UNIVERSIDADE EVANGÉLICA DE GOIÁS – UniEVANGÉLICA

BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

JOSÉ LUIZ PINTO JÚNIOR

ANÁLISE E PROJETO DE SOFTWARE PARA MICRO E PEQUENAS EMPRESAS

Anápolis

2021JOSÉ LUIZ PINTO JÚNIOR

ANÁLISE E PROJETO DE SOFTWARE PARA MICRO E PEQUENAS EMPRESAS

Trabalho de Conclusão de Curso I apresentado como requisito parcial para a obtenção de grau no curso de Bacharelado em Engenharia de Computação do Universidade Evangélica de Goiás – UniEVANGÉLICA.

Orientador(a): Profa. Ma. Natasha Sophie Pereira.

Anápolis

2021

JOSÉ LUIZ PINTO JÚNIOR

ANÁLISE E PROJETO DE SOFTWARE PARA MICRO E PEQUENAS EMPRESAS

Trabalho de Conclusão de Curso I apresentado como requisito parcial para a obtenção de grau no curso de Bacharelado em Engenharia de Computação do Universidade Evangélica de Goiás – UniEVANGÉLICA.

Aprovado pela banca examinadora em 9 de junho de 2021, composta por:

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

Profa. Ma. Natasha Sophie Pereira

Orientador

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

Prof. Eduardo Ferreira de Souza

Professor Convidado

Anápolis

2021

**RESUMO**

Cada vez mais, o ambiente competitivo é caracterizado por sua demanda por inovação contínua nas empresas, o que permite que seu desempenho seja aprimorado e seus lucros aumentem constantemente. Mesmo que as Micro e Pequenas Empresas (MPEs) reconheçam a importância da TI e se esforcem para utilizá-la a fim de aumentar sua competitividade no mercado e criar novas oportunidades de negócios, elas têm dificuldade em adotar a TI devido à falta de mão de obra, e de recursos financeiros e físicos. Deste modo, este trabalho visa dar suporte ao aumento da competitividade de uma pequena empresa de informática, localizada em Goianésia – GO, ao realizar um projeto de software que auxilie na gestão dos processos de negócios, facilitando a operacionalidade da empresa. Os requisitos funcionais e não-funcionais serão levantados através de reuniões com os *stakeholders*, a fim de nortear as características que o *software* deverá ter.Utilizar-se-á então a Linguagem Unificada de Modelagem (da sigla em inglês, UML) para mapear as funcionalidades do *software*, facilitando a compreensão da sua proposta. Até o momento, conseguiu-se compilar informações sobre arquitetura de software, bem como a importância de se realizar um planejamento correto antes mesmo de se escrever uma linha de código.

Palavras-chave: UML. Projeto de *software*. Arquitetura de *software*.

**Lista de Abreviações e Siglas**

APICS Production and Inventory Control Society

DMS Documento de Modelagem de Software

OMG Object Management Group

OMT Object Modeling Technique

OOSE Object Oriented Software Engineering

PME Pequenas e Médias Empresas

RUP Rational Unified Process

TI Tecnologia da informação

UML Unified Modeling Language

**Lista de Figuras**

[Figura 1 - Diagrama que ilustra o planejamento da arquitetura de software. 11](#_Toc75551052)

[Figura 2 - Relacionamentos em um diagrama de classe. 12](#_Toc75551053)

[Figura 3 - Exemplo de diagrama de sequência. 13](#_Toc75551054)

[Figura 4 - Exemplo de diagrama de Caso de Uso 17](#_Toc75551055)

[Figura 5 – Exemplo de Diagrama de Sequência 18](#_Toc75551056)

[Figura 6 - Exemplo de Diagrama de Colaboração 19](#_Toc75551057)

[Figura 7 - Exemplo de Diagrama de Máquina de Estados 19](#_Toc75551058)

[Figura 8 - Exemplo de Diagrama de Atividade 20](#_Toc75551059)

[Figura 9 - Exemplo de Diagrama Componente 21](#_Toc75551060)

[Figura 10 - Exemplo de Diagrama de Implantação 21](#_Toc75551061)

[Figura 11 - Modelo em cascata 25](#_Toc75551062)

Sumário

[1. Introdução 8](#_Toc75551063)

[1.1 Problema 8](#_Toc75551064)

[1.2 Objetivos 8](#_Toc75551065)

[1.2.1 Objetivo Geral 8](#_Toc75551066)

[1.2.2 Objetivos Específicos 8](#_Toc75551067)

[1.3 Justificativa 8](#_Toc75551068)

[1.4 Cronograma 9](#_Toc75551069)

[2. Fundamentação Teórica 11](#_Toc75551070)

[2.1 Arquitetura de Software 11](#_Toc75551071)

[2.2 Papel da Arquitetura de Software 12](#_Toc75551072)

[2.3 Foco nos Elementos Significativos 13](#_Toc75551073)

[2.4 Balanceamento das Necessidades dos Stakeholders 14](#_Toc75551074)

[2.5 A UML 14](#_Toc75551075)

[2.5.1 Representação da Arquitetura 16](#_Toc75551076)

[2.5.2 Diagrama de Caso de Uso 16](#_Toc75551077)

[2.5.3 Diagrama de Classes 17](#_Toc75551078)

[2.5.4 Diagrama de Sequência 17](#_Toc75551079)

[2.5.5 Diagrama de Colaboração 18](#_Toc75551080)

[2.5.6 Diagrama de Máquina de Estados 19](#_Toc75551081)

[2.6 Software de gestão de estoques 22](#_Toc75551082)

[2.7 Processo de software 23](#_Toc75551083)

[3. Metodologia 24](#_Toc75551084)

[3.1 Concepção 24](#_Toc75551087)

[3.2 Elaboração 24](#_Toc75551088)

[3.3 Modelo Cascata 25](#_Toc75551089)

[4. Resultados 27](#_Toc75551090)

[4.1 Resultados Obtidos 27](#_Toc75551091)

[4.2 Resultados esperados 27](#_Toc75551092)

[Referências 28](#_Toc75551093)

# Introdução

Cada vez mais, o ambiente competitivo é caracterizado por sua demanda por inovação contínua nas empresas, o que permite que seu desempenho seja aprimorado e seus lucros aumentem constantemente (REICHHELD, 1993). Como o papel de pequenas empresas vem crescendo e elas se tornaram uma parte importante da economia nacional, sua sobrevivência e crescimento são de grande impacto. A adoção de Tecnologias da Informação (TI) em Micro e Pequenas Empresas (MPEs) tem sido pesquisada ativamente, uma vez que a TI pode fornecer vantagens competitivas importantes para estes negócios (EIKEBROKK; OLSEN, 2007).

## Problema

Mesmo que as PMEs reconheçam a importância da TI e se esforcem para utilizá-la a fim de aumentar sua competitividade no mercado e criar novas oportunidades de negócios, elas têm dificuldade em adotar a TI devido à falta de mão de obra, e de recursos financeiros e físicos (CRAGG; CALDEIRA; WARD, 2011). Segundo dados do DataSebrae (2015), 24% dos pequenos negócios não utilizam computadores e 39% não utilizam *software* para gerir o negócio de forma integrada. Desta maneira, busca-se responder o seguinte questionamento: como aumentar a competitividade de uma pequena empresa com baixos recursos financeiros e humanos?

## Objetivos

### 1.2.1 Objetivo Geral

Desenvolver um projeto de *software* que auxilie na gestão de estoque de MPEs de modo a facilitar a operacionalização do negócio e trazer vantagens competitivas.

### 1.2.2 Objetivos Específicos

* Entender as necessidades apresentadas pelos *stakeholders*, bem como os requisitos funcionais e não-funcionais necessários para o desenvolvimento do projeto de *software*;
* Apontar possíveis restrições que possam ocorrer durante o planejamento do projeto.
* Implementar o Documento de Modelagem do Sistema (DMS) para o *software* proposto.

## Justificativa

A Sociedade de Produção e Controle de Inventário Americana, do inglês *American Production and Inventory Control Society* (APICS), define a gestão de estoque, um dos processos a serem facilitados com o uso do *software*, como o ramo da gestão empresarial que se preocupa com o planejamento e controle de estoques (TOOMEY, 2000). O gerenciamento de estoque é uma questão crítica de gerenciamento para a maioria das empresas, se tornando uma forma importante de aumentar sua força competitiva, portanto, uma questão importante para a maioria dos negócios (QUAYLE, 2003).

O gerenciamento de estoque eficaz deve incluir uma solução abrangente, devendo incluir *software* adequado. Quando se trata de pequenas empresas, o *software* deve ser acessível e fácil de usar. Pequenas empresas preferem usar o Microsoft Office ®, em vez de sistemas de informação especializados que são caros, porém mais eficientes (ILIASHENKO; SHIROKOVA, 2014).

Portanto, o desenvolvimento de um *software* de gestão empresarial resultará em uma administração mais assertiva da empresa, evitando desperdícios como excesso de estoque e trazendo mais segurança e controle aos colaboradores, aumentando sua força competitiva.

## Cronograma

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Atividade | 2021 | | | | | | | | | | | | | | | | | | | |
| Fev | | Mar | | Abr | | Maio | | Jun | | Jul | | Ago | | Setr | | Out | | Nov | |
| 1ª Quinzena | 2ª Quinzena | 1ª Quinzena | 2ª Quinzena | 1ª Quinzena | 2ª Quinzena | 1ª Quinzena | 2ª Quinzena | 1ª Quinzena | 2ª Quinzena | 1ª Quinzena | 2ª Quinzena | 1ª Quinzena | 2ª Quinzena | 1ª Quinzena | 2ª Quinzena | 1ª Quinzena | 2ª Quinzena | 1ª Quinzena | 2ª Quinzena |
| Entender as necessidades apresentadas pelos stakeholders |  |  | x | x | x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Levantamento dos requisitos não funcionais |  |  |  |  |  |  |  |  |  |  | x |  |  |  |  |  |  |  |  |  |
| Levantamento dos requisitos funcionais |  |  |  |  |  |  |  |  |  |  | x |  |  |  |  |  |  |  |  |  |
| Apontar possíveis restrições que possam ocorrer durante o planejamento do projeto |  |  |  |  |  |  |  |  |  |  |  | x |  |  |  |  |  |  |  |  |
| Realizar a modelagem dos diagramas da UML pertinentes |  |  |  |  |  |  |  |  |  |  |  |  |  | x | x | x | x | x |  |  |
| Validação com os stakeholders |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | x | x | x |

# Fundamentação Teórica

## Arquitetura de Software

A arquitetura de *software* de um programa ou sistema de computação é a estrutura, ou estruturas, do sistema que compreendem componentes de *software*, as propriedades externamente visíveis desses componentes e as relações entre eles. A arquitetura de *software* normalmente desempenha um papel fundamental como ponte entre os requisitos e a implementação. Uma arquitetura é o conjunto de decisões significativas sobre a organização de um sistema de *software*, a seleção de elementos estruturais e suas interfaces pelas quais o sistema é composto, juntamente com seu comportamento conforme especificado nas colaborações entre esses elementos, a composição desses elementos em subsistemas progressivamente maiores e o estilo arquitetônico que orienta essa organização - esses elementos e suas interfaces, suas colaborações e sua composição (GRADY; RUMBAUGH; JACOBSON, 2005).

Ao fornecer uma descrição abstrata de um sistema, a arquitetura expõe certas propriedades, enquanto oculta outras. Idealmente, esta representação fornece um guia intelectualmente tratável para o sistema geral, permite que os desenvolvedores reflitam sobre a capacidade de um sistema para satisfazer certos requisitos e sugere um plano para a construção e composição do sistema (GRADY; RUMBAUGH; JACOBSON, 2005). A Figura 1 ilustra, em forma de diagrama, o planejamento da arquitetura de software

|  |
| --- |
| Figura 1 - Diagrama que ilustra o planejamento da arquitetura de software. |
|  |
| Fonte: próprio autor. |

## Papel da Arquitetura de Software

No cerne de todas as inúmeras definições de arquitetura de *software* está a noção de que a arquitetura de um sistema descreve sua estrutura bruta. Essa estrutura ilumina as decisões de nível superior, incluindo coisas como de que maneira o sistema é composto de partes que interagem, quais são os principais caminhos de interação e quais são as propriedades principais das partes que a compõe. Além disso, uma descrição da arquitetura inclui informações suficientes para permitir uma análise de alto nível e uma avaliação crítica do sistema (GRADY; RUMBAUGH; JACOBSON, 2005).

A arquitetura de *software* é definida como a estrutura organizacional de um sistema de *software*, incluindo componentes, conexões, restrições e fundamentos. Os componentes podem ser pequenos pedaços de código, como módulos, ou pedaços maiores, como um sistema autônomo de gerenciamento de banco de dados. As conexões na arquitetura são abstrações de como os componentes interagem em um sistema, por exemplo, chamadas de procedimento. A arquitetura tem várias restrições e fundamentos associados a ela, incluindo as restrições na seleção de componentes e os fundamentos para a escolha de um componente específico em uma determinada situação (GUEDES, 2011). A Figura 2 exemplifica um diagrama de classe UML contendo algumas representações estruturais de um sistema de processamento de matrícula do aluno no curso. A Figura 2 apresenta três classes: Curso, Estudante e Professor. Todas as três classes se relacionam entre si com a ajuda de relacionamento de multi-dependência definidos na UML (do inglês *Unified Modeling Language* - Linguagem de Modelagem Unificada), vários atributos de classes são mostrados nas respectivas caixas de cada classe.

|  |
| --- |
| Figura 2 - Relacionamentos em um diagrama de classe. |
|  |
| Fonte: próprio autor. |

Além da estrutura do programa e do relacionamento entre suas partes, a definição da arquitetura é projetar a estrutura do *software* e a interação dos objetos (comportamento) antes da fase de projeto detalhado. O comportamento do *software* é aquilo que ele realmente faz após ser implementado no ambiente de produção. Além de definir os elementos estruturais, a arquitetura define o comportamento do sistema de *software* ao definir as interações entre os elementos estruturais (GUEDES, 2011).

|  |
| --- |
| Figura 3 - Exemplo de diagrama de sequência. |
|  |
| Fonte: Guedes (2011, p. 34). |

## Foco nos Elementos Significativos

Embora a arquitetura de *software* defina a estrutura e o comportamento, ela não se preocupa em definir toda a estrutura e todo o comportamento, preocupando-se apenas com os elementos considerados significativos. Os elementos significativos são aqueles que têm um efeito longo e duradouro, como os principais elementos estruturais, os elementos associados ao comportamento essencial e os elementos que tratam de qualidades significativas, como confiabilidade e escalabilidade (GUEDES, 2011). Em geral, a arquitetura de *software* não se preocupa com os detalhes refinados desses elementos. A arquitetura incorpora informações sobre como os elementos se relacionam entre si, omitindo especificamente certas informações sobre os elementos que não pertencem à sua interação.

Assim, uma arquitetura é antes de tudo uma abstração de um sistema que suprime detalhes de elementos que não afetam como eles usam, são usados por, se relacionam ou interagem com outros elementos. Como a arquitetura se concentra apenas em elementos significativos, ela nos fornece uma perspectiva particular do sistema em consideração - a perspectiva que é mais relevante para o arquiteto. Nesse sentido, a arquitetura é uma abstração do sistema que ajuda um arquiteto a gerenciar sua complexidade (GRADY; RUMBAUGH; JACOBSON, 2005).

## Balanceamento das Necessidades dos Stakeholders

A arquitetura é criada para atender a um conjunto de necessidades dos *stakeholders*. No entanto, muitas vezes não é possível atender a todas as necessidades expressas. Por exemplo, uma parte interessada pode solicitar alguma funcionalidade dentro de um prazo especificado, mas essas duas necessidades (funcionalidade e prazo) são mutuamente exclusivas. O escopo pode ser reduzido para atender ao cronograma ou todas as funcionalidades podem ser fornecidas em um período de tempo estendido (LAGO; AVGERIOU; HILLIARD, 2010).

Da mesma forma, diferentes partes interessadas podem expressar necessidades conflitantes e, novamente, um equilíbrio apropriado deve ser alcançado. Fazer trocas é, portanto, um aspecto essencial do processo de arquitetura, e a negociação, uma característica essencial do arquiteto de *software*. Outro desafio para o arquiteto é que as partes interessadas não estão apenas preocupadas com o fato de o sistema fornecer a funcionalidade necessária. Muitas das preocupações listadas são de natureza não funcional, pois não contribuem para a funcionalidade do sistema (por exemplo, as preocupações com relação a custos e programação). No entanto, tais preocupações representam qualidades ou restrições do sistema. Requisitos não funcionais são frequentemente os requisitos mais significativos no que diz respeito a um arquiteto (LAGO; AVGERIOU; HILLIARD, 2010).

## A UML

A Linguagem de Modelagem Unificada, do inglês *Unified Modeling Language* (UML), permite que analistas, arquitetos de *software* e desenvolvedores especifiquem, visualizem e documentem um sistema inteiro. Ao fornecer uma "linguagem" comum para tratar de sistemas complexos, a UML permite que muitos contribuidores diferentes, com perspectivas diversas, se comuniquem em bases comuns (GRADY; RUMBAUGH; JACOBSON, 2005).

As linguagens de modelagem orientadas a objetos surgiram em algum momento entre meados dos anos 1970 e o final dos anos 1980, quando metodologistas, diante de um novo gênero de linguagens de programação orientadas a objetos e aplicações cada vez mais complexas, começaram a experimentar abordagens alternativas para análise e design (GRADY; RUMBAUGH; JACOBSON, 2005). O número de métodos orientados a objetos aumentou de menos de 10 para mais de 50 durante o período entre 1989 e 1994. Muitos usuários desses métodos tiveram problemas para encontrar uma linguagem de modelagem que atendesse completamente às suas necessidades, alimentando assim as chamadas guerras de métodos (GRADY; RUMBAUGH; JACOBSON, 2005).

Aprendendo com a experiência, novas gerações desses métodos começaram a aparecer, com alguns métodos claramente proeminentes emergindo, mais notavelmente Booch, OOSE (Object-Oriented Software Engineering) e OMT (Object Modeling Technique). Cada um deles era um método completo, embora cada um fosse reconhecido como tendo seus pontos fortes e fracos. Em termos simples, o método Booch foi particularmente expressivo durante as fases de design e construção de projetos, o OOSE forneceu excelente suporte para casos de uso como uma forma de conduzir a captura de requisitos, análise, design de alto nível e OMT-2 foi mais útil para análise e sistemas de informação intensivos de dados. O componente comportamental de muitos métodos orientados a objetos, incluindo o método Booch e OMT, era a linguagem dos gráficos de estados, inventada por David Harel. Antes dessa adoção orientada a objetos, os gráficos de estados eram usados principalmente no domínio da decomposição funcional e análise estruturada, e levaram ao desenvolvimento de modelos e ferramentas executáveis que geravam código de execução completo (GRADY; RUMBAUGH; JACOBSON, 2005).

O esforço UML começou oficialmente em outubro de 1994, quando Rumbaugh se juntou com Booch na Rational. O foco inicial do projeto foi a unificação dos métodos Booch e OMT. O rascunho da versão 0.8 do Método Unificado (como era então chamado) foi lançado em outubro de 1995. Na mesma época, Jacobson se juntou à Rational e o escopo do projeto UML foi expandido para incorporar OOSE. Estes esforços resultaram no lançamento dos documentos UML versão 0.9 em junho de 1996 (GRADY; RUMBAUGH; JACOBSON, 2005).

Durante esse tempo, também ficou claro que muitas organizações de software viam a UML como estratégica para seus negócios. Estabeleceu-se um consórcio UML, com várias organizações dispostas a dedicar recursos para trabalhar em uma definição UML forte e completa. Os parceiros que contribuíram para a definição UML 1.0 incluíram Digital Equipment Corporation, Hewlett-Packard, ILogix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational, Texas Instruments e Unisys. Essa colaboração resultou na UML 1.0, uma linguagem de modelagem bem definida, expressiva, poderosa e aplicável a um amplo espectro de domínios de problemas. A UML 1.0 foi oferecida para padronização ao Object Management Group (OMG) em janeiro de 1997, em resposta à sua solicitação de proposta para uma linguagem de modelagem padrão (GRADY; RUMBAUGH; JACOBSON, 2005).

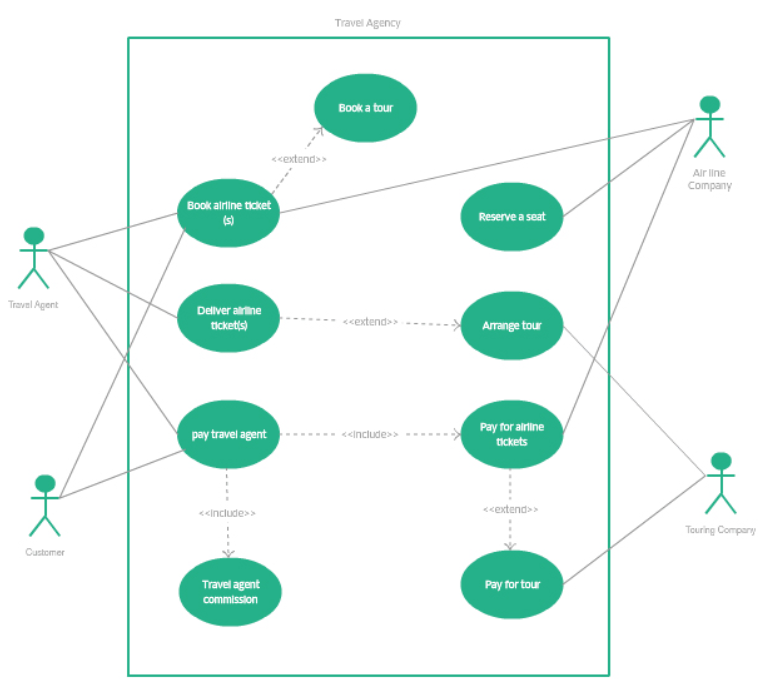
### Representação da Arquitetura

A arquitetura de uma aplicação utilizando a UML é representada seguindo as recomendações do *Rational Unified Process* (RUP). Cada diagrama UML é projetado para permitir que desenvolvedores e clientes visualizem um sistema de *software* a partir de uma perspectiva diferente e em vários graus de abstração. Os diagramas UML comumente criados em ferramentas de modelagem visual incluem, segundo Grady, Rumbaugh e Jacobson (2005):

### Diagrama de Caso de Uso

Os casos de uso especificam o comportamento esperado (o quê), e não o método exato de fazê-lo acontecer (como). Os casos de uso, uma vez especificados, podem ser denotados como representação textual e visual, utilizando-se de diagramas. Um conceito chave da modelagem de caso de uso é que ela auxilia a projetar um sistema da perspectiva do usuário final. É uma técnica eficaz para comunicar o comportamento do sistema nos termos do usuário, especificando todos os comportamentos do sistema visíveis externamente. A Figura 4 exemplifica um diagrama de caso de uso.

Figura 4 - Exemplo de diagrama de Caso de Uso



Fonte: https://creately.com/blog/diagrams/uml-diagram-types-examples/

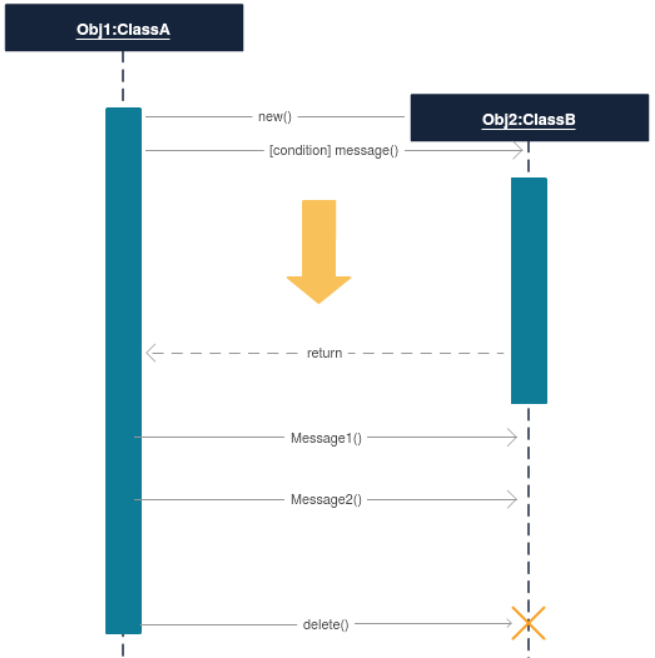
### Diagrama de Classes

Modela a estrutura e o conteúdo das classes usando elementos de *design* como classes, pacotes e objetos. Ele também exibe relacionamentos como contenção, herança, associações e outros.

### Diagrama de Sequência

Exibe a sequência de tempo dos objetos que participam da interação. Consiste na dimensão vertical (tempo) e dimensão horizontal (objetos diferentes), conforme ilustrado na Figura 5.

Figura 5 – Exemplo de Diagrama de Sequência

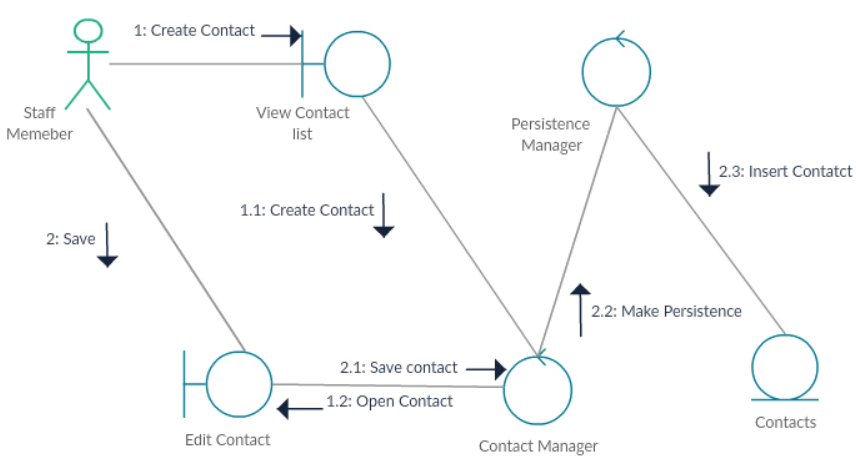


Fonte: https://creately.com/blog/diagrams/uml-diagram-types-examples/

### Diagrama de Colaboração

Exibe uma interação organizada em torno dos objetos e seus vínculos entre si. Os números são usados para mostrar a sequência de mensagens. Um exemplo deste diagrama pode ser encontrado na Figura 6.

Figura 6 - Exemplo de Diagrama de Colaboração

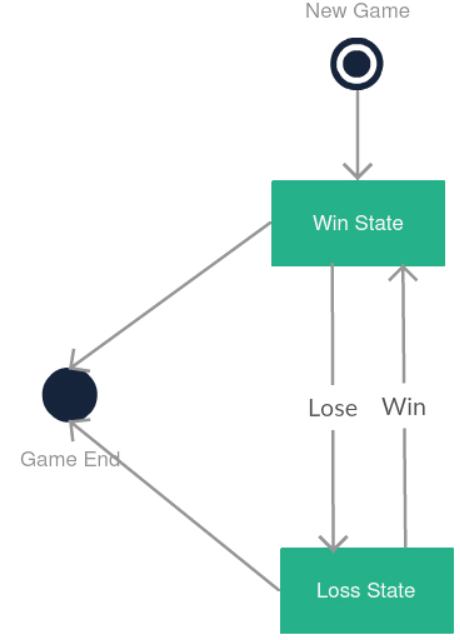


Fonte: https://creately.com/blog/diagrams/uml-diagram-types-examples/

### Diagrama de Máquina de Estados

Exibe as sequências de estados pelos quais um objeto de uma interação passa durante sua vida em resposta a estímulos recebidos, juntamente com suas respostas e ações, conforme exemplificado na Figura 7.

Figura 7 - Exemplo de Diagrama de Máquina de Estados

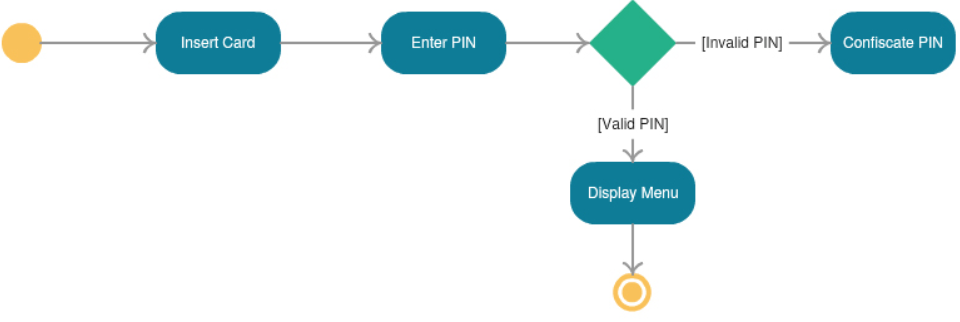


Fonte: https://creately.com/blog/diagrams/uml-diagram-types-examples/

#### 2.5.7 Diagrama de Atividade

Exibe um diagrama de estado especial onde a maioria dos estados são estados de ação e a maioria das transições são disparadas pela conclusão das ações nos estados de origem. Este diagrama se concentra em fluxos orientados por processamento interno, exemplificado na Figura 8.

Figura 8 - Exemplo de Diagrama de Atividade

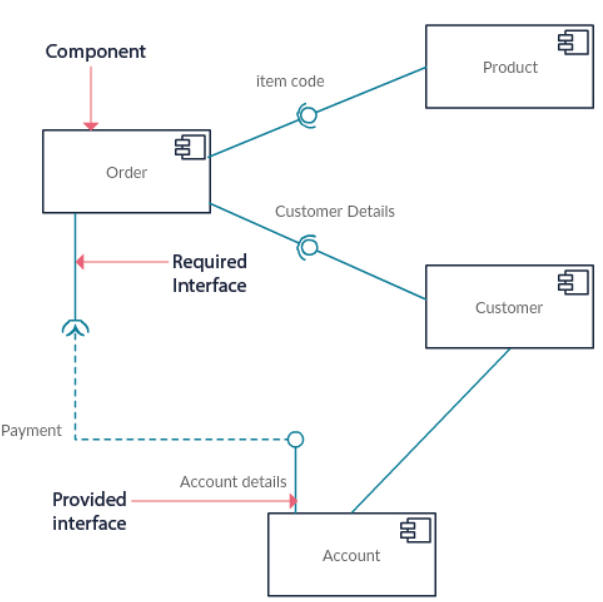


Fonte: https://creately.com/blog/diagrams/uml-diagram-types-examples/

#### 2.5.8 Diagrama Componente

Exibe a estrutura empacotada de alto nível do próprio código. As dependências entre os componentes são mostradas, incluindo componentes de código-fonte, componentes de código binário e componentes executáveis. Alguns componentes existem em tempo de compilação, em tempo de link, em tempos de execução, bem como em mais de uma vez. Um exemplo deste diagrama pode ser observado na Figura 9.

Figura 9 - Exemplo de Diagrama Componente

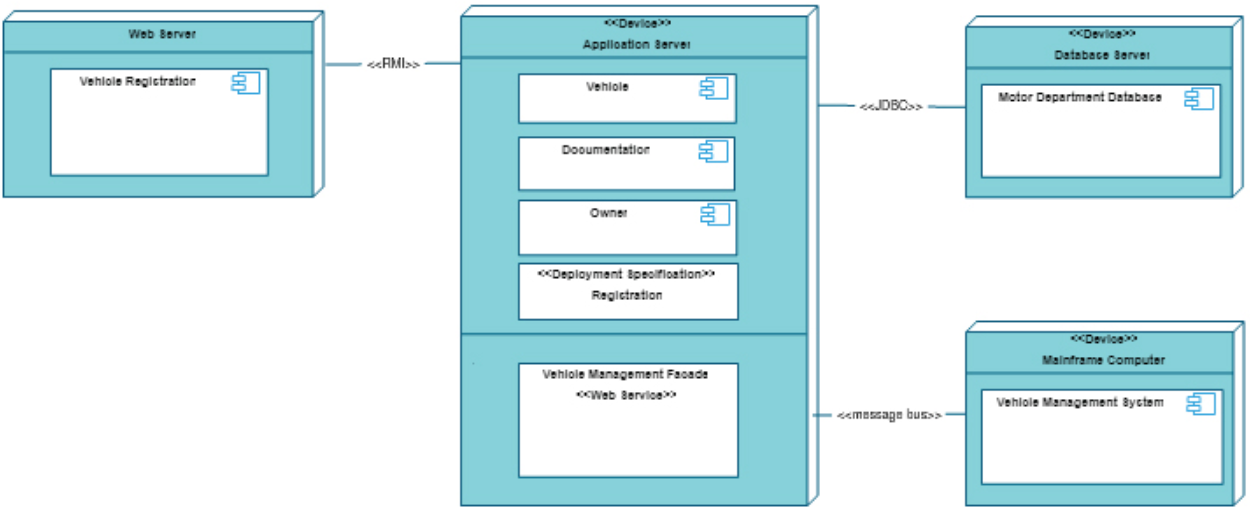


Fonte: https://creately.com/blog/diagrams/uml-diagram-types-examples/

#### 2.5.9 Diagrama de Implantação

Exibe a configuração de elementos de processamento em tempo de execução e os componentes de *software*, processos e objetos que vivem neles. Instâncias de componentes de *software* representam manifestações em tempo de execução de unidades de código. Um exemplo de um diagrama de implantação é ilustrado na Figura 10.

Figura 10 - Exemplo de Diagrama de Implantação



Fonte: <https://creately.com/blog/diagrams/uml-diagram-types-examples/>

## Software de gestão de estoques

O sistema de gerenciamento de estoque é um sistema baseado em computador destinado a verificar intensidades de estoque, requisições (demandas), transações e suprimentos. O software de gerenciamento de estoque é um instrumento utilizado para consolidar informações de estoque que anteriormente eram coletadas em formato de cópia impressa ou planilha. Para manter os registros de contas com competência, os proprietários das empresas devem implementar um excelente sistema de gerenciamento de estoque. O estoque nas empresas constitui uma grande parte do investimento que deve ser organizado em sequência para maximizar a obtenção de ganho financeiro. Os estoques registrados manualmente são pouco confiáveis e incompetentes, exceto os que são bem coordenados (FRANK; LYNCH; REGO, 2009).

Desenvolvedores de *software* utilizam ANGULARJS, PHP, HTML, CSS e banco de dados NoSQL para projetos e implementação de sistemas de gerenciamento de inventário. HTML é usado principalmente para projetar a interface do sistema de inventário e estruturá-la para ser bem definida. CSS também é usado junto com HTML no design do sistema de inventário. ANGULARJS é utilizado para todo o desenvolvimento da aplicação enquanto o PHP é aplicado para o desenvolvimento de todo o backend do sistema para gerenciamento de banco de dados e APIs. O banco de dados NoSQL é o banco de dados usado para armazenar todos os dados que são utilizados durante o desenvolvimento do sistema de inventário (FAYYAD; PIATETSKY-SHAPIRO; SMYTH, 1996).

Agarwal, Lal Yadav e Anand (2013) pesquisaram o algoritmo Apriori que foi desenvolvido em java e era uma associação que conduz o aplicativo de mineração de dados a serem utilizados no *software* de gestão de estoque. Este algoritmo poderia ser aplicado para gerenciar negociações de mercadorias, fornecendo aos comerciantes detalhes sobre a previsão das tendências de vendas do produto, bem como as atitudes do consumidor.

## Processo de software

Um processo de *software* é um conjunto de atividades que levam à produção de um produto de *software*. Essas atividades podem envolver o desenvolvimento de software a partir do zero em uma linguagem padrão de programação como Java ou C, no entanto, aplicações não são necessariamente desenvolvidas dessa forma. Atualmente, novos softwares de negócios são desenvolvidos por meio da extensão e modificação de sistemas existentes ou por meio da configuração e integração de prateleira ou componentes do sistema (SOMMERVILLE, 2011).

Segundo Presman (2011), o processo de software é definido como uma metodologia para as atividades, ações e tarefas necessárias para desenvolver um *software* de alta qualidade. Um processo de *software* define a abordagem adotada conforme um *software* é elaborado pela engenharia.

# Metodologia

A metodologia de pesquisa pode ser definida como um tipo de técnica de pesquisa e design, de coleta e análise de dados para criar evidências que possam apoiar a pesquisa. A metodologia explica as maneiras de estudar um problema e a razão de usar o método e a técnica propostos (BARADA, 2013).

Esta pesquisa se classifica como qualitativa, caracterizada pelo uso de métodos que procuram investigar traços inerentes aos objetos de investigação e que tendem a ser de natureza mais interpretativa (GRIX, 2010). Os pontos fortes da pesquisa qualitativa estão enraizados principalmente em sua abordagem indutiva, em que o pesquisador coleta dados e desenvolve teorias como resultado da análise destes dados, seu foco em fenômenos ou pessoas específicas e sua ênfase em palavras ao invés de números (MAXWELL, 2013).



## Concepção

O início do planejamento do projeto se fará com o levantamento dos requisitos funcionais e não-funcionais através de entrevista com os *stakeholders*, caracterizados, neste caso, pelo proprietário da empresa. Então, será feita uma modelagem dos dados obtidos na entrevista, mapeando as funcionalidades que o *software* deverá apresentar e suas interações com o usuário, utilizando a Linguagem Unificada de Modelagem, do inglês *Unified Modeling Language* (UML).

Optou-se por representar as características do *software* usando a UML, uma notação popular e amplamente suportada (SIAU; CAO, 2001). A UML provou ser muito adequada para esse propósito, uma vez que possui mecanismos de extensibilidade embutidos que permitem que extensões específicas sejam realizadas em certos domínios, caso necessário. Com esses mecanismos, espera-se ser possível representar as funcionalidades requeridas pelos *stakeholders* de maneira visual, com a utilização de diagramas, facilitando a compreensão da proposta do *software* em questão.

## Elaboração

Para elaboração do projeto de *software*, será utilizado o Documento de Modelagem de Software (DMS). O documento em questão oferece um guia passo a passo de como elaborar um projeto de *software*, garantindo que pontos essenciais, como confecção de atas das reuniões com os *stakeholders*, sejam realizadas.

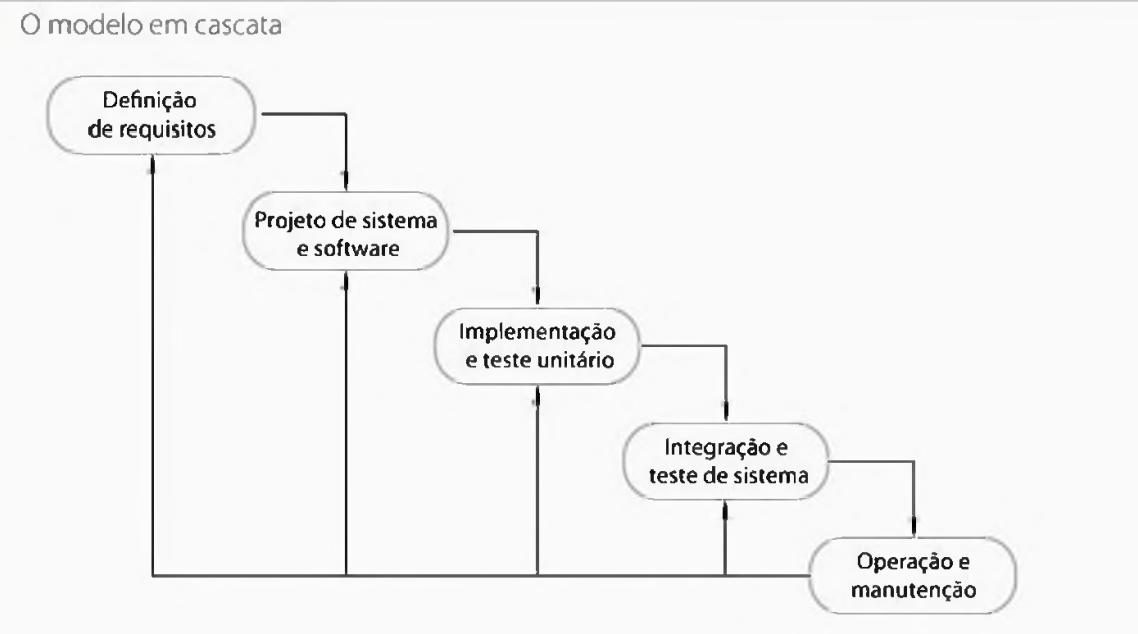
## Modelo Cascata

Se desenvolverá o projeto de *software* baseando-se no modelo cascata. O modelo em cascata é consistente com outros modelos de processos de engenharia, e a documentação é produzida em cada fase do ciclo. Dessa forma, o processo torna-se visível, e os gerentes podem monitorar o progresso de acordo com o plano de desenvolvimento (SOMMERVILLE, 2011).

Em princípio, o modelo cascata deve ser usado apenas quando os requisitos são bem compreendidos e pouco provavelmente venham a ser radicalmente alterados durante o desenvolvimento do sistema. No entanto, o modelo em cascata reflete o tipo de processo usado em outros projetos de engenharia. Como é mais fácil usar um modelo de gerenciamento comum para vários tipos de projeto, processos de software baseados no modelo em cascata ainda são comumente utilizados (SOMMERVILLE, 2011).

Um esquema de modelo em cascata é apresentado na Figura 11. Começa-se com a definição de requisitos. Os serviços, restrições e metas do sistema são estabelecidos por meio de consulta aos usuários. Em seguida, são definidos em detalhes e funcionam como uma especificação do sistema.

Figura 11 - Modelo em cascata



Fonte: Sommerville (2011, p. 20)

A segunda etapa é o projeto do software em si. O processo de projeto de sistemas aloca os requisitos tanto para sistemas de *hardware* como para sistemas de *software*, por meio da definição de uma arquitetura geral do sistema. O projeto de *software* envolve identificação e descrição das abstrações fundamentais do sistema de *software* e seus relacionamentos (SOMMERVILLE, 2011)

Depois do projeto estabelecido, faz-se a implementação e teste do software. Durante esse estágio, o projeto do *software* é desenvolvido como um conjunto de programas ou unidades de programa. O teste unitário envolve a verificação de que cada unidade atenda a sua especificação (SOMMERVILLE, 2011).

Realiza-se então a integração e teste do sistema. As unidades individuais do programa ou programas devem ser integradas e testadas como um sistema completo para assegurar que os requisitos do software tenham sido atendidos. Após o teste, o sistema de *software* é entregue ao cliente (SOMMERVILLE, 2011).

Normalmente, a fase de operação e manutenção é a mais longa do ciclo de vida de um projeto em cascata. O sistema é instalado e colocado em uso. A manutenção envolve a correção de erros que não foram descobertos em estágios iniciais do ciclo de vida, com melhora da implementação das unidades do sistema e ampliação de seus serviços em resposta às descobertas de novos requisitos (SOMMERVILLE, 2011).

# Resultados

## Resultados Obtidos

Até o momento, conseguiu-se compilar informações sobre arquitetura de software, bem como a importância de se realizar um planejamento antes mesmo de se escrever uma linha de código.

## Resultados esperados

Com a elaboração deste trabalho, espera-se apresentar um projeto de software que auxilie os stakeholders nos processos de gestão do negócio, especificamente os processos de vendas e gestão de estoque, aumentando a competitividade e eficiência da empresa. Espera-se que o software seja aplicável em um contexto de pequena e média empresa, apresentando bom custo-benefício e facilidade de uso.

Espera-se que o cliente do *software* possa dar entrada de novos produtos a partir da leitura do código de barras do mesmo. Caso o produto não apresente código de barras originalmente, espera-se que o usuário do *software* possa imprimir uma etiqueta própria, reconhecível pelo programa. Assim, a entrada de produtos seria feita de maneira rápida e as informações armazenadas no banco de dados da empresa. De maneira semelhante, no momento da venda, o cliente utilizaria o código de barras para registrar a saída do produto do estoque. Assim, diminui-se a ocorrência de erros de inventário, garantindo maior assertividade na gestão de estoques por parte do micro e pequeno empresário.

# Referências

AGARWAL, P.; LAL YADAV, M.; ANAND, N. Study on Apriori Algorithm and its Application in Grocery Store. **International Journal of Computer Applications**, [*s. l.*], v. 74, n. 14, 2013. Available at: https://doi.org/10.5120/12950-9882

BARADA, V. Sarah J. Tracy, Qualitative Research Methods: Collecting Evidence, Crafting Analysis, Communicating Impact. **Revija za sociologiju**, [*s. l.*], 2013. Available at: https://doi.org/10.5613/rzs.43.1.6

CRAGG, P.; CALDEIRA, M.; WARD, J. Organizational information systems competences in small and medium-sized enterprises. **Information & Management**, [*s. l.*], v. 48, n. 8, p. 353–363, 2011. Available at: https://doi.org/10.1016/j.im.2011.08.003

DATASEBRAE. **Tecnologia da Informação**. [*S. l.*], 2015. Available at: https://datasebrae.com.br/tecnologias-de-informacao-e-comunicacao/. Acesso em: 23 fev. 2021.

EIKEBROKK, T. R.; OLSEN, D. H. An empirical investigation of competency factors affecting e-business success in European SMEs. **Information & Management**, [*s. l.*], v. 44, n. 4, p. 364–383, 2007. Available at: https://doi.org/10.1016/j.im.2007.02.004

FAYYAD, U.; PIATETSKY-SHAPIRO, G.; SMYTH, P. From data mining to knowledge discovery in databases. **AI Magazine**, [*s. l.*], v. 17, n. 3, 1996.

FRANK, M. M.; LYNCH, L. J.; REGO, S. O. Tax reporting aggressiveness and its relation to aggressive financial reporting. *In*: , 2009. **Accounting Review**. [*S. l.: s. n.*], 2009. Available at: https://doi.org/10.2308/accr.2009.84.2.467

GRADY, B.; RUMBAUGH, J.; JACOBSON, I. **Unified Modeling Language User Guide**. [*S. l.: s. n.*], 2005.

GRIX, J. **The Foundations of Research**. London: Macmillan Education UK, 2010. Available at: https://doi.org/10.1007/978-0-230-36490-5

GUEDES, G. T. A. **UML 2: uma abordagem prática**. São Paulo: Novatec, 2011.

ILIASHENKO, O. Y.; SHIROKOVA, S. V. Application of Database Technology to Improve the Efficiency of Inventory Management for Small Businesses. **WSEAS Transactions on Business and Economics**, [*s. l.*], v. 11, p. 810–818, 2014.

LAGO, P.; AVGERIOU, P.; HILLIARD, R. Guest Editors’ Introduction: Software Architecture: Framing Stakeholders’ Concerns. **IEEE Software**, [*s. l.*], v. 27, n. 6, p. 20–24, 2010. Available at: https://doi.org/10.1109/MS.2010.142. Acesso em: 12 mar. 2021.

MAXWELL, J. A. A Model for qualitative research design. **Qualitative Research Design: An Interactive Approach**, [*s. l.*], 2013.

QUAYLE, M. A study of supply chain management practice in UK industrial SMEs. **Supply Chain Management: An International Journal**, [*s. l.*], v. 8, n. 1, p. 79–86, 2003. Available at: https://doi.org/10.1108/13598540310463387

REICHHELD, F. F. Loyalty-based management. **Harvard Business Review**, [*s. l.*], v. 71, n. 2, p. 64–73, 1993.

SIAU, K.; CAO, Q. Unified Modeling Language. **Journal of Database Management**, [*s. l.*], v. 12, n. 1, p. 26–34, 2001. Available at: https://doi.org/10.4018/jdm.2001010103

SOMMERVILLE, I. **Engenharia de Software**. [*S. l.*]: Pearson Brasil, 2011.