiarizadas com artefatos de documentação;
* Uma versão operacional do programa só ficará disponível ao final do projeto, aumentando o risco de impacto por erros não identificados durante o processo de desenvolvimento.

### 2.5.2 Modelos de Processo Evolucionários

Um modelo de processo evolucionário adota o fluxo iterativo e apresenta características que permitem o desenvolvimento de um produto cada vez mais completo a cada iteração. Trata-se de um modelo de processo projetado especificamente para o desenvolvimento de produtos que crescem e mudam. A aplicação pode ser adequada para cenários onde as regras de negócio e requisitos podem sofrer alterações, quando não se tem tempo hábil para o desenvolvimento e entrega de um produto completo ou quando requisitos não estão bem definidos, por exemplo (PRESSMAN, 2016).

### 

#### 2.5.2.1 Prototipação

O paradigma da prototipação consiste, basicamente, na construção de uma “demonstração” do software ou de parte dele (geralmente construída rapidamente, incorporando funcionalidades básicas), para que seja avaliado pelos envolvidos no projeto a fim de auxiliar na compreensão do que exatamente deve ser desenvolvido. É uma abordagem recomendada para casos em que se tem uma ideia geral do que deve ser construído, porém os requisitos não são suficientes ou não estão bem definidos (PRESSMAN, 2016).

O modelo adota um fluxo iterativo, incorporando as etapas conforme representado na Figura 4. O ponto de partida é a atividade de comunicação, na qual são definidos os objetivos gerais do software, identificados os requisitos e quais áreas necessitam de uma definição mais ampla. Com isso, uma iteração de prototipação é planejada, na qual desenvolve-se um modelo do protótipo (normalmente priorizando aspectos visuais do software), que é construído e na sequência deve ser avaliado e receber feedback dos envolvidos. Com isso, é possível refinar os requisitos e ter maior clareza das necessidades que o software deve atender. A próxima iteração dedica-se na melhoria do protótipo e assim sucessivamente, obtendo a cada iteração um produto mais próximo do atendimento dos requisitos, até que o objetivo do processo seja cumprido (PRESSMAN, 2016).

Figura 4: Paradigma de prototipação

|  |
| --- |
|  |

Fonte: Pressman (2016, p. 45)

De acordo com Wazlawick (2013), o modelo de processo de prototipação distingue-se em duas técnicas: *throw-away* e *cornerstone*.

A abordagem *throw-away*, consiste na construção de protótipos usados unicamente com o objetivo de estudar aspectos do sistema, entender melhor seus requisitos e reduzir riscos, sendo descartado após o cumprimento de sua finalidade (CRINNION, 1991, apud WAZLAWICK, 2013, p. 37).

Na abordagem *cornerstone*, o protótipo é utilizado com as mesmas finalidades da abordagem throw-away, porém, ao invés de ser descartado, o protótipo será parte do sistema final, sendo evoluído até que se torne um sistema “entregável”. Por essa característica, requer um planejamento mais cuidadoso, visando evitar que um protótipo com erros componha o produto final. O modelo de prototipação evolucionária se baseia nessa técnica (BUDDLE et al., 1992, apud. WAZLAWICK, 2013, p. 37).

#### 2.5.2.2 Modelo orientado a cronograma

De acordo com Wazlawick (2013), o modelo de processo orientado a cronograma é uma variação melhor estruturada do paradigma de prototipação. De modo geral, o modelo adota um fluxo iterativo, para o atendimento gradual dos requisitos, ordenados por prioridade, até que todos os requisitos sejam atendidos ou o tempo se esgote. Com isso, espera-se que ao final do projeto, mesmo que nem todos os requisitos tenham sido atendidos, ao menos um conjunto de requisitos com maior prioridade foi atendido. Trata-se de uma abordagem adequada, principalmente, para projetos com uma data limite de entrega intransferível.

A Figura 5 apresenta o diagrama de atividades UML para o modelo. Antes de iniciar as iterações de construção, é necessário realizar a análise de requisitos, com base na qual será desenvolvido o projeto arquitetural, que organizará os requisitos em unidades funcionais, apresentando também a forma como se interconectam e colaboram. Com o projeto arquitetural completo, é possível iniciar as iterações, que esforçam-se em detalhar e atender os requisitos (WAZLAWICK, 2013).

O processo se encerra quando todos os requisitos forem atendidos, ou o tempo disponível para desenvolvimento se esgote, por isso a importância da priorização. Para que a técnica seja bem sucedida, é importante ter clareza dos requisitos e sua importância para o cliente e das dependências entre os componentes funcionais, para a correta priorização (WAZLAWICK, 2013).

Figura 5: Modelo de processo orientado a cronograma

|  |
| --- |
|  |

Fonte: Wazlawick (2013, p. 38)

Segundo Ellis (2010, apud WAZLAWICK, 2013, p. 37),

Uma das principais vantagens desse modelo é o fato de colocar funcionalidades úteis nas mãos do cliente antes de completar o projeto. Se os estágios forem planejados cuidadosamente, funcionalidades importantes estarão disponíveis muito mais cedo do que com outros ciclos de vida. Além disso, esse modelo provê entregas mais cedo e de forma contínua, o que pode aliviar um pouco a pressão de cronograma colocada na equipe.

## 2.6 REQUISITOS DE SOFTWARE

Para Sommerville (2011), os requisitos de um sistema refletem as necessidades dos clientes, compreendendo descrições do que o sistema deve fazer, os serviços que oferece e as restrições a seu funcionamento. O processo de descobrir, analisar, documentar e verificar requisitos é chamado engenharia de requisitos.

Sommerville (2011) classifica requisitos em dois grupos: requisitos de usuário e requisitos de sistema. Requisitos de usuário são requisitos de alto nível que expressam os serviços que o sistema deve fornecer e sob quais restrições irá operar, normalmente através de linguagem natural e diagramas. Um requisito de usuário pode gerar um ou mais requisitos de sistema. Requisitos de sistema são versões expandidas dos requisitos de usuário, servem como ponto de partida para o projeto do sistema.

### 2.6.1 Requisitos Funcionais e Não Funcionais

De acordo com Sommerville (2011), os requisitos de sistema são frequentemente classificados em requisitos funcionais e não funcionais. Requisitos funcionais são declarações de funcionalidades que o sistema deve implementar e regras de negócio, algumas vezes também podem indicar o que o sistema não deve implementar. Requisitos não funcionais, por sua vez são requisitos que não estão diretamente relacionados com os serviços específicos oferecidos pelo sistema, são restrições aos serviços ou funções oferecidos pelo sistema.

Segundo o autor, requisitos funcionais devem apresentar completude (todas as necessidades do usuário devem ser identificadas) e consistência (não devem apresentar contradição. Na prática, é praticamente impossível alcançar essas características na análise de requisitos funcionais, pois as inconsistências tendem a surgir após uma análise mais profunda ou após a entrega total ou parcial do produto.

Ainda de acordo com Sommerville (2011), ao contrário de requisitos funcionais, requisitos não funcionais podem impactar em uma ou mais funcionalidade, por vezes até mesmo inutilizando o sistema. Requisitos não funcionais devem ser mensuráveis, para que seja possível verificar o atendimento ou não do requisito. Podem ser classificados em:

1. Requisitos de produto: determinam características de comportamento do software, como desempenho e confiabilidade, por exemplo;
2. Requisitos organizacionais: resultantes de políticas e procedimentos organizacionais, pode-se citar como exemplo restrições de linguagem de programação e forma de acesso ao sistema;
3. Requisitos externos: são requisitos que derivam de fatores externos, como órgãos reguladores e legislação, por exemplo.

### 2.6.2 Engenharia de Requisitos

De acordo com Pressman (2016) a engenharia de requisitos é uma ação de engenharia de software que ocorre entre as atividades de comunicação e modelagem. Ela deve ser adaptada às necessidades do projeto, porém abrange, de modo geral sete tarefas distintas, respectivamente descritas a seguir:

1. Concepção. A concepção busca o entendimento básico do problema, a identificação dos interessados na solução e qual sua natureza.
2. Levantamento: busca-se compreender os objetivos do sistema para os envolvidos, estabelecer as metas de negócio, bem como estabelecer métodos de atribuição de prioridades.
3. Elaboração: esta etapa concentra-se no refinamento das informações obtidas nas duas etapas anteriores, buscando desenvolver um modelo de requisitos refinado.
4. Negociação: junto dos interessados no projeto, resolvem-se conflitos entre requisitos e limitações tecnológicas, também nessa etapa os requisitos devem ser ordenados e priorizados.
5. Especificação: na especificação de requisitos, desenvolve-se uma descrição detalhada dos requisitos, que pode ser apresentada em linguagem natural, modelos gráficos, modelos matemáticos ou casos de uso, individualmente ou combinados.
6. Validação: a validação de requisitos examina a especificação para garantir que todos os requisitos de software tenham de forma correta, não ambígua e consistente.
7. Gestão: a gestão de requisitos é um conjunto de atividades que visa a identificação, controle e acompanhamento das mudanças de requisitos

O documento de requisitos de software, também chamado de Especificação de Requisitos de Software, é uma declaração oficial de o que os desenvolvedores do sistema devem implementar. Deve incluir tanto requisitos de usuário quanto requisitos de sistema, descritos separadamente ou não, dependendo da quantidade de requisitos e do padrão adotado. Ainda de acordo com o padrão adotado, por conter mais informações, como arquitetura e modelos de alto nível e aspectos da evolução, por exemplo (SOMMERVILLE, 2011).

## 2.7 UML

A Linguagem de Modelagem Unificada (em inglês, Unified Modeling Language, UML) é uma linguagem padrão para a descrição e documentação e projetos de software. Pode ser usada para visualização, especificação, documentação e construção de artefatos de um software. (BOOCH e JACOBSEN, 2004, apud PRESSMAN, 2016, p. 869).

De acordo com Bezerra (2015), a UML define elementos visuais que podem ser utilizados na modelagem de sistemas, em especial aqueles que adotam o paradigma de orientação a objetos. Através dos elementos gráficos da linguagem, é possível construir diagramas que representam diversas perspectivas de um sistema.

De acordo com UML.org (2020, tradução minha), a versão 2.0 da linguagem define treze tipos de diagramas, dos quais seis representam a estrutura estática do aplicativo, três representam comportamentos gerais, e quatro representam aspectos de interações.

### 2.7.1 Diagrama de Caso de Uso

A modelagem de caso de uso é frequentemente utilizada no apoio à elicitação de requisitos. Um caso de uso pode ser considerado um cenário simples que descreve o que o usuário espera de um sistema (SOMMERVILLE, 2011).

De acordo com Bezerra (2015), um caso de uso é definido como a “especificação de uma sequência completa de interações entre um sistema e um ou mais agentes externos a esse sistema”, que corresponde ao relato de uso de uma funcionalidade. O Modelo de Caso de Uso (MCU) de uso abrange a perspectiva externa do sistema, o que significa dizer que não deve revelar a estrutura e comportamento internos. Com a compreensão do MCU, é possível saber quais são as funcionalidades oferecidas pelo sistema e quais os resultados externos produzidos por cada uma delas. O diagrama UML que representa o MCU é o Diagrama de Caso de Uso (DCU).

O DCU é composto por três elementos principais: casos de uso, atores e relacionamentos. Também é possível representar as fronteiras do sistema através de um retângulo. Cada um dos elementos de um DCU é descrito a seguir, de acordo com Bezerra (2015):

#### 2.7.1.1 Caso de uso

Um caso de uso corresponde a uma funcionalidade do sistema. É graficamente representado por uma elipse, com o nome do caso de uso posicionado abaixo ou dentro da figura, conforme representado na Figura 6. A descrição textual de um caso de uso relata as interações entre os elementos externos de um sistema e o sistema em si. Diferentemente da forma de representação gráfica, a UML não define um padrão para a estrutura da descrição de um caso de uso

#### 2.7.1.2 Atores

São agentes externos que interagem com o sistema, normalmente responsáveis pelo início da execução de um caso de uso. Um ator corresponde a um papel em relação ao sistema, por isso é uma boa prática nomear os atores com o papel que desempenham, ao invés de nomear os indivíduos. Esses papéis podem ser descritos por cargos, organizações ou divisões de uma organização, outros sistemas, ou equipamentos com os quais o sistema se comunica. A notação UML para representar atores em um DCU é a figura de um boneco “de palitos”, com seu papel posicionado abaixo da figura, conforme apresentado na Figura 6.

#### 2.7.1.3 Relacionamentos

O terceiro componente de um DCU são os relacionamentos, cuja função é relacionar atores e casos de uso. Um ator deve estar relacionado a um ou mais casos de uso, pode haver também relacionamentos entre casos de uso ou entre atores. Os relacionamentos são classificados em: comunicação, inclusão, extensão e generalização

##### 2.7.1.3.1 Relacionamento de comunicação

Um relacionamento de comunicação ocorre entre um ator e um caso de uso, indicando que o ator interage (troca informações) com o sistema, através do caso de uso que se relaciona. A representação gráfica do relacionamento de comunicação é um segmento de reta ligando o ator e o caso de uso, conforme representado na Figura 6. De acordo com a UML, pode-se também imprimir o sentido da reta, para denotar onde a interação foi iniciada, apesar de ter pouco uso prático e não ser muito usual.

Figura 6: Representação gráfica UML de Ator, Caso de Uso e Relacionamento de comunicação

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 70)

##### 2.7.1.3.2 relacionamento de inclusão

O relacionamento de inclusão ocorre somente entre casos de uso. Quando dois ou mais casos de uso incluem uma sequência comum de interações, a sequência comum pode ser descrita em outro caso de uso, que será incluído por cada um dos casos de uso que a compartilham. Isso evita a repetição da descrição da sequência comum, além de tornar a descrição dos casos de uso mais simples e facilitar a manutenção. O relacionamento de inclusão entre um caso de uso inclusor A (aquele que inclui o comportamento) e caso de uso incluso B (aquele cujo comportamento é incluído por outros) é representado por uma seta direcionada de A para B, tendo seu eixo tracejado e rotulado com o estereótipo *include*, demonstrado na Figura 7.

Figura 7: Exemplo de relacionamento de inclusão

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 71)

##### 2.7.1.3.3 Relacionamento de extensão

Assim como o relacionamento de inclusão, o relacionamento de extensão ocorre somente entre casos de uso. Esse relacionamento é usado para modelar situações em que diferentes casos de uso (extensores) podem ser inseridos em cenários específicos de um caso de uso (estendido). Entretanto, o caso de uso estendido por outros casos de uso deve ser descrito com completude, sem indicar que deve necessariamente haver uma extensão em seu comportamento. A representação gráfica de um relacionamento de extensão, em que um caso de uso A estende um caso de uso B, é representado por uma seta direcionada de A para B, com eixo tracejado e rotulado com o estereótipo *extend*, conforme exemplo da Figura 8.

Figura 8: Exemplo de relacionamento de extensão

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 72)

##### 2.7.1.3.4 Relacionamento de generalização

O relacionamento de generalização pode existir entre dois casos de uso ou entre dois atores. Quando entre casos de uso, permite que um caso de uso B herde o comportamento e os relacionamentos de A, podendo incluir novos comportamentos e relacionamentos. Entre atores, a generalização permite que o ator B, herdeiro de A, interaja com todos os casos de uso de A, podendo interagir com novos casos de uso, restritos a B. Uma seta com a ponta em formato de triângulo, na direção do caso de uso ou ator do qual as características são herdadas, com uma linha sólida compondo o eixo, é o elemento gráfico responsável pela representação desse relacionamento, exemplificado na Figura 9 (generalização entre atores) e Figura 10 (generalização entre casos de uso).

Figura 9: Exemplo de relacionamento de generalização entre atores

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 72)

Figura 10: Exemplo de relacionamento de generalização entre casos de uso

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 72)

#### 2.7.1.4 Fronteira do sistema

A fronteira do sistema é utilizada para enfatizar a divisão entre o interior e exterior do sistema. Para isso, utiliza-se um retângulo no interior do qual são inseridos os casos de uso, os atores devem ser posicionados fora do retângulo, conforme exemplificado na Figura 11.

Figura 11: Exemplo de caso de uso com representação de fronteira de sistema

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 70)

### 2.7.2 Diagrama de Classes

O modelo de classes é utilizado durante a maior parte do desenvolvimento iterativo e incremental de um Sistema de Software Orientado a Objeto (SSOO) e evoluiu durante as iterações. À medida que as iterações ocorrem o modelo de classes evolui, sendo incrementado com novos detalhes descobertos durante o desenvolvimento. (BEZERRA, 2015).

Conforme Bezerra (2015), o modelo de classes é utilizado durante a maior parte do desenvolvimento iterativo, através do qual evolui à medida que novos detalhes são descobertos e incrementados ao modelo. Durante o processo, o modelo de classes passa por três estágios sucessivos de abstração: análise, especificação e implementação.

O Diagrama de Classes é utilizado na construção do modelo de classes desde o nível de análise até o nível de especificação. No nível de análise, o objetivo do modelo de dados é descrever o problema a ser resolvido pelo software, sem considerar características de solução. Na etapa de especificação, detalhes da solução a ser utilizada são descritos em alto nível de abstração (BEZERRA, 2015). A seguir, são apresentados os elementos de um diagrama de classes UML.

#### 2.7.2.1 Classes

Uma classe é representada por uma “caixa” com, no máximo três compartimentos. O primeiro compartimento (de cima para baixo) é obrigatório e contém o nome da classe. Por convenção o nome deve ser apresentado no singular, e com as palavras que o compõem iniciando com letra maiúscula. Os compartimentos seguintes contém, respectivamente, os atributos e os métodos da classe (BEZERRA, 2015).

De acordo com Pressman (2016), atributos podem ser valores calculados pela classe a partir de suas variáveis de instância, ou valores obtidos dos objetos que a compõem. Cada atributo pode ter um nome, um tipo e um nível de visibilidade, sendo obrigatório somente o nome. O atributo pode ainda ser caracterizado como estático ou de classe, sublinhando um atributo estático. Os métodos de uma classe correspondem às operações que ela pode executar. Também apresentam um nível de visibilidade e nome, junto de seus parâmetros e retornos, com o tipo indicado.

A visibilidade de atributos e métodos é representada pelo sinal –, #, ~ ou +, precedente ao nome do atributo para indicar, respectivamente, as visibilidades private, protected, package ou public. O tipo de um atributo, parâmetro ou retorno pode corresponder a um tipo primitivo, ou a um objeto. (PRESSMAN, 2016). A Figura 12 ilustra a representação de uma classe, com atributos e métodos.

Figura 12: Representação de uma Classe em UML

|  |
| --- |
|  |

Fonte: Pressman (2016, p. 870)

#### 2.7.2.2 Associações

De acordo com Bezerra (2015), Cada ocorrência de uma classe é chamada de objeto ou instância. Durante a execução de um sistema, objetos se relacionam uns com os outros, o que permite que troquem mensagens e colaborem entre si para produzir as funcionalidades do sistema. A associação é o elemento utilizado para indicar um relacionamento entre classes, representado por uma linha ligando as classes às quais pertencem os objetos relacionados, conforme exemplo da Figura 13. Associações possuem diversas características, como multiplicidade, nome, direção de leitura, papéis, tipos de participação e conectividade que serão descritas a seguir.

Figura 13: Exemplos de associações entre classes

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 114)

#### 2.7.2.3 Multiplicidades

Conforme Bezerra (2015), cada associação em um diagrama de classes possui duas multiplicidades, uma em cada extremo da linha que a representa. A multiplicidade de uma associação permite a representação dos limites inferior e superior da quantidade de objetos aos quais o outro objeto pode estar associado. Os símbolos possíveis para representar uma multiplicidade estão descritos no Quadro 2.

Quadro 2: Simbologia para representar multiplicidades

|  |  |
| --- | --- |
| **Nome** | **Simbologia** |
| Apenas um | 1 |
| Zero ou muitos | 0..\* |
| Um ou Muitos | 1..\* |
| Zero ou um | 0..1 |
| Intervalo específico | Iᵢ..Iₛ |

Fonte: Bezerra (2015, p. 115)

Conectividade é o nome dado ao tipo de associação entre duas classes. Ela depende dos símbolos de multiplicidade utilizados na associação (BEZERRA, 2015). O Quadro 3 relaciona os símbolos de multiplicidade utilizados na associação e a conectividade correspondente.

Quadro 3: Conectividades em relação às multiplicidades

|  |  |  |
| --- | --- | --- |
| **Conectividade** | **Multiplicidade de um extremo** | **Multiplicidade do outro extremo** |
| Um para um | 0..1 ou 1 | 0..1 ou 1 |
| Um para muitos | 0..1 ou 1 | \* ou 1..\* ou 0..\* |
| Muitos para Muitos | \* ou 1..\* ou 0..\* | \* ou 1..\* ou 0..\* |

Fonte: Bezerra (2015, p. 116)

A Figura 14 apresenta um exemplo de associação com notação de multiplicidade.

Figura 14: Exemplos de associação com uso de símbolos de multiplicidade

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 115)

#### 2.7.2.4 Participações

Participação é o nome dado à característica de necessidade ou não da existência de uma associação entre objetos. Uma participação pode ser obrigatória ou opcional, dependendo da multiplicidade da associação. Se o valor mínimo de uma associação é igual a 1, a participação é obrigatória, caso contrário, é opcional (BEZERRA, 2015).

#### 2.7.2.5 Nome de associação, direção de leitura e papéis

De acordo com Bezerra (2015), o nome, direção de leitura e papéis são recursos UML de notação que permitem esclarecer o significado de uma associação. O nome da associação deve ser posicionado no meio da linha da associação e deve fornecer um significado semântico a ela. A direção de leitura é representada por um pequeno triângulo posicionado próximo ao lado do nome da associação referente à direção que a associação deve ser lida. Por fim, os papéis, posicionados na linha de associação, ao lado do objeto, tem o objetivo de esclarecer a responsabilidade do objeto na associação. Esses recursos são opcionais e devem ser sabiamente empregados para esclarecer a finalidade de uma associação, sua utilização é exemplificada na Figura 15.

Figura 15: Exemplo de utilização dos recursos UML nome da associação, direção de leitura e papéis

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 118)

#### 2.7.2.6 Classes associativas

Classes associativas são classes ligadas a associações e normalmente aparecem quando é necessário manter informações da associação. É comum que sejam ligadas a associações muitos para muitos, entretanto, podem estar ligadas a associações de qualquer conectividade. Uma classe associativa pode, inclusive, participar de outros relacionamentos. Também pode ser substituída por um classe normal, associada a cada uma das classes associadas na relação que daria origem a uma classe associativa. Na UML, uma classe associativa é representada pela mesma notação utilizada para um classe comum, porém, liga-se a uma associação através de uma linha tracejada. A Figura 16 apresenta a representação da classe associativa Emprego (BEZERRA, 2015).

Figura 16: Exemplo de representação de uma classe associativa

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 119)

#### 2.7.2.7 Agregações e composições

Em um diagrama de classes UML, relações de todo-parte significam que o objeto parte está contido no objeto todo, e o objeto todo contém o objeto parte. Esse tipo de relação é dividido em dois tipos: a agregação e a composição (BEZERRA, 2015).

De acordo com Pressman (2016), tanto a composição quanto a agregação representam relações de todo-parte, entretanto, na composição, os objetos parte não têm papel a desempenhar no sistema de software sem a existência de um objeto todo. A relação de agregação é representada por um losango vazio ao lado da classe todo, na relação de composição, esse losango mantém o posicionamento, porém o losango é completamente preenchido. A Figura 17 apresenta uma relação de agregação entre as classes *College* e *Building*, e uma relação de composição entre as classes *College* e *Course*.

Figura 17: Exemplo de notação de relações de agregação e composição

|  |
| --- |
|  |

Fonte: Pressman (2016, p. 873)

#### 2.7.2.8 Generalizações e especializações

Conforme Bezerra (2015), relacionamentos generalização/especialização, também chamado de relacionamento de herança, indica uma relação entre classes, onde uma classe A, chamada de subclasse, herda atributos e métodos de uma classe mais genérica B, denominada superclasse. Declara-se então que a classe A é uma especialização de B, e B é uma generalização de A. A notação UML para a relação de herança é representada por uma flecha partindo da subclasse em direção à superclasse, conforme representado na Figura 18.

Figura 18: Exemplo de notação de relação de herança.

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 129)

### 2.7.3 Diagrama de Sequência

De acordo com Pressman (2016), um diagrama de sequência é utilizado para indicar as comunicações dinâmicas entre objetos durante a execução de uma tarefa. Ele mostra a troca de mensagens entre objetos em ordem temporal. Pode ser usado para demonstrar as interações em um caso de uso, ou em um cenário de um sistema.

#### 2.7.3.1 Atores

Conforme Bezerra (2015) Em um diagrama de sequência, podem ser representados os atores que participam da realização do caso de uso. A notação gráfica é a mesma utilizada no diagrama de uso, descrita na seção 2.7.1.2.

#### 2.7.3.2 Objetos

Um objeto é representado por uma caixa, que deve conter, obrigatoriamente o tipo do objeto (normalmente o nome da classe à qual o objeto corresponde) precedido de dois pontos. Pode-se opcionalmente, declarar antes dos dois pontos o nome dado ao objeto. Esse texto deve ser sublinhado (PRESSMAN, 2016).

Conforme Bezerra (2015), quando deseja-se referenciar uma ocorrência de um objeto em uma coleção, a notação é a referência de seu índice entre colchetes.

A Figura 19 apresenta três diferentes notações para objetos em um diagrama de sequência UML.

Figura 19: Exemplos de notação UML para objetos em um diagrama de sequência

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 200)

De acordo com Bezerra (2015), quando métodos estáticos são chamados, pode-se representar somente a classe que o implementa. Nesse caso, a notação também ocorre por uma caixa, porém somente com o nome da classe e sem sublinhar o texto.

#### 2.7.3.3 Linha de vida

Em um diagrama de sequência, a linha de vida de um objeto é composta por duas partes: cabeça e cauda. A cabeça, corresponde à notação do objeto, e a cauda é representada por uma linha vertical tracejada. Atores também podem apresentar linha de vida. Nesse caso, a cabeça é substituída pela notação gráfica do ator (BEZERRA, 2015). A Figura 20 apresenta as representações gráficas de linhas de vida de atores e objetos.

Figura 20: Notação UML para linha de vida

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 204)

#### 2.7.3.4 Mensagens

Uma mensagem é uma solicitação de execução de uma operação em outro objeto, é por meio do envio de mensagens que os objetos de um sistema colaboram entre si para prover funcionalidades. A notação UML para uma mensagem em um diagrama de sequência é uma flecha ligando a linha de vida do objeto de envio à linha de vida do objeto receptor. O posicionamento vertical de uma mensagem em uma linha de vida, indica sua posição temporal de envio. Quanto mais acima se posiciona, mais cedo foi enviada. Da mesma forma, quanto mais abaixo se posiciona, mais tarde foi enviada. Através disso é possível identificar a sequência de mensagens enviadas e recebidas durante a realização de um caso de uso. O formato da da seta indica o tipo de mensagem que está sendo enviada, conforme Quadro 4. O rótulo da mensagem deve ser posicionado acima da linha da seta (BEZERRA, 2015).

Quadro 4: Notação gráfica para tipos de mensagem em um diagrama de sequência

|  |  |
| --- | --- |
| **Tipo de mensagem** | **Notação gráfica** |
| Mensagem Síncrona |  |
| Mensagem assíncrona |  |
| Mensagem de retorno |  |
| Mensagem de sinal |  |

Fonte: Adaptado de Bezerra (2015, p. 205)

De acordo com Pressman (2016), o rótulo de mensagens síncronas e assíncronas deve identificar o método chamado e, opcionalmente incluir os parâmetros, seus tipos, e o tipo de retorno. No caso de mensagens de retorno, o rótulo é opcional.

Conforme Bezerra (2015), mensagens de sinal são utilizadas, principalmente, como uma opção de notação para criação ou destruição de objetos. Nesse caso, os rótulos devem ser, respectivamente <<*create*>> e <<*destroy*>>.

Existem ainda as mensagens reflexivas, que indicam o envio de mensagem de um objeto a si mesmo. Nesse caso, a notação deve ser rotulada, porém a seta segue o formato apresentado na Figura 21 (BEZERRA, 2015).

Figura 21: Notação UML para mensagens reflexivas

|  |
| --- |
|  |

Fonte: Bezerra (2015, p. 205)

#### 2.7.3.5 Ocorrências de execução

Uma ocorrência de execução[[3]](#footnote-4) representa o tempo durante o qual um objeto realiza uma operação. A notação gráfica determinada pela UML é a de um retângulo vazio, disposto sob a linha de vida, na posição vertical correspondente ao período de atividade. O início da execução é, portanto, indicado pela parte superior do retângulo, e normalmente está alinhado com a recepção de uma mensagem. A parte de baixo do retângulo indica o fim da execução e normalmente substitui a notação de uma mensagem de retorno, visto que o final da execução já está representado (BEZERRA, 2015).

Na Figura 22, observa-se uma ocorrência de execução na linha de vida do objeto Drawing, iniciada pela chamada do método *getFigureAt*.

Figura 22: Exemplo de notação de ocorrência de execução

|  |
| --- |
|  |

Fonte: Adaptado de Pressman (2016, p. 878)

#### 2.7.3.6 Criação e destruição de objetos

Conforme Bezerra (2015), os momentos de criação e destruição de objetos também podem ser representados em um diagrama de sequência. A criação de um objeto corresponde ao momento em que ele passa a existir e realizar suas responsabilidades no sistema. Um objeto é destruído quando não realiza mais operações na execução. As chamadas de criação e destruição de um objeto explícitas são realizadas através da notação apresentada na seção 2.7.3.4.

Se um objeto existe desde o início da execução de uma sequência, então ele deve ser posicionado no topo. Entretanto, caso ele seja criado em determinado momento, deve ser verticalmente posicionado alinhado à mensagem de criação, que pode ser uma chamada ao objeto, ou uma mensagem com o rótulo <<*create*>>. A destruição de um objeto pode ser indicada por uma mensagem com o rótulo <<*destroy*>>, ou por um “X” no final de sua linha de vida.

## 2.8 ARQUITETURA DE SOFTWARE

De acordo com Bass, Clements e Kazman (2003, apud PRESSMAN 2016, p. 254), a arquitetura de um software é “a estrutura, ou estruturas, do sistema, o que abrange os componentes de software, as propriedades externamente visíveis desses componentes e as relações entre eles.”. Conforme Pressman (2016), a arquitetura é uma representação que permite analisar a efetividade do projeto ao atender os requisitos, considerar alternativas de arquitetura em um estágio em que ainda é fácil fazer mudanças de projeto e, por fim, reduzir os riscos associados à construção do software.

Para Sommerville (2011), a arquitetura de software identifica os principais componentes estruturais de um sistema e como se relacionam, unindo a engenharia de requisitos ao projeto. Para Bosch (2000, apud SOMMERVILLE, 2011) características da arquitetura de software são capazes de afetar o desempenho, robustez, capacidade de distribuição e manutenibilidade de um sistema.

De acordo com Pressman (2016), um padrão arquitetural é a descrição de uma forma padronizada de organizar um sistema. Um estilo arquitetural, por sua vez é um padrão arquitetural de referência, cuja aplicação é difundida e identificada em diversos sistemas. Sommerville (2011), entretanto, trata padrão arquitetural como um termo equivalente a estilo arquitetural, nesse trabalho essa é a nomenclatura adotada.

Sommerville (2011). A arquitetura de um software pode se basear em um determinado padrão ou estilo de arquitetura. Um padrão de arquitetura é uma forma de organizar o sistema, aplicada em diversos sistemas de software. Um padrão busca descrever a essência da arquitetura de um software e seu fluxo de trabalho, com base na percepção de características arquiteturais entre as aplicações observadas.

Cada padrão de arquitetura descreve uma categoria de sistema que engloba:

“(1) um conjunto de componentes (por exemplo, um banco de dados, mó- dulos computacionais) que realiza uma função exigida por um sistema, (2) um conjunto de conectores que habilitam a “comunicação, coordenação e coope- ração” entre os componentes, (3) restrições que definem como os componen- tes podem ser integrados para formar o sistema e (4) modelos semânticos que permitem a um projetista compreender as propriedades gerais de um sistema por meio da análise das propriedades conhecidas de suas partes constituint (BASS; CLEMENT; KAZMAN, 2003, apud PRESSMAN 2016, p. 258).”

Ao tomar decisões sobre a arquitetura de um software, deve-se conhecer os padrões comuns, bem como suas aplicações e pontos fracos e fortes (SOMMERVILLE, 2011). A seguir, serão descritos três padrões de arquitetura de software: cliente-servidor, MVC e arquitetura de transação (ou *transaction script*).

### 2.8.1 Arquitetura Cliente-Servidor

De acordo com Sommerville (2011), o padrão de arquitetura cliente-servidor aborda uma organização muito usada em sistemas distribuídos em tempo de execução. Em uma arquitetura cliente-servidor, a funcionalidade do sistema está organizada em serviços, onde cada serviço é prestado por um servidor. Os clientes são os usuários desses serviços que acessam os servidores para fazer uso deles. Os principais componentes desse padrão são:

1. Um conjunto de servidores que oferecem serviços a outros componentes, como por exemplo: servidores de arquivos e servidores web;
2. Um conjunto de clientes que consomem os serviços oferecidos pelos servidores;
3. Uma rede que permite aos clientes acessar esses serviços.

A maioria dos sistemas cliente-servidor é implementada como sistemas distribuídos, conectados através de protocolos de internet. Entretanto, vale ressaltar que pode-se implementar um modelo lógico de serviços independentes em um único computador (SOMMERVILLE, 2011). Uma arquitetura cliente-servidor para uma biblioteca de filmes é ilustrada na Figura 23, onde os servidores provêm acesso a serviços específicos, que são acessados pelos clientes através da rede, nesse caso, a Internet.

Figura 23: Exemplo de arquitetura cliente-servidor para uma biblioteca de filmes

|  |
| --- |
|  |

Fonte: Sommerville (2011, p. 114)

De acordo com Sommerville (2011), entre as vantagens da implementação dessa arquitetura, pode-se citar:

1. O reúso da transformação é de fácil compreensão e suporte;
2. Estilo de workflow corresponde à estrutura de muitos processos de negócios.
3. Evolução por adição de transformações é simples.
4. Pode ser implementado tanto como um sistema sequencial quanto concorrente.

Ainda segundo o autor, entre as desvantagens, há a necessidade de um protocolo e formato padrão para transferência de dados, da mesma forma, as saídas devem ser geradas em um formato acordado.

### 2.8.2 Arquitetura MVC

O padrão Model-View-Controller (MVC) divide uma aplicação em três partes interconectadas: *model*, *view* e *controller* (ou, modelo, visão e controlador). O MVC utiliza uma solução já definida para separar componentes e as responsabilidades de cada um, reduzindo a dependência entre eles (ZENKER et al., 2019).

Segundo Bezerra (2015), o MVC é um padrão arquitetural que propõe uma forma de organizar a interface com o usuário e descreve de que forma o estado dessa interface deve ser atualizado. Busca a separação de entre a lógica da apresentação e a lógica da aplicação, através da separação funcional em três principais componentes.

O componente *model* é a parte da aplicação que contém os dados e suas validações. Esse componente corresponde ao estado, à estrutura e ao comportamento dos dados manipulados pela aplicação. Fornece operações em sua interface que permitem ao restante da aplicação manipular os dados para operações de edição, exclusão, atualização, etc. (BEZERRA, 2015).

O componente *view* do MVC representa cada uma das possíveis formas de apresentação de informações provenientes do *model*, e fornece uma interface para o usuário interagir com a aplicação. O componente de visualização não deve conter inteligência, deve apenas implementar lógica de apresentação, como repetições para exibição de itens em lista e exibição de informações recebidas do *model*, por exemplo. Cada *view* está associada a um componente controlador (o *controller*) que auxilia na implementação da interface gráfica, o Controller (BEZERRA, 2015).

De acordo com Zenker et al. (2019), desde a criação do padrão original, em 1979, diversas variações foram criadas para acompanhar novas demandas de interação com o usuário, aplicando-se a vários tipos de projeto, como *desktop*, *web* e *mobile*. Em sua essência, o fluxo de controle do MVC ocorre da seguinte maneira, conforme Wilson (2015, apud ZENKER et al., 2019, p. 97):

1. O controlador (controller) é responsável por controlar e mapear as ações do usuário para que o modelo entenda, implementando a regra de negócio do software. Na prática, o controlador analisa uma solicitação do usuário e a partir disso determina o que deve ser feito.
2. O modelo (model) é responsável pela manutenção dos dados, representa o estado das entidades do banco de dados, que tem atributos e operações;
3. Por fim, a visão (view) inclui os elementos de interação com o usuário, nela estão contidos os componentes visuais e o código com a interface a ser mostrada. Ela não faz nenhuma pesquisa diretamente no banco, apenas recebe, manipula e mostra os dados. Em geral, cada modelo está associado à diversas visões, que permitem a realização das diferentes operações associadas ao modelo

A Figura 24 ilustra a interação entre os componentes em uma arquitetura MVC.

Figura 24: Padrão MVC

|  |
| --- |
|  |

Fonte: Zenker et al., (2019, p. 97)

### 2.8.3 Arquitetura de Transação (Transaction Script)

Para Fowler (2002, tradução minha), a maioria das aplicações corporativas pode ser concebida como uma série de transações. Uma transação pode ser uma exibição ou alteração de alguma informação, por exemplo. Cada interação com o sistema contém uma certa quantidade de lógica, simples ou complexa. Um *transaction script* (script de transação) organiza toda essa lógica em um único procedimento, denominado transação (transaction). Uma arquitetura *Transaction Script*, portanto, organiza a aplicação e suas funcionalidades em um conjunto de transações que podem ser realizadas por um sistema.

O termo *transaction script* é utilizado porque, para a maioria das operações, é realizada uma transação com o banco de dados, que pode ocorrer através de uma chamada direta, ou por intermédio de uma camada de encapsulamento de dados. Cada transação possui seu próprio *transaction script* que inclui todas as instruções necessárias para a realização de uma operação, pode ainda incluir subtarefas em comum com outros *transaction scripts*, que podem ser implementadas como subtarefas (FOWLER, 2002, tradução minha).

Como em qualquer programa, o código deve ser separado em módulos, Fowler (2002, tradução minha) recomenda que os *transaction scripts* sejam agrupadas em classes. A Figura 25 apresenta um diagrama de classes de uma implementação em Transaction Script, onde cada uma das classes herdeiras de *Transaction Script* implementam diferentes funcionalidades. Se a aplicação envolve validações, cálculos e regras de negócio complexas, então o padrão arquitetural *Domain Driven* pode ser uma boa abordagem. Entretanto, caso implemente validações e regras de negócio simples, então *transaction script* é a abordagem adequada.

Um dos benefícios dessa abordagem é o fato de precisar se preocupar individualmente com as funcionalidades, visto que cada transação é implementada forma independente. Por outro lado, à medida que a complexidade da aplicação aumenta, pode se tornar difícil manter um bom design arquitetural e prevenir *transaction scripts* duplicados (FOWLER, 2002, tradução minha).

Figura 25: Diagrama de classes de uma aplicação implementada no padrão Transaction Script

|  |
| --- |
|  |

Fonte: Fowler (2002, p. 111)

## 2.9 TESTES

A disciplina de Testes busca verificar os resultados da implementação, através do planejamento, desenho e realização das atividades do processo de testes. Testar um software é indispensável para a identificação de defeitos e avaliação do grau de qualidade[[4]](#footnote-5) de um produto e de seus componentes. Fatores como cronograma e orçamento frequentemente impõem restrições à quantidade de testes que é possível executar (PAULA FILHO, 2009).

Segundo Paula Filho (2009), “um teste é uma atividade na qual um produto, sistema ou componente é executado sob condições especificadas, com obser- vação e registro dos resultados e avaliação de um ou mais aspectos”. Cada teste tem pelo menos um objetivo de teste, que é um conjunto identificado de características que devem ser testadas, mensuradas e depois comparadas em relação ao comportamento esperado, conforme descrito na documentação do software. Um critério de teste é um critério que o componente testado deve satisfazer para passar em um dado teste. Uma bateria de testes é uma implementação de um teste que atende a um objetivo específico.

Segundo Glass (2003, apud PAULA FILHO, 2009) remover defeitos é a etapa do processo de software que mais consome tempo. Sendo quase impossível executar uma boa remoção de erros sem uso de ferramentas, e raramente são utilizadas ferramentas além de depuradores.

É importante que os testes sejam bem planejados e desenhados para melhor aproveitamento dos recursos alocados para eles. Nem sempre a detecção de um defeito por um teste é algo óbvio, por isso durante e após a realização de um teste, os resultados devem ser inspecionados, comparando os resultados previstos e os resultados obtidos (PAULA FILHO, 2009).

O planejamento e o desenho de testes devem ser feitos por pessoas que conheçam a metodologia de testes usada, enquanto a realização dos testes baseada em um desenho bem definido, pode ser feita por uma pessoa menos experiente ou de forma automatizada. Recomenda-se que os testes sejam feitos, se não por testadores, ao menos por outra pessoa que não aquela que desenvolveu o software, pois frequentemente têm maior dificuldade para identificar problemas nos produtos de seu trabalho (PAULA FILHO, 2009).

Um procedimento de teste é um conjunto detalhado de instruções para execução de um teste, normalmente derivado da descrição de um caso de uso. Um caso de teste, por sua vez, é a especificação das entradas, resultados previstos e condições de execuções para um item a se testar. Um teste pode ser visto como uma coleção desses dois elementos (PAULA FILHO, 2009).

Um procedimento de teste pode ser executado de forma manual ou automatizada. Um script de teste é uma representação formalizada de um procedimento de teste, normalmente é associado a um script executável de forma automatizada, mas testes manuais também podem utilizar representações mais estruturadas (PAULA FILHO, 2009).

O objetivo de um caso de teste não é demonstrar que um programa funciona, e sim encontrar defeitos. Para isso, deve obrigatoriamente incluir uma descrição de saídas esperadas, com as quais será comparado o resultado da execução, e um conjunto de entradas. Devem existir casos de testes tanto para entradas válidas como inválidas. Um caso de teste pode invocar vários procedimentos de teste, pode também depender do resultado de um outro teste, por isso a execução de casos de testes deve ter uma ordem especificada (PAULA FILHO, 2009).

A quantidade potencial de defeitos que podem ser detectados na execução de um testes é denominada cobertura de testes. O objetivo central de toda a disciplina de testes é maximizar a cobertura, que pode ser potencializada com ferramentas de automação (PAULA FILHO, 2009).

Testes de software podem ser classificados de acordo com diferentes características, o que implica em uma nomenclatura diversa. As características pelas quais um teste de software pode ser classificado são: o papel que desempenham dentro do processo de software, visibilidade em relação ao componente de teste; formalismo na execução; e automação da execução (PAULA FILHO, 2009).

## 2.10 BANCO DE DADOS

Segundo Cardoso e Cardoso (2012), o conceito de banco de dados surgiu diante da necessidade de armazenar dados e disponibilizá-los para consulta e acesso, em um cenário onde, até então, os dados eram mantidos em grupos de registros, armazenados em sistema de arquivos. Essa forma de armazenar dados recorrentemente apresentava problemas de incompatibilidade, redundância, inconsistência, dificuldade de acesso e problemas de segurança. Nesse contexto, surgiu o conceito de banco de dados: uma coleção de dados organizada em uma estrutura para armazenamento de informações e com propriedades determinadas.

Machado (2014) define banco de dados como um conjunto de dados devidamente relacionados, organizado de forma lógica e com um significado inerente, que representam e refletem um aspecto do mundo real. Além disso, deve ser projetado, construído e preenchido com valores de dados para um propósito específico, visando atender um conjunto predefinido de usuários e de aplicações.

### 2.10.1 Modelos de Dados

Para serem armazenados em um sistema de banco de dados, os dados devem ser dispostos de forma organizada e coerente. Um modelo de dados é um conjunto de ferramentas utilizadas para fornecer uma representação dos conceitos e estruturas lógica e física, junto de suas propriedades, em um sistema de banco de dados. Os modelos podem ser classificados de acordo com seu nível de abstração, entre modelos de alto nível e modelos de baixo nível (CARDOSO, CARDOSO, 2012).

Um modelo de alto nível é conceitual e fornece uma visão da estrutura de um banco de dados próxima de como ela é vista naturalmente, enquanto um modelo de baixo nível (também chamado de modelo físico) fornece uma visão de como os dados são realmente armazenados em um computador (CARDOSO, CARDOSO, 2012).

#### 2.10.1.1 Modelo de entidade-relacionamento

Conforme Cardoso e Cardoso (2012), nesse modelo, o mundo real é representado por um conjunto de objetos básicos, denominados entidades, e os relacionamentos entre elas. As entidades são descritas por meio de atributos, que representam os dados associados ao objeto representado. Os relacionamentos correspondem às associações entre as entidades. A Figura 26 apresenta um exemplo de modelo de entidade-relacionamento.

Figura 26: Exemplo de modelo de entidade-relacionamento

|  |
| --- |
|  |

Fonte: Cardoso e Cardoso (2012, p. 21)

### 2.10.2 Sistema Gerenciador de Banco de Dados

Conforme Cardoso e Cardoso (2012), um sistema gerenciador de banco de dados é um software projetado para guardar e gerenciar bancos de dados, permitindo que usuários criem e manipulem bancos de dados com diferentes propósitos. Seu objetivo é proporcionar o acesso fácil e eficiente a informações armazenadas. Um SGBD possui uma estrutura que mantém não somente os dados, mas também a forma como são armazenados, contendo uma descrição completa desse banco.

Para possibilitar a definição, manipulação e consulta de dados em um banco de dados, um sistema de banco de dados possui uma linguagem de consulta de alto nível, denominada Structured Query Language (SQL). A linguagem SQL pode ser dividida entre Data-Definition Language (DDL) e Data Manipulation Language (DML), cuja função é, respectivamente, definir esquemas e estruturas de bancos de dados; e manipular (consultar, inserir, remover ou modificar) informações armazenadas (CARDOSO, CARDOSO, 2012).

Segundo as autoras, por meio de um programa de aplicação é possível utilizar consultas SQL para conversar com o SGBD e fazer manipulações e consultas com os dados. O conjunto formado por um banco de dados, um SGBD e as aplicações que o manipulam é chamado de sistema de banco de dados, e é ilustrado na Figura 27, onde fica mais clara a função exercida por um SGBD.

Figura 27: Representação de um sistema de banco de dados

|  |
| --- |
|  |

Fonte: Cardoso e Cardoso (2012, p. 18)

#### 2.10.2.1 MySQL e MariaDB

O MySQL é um software da categoria SGBD apreciada por várias empresas, entidades e pessoas pois possui um servidor confiável, rápido e de fácil utilização, que pode ser utilizado com grandes bancos de dados, inclusive em aplicações voltadas para a internet (MANZANO, 2011).

De acordo com Manzano (2011) Grande parte de seu sucesso está associado à fácil integração com a linguagem PHP. O SGBD apresenta as seguintes características: portabilidade para diversos sistemas operacionais; compatibilidade com *wwwdrivers* ODBC, JDBC e .NET; módulos de interfaceamento com as linguagens de programação Java, C, C++, Python, Perl, PHP e Ruby; facilidade de uso; excelente desempenho e estabilidade, exigindo poucos recursos de hardware; além de uma versão baseada na filosofia de software livre.

Até a compra do MySQL pela Oracle, em 2009, sua distribuição era baseada em software livre. Depois disso, foi criado o MariaDB, uma versão *open source*do MySQL, que implementa estrutura e comandos semelhantes e o funcionamento idêntico (BARBOZA, FREITAS, 2018).

## 2.11 PROGRESSIVE WEB APPS

Segundo Trindade e Affini (2018), aplicativos móveis (Apps) são softwares projetados para execução em dispositivos móveis, como *smartphones* ou *tablets.* A instalação de Apps pode ser realizada através de lojas online, como a App Store para dispositivos com sistema operacional iOS, da fabricante Apple; Play Store para dispositivos Android, sistema operacional desenvolvido pela Google; entre outros.

Ainda segundo as autoras, um aplicativo desenvolvido para uma plataforma Android não é compatível com a plataforma iOS, por exemplo, e vice-versa. Cada plataforma possui suas próprias tecnologias para construção de aplicativos, o que implica na necessidade de desenvolvimento de um novo aplicativo para cada plataforma, sob as tecnologias definidas. Isso torna o desenvolvimento de aplicativos custoso e demorado, pois há pouca portabilidade entre as plataformas, exigindo que um único aplicativo seja reescrito diversas vezes, para compatibilizar com as plataformas.

De acordo com Lima (2017), em 2007, a visão original de Steve Jobs para aplicativos é de que seriam escritos com as mesmas tecnologias utilizadas para construir aplicações Web. Entretanto, em 2008 foi lançada a Apple Store, uma plataforma que permitia aos usuários do sistema operacional iOS baixar aplicativos nativos, resultando no crescimento do número de aplicativos desenvolvidos e disponibilizados através das diversas lojas de aplicativos disponíveis para as plataformas. Porém, o declínio dos aplicativos nativos vem sendo observado.

Segundo Lima (2017), esse declínio pode ser percebido através da redução de *download* de novos *apps*; grande parte dos aplicativos disponíveis nas lojas online sequer são baixados; além disso, mais de 80% do tempo de usuários de *smartphones* é gasto no uso de até 5 aplicativos. Esses fatos indicam que usuários tendem a baixar somente aplicativos que de fato irão utilizar.

Nesse contexto, *Progressive Web Apps* (PWAs) surgem como uma boa alternativa, pois, assim como aplicativos, podem ser acessados por um navegador *web* e instalados, para serem executados a partir das telas iniciais dos dispositivos, além de entregar a interface e (quase todos) os recursos de interação com hardware possíveis em um aplicativo com a praticidade de um site (ROCK CONTENT, 2019).

De acordo com o *Google* (2020, apud TRINDADE, AFFINI, 2018), um PWA é caracterizado por:

Progressivo: deve atender qualquer usuário independentemente do navegador ou sistema operacional utilizado;

* Responsivo: deve se adequar a diferentes formatos de exibição (*desktop*, *tablet*, *mobile*), ou seja, diferentes resoluções;
* Independente de conectividade: deve funcionar mesmo com conexões lentas ou até mesmo sem conexão, recurso possibilitado por um *Service Worker*;
* *App-like*: adotar layout, recursos e navegação semelhantes a aplicativos;
* Atualizado: deve se manter atualizado de forma automática e discreta, possibilitado por um *Service Worker*;
* Seguro: deve implementar protocolo HTTPS para evitar invasões e adulterações durante a troca de dados;
* Encontrável: pode ser identificado como um “aplicativo” através de um arquivo de manifesto e ao escopo de registro do *Service Worker*;
* Reengajável: permite o engajamento de usuários por meio de notificações *push*;
* Instalável: fornece uma forma de instalar a aplicação, sem necessidade de acessar uma loja de *apps*;
* Linkável: não requer instalação e pode ser compartilhado por meio de URL.

Entre as vantagens dessa tecnologia, destacam-se a retenção e economia. No que tange a retenção, um PWA requer um número reduzido de etapas necessárias para acesso às funcionalidades do produto em relação a um aplicativo comum. Um PWA pode ser acessado diretamente através de um navegador web e fornece ao usuário a opção de instalação simplificada, enquanto a interação com um aplicativo comum requer que o usuário ao menos faça a busca em uma loja online, faça o *download* e execute, para então experimentar suas funcionalidades. Conforme comentado anteriormente, um PWA não exige a construção de uma nova aplicação para cada plataforma e com isso, os benefícios de economia estão associados à dispensa da necessidade de uma equipe especializada em desenvolvimento para cada plataforma, além do tempo investido para construção do mesmo aplicativo por mais de uma vez (LIMA, 2017).

### 2.11.1 Requisitos

Segundo Trindade e Affini (2018), para iniciar um projeto *web* utilizando conceitos de PWA são necessários três requisitos: Conexão HTTPS, *Web App Manifest* e *Service Worker*. Vale ressaltar que o funcionamento do *app manifest* ou do *service worker* depende da compatibilidade com o navegador, entretanto, o navegador *Google Chrome* já possui compatibilidade para dispositivos Android e iOS, enquanto o navegador Safari já sinaliza desenvolvimento de suporte para PWA e estudos apontam investimento massivo em PWA pela empresa *Microsoft*. Apesar disso, a aplicação deve funcionar mesmo sem compatibilidade com a tecnologia, visto que é uma aplicação *web*.

HTTPS é a sigla em inglês de *Hyper Text Transfer Protocol Secure* (“Protocolo de transferência de hipertexto seguro”, em português), é uma implementação do protocolo HTTP com uma camada de segurança que utiliza o protocolo SSL/TLS. Com isso, os dados são transmitidos por uma conexão criptografada e é possível verificar a autenticidade do servidor e do cliente por meio de certificados digitais Wikipedia (2018, apud TRINDADE, AFFINI, 2018).

O Manifesto do Aplicativo *Web*, conhecido como *Web App Manifest*, é um arquivo em formato JSON[[5]](#footnote-6) que fornece informações sobre o aplicativo (como nome, autor, ícone e descrição) e como deve se comportar quando instalado no dispositivo. Um *web app manifest* faz parte de uma coleção de tecnologias de *Progressive Web Apps*, que permite a instalação em um dispositivo sem estar em uma loja de aplicativos, além de recursos como trabalhar *offline* e receber notificações *push* (TRINDADE, AFFINI, 2018).

Por fim, um *Service Worker* é um script que o navegador executa em segundo plano, possibilitando recursos que não precisam de uma página ou de interação com o usuário, como notificações *push* e sincronização em segundo plano. Além disso, são capazes de interceptar e tratar solicitações de rede, incluindo o gerenciamento programático de um *cache* de respostas. Comportam-se como uma camada entre a aplicação e o navegador, que possibilita criação de experiências offline, interceptação de requisições de rede e atualização de repositórios residentes no servidor, fornecendo uma boa experiência ao usuário (TRINDADE, AFFINI, 2018).

## 2.12 APLICAÇÕES WEB

Segundo Winckler e Pimenta (2002), uma aplicação web pode ser definida como um software que utiliza a Web como ambiente de execução, pode ser classificada como um Site Web ou um sistema Web (CONALLEM, 2000, apud WINCKLER; PIMENTA, 2002, p. 11). Um Site Web é um sistema hipermídia distribuído, cujo propósito é permitir pesquisa e acesso direto a documentos e informações através da internet para computadores clientes. O acesso ocorre através de um software chamado *browser* (navegador), instalado no computador do cliente, utilizando a infraestrutura da internet, sob o protocolo HTTP (Hyper Text Transfer Protocol). O conteúdo disponível em um Site Web é estático, definido por um arquivo ou documento pré-formatado.

Uma aplicação web é um sistema web que permite aos usuários executarem lógica de negócio e persistir dados através da interação com a interface da aplicação. Diferentemente de sites web, o conteúdo de uma aplicação web é dinamicamente construído, baseado na interação do usuário com as páginas, através do *browser* (WINCKLER; PIMENTA, 2002).

Segundo Miletto e Bertagnolli (2014), um sistema web pode ser considerado um sistema distribuído, implementado sob a arquitetura cliente-servidor, como ilustrado na Figura 28. Na figura, o lado cliente é acessa a aplicação a partir de um navegador web (*browser*); através da internet, são feitas solicitações ao servidor web remoto, que abriga a aplicação e é responsável por interpretar a solicitação, e responde com o respectivo documento. Além disso, a aplicação pode interagir com o sistema de arquivos local através da operação de um banco de dados ou de arquivos propriamente ditos.

Figura 28: Implementação cliente-servidor de uma aplicação Web

|  |
| --- |
|  |

Fonte: (WINCKLER; PIMENTA, 2002, p. 12)

Ainda segundo Miletto, Franco e Bertagnolli (2014), é considerada uma boa prática na construção de sistemas web, estruturar a implementação em camadas com responsabilidades específicas, conforme representado na Figura 29. Essa separação em camadas fornece robustez e consistência, além de reduzir o acoplamento entre os componentes.

Figura 29: Camadas de uma aplicação web

|  |
| --- |
|  |

Fonte: Miletto, Franco e Bertagnolli (2014, p. 6)

### 2.12.1 Construção

Cada camada de uma aplicação web pode ser construída através do uso de uma gama de tecnologias. Fundamentalmente, a camada de apresentação é executada no lado do cliente e é construída com o uso da tecnologia HTML, estilizada com CSS e pode ter o comportamento definido pela linguagem *JavaScript*. As camadas de processamento da aplicação, gerenciamento de dados e banco de dados, por sua vez, ficam localizadas no lado do servidor. A camada de processamento da aplicação é construída com o uso de alguma linguagem de programação do lado do servidor, como PHP, por exemplo, que com o uso de linguagem SQL (ver seção 2.10) realiza a manipulação de dados (MILETTO; FRANCO; BERTAGNOLLI, 2014).

#### 2.12.1.1 HTML

De acordo com MDN Web Docs (2020c), HTML (Linguagem de Marcação de Hipertexto, ou *Hyper Text Markup Language,* em inglês) é o código utilizado para estruturar uma página web e seu conteúdo. Consiste em uma série de elementos utilizados para delimitar ou agrupar partes do conteúdo para que apareça ou atua de determinada maneira. Um elemento é composto por, respectivamente, uma *tag* de abertura, o conteúdo, e uma *tag* de fechamento, conforme ilustrado na figura 30.

Figura 30: Estrutura de um elemento HTML

|  |
| --- |
|  |

Fonte: MDN Web Docs (2020c).

#### 2.12.1.2 CSS

Segundo MDN Web Docs (2020a), CSS (*Cascade Style Sheet,* ou Folha de Estilo em Cascata, em português), é o código utilizado para estilizar uma página web. Assim como o HTML, CSS é uma linguagem de folhas de estilos que permite aplicar estilos seletivamente a elementos em documentos, como deixar a cor de um texto vermelha, por exemplo. Para que um arquivo CSS atue sobre uma página web, ele deve ser aplicado ao documento HTML. Um arquivo CSS é composto uma série de conjunto de regras para definir o estilo de uma página web. Conforme apresentado na Figura 31, um conjunto de regras é definido por:

1. Seletor (*selector*): indica o elemento HTML ao qual o estilo será aplicado;
2. Declaração (*declaration*): define uma propriedade (property) e um valor (*property value*) que serão aplicados ao seletor;

Figura 31: estrutura de um conjunto de regras CSS

|  |
| --- |
|  |

Fonte: MDN Web Docs (2020a).

#### 2.12.1.3 JavaScript

JavaScript é uma linguagem de programação multi-paradigma utilizada principalmente do lado do cliente para criar páginas web dinâmicas, mas também é utilizado do lado do servidor, através da plataforma *NodeJs*. Do lado do cliente, fornece interação com uma série de APIs, que permite que a linguagem manipule o conteúdo de páginas *web*, manipule dados, interaja com o dispositivo que executa o navegador, entre outras funcionalidades (MDN WEB DOCS, 2020e).

Em websites modernos, é comum que se recuperem dados individuais de um servidor para atualizar partes de uma página sem precisar recarregá-la completamente. Isso é possível através de uma técnica denominada Ajax (*Asynchronus JavaScript and XML*) que permite o envio de requisições (*requests*) para um servidor, que deve retornar o conteúdo que será utilizado para atualizar parte de uma página web (MDN WEB DOCS, 2020b, tradução minha)

#### 2.12.1.4 Frameworks

“Um *framework* é uma estrutura genérica estendida para se criar uma aplicação ou subsistema mais específico”. (SOMMERVILLE, 2011, p. 300). De acordo com MDN Web Docs (2020f, tradução minha), do lado do servidor, frameworks web fornecem ferramentas e bibliotecas para simplificar operações comuns no desenvolvimento *web*. A adoção de um *framework* para desenvolvimento não é obrigatória, mas é altamente recomendada, uma vez que simplifica tarefas do dia-a-dia de um desenvolvedor. Entre as funções frequentemente fornecidas por um *framework*, pode-se citar: encapsulamento de conexões HTTP e interpretação dos métodos (GET, POST, PUT, PATCH, DELETE, etc.), roteamento de requisições, abstração e simplificação do acesso ao banco de dados e renderização de dados.

##### 2.12.1.4.1 Express

*NodeJs* (ou, somente *Node*, com menos formalidade) é um ambiente em tempo de execução e multiplataforma que permite a criação de aplicativos e ferramentas do lado do servidor sob linguagem *JavaScript*. Entre as principais vantagens oferecidas, destacam-se: performance excelente, código escrito em *JavaScript* (dispensa necessidade de aprendizado de uma outra linguagem para o *backend*), acesso a diversos pacotes (disponíveis no NPM), compatível com diversos sistemas operacionais, comunidade e ecossistemas muito ativos (MDN WEB DOCS, 2020d).

Express é o framework Node mais popular e a biblioteca originária de uma série de outros frameworks Node. É bastante minimalista, em contrapartida oferece bastante flexibilidade aos desenvolvedores para modelar as soluções. Além disso, fornece uma série de bibliotecas para trabalhar com cookies, sessões, login de usuários, parâmetros de URL, verbos HTTP, cabeçalhos, etc (MDN WEB DOCS, 2020d).

##### 2.12.1.4.1 VueJS

Conforme Galdino (2020) é um *framework* progressivo e flexível, baseado na linguagem *JavaScript*, para a construção de interfaces de usuário. É ideal para prototipagem rápida pois oferece uma maneira fácil e flexível de ligação de dados reativos e componentes reutilizáveis, mas também pode suportar aplicações web complexas e escaláveis. Além disso, fornece boa integração com as diversas bibliotecas *JavaScript*.

De acordo com Picollo (2020), a arquitetura de uma aplicação construída sob *VueJS* é baseada em componentes criados com a sintaxe de HTML, CSS e *JavaScript* em um único arquivo de extensão .vue, o que facilita o isolamento e a manutenção de funcionalidades. Assim, cada componente constitui um escopo isolado dos demais, tanto em lógica quantos nos estilos.

## 2.13 RESTFul Web Services

Para entender o que é um RESTFul *Web Service*, é necessário, primeiramente, compreender o que é um *web service*. De acordo com Breitman (2005), não há uma definição única para o termo *Web Service*, porém a autora destaca algumas definições consideradas mais esclarecedoras. Com base na identificação de pontos comuns entre elas, pode-se afirmar que um web service é um tipo de aplicação *web*, identificado através de uma URI, baseado no padrão XML, que fornece interfaces através das quais outras aplicações podem utilizar ou invocar os métodos e funcionalidades implementadas.

Segundo Salvadori (2015), um *web service* pode ser implementado através do padrão SOAP (*Simple Object Access Protocol*), que utiliza notação XML para a troca de informações; ou através de REST, que em sua maioria utiliza JSON para a troca de mensagens. Dessa forma, pode-se dizer que em uma definição mais recente, um *web service* não opera restritivamente sob XML, na verdade define um formato padrão para a comunicação com as aplicações clientes.

Por fim, De acordo com Machado, Franco e Prestes (2016), REST (Representational State Transfer) é uma abordagem para o desenvolvimento de aplicações com baixo acoplamento, escalabilidade e simplicidade arquitetural, viabilizada pela adoção de um conjunto de de restrições. As principais restrições REST são:

“As aplicações devem respeitar o modelo cliente/servidor; devem existir intermediários entre o cliente e o servidor e, nesses mediadores, ser possível a utilização de cache, por exemplo, o *proxy* do HTTP; qualquer tipo de recurso deve ser definido por meio de *Uniform Resource Identifiers* (URIs), da mesma forma como acontece com os links na Web”; deve-se delimitar um conjunto de métodos, por exemplo, HTTP GET, POST, PUT, etc.; não se deve possuir um estado para a aplicação (*stateless*) (MACHADO; FRANCO; PRESTES, 2016, p. 193).”

Recomenda-se o uso de uma abordagem REST quando um serviço necessita ser incorporado em uma página Web, através de Ajax, por exemplo.

# 3 MATERIAL E MÉTODOS

Este trabalho caracteriza-se como uma pesquisa qualitativa, na qual apresenta-se o desenvolvimento de um protótipo de aplicativo multiplataforma que permite a busca e divulgação de produtores locais de alimentos. Tendo como base o notável aumento pela busca por hábitos alimentares mais saudáveis, bem como o fomento da economia local, considerando o uso de software como potenciais soluções para os mais diversos problemas encontrados em uma sociedade, mais especificamente o uso de aplicativos como solução para problemas pontuais.

## 3.1 PESQUISA BIBLIOGRÁFICA

Buscando compreender os domínios dos problemas em comum enfrentados na busca por hábitos alimentares mais saudáveis e pelo fomento da economia local, iniciou-se uma pesquisa bibliográfica com o objetivo de compreender o cenário atual relacionado a estes dois assuntos, e de que forma o uso de software poderia preencher as lacunas identificadas.

Na sequência, foi realizada também uma pesquisa bibliográfica acerca dos assuntos relacionados ao desenvolvimento de software e especificamente de aplicativos, a fim de construir uma base de conhecimento sólida para o desenvolvimento da solução consistente e efetiva.

## 3.2 IDENTIFICAÇÃO DE STAKEHOLDERS

A partir da escolha de um problema específico para propor uma solução, com base nos resultados da pesquisa bibliográfica acerca da problemática, foram identificados os stakeholders, isto é, os indivíduos impactados pela solução proposta.

## 3.3 ANÁLISE DE REQUISITOS

Como a proposta da solução baseou-se somente na pesquisa bibliográfica, buscou-se identificar requisitos mínimos funcionais e não funcionais para a construção de um protótipo com funcionalidades genéricas que atendam às necessidades dos stakeholders dentro dos domínios do problema.

## 3.4 PLANEJAMENTO

Avaliando os requisitos, planejou-se a solução, adotando para o desenvolvimento do protótipo o modelo orientado a cronograma, tendo em vista os prazos do projeto, a forma de elicitação de requisitos e a característica de protótipo atribuída à proposta de solução.

Na primeira etapa, foi desenvolvido um modelo de alto nível, onde foram desenvolvidos os diagramas de caso de uso e de classe. Com base no documento de requisitos, juntamente do Diagrama de Caso de Uso e na compreensão das dependências e criticidade de cada caso de uso, foi possível ordenar por prioridade os conjuntos funcionais que deveriam compor o protótipo. Dessa forma, cada uma das iterações dedicou-se a entregar um artefato de software que atenda ao caso de uso mais prioritário ainda não tratado.

Cada iteração responsável pela entrega de uma funcionalidade correspondente a um caso de uso incluiu, em respectiva ordem, as atividades de modelagem, construção e entrega, compostas pelas ações de:

1. Modelagem: especificação do caso de uso e desenvolvimento de diagrama de sequência correspondente ao caso de uso;
2. Construção: documentação dos casos de teste, codificação e execução dos casos de teste;
3. Entrega: testes de integração e entrega do incremento.

## 3.5 MODELAGEM

Para desenvolvimento dos diagramas mencionados a seguir, utilizou-se o software StarUML, devido a amplitude dos recursos voltados à linguagem UML, sendo esse seu foco principal.

### 3.5.1 Diagrama de Casos de Uso

Após o levantamento de requisitos e planejamento, foi desenvolvido o diagrama de caso de uso, baseado nos requisitos funcionais. Com isso, foi possível visualizar quais funcionalidades deveriam compor o protótipo.

A partir da primeira iteração, o caso de uso correspondente foi especificado e descrito com o uso do editor de texto Google Docs, fornecendo informações de como espera-se que ocorra a interação com o protótipo.

### 3.5.2 Diagrama de Classes

Visando identificar as classes e respectivos atributos e funções, necessários para o funcionamento da aplicação, desenvolveu-se o diagrama de classes. O diagrama fornece também uma visão do modelo relacional, onde os atributos correspondem aos atributos, as classes às entidades e as relações entre classes às relações entre entidades. O diagrama foi incrementado conforme necessário nas iterações seguintes.

### 3.5.3 Diagrama de Sequência

Com base nos artefatos da primeira iteração, somado à descrição do caso de uso, desenvolveu-se o diagrama de sequência. Com isso, criou-se uma visão do fluxo de processos que devem ser codificados para a viabilização do caso de uso.

## 3.6 CONSTRUÇÃO

A construção, realizada a cada iteração responsável pela entrega de um incremento correspondente a um caso de uso, incorporou três principais ações, em respectiva ordem: identificação e documentação dos casos de teste, construção e execução dos casos de teste.

### 3.6.1 Documentação dos Casos de Teste

Sabendo-se do comportamento esperado pelo sistema, informação fornecida pelos artefatos produzidos até o momento, os casos de teste foram identificados e documentados, para que ao final da construção do caso de uso os mesmos sejam executados a fim de garantir o correto atendimento do caso de uso. Para a documentação, utilizou-se o editor de texto Google Docs.

### 3.6.2 Codificação

Para a codificação, optou-se por construir a aplicação utilizando arquitetura cliente-servidor, separada em três camadas: cliente, aplicação e dados.

O papel do cliente é desempenhado por uma aplicação construída com o uso das tecnologias HTML e CSS, sob a framework VueJs, a qual é baseada na linguagem de programação JavaScript. Aplica-se também à camada de cliente, o conceito de Progressive Web App, tecnologia que permite que a mesma aplicação seja acessada tanto através de um navegador web, como seja instalada em dispositivos móveis e se comporte de maneira semelhante a um aplicativo nativo. A arquitetura escolhida para a construção da aplicação cliente foi a arquitetura MVC.

A camada de aplicação foi centralizada em um webservice, construído utilizando a linguagem de programação JavaScript, sob a framework Express, aplicando o padrão API REST em uma arquitetura de transação.

Por fim, a responsabilidade da camada de dados é desempenhada por um banco de dados PostgreSQL, cuja estrutura é definida no código da camada de aplicação, com o uso da biblioteca JavaScript Knex. Optou-se por utilizar essa biblioteca pois fornece uma camada de abstração entre a aplicação e a base de dados, permitindo que a mesma estrutura seja utilizada em diversas plataformas de banco de dados sem a necessidade de construção de scripts DDL específicos para cada tipo de banco de dados. Dessa forma, é possível migrar a estrutura da base de dados da aplicação sem grandes complicações, visto que está definida em um código comum aplicável a diferentes tipos de bases de dados, além de permitir o versionamento da estrutura.

Definidos os papéis, a codificação da solução fez-se na seguinte sequência:

1. Quando necessaŕio, codificação da estrutura da base de dados necessária para o caso de uso, utilizando migrations da biblioteca JavaScript Knex;
2. Quando necessário, codificação dos recursos REST que serão consumidos no caso de uso;
3. Codificação da interface gráfica através da qual o cliente poderá efetivar o caso de uso, bem como as funções que deverão consumir recursos da API REST.

### 3.6.3 Execução dos Casos de Teste

Executaram-se os casos de teste conforme descritos na documentação, para garantia de que o recurso foi implementado com sucesso. Caso a execução do teste tenha identificado algum problema, retoma-se a ação de construção, buscando identificar e solucioná-lo.

## 3.7 ENTREGA

Na etapa de entrega, foram realizados os testes de integração, incluindo o novo incremento de software. Caso sejam identificados erros nos testes de integração, retoma-se a ação de construção, buscando identificá-lo e corrigi-lo. Caso contrário, o incremento de software é entregue e obtêm-se uma versão mais completa do protótipo.

# 4 RESULTADOS E DISCUSSÕES

Trata-se da recapitulação sintética dos resultados da pesquisa, ressaltando o alcance e as consequências de suas contribuições, bem como seu possível mérito.

# 5 CONCLUSÃO

Este item pode também ser chamado de CONSIDERAÇÕES FINAIS.

É a parte final do trabalho que deve confrontar o que foi evidenciado na interpretação dos resultados, com as hipóteses ou com o enunciado do problema. A redação deve ser precisa e categórica.

# 

# REFERÊNCIAS

AZEVEDO, Eliane. **O Ativismo Alimentar na Perspectiva do Locavorismo**. Ambiente & Sociedade. São Paulo, v. 18, n. 3, p. 81-98, jul.-set 2015. Disponível em: <<https://doi.org/10.1590/1809-4422ASOC740V1832015>>. Acesso em: 18 abr. 2020.

BARBOZA, Fabrício Felipe Meleto; FREITAS, Pedro Henrique Chagas. **Modelagem e desenvolvimento de banco de dados.** Porto Alegre: SAGAH, 2018. E-book.

BEZERRA, Eduardo. **Princípios de análise e projeto de sistemas com UML.** 3 ed. Rio de Janeiro: Elsevier, 2015. E-book.

CARDOSO, Virgínia; CARDOSO, Giselle. **Sistema de banco de dados:** uma abordagem introdutória e aplicada. São Paulo: Saraiva, 2012. E-book.

CARVALHO, André C. P. L. F. de; LORENA, Ana Carolina. **Introdução à computação**: Hardware, Software e Dados. Rio de Janeiro: LTC, 2017. E-book.

FOWLER, Martin. **Patterns of enterprise application architecture.** Addison-Wesley, 2002. E-book.

GALDINO, Fabricio. **VueJs Tutorial.** 2017. Disponível em: <<https://www.devmedia.com.br/vue-js-tutorial/38042>>. Acesso em: 22 jun. 2020.

HALWEIL, Brian. **The Argument for Local Food**. World Watch, p. 20-27, Mai.-jun, 2003. Disponível em: <<https://www.iatp.org/documents/the-argument-for-local-food>> Acesso em: 22 abr. 2020.

BREITMAN, Katin Koogan. **Web semântica:** a internet do futuro. Rio de Janeiro: LCT, 2005. E-book.

LIMA, Matheus. **Introdução aos Progressive Web Apps**. 2017. Disponível em: <[https://medium.com/tableless/introdu%C3%A7%C3%A3o-aos-progressive-web-apps-ad47ba24cddb](https://medium.com/tableless/introdução-aos-progressive-web-apps-ad47ba24cddb)>. Acesso em 21 jun. 2020.

MACHADO, Felipe Nery Rodrigues. **Projeto e implementação de banco de dados**. 3 ed. São Paulo: Érica, 2014. E-book.

MACHADO, Rodrigo Prestes; FRANCO, Márcia H. I.; BERTAGNOLLI; Silvia de Castro. **Desenvolvimento de software III**: programação de sistemas web orientada a objetos em Java. Porto Alegre: Bookman, 2016. E-book.

MANZANO, José Augusto N. G. **Mysql 5.5 interativo:** guia essencial de orientação e desenvolvimento. São Paulo: Érica, 2011. E-book.

MDN WEB DOCS. **CSS básico**. Disponível em: <<https://developer.mozilla.org/pt-BR/docs/Aprender/Getting_started_with_the_web/CSS_basico>>. Acesso em: 22 jun. 2020a;

\_\_\_\_\_\_\_. **Fetching data from the server.** Disponível em: <<https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/Client-side_web_APIs/Fetching_data>>. Acesso em: 22 jun. 2020b.

\_\_\_\_\_\_\_. **HTML básico**. Disponível em: <<https://developer.mozilla.org/pt-BR/docs/Aprender/Getting_started_with_the_web/HTML_basico>>. Acesso em: 22 jun. 2020c.

\_\_\_\_\_\_\_. **Introdução Express/Node.** Disponível em: <[https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Express\_Nodejs/Introdu%C3%A7%C3%A3o](https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Express_Nodejs/Introdução)>. Acesso em: 22 jun. 2020d.

\_\_\_\_\_\_\_. **JavaScript.** Disponível em: <<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>>. Acesso em: 22 jun. 2020e.

\_\_\_\_\_\_\_. **Server-side web frameworks.** Disponível em: <<https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/First_steps/Web_frameworks>>. Acesso em: 22 jun. 2020f.

PAULA FILHO, Wilson de Pádua. **Engenharia de software:** fundamentos, métodos e padrões. 3 ed. Rio de Janeiro: LTC, 2009. E-book.

PICOLLO, Lucas. **Vue JS:** o que é, como funciona e vantagens. Disponível em: <<https://blog.geekhunter.com.br/vue-js-so-vejo-vantagens-e-voce/>>. Acesso em: 22 jun. 2020.

PRESSMAN, Roger S; MAXIM, Bruce R. **Engenharia de Software:** uma abordagem profissional. 8 ed. Porto Alegre: AMGH, 2016. E-book.

RAINER JR, R. Kelly; CEGIELSKI, Casey G. **Introdução a Sistemas de Informação**: apoiando e transformando negócios na era da mobilidade. 5 ed. Rio de Janeiro: Elsevier, 2016. E-book.

ROCK CONTENT. **Entenda o que são Progressive Web Apps (PWAs) e veja os melhores exemplos do mercado.** 2019.Disponível em: <<https://rockcontent.com/blog/progressive-web-apps/>>. Acesso em 21 jun. 2020.

RUDY, Kathy. **Locavores, Feminism, and the Question of Meat.** The Journal of American Culture, v. 35, n. 1, p. 26-36, mar. 2012. Disponível em: <<https://doi.org/10.1111/j.1542-734X.2011.00795.x>>. Acesso em: 23 abr. 2020.

SALVADORI, Ivan Luiz. **Desenvolvimento de Web APIs RESTFul Semânticas Baseadas em JSON.** 2015. Dissertação (Mestrado em Ciência da Computação) - Departamento de Informática e Estatística. Universidade Federal de Santa Catarina, Florianópolis, 2015. Disponível em: <<https://repositorio.ufsc.br/bitstream/handle/123456789/132469/333102.pdf?sequence=1&isAllowed=y>>. Acesso em: 22 jun. 2020

SBROCCO, José Henrique Teixeira de Carvalho; MACEDO, Paulo Cesar de. **Metodologias Ágeis:** engenharia de software sob medida. São Paulo: Érica, 2012. E-book.

SCHACH, Stephen R. **Engenharia de software:** os paradigmas clássico e orientado a objetos. 7 ed. Porto Alegre: AMGH, 2010. E-book.

SOMMERVILLE, Ian. **Engenharia de Software**. 9 ed. São Paulo: Pearson Prentice Hall, 2011. E-book.

THOMPSON, Edward Jr; HARPER, Alethea Marie; KRAUS, Sibella. **Think Globally - Eat Locally**. San Francisco Foodshed Assessment, 2008. E-book.

TRINDADE, Patrícia Esteves; AFFINI, Leticia Passos. **APONTAMENTO A CERCA DO PROGRESSIVE WEB APPS**. In: III Jornada Internacional GEMInIS (JIG 2018). São Paulo, 2019. Disponível em: <<https://www.doity.com.br/anais/jig2018/trabalho/81964>>. Acesso em 21 jun. 2020.

UML.org. **What Is UML**. 2020. Disponível em: <<https://www.uml.org/what-is-uml.htm>>. Acesso em: 11 jun. 2020

WAZLAWICK, Raul Sidnei. **Engenharia de software:** conceitos e práticas. Rio de Janeiro: Elsevier, 2013. E-book.

WINCKLER, Marco; PIMENTA, Marcelo Soares. **Avaliação de Usabilidade de Sites Web.** *In:* Escola de Informática da SBC SUL (ERI 2002) ed. Porto alegre: Sociedade Brasileira de Computação, 2002. Disponível em: <<https://www.irit.fr/~Marco.Winckler/2002-winckler-pimenta-ERI-2002-cap3.pdf>.> Acesso em: 23 jun. 2020.

ZENKER, Aline Maciel et al. **Arquitetura de Sistemas.** Porto Alegre: SAGAH, 2019. E-book.

1. Também chamados de “ciclo de vida”, conforme Wazlawick (2013). [↑](#footnote-ref-2)
2. Também denominado prescritivo, ou tradicional (PRESSMAN, 2016) [↑](#footnote-ref-3)
3. Também chamada de barra de ativação, conforme Pressman (2016). [↑](#footnote-ref-4)
4. Quanto maior o número de defeitos detectados em um software, provavelmente maior também o número de defeitos não-detectados. (PAULA FILHO, 2009) [↑](#footnote-ref-5)
5. JavaScript Object Notation (JSON), é um modelo para armazenamento de transmissão de informações em formato de texto. Bastante utilizado por aplicações *Web* devido a sua estrutura mais compacta do que o modelo XML (TRINDADE, AFFINI, 2018). [↑](#footnote-ref-6)