Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Curso de Graduação em Engenharia Elétrica

Polyana Brandão Marcelino

Documentação e Reestruturação de *Software* de Proteção

Campina Grande, Paraíba
Junho de 2016

Polyana Brandão Marcelino

Documentação e Reestruturação de *Software* de Proteção

Trabalho de Conclusão de Curso submetido à Coordenação do Curso de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Orientador:

Professor Benemar Alencar de Souza, D. Sc.

Campina Grande, Paraíba

Junho de 2016

Polyana Brandão Marcelino

Documentação e reestruturação de *software* de proteção

Trabalho de Conclusão de Curso submetido à Coordenação do Curso de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Aprovado em / /

Professor Avaliador
Universidade Federal de Campina Grande
Avaliador

Professor Benemar Alencar de Souza, D. Sc.
Universidade Federal de Campina Grande
Orientador, UFCG

Campina Grande, Paraíba

Junho de 2016

Agradecimentos

Agradeço:

A Deus, pelas oportunidades proporcionadas durante a vida acadêmica.

Aos meus pais Paulo e Rachel e à minha irmã, pela confiança, amor e incentivos constantes para a superação dos obstáculos.

Ao professor Benemar, pela orientação, oportunidade e confiança na realização dessa atividade.

À professora Núbia, à professora Fátima, a Flávio e a Bianca, pelos auxílios prestados ao longo da realização deste trabalho.

Aos colegas do Laboratório de Sistemas de Potência, pelo companheirismo e colaboração.

Aos amigos feitos ao longo desses anos de curso, pela amizade, compreensão e paciência.

E aos professores, funcionários e colegas de curso, pela contribuição em minha formação acadêmica.

Resumo

Este trabalho de conclusão de curso abrange estudo de documentação e reestruturação de *software*, bem como a aplicação desses processos em aplicativo de proteção desenvolvido pela Companhia Hidro Elétrica do São Francisco (CHESF) em projeto de Pesquisa e Desenvolvimento (P&D) em parceria com a Universidade Federal de Campina Grande (UFCG) e pesquisadores de demais instituições brasileiras. A importância deste estudo verifica-se com o papel que o *software* passou a desempenhar e com as crescentes exigências de confiabilidade, economia e desempenho ao longo da evolução computacional. Diante desse contexto, a necessidade de documentar e prover suporte ao reuso do *software* ADDEP, a fim de organizar o sistema permitindo seu entendimento em termos de componentes e inter-relacionamentos, motivou este trabalho.

Sumário

Lista de Figuras

1. Introdução

Há aproximadamente quatro décadas, *software* constituía pequena parcela dos sistemas computacionais quando comparado ao *hardware*, tendo custo de desenvolvimento e manutenção desprezível. Atualmente o *software* é responsável por significativa porção dos sistemas computacionais, encontrado desde aplicações para uso pessoal, tais como editores de texto e planilhas eletrônicas, à sistemas de grande porte, como por exemplo os utilizados no controle e supervisão dos sistemas de geração/distribuição de energia [1].

Ao longo das últimas décadas, o *software* tornou-se maior e mais complexo, aumentando-se a exigência de confiabilidade, economia e desempenho por parte do mercado. E, assim como os demais sistemas, o *software* está sujeito a mudanças regulares, devendo ser codificado e documentado de forma a atenuar os custos envolvidos nesse processo.

O processo de desenvolvimento de um sistema de *software* vai desde a concepção do sistema, quando requisitos são elicitados e analisados, até sua concreta implementação. Na fase inicial, há interesse em compreender a funcionalidade que o sistema deverá prover sem que haja necessidade de se tomar qualquer decisão a nível estrutural ou arquitetural. Após essa etapa, a funcionalidade do sistema é particionada a fim de identificar possíveis subsistemas ou módulos, visando encontrar ou apresentar uma arquitetura que satisfaça às necessidades requeridas. Na fase de arquitetura, define-se a divisão de funções entre subsistemas ou módulos, bem como os mecanismos de interação entre eles e a representação da informação compartilhada [1].

Diante da necessidade de documentar, refatorar e reestruturar o *software* ADDEP, desenvolvido pela Companhia Hidro Elétrica do São Francisco (CHESF), em projeto de Pesquisa e Desenvolvimento (P&D) em parceria com a Universidade Federal de Campina Grande (UFCG) e pesquisadores de outras instituições brasileiras, realizou-se os processos de documentação, e reestruturação desse *software*.

ADDEP é um programa de aplicação que realiza o diagnóstico de distúrbios e avaliação do desempenho de funções de proteção. Ele tem o objetivo de otimizar a elaboração dos Relatórios de Análise de Desempenho da Proteção (RADP), que devem ser elaborados pelas Concessionárias de Energia Elétrica e enviados ao Operador Nacional do Sistema Elétrico (ONS) após a ocorrência de distúrbios.

Este trabalho apresenta os procedimentos envolvidos nesse processo organizados em seis capítulos, incluindo esta introdução (Capítulo 1). O Capítulo 2 apresenta o *software* ADDEP, programa no qual a reestruturação e a documentação foram aplicadas. O Capítulo 3 contém a fundamentação teórica relativa às atividades realizadas. O Capítulo 4 aborda sobre a documentação e a reestruturação realizada no *software* ADDEP. No Capítulo 5 são feitas as Considerações Finais.

2. Apresentação do *Software* ADDEP

Os procedimentos e requisitos necessários para as atividades de planejamento da operação eletroenergética e administração da transmissão no Sistema Interligado Nacional (SIN) são estabelecidos pelos Procedimentos de Rede, que consistem em documentos normativos elaborados pelo Operador Nacional do Sistema Elétrico (ONS), com participação dos agentes, e aprovados pela Agência Nacional de Energia Elétrica (ANEEL) [2]. Dentre os Procedimentos de Rede, destaca-se o submódulo de Avaliação de Desempenho dos Sistemas de Proteção, cujo objetivo é estabelecer diretrizes para atividades como: coleta de dados, avaliação do desempenho dos sistemas de proteção e recomendação de ações corretivas e gerenciais.

Uma das etapas desse módulo consiste na elaboração dos Relatórios de Análise de Desempenho da Proteção (RADP), os quais são enviados ao ONS pelas concessionárias após a ocorrência de cada distúrbio. Durante a elaboração dos RADP, consideram-se informações a respeito das características do distúrbio, de suas possíveis causas e também sobre possíveis operações indevidas dos dispositivos de proteção.

Esse diagnóstico é tipicamente realizado com base em informações fornecidas pelas equipes de operação e manutenção, bem como na análise de oscilografias obtidas de relés numéricos de proteção e Registradores Digitais de Perturbações (RDP). Entretanto, a depender da quantidade e qualidade das informações iniciais sobre o defeito, a elaboração dos RADP pode se tornar complicada e demorada, o que vai de encontro aos interesses das concessionárias.

Os valores das grandezas e os sinais registrados durante o evento de falta são armazenados em arquivos de dados para posterior análise. Atualmente estes arquivos codificados de acordo com o padrão COMTRADE, formato padronizado pelo Instituto de Engenheiros Eletricistas e Eletrônicos (IEEE) para oscilografia digital. O formato COMTRADE define três tipos de arquivos: cabeçalho, configuração e dados [3].

Arquivos de Cabeçalho armazenam informações criadas pelo originador dos dados. Arquivos de Configuração possuem terminação ".cfg" e contêm informações que permitem interpretar e associar os dados com os valores armazenados no arquivo de dados. Arquivos de Dados possuem terminação ".dat" e contêm valores das amostras dos canais monitorados.

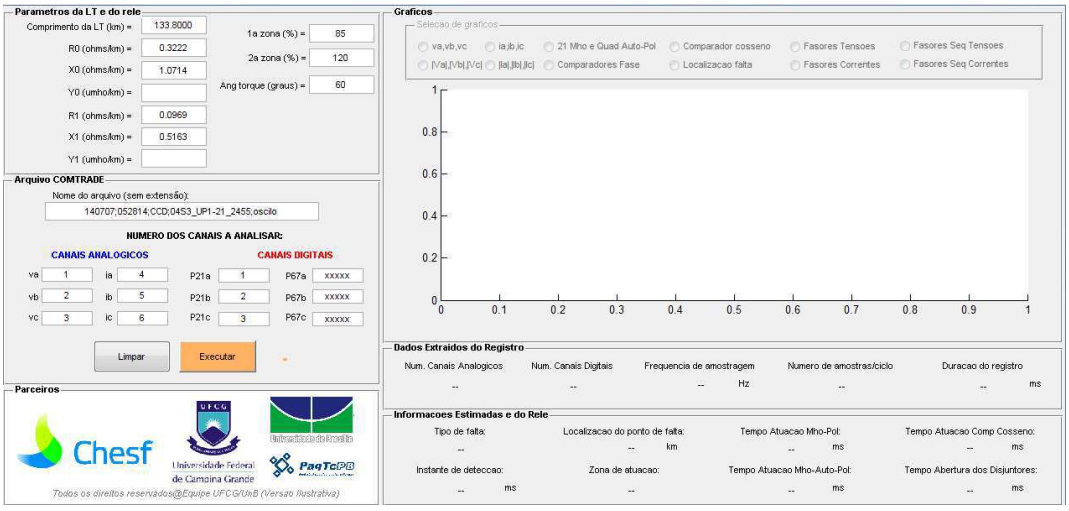
Com objetivo de auxiliar o especialista durante a elaboração do RADP, a CHESF iniciou, em 2013, um projeto de Pesquisa e Desenvolvimento (P&D) em parceria com a Universidade Federal de Campina Grande (UFCG) e pesquisadores de outras instituições brasileiras visando desenvolver um programa de aplicação para diagnóstico de distúrbios e avaliação do desempenho de funções de proteção, o qual foi denominado de ADDEP acrônimo de *Análise de Distúrbios e do Desempenho*

da Proteção.

O ADDEP contém funções para leitura de arquivos COMTRADE dos tipos configuração e dados, estimação fasorial, detecção, classificação e localização de faltas, rotinas para estimação do tempo de abertura dos disjuntores e funções de proteção de distância (elementos mho e quadrilateral) [4].

No decorrer deste trabalho, utilizou-se a versão V0 do ADDEP, implementada no *software* Matlab e referente ao ano de 2015. Essa versão apresenta apenas uma interface gráfica, exibida na Figura 1, na qual o usuário insere os dados requeridos e onde são exibidos os dados de saída e os resultados gráficos. Na parte esquerda dessa interface inserem-se os parâmetros da linha de transmissão, da rede e dados referentes aos arquivos Comtrade. No lado direito da tela de interface, são exibidos os dados extraídos do registro, as informações estimadas e do relé, e os resultados gráficos do sistema.

Figura 1: Interface gráfica do *software* ADDEP.



3. Fundamentação Teórica

Um programa computacional visa solucionar um problema que define ou está contido em domínios sujeitos às leis da física, da matemática, do direito, do mercado financeiro, etc [5]. Em seu desenvolvimento, busca-se construir modelos que coloquem em termos de algoritmos o domínio da aplicação. Uma tecnologia utilizada nesse processo é a Orientação a Objetos. Modelos orientados a objetos são implementados por meio de linguagem de programação orientada a objetos.

Sistemas orientados a objetos, quando projetados corretamente, são flexíveis a mudanças, possuem estruturas bem conhecidas e proporcionam a implementação de componentes reutilizáveis. Métodos de desenvolvimento de *software* anteriores ao surgimento do paradigma Orientação a Objetos organizavam a especificação do sistema de acordo com suas funções ou com dados manipulados. Esses métodos costumavam apresentar dificuldades na transição entre etapas no processo de desenvolvimento [5].

A Metodologia de Desenvolvimento de *Software* (MDS) pode basear-se em diversos modelos e fazer uso de diferentes processos. Ciclo de vida, cascata e prototipação são exemplos de modelos de MDS. Análise Estruturada, Orientação a Objetos, Orientação a Eventos e *Agile* são exemplos de processos de MDS. MDS envolve etapas de comunicação, planejamento, modelagem, construção e implantação. Ao longo do processo de desenvolvimento, deve-se adotar práticas como código padronizado, rotinas de testes, refatoração, comunicação eficiente.

3.1. **Conceitos**

3.1.1. *Software*

Software é comumente definido como um agrupamento de comandos escritos em uma linguagem de programação para criar ações dentro do programa e permitir seu funcionamento. No entanto, o termo *software* não se restringe apenas aos programas de computadores associados com uma aplicação, mas também envolve toda a documentação necessária para instalação, uso, desenvolvimento e manutenção dos programas [1].

3.1.2. Método

Elemento que descreve comportamento. Seu nome deve remeter a uma ação e ter inicial minúscula. O método possui a seguinte estrutura:

<modificador> <tipo de retorno> <nome> (<lista de argumentos>) {<bloco>}

Modificadores definem o tipo de acesso ao método, isto é, tipo de encapsulamento. Eles podem ser *public*, *protected* ou *private*. *Public* permite classes de qualquer escopo acessar o método. *Protected* permite classes do mesmo escopo acessarem o método. *Private* permite apenas a própria classe acessar o método. O conceito de modificador se estende a atributos [5]. Tipo de retorno indica o tipo de variável retornada pelo método. O nome identifica o método. A lista de argumentos informa os tipos de valores que deverão ser passados ao método. Bloco contém a implementação do método.

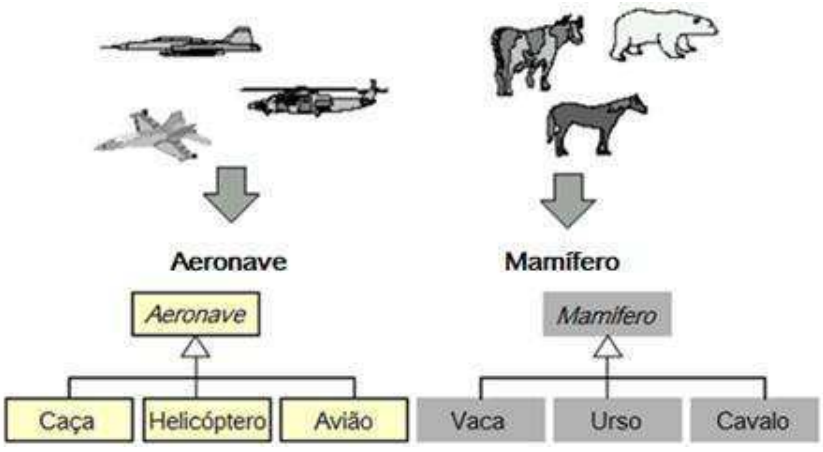
3.1.3. Classe

Elemento que contém atributos e métodos que descrevem um conjunto de objetos com interfaces semelhantes [5]. Costuma ser nomeada por substantivo e ter inicial maiúscula e pode ser concreta ou abstrata. Classes trabalham em conjunto com outras classes, sendo a comunicação entre elas é denominada relacionamento.

Classe concreta contém a assinatura e implementação de todos os métodos. Classe abstrata é aquela cuja finalidade principal é definir uma interface comum para suas subclasses [6]. Ela posterga parte de, ou toda, sua implementação para operações definidas nas subclasses. Portanto, uma classe abstrata deve possuir pelo menos um método abstrato e não pode ser instanciada. Nomes de classes abstratas devem ter notação em itálico.

Métodos abstratos são aqueles que apenas determinam a existência de um comportamento, não definindo uma implementação [6]. O tipo em itálico também é usado para denotar métodos abstratos. Uma classe que possui apenas métodos abstratos é denominada classe virtual pura.

Figura 2: Exemplo de abstração expressa em classes.



Fonte: Adaptado de [5].

Classes abstratas capturam as propriedades e comportamentos essenciais dos objetos. A Figura 2 exibe as classes abstratas “Aeronave” e “Mamífero”. A primeira contém a essência das classes “Caça”, “Helicóptero”, “Avião”; a segunda contém a essência das classes “Vaca”, “Urso”, “Cavalo”. As classes “Aeronave” e “Mamífero” são classificadas de Classe Base, Classe Mãe ou SuperClasse. As demais classes são classificadas de Classe Derivada, Classe Filha ou SubClasse. Elas se relacionam por meio de Herança.

3.1.4. Herança

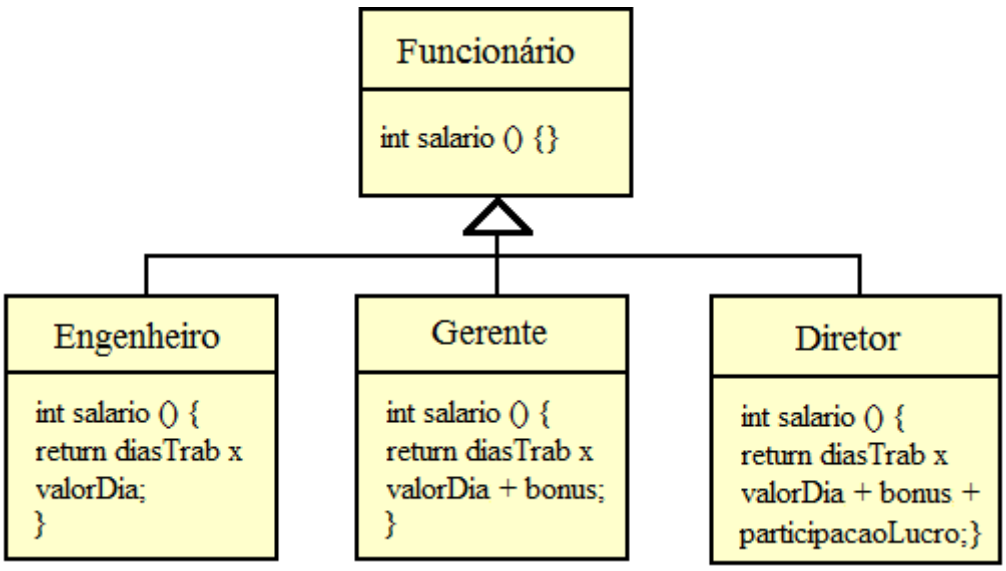
Processo pelo qual classes derivadas incorporam as propriedades e métodos de suas classes bases respectivas. Classes derivadas podem adicionar novas propriedades e métodos, e redefinir a implementação dos métodos herdados.

3.1.5. Polimorfismo

Mecanismo pelo qual um método assume diferentes interfaces para realizar uma mesma implementação, ou assume diferentes implementações para uma mesma interface. O primeiro caso é denominado *Overloading* e o segundo *Overriding*.

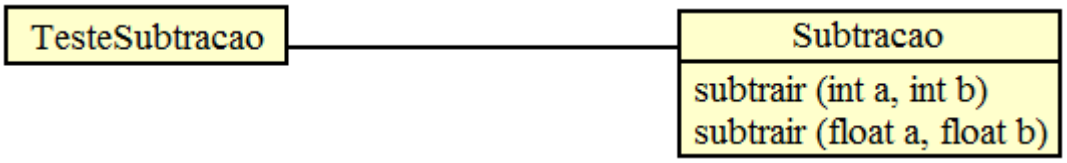
A Figura 3 contém exemplo de *Overriding*. A interface do método “salário” é a mesma em todas as classes, mas a sua implementação é diferente. A Figura 4 contém exemplo de *Overloading*. A implementação do método “subtrair” é a mesma, o nome do método é mantido, mas os tipos dos argumentos são distintos, o que caracteriza mudança na interface.

Figura 3: Exemplo de Polimorfismo por meio de *Overriding*.



Fonte: Adaptado de [5].

Figura 4: Exemplo de Polimorfismo por meio de *Overloading*.



Fonte: Adaptado de [5].

3.2. **Tipos de *Software***

Software pode ser classificado de Programas de Aplicação, *Toolkits* e *Frameworks* [6].

Programas de Aplicação constituem aplicativos para a solução de problemas específicos. Editores de documentos e planilhas são exemplos de programas de aplicação. Nesse tipo de *software*, as prioridades são reusabilidade interna e facilidade de manutenção e extensão. Frequentemente Programas de Aplicação incorporam classes de uma ou mais bibliotecas de classes pré-definidas. Essas são denominadas *Toolkit*s ou Bibliotecas de Classes.

Toolkits fornecem funcionalidades para auxiliar a implementação de um Programa de Aplicação. Dessa forma, evita-se que o implementador recodifique funcionalidades comuns. Eles constituem conjunto de classes projetado para fornecer funcionalidades específicas de uso genérico. Dessa forma, no desenvolvimento de *Toolkits* prioriza-se a reusabilidade. Uma biblioteca matemática é um exemplo de *Toolkit*. Ela deve conter funcionalidades específicas, realizar operações matemáticas, e ser implementada de forma que possa ser empregada em diferentes Programas de Aplicação.

Framework, também denominado Arcabouço de Classes, constitui um conjunto de classes cooperantes que constroem um projeto reutilizável para uma determinada categoria de *software*. *Frameworks* são orientados à criação de Programas de Aplicação de natureza específica, como por exemplo, editores gráficos. A aplicação é customizada por meio da criação de subclasses específicas, derivadas das classes abstratas definidas pelo *Framework*.

Frameworks determinam a arquitetura da aplicação. Eles definem a estrutura geral do Programa de Aplicação, sua divisão em classes e objetos, os relacionamentos dos componentes e o fluxo de controle. Dessa forma, o projetista/implementador concentra-se apenas nos aspectos específicos da aplicação [6]. *Framework* deve poder ser aplicado em diversos Programas de Aplicação no domínio para o qual foi projetado. Dessa forma, suas prioridades são flexibilidade e extensibilidade.

O ADDEP, objeto dos processos de documentação e reestruturação apresentados neste trabalho, caracteriza-se como Programa de Aplicação. Isso ocorre por esse

software ter por objetivo solucionar um problema específico na área de proteção, otimizar a elaboração dos Relatórios de Análise de Desempenho da Proteção (RADP).

3.3. **Análise de *Software***

A análise de *software* orientado a objetos possibilita muitas abordagens diferentes. Pode-se escrever a descrição do problema e separar os substantivos e verbos para criar as classes e operações correspondentes; concentrar-se sobre as colaborações e responsabilidades do sistema; ou ainda, modelar o mundo real e, na fase de projeto, traduzir os objetos encontrados durante a análise. As duas técnicas mais comuns para a reutilização de funcionalidade em sistemas orientados a objetos são Composição de Objetos e Herança de Classe [6].

Na Composição de Objetos, funcionalidades complexas são obtidas pela montagem e/ou composição de objetos. Ela requer que os objetos que estão sendo compostos tenham interfaces bem definidas. Como os detalhes internos dos objetos não são visíveis aos demais, não há violação do princípio de encapsulamento. Devido a isso, essa técnica é chamada "reutilização de caixa preta". O encapsulamento é importante, em termos de segurança, para proteger os atributos dos objetos de terem seus valores corrompidos; em termos de independência, para proteger outros objetos de complicações por dependência de sua estrutura interna.

Composição de Objetos é definida dinamicamente em tempo de execução pela obtenção de referências a outros objetos. Ela requer que os objetos respeitem as interfaces dos demais, o que exige interfaces cuidadosamente projetadas. Como os objetos são acessados exclusivamente por meio de suas interfaces, não há violação do princípio do encapsulamento. Objetos podem ser substituídos por outros de mesma interface em tempo de execução. Vantagens e desvantagens da Composição de Objetos são exibidas na Figura 5 [6].

Figura 5: Principais vantagens e desvantagens da Composição de Objetos.

Herança de Classe permite definir classes pela extensão de outras. Constitui um mecanismo para compartilhamento de código e de representação [6]. Reutilização por meio de herança é também denominada “reutilização de caixa branca”, pois permite que atributos e métodos da classe base sejam acessados pelas classes derivadas.

Devido às classes derivadas herdarem a implementação dos métodos da classe base, há limitação da flexibilidade e reusabilidade do código. Uma solução para essa questão é a classe base ser abstrata, de preferência virtual pura [6].

Herança de Classe é definida em tempo de compilação e é simples de ser utilizada. Ela facilita modificar a implementação durante o reuso. No entanto, não é possível mudar implementações herdadas das classes ancestrais em tempo de execução. As principais vantagens e desvantagens da Herança de Classe são exibidas na Figura 6 [6].

Figura 6: Principais vantagens e desvantagens da Herança de Classe.

A reutilização por Herança de Classe torna mais fácil criar novos componentes que podem ser obtidos pela Composição de outros já existentes. Assim essas duas técnicas, Herança de Classe e Composição de Objetos, trabalham em conjunto na implementação de novas funcionalidades [6].

Delegação é um mecanismo que torna Composição de Objetos tão eficaz para fins de reutilização quanto a Herança de Classes [6]. Na Delegação, dois objetos são envolvidos no tratamento de uma solicitação, e o objeto receptor delega operações para o objeto delegado. Esse processo é análogo, na Herança de Classe, à postergação de solicitações enviadas às classes derivadas pela classe base por meio da variável membro *this*.

A Delegação apresenta a desvantagem das técnicas que tornam o *software* mais flexível por meio da Composição de Objetos: o *software* dinâmico, por ser altamente parametrizado, é mais difícil de ser compreendido do que o *software* estático baseado em Herança de Classe. A aplicação da Delegação é aconselhável de acordo com as condições do contexto e com a experiência do programador com seu uso. Recomenda-se utilizá-la em formas altamente estilizadas, isto é, por meio de Padrões de Projeto catalogados [6].

Outra técnica para reutilização de funcionalidade, não estritamente orientada a objetos, são os Tipos Parametrizados, conhecidos em C++ como *Templates*. Nessa técnica define-se um tipo genérico, que suporta todos os demais. Os tipos não especificados são fornecidos como parâmetros no ponto de utilização. Tipos Parametrizados constituem uma terceira maneira de efetuar a reutilização de comportamentos em sistemas orientados a objetos (além da Herança de Classe e da Composição de Objetos).

Herança de Classe, Composição de Objetos e Tipos Parametrizados podem ser utilizadas separadamente ou em conjunto. A decisão de qual dessas técnicas empregar deve levar em consideração a natureza do *software*, as restrições de implementação e a experiência do programador.

As técnicas de reutilização de funcionalidades empregadas para a reestruturação do *software* ADDEP foram Herança de Classe e Composição de Objetos.

A Herança de Classe foi utilizada por meio de um padrão de projeto catalogado, padrão *Strategy*, no processo de leitura de arquivo Comtrade. Essa técnica foi aplicada uma vez que classes relacionadas diferiam apenas no comportamento, necessitando-se da variação de um mesmo algoritmo.

A Composição de Objetos foi a técnica predominante adotada no processo de reestruturação. Isso ocorreu devido à forma de análise realizada do *software*, na qual se modelou o mundo real e, na fase de projeto, traduziram-se os objetos encontrados.

O mecanismo de delegação não foi empregado, visto ser desnecessário o envolvimento de mais de um objeto no tratamento das solicitações no *software* ADDEP.

Tipos Parametrizados não foram utilizados, visto ser desnecessária a definição dos tipos dos parâmetros em tempo de execução no *software* ADDEP.

3.4. **Ciclo de Vida de *Software***

O ciclo de vida de um *software* orientado a objetos (Figura 7) possui várias fases: prototipação, expansão e consolidação [6].

Figura 7: Ciclo de vida de um *software*.

Fonte:[6].

Prototipação é a fase inicial de um *software*. Nela ocorrem as primeiras implementações, com a finalidade de atender um conjunto inicial de requisitos. O *software* reflete as entidades no domínio do problema inicial e é colocado em funcionamento. Uma vez que isso ocorre, sua evolução é governada por duas necessidades conflitantes: (1) o *software* deve satisfazer mais requisitos, e (2) o *software* deve ser mais reutilizável.

Comumente novos requisitos acrescentam classes e operações. Ao passar pelo processo de expansão, o *software* eventualmente tornar-se inflexível e rígido para permitir mudanças. Para continuar a evoluir, o *software* deve passar pelo processo de refatoração [6].

Na fase de consolidação novos objetos são produzidos, frequentemente pela decomposição de objetos existentes. A necessidade contínua de satisfazer novos requisitos e possibilitar reutilização faz o *software* passar por repetidas fases de expansão e consolidação, tornando-se mais genérico ao longo desse processo.

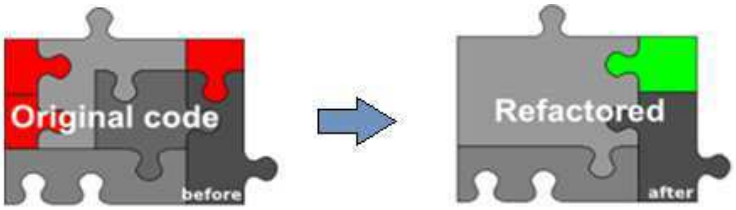
Algumas estruturas de classes e objetos proporcionam mais robustez ao sistema frente às mudanças de requisitos, diminuindo a necessidade de refatorações futuras [6]. Sistemas bem projetados facilitam a identificação de mudanças necessárias e mitigam riscos de erros durante esse processo.

3.5. **Refatoração**

Refatoração é o processo de otimização do *software*, melhorando sua estrutura interna, sem modificar sua funcionalidade para o usuário [7]. Recomenda-se aplicar esse processo quando deseja-se expandir um *software*, facilitar sua compreensão ou identificar e solucionar *bugs*.

A Figura 8 faz uma analogia a esse processo por meio de peças de quebra-cabeça. Observe que a funcionalidade da peça vista externamente não foi alterada, no entanto, realizaram-se mudanças estruturais internas.

Figura 8: Analogia ao processo de Refatoração.



Fonte: [8].

O processo de refatoração proporciona as seguintes vantagens [7]:

- Melhora o *design* do *software*;
- Facilita o entendimento do código;
- Facilita a detecção de *bugs*;
- Otimiza o processo de desenvolvimento do código.

3.5.1. Etapas do Processo de Refatoração

As quatro etapas principais aplicadas no processo de refatoração consistem em:

- Construir um conjunto sólido de testes para cada seção de código. Testes são essenciais para detectar erros ocorridos durante a refatoração. O programa deve ser refatorado em pequenas etapas para facilitar a detecção de erros, caso eles ocorram.
- Decompor métodos longos. Peças menores de código facilitam o manuseio, sendo mais fácil utilizá-las e reutilizá-las. Portanto, dessa forma, evita-se duplicação de código, otimizando a reutilização.
- Remover variáveis temporárias, pois são úteis apenas em suas próprias rotinas. Dessa forma, substitui-se essas variáveis por métodos de pergunta, pois esses são acessíveis aos demais membros da classe, proporcionando um design limpo sem métodos complexos e longos.
- Substituir operações com múltiplos comandos condicionais por estrutura de classe baseada em polimorfismo. Padrões de Projeto podem ser aplicados nessa situação.

Devido ao escopo deste trabalho envolver documentação e reestruturação do *software*, e não envolver codificação, o processo de refatoração restringiu-se a decompor segmentos de código extensos em módulos menores e em substituir múltiplos comandos condicionais por estrutura de classe baseada em polimorfismo por meio de padrões de projeto.

Esses procedimentos foram realizados durante a reestruturação do *software* e podem ser vistos por meio da documentação elaborada.

3.6. **Padrões de Projeto**

A chave para a reutilização consiste em antecipar novos requisitos e em projetar sistemas de modo a otimizar sua evolução. No projeto de um sistema robusto, devem-se analisar possíveis necessidades de mudança ao longo da vida do *software*. Um projeto que desconsidera essa possibilidade está sujeito ao risco de grande reformulação. Mudanças podem envolver redefinições e reimplementações de classes, modificação de requisitos e retestagem do sistema. Modificações não planejadas devem ser evitadas, pois geram custos adicionais ao projeto.

Padrões de Projeto otimizam processos ao garantirem que mudanças ocorram segundo maneiras específicas. Cada padrão permite algum aspecto da estrutura do sistema variar independentemente de outros, tornando-o mais robusto em relação a um tipo particular de mudança [6].

Há três tipos de Padrões de Projeto: Padrões de Criação, Padrões Estruturais e Padrões Comportamentais.

Padrões de Criação abstraem o processo de instanciação. Eles ajudam sistemas tornarem-se independentes do modo como seus objetos são criados, compostos e representados [6]. Padrão de criação de classe usa herança para variar a classe instanciada, enquanto que Padrão de Criação de objeto delega a instanciação para outro objeto.

Padrões Estruturais atuam no modo como classes e objetos são compostos para formar estruturas maiores. Padrões Estruturais de classes utilizam herança para compor interfaces ou implementações. Padrões Estruturais de objetos descrevem maneiras de compor objetos para obter novas funcionalidades.

Padrões Comportamentais atuam nos algoritmos e atribuições de responsabilidade entre classes e objetos, e descrevem a comunicação entre eles. Padrões Comportamentais de classe utilizam herança para distribuir o comportamento entre classes. Padrões Comportamentais de objetos utilizam composição de objetos em vez de herança.

O algoritmo proposto por [6] para selecionar no catálogo de padrões os adequados ao sistema a ser desenvolvido é expresso a seguir:

- Considere como Padrões de Projeto solucionam problemas de projeto;
- Examine a intenção dos padrões;
- Estude como os padrões se inter-relacionam;
- Estude padrões de finalidades semelhantes;

- Examine possíveis causas de reformulação do projeto;
- Considere o que deveria ser modificado futuramente em seu projeto.

Ao selecionar os padrão adequados, a metodologia proposta por [6] para aplicá-los é expressa a seguir:

- Leia o padrão por inteiro para obter uma visão geral;
- Estude sua estrutura, seus participantes e suas colaborações;
- Analise um exemplo do padrão codificado;
- Escolha nomes significativos para os participantes do padrão no contexto da aplicação;
- Defina as classes e operações participantes.

Padrões de Projeto facilitam a reutilização de arquiteturas bem sucedidas; otimizam projeto, documentação e manutenção do sistema; reduzem a quantidade de refatorações futuras. Eles constituem uma ferramenta importante nos projetos orientados a objetos, no entanto devem ser utilizados com cautela, após se observar as conseqüências e avaliar o custo-benefício do uso.

No processo de reestruturação do *software* ADDEP, adotou-se o padrão de projeto *Strategy* para eliminar comandos condicionais que continham variações do algoritmo de leitura do arquivo Comtrade ".dat".

3.7. *Unified Modeling Language* (UML)

Até o ano de 1996, não havia método padrão para a modelagem de sistemas. Três metodologias principais eram utilizadas pelos desenvolvedores da época: *Object Modeling Technique* (OMT), *Object Oriented Software Engineering* (OOSE) e o Método de Booch. Visando unificar essas metodologias e definir um padrão, em 1996 foi desenvolvida a primeira versão da UML, que atualmente encontra-se na versão 2.5 e constitui a linguagem padrão de modelagem de *software*.

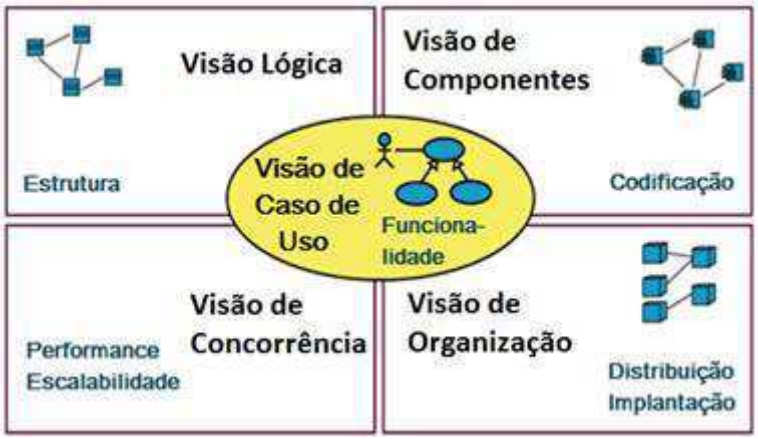
UML (*Unified Modeling Language*) é uma linguagem mantida pelo *Object Management Group* que visa modelar sistemas por meio de diagramas, em termos do paradigma de orientação a objeto. Modelos UML especificam a estrutura e o comportamento do sistema, servem como mapa para sua construção e documentação, e atuam no gerenciamento de versões [5].

3.7.1. Visões UML

Um sistema é composto por diversos aspectos: funcional (que é sua estrutura estática e suas interações dinâmicas), não funcional (requisitos de tempo, confiabilidade, desenvolvimento, etc.) e aspectos organizacionais (organização do trabalho, mapeamento dos módulos de código, etc.). Então o sistema é descrito em

um certo número de visões, cada uma representando uma projeção da descrição completa e mostrando aspectos particulares do sistema. Cada visão é descrita por um número de diagramas que contém informações que dão ênfase aos aspectos particulares do sistema. Existe, em alguns casos, a sobreposição entre os diagramas, o que significa que um mesmo diagrama pode fazer parte de mais de uma visão [9]. As visões que compõem um sistema são: Visão de Componentes, Visão de Casos de Uso, Visão Lógica, Visão de Organização e Visão de Concorrência. A Figura 9 apresenta esse esquema de visões.

Figura 9: Esquema de Visões.



Fonte: Adaptado de [5].

Visão de Casos de Uso: descreve o comportamento do sistema sob o ponto de vista do usuário final, analista e equipe de teste. Os aspectos estáticos dessa visão são representados pelo diagrama de casos de uso; os aspectos dinâmicos pelos diagramas de interação, máquina de estados e atividades.

Visão Lógica: também denominada Visão de Projeto, descreve as classes e colaborações que compõem o vocabulário do sistema. Ela proporciona suporte aos requisitos funcionais que o sistema deverá fornecer aos usuários finais. Os aspectos estáticos dessa visão são representados pelos diagramas de classes e objetos; os aspectos dinâmicos pelos diagramas de máquina de estados, sequência, comunicação e atividades.

Visão de Componentes: descreve a implementação dos módulos e dependências do sistema; gerencia a configuração das versões do sistema. Os aspectos estáticos dessa visão são representados pelo diagrama de componentes.

Visão de Concorrência: descreve os processos, comunicação e concorrência das linhas de execução de processos paralelos (*threads*) que compõem os mecanismos de concorrência e de sincronização do sistema. Ela engloba questões relativas ao desempenho e escalabilidade. Escalabilidade de *software* diz respeito à capacidade do sistema assimilar uma carga crescente de conexões ou processos. Os aspectos estáticos dessa visão são representados pelos diagramas de componentes e de

implantação; os aspectos dinâmicos pelos diagramas de máquina de estados, sequência, comunicação e de atividades.

Visão de Organização: descreve a organização física do sistema, isto é, a topologia do *hardware* em que o sistema é executado. Os aspectos estáticos dessa visão são representados pelos diagramas de implantação; os aspectos dinâmicos pelos diagramas de máquina de estado, de interação e de atividades.

3.7.2. Diagramas UML

Para se obter visões de um sistema, ou parte dele, sob diferentes perspectivas, utilizam-se diagramas. Eles constituem representações gráficas de um conjunto de elementos e definem características, dinâmica, comportamento e estrutura lógica do processo de *software*. A Figura 10 exibe diagramas UML classificados de estruturais e comportamentais. Diagramas estruturais são estáticos e tem por objetivo modelar a estrutura do sistema. Diagramas comportamentais são dinâmicos e visam modelar o comportamento do sistema.

Figura 10: Diagramas UML classificados de Estruturais e Comportamentais.

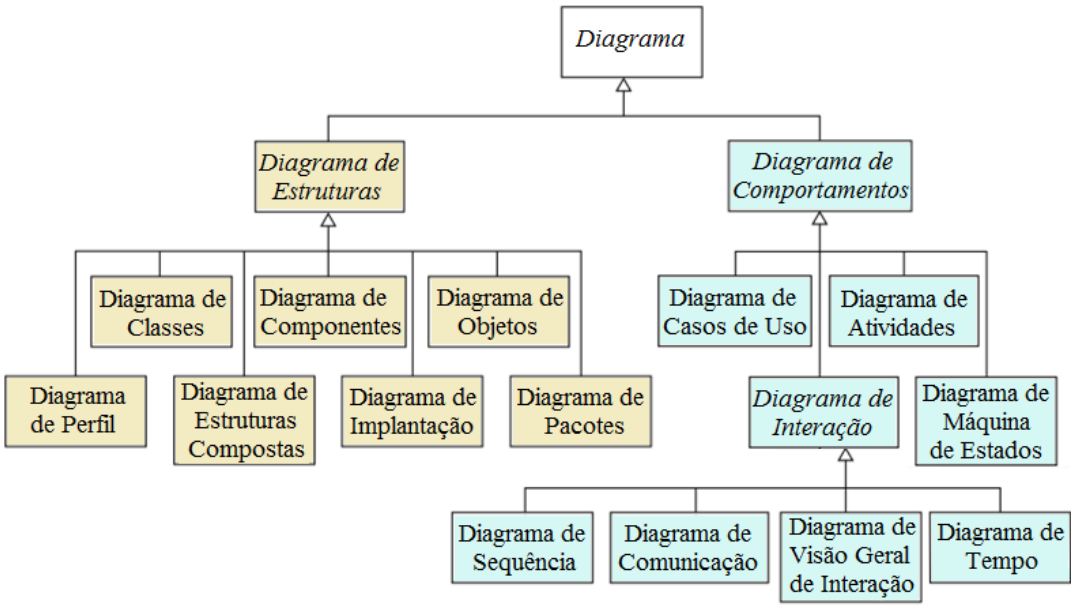


Diagrama de Casos de Uso: diagrama comportamental que descreve requisitos funcionais do sistema em termos de atores, casos de uso, relacionamentos e fronteira. Ele descreve o que o sistema faz sem especificar como deve ser feito.

Atores representam papéis desempenhados por elementos externos ao sistema que interagem com ele. Usuário, *software* e *hardware* são exemplos de atores. Eles são modelados graficamente por meio do boneco palito. Casos de uso representam funcionalidades do sistema. São representados graficamente por elipses, e seus nomes devem ser iniciados por verbos. Fronteira do sistema é um elemento

opcional utilizado para definir o escopo do diagrama. Sua notação é feita por meio de um retângulo circunscrito aos casos de uso, mas sem englobar os atores.

Elementos do Diagrama de Casos de Uso interagem por meio de relacionamentos. Esses se classificam de associação; generalização; dependência, que pode ser por extensão (*extend*) ou inclusão (*include*). Relacionamento do tipo associação conecta atores a casos de uso. Relacionamento do tipo generalização refere-se ao elemento filho herdar comportamento e significado do elemento pai, válido para atores e casos de uso. O elemento filho pode sobrescrever comportamentos do pai e substituí-lo em qualquer lugar em que apareça. Recomenda-se localizar graficamente o elemento filho abaixo do elemento pai.

Relacionamento de dependência "*extend*" representa uma variação/extensão do comportamento do caso de uso base. O caso de uso estendido só é executado sob certas circunstâncias. Dessa forma, ações *defaults* devem ser associadas ao caso de uso base e ações esporádicas ao caso de uso estendido. Relacionamento de dependência "*include*" evita repetição ao fatorar uma atividade comum a dois ou mais casos de uso.

Além do diagrama, casos de uso podem ser descritos por meio de documentos. A descrição pode ser informal, típica ou detalhada. A primeira contém o nome do caso de uso e uma descrição textual de sua funcionalidade. A segunda contém a identificação do ator que iniciou o caso de uso, os pré-requisitos (se houver) do caso de uso e uma descrição textual do fluxo normal e dos fluxos alternativos (se houver). A terceira contém, além do conteúdo da descrição típica, estrutura de dados e regras de negócio.

Diagrama de Máquina de Estados: diagrama comportamental que captura o ciclo de vida do objeto, os estados durante sua existência e a transição de estados em resposta a eventos de ativação ou devido a ações internas do código.

Estado é a condição do objeto em determinado momento. Sua representação gráfica é um retângulo com bordas arredondadas. O estado inicial é representado por um círculo, e o estado final por um círculo dentro de uma circunferência. O diagrama deve possuir um estado inicial e um ou mais estados finais.

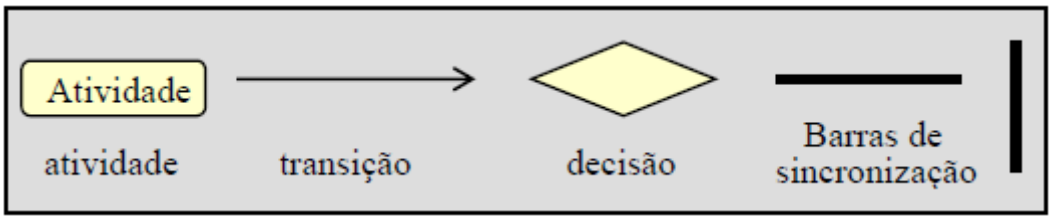
Transição é o relacionamento que indica mudança de estado. É representada graficamente por uma linha contínua com uma seta, que parte do estado inicial e aponta para o estado final. Transições podem ser acionadas por eventos ou por ações internas do código referente ao estado de origem. Evento de ativação é um estímulo capaz de disparar transição de estado em um objeto. Condição de guarda é a expressão, expressa entre colchetes, que define o disparo do evento de ativação.

Diagrama de Atividades: diagrama comportamental que modela o fluxo sequencial das atividades. Elementos gráficos desse diagrama são exibidos na Figura 11. Losango representa notação de decisão. Barras de sincronização representam a execução de fluxos concorrentes ou paralelos. Elas podem ser

barras de bifurcação, um fluxo de entrada e dois ou mais de saída; ou de união, dois ou mais fluxos de entrada e um de saída.

Esse diagrama também pode ser representado utilizando o elemento raia (*Swin Lane*). Raias definem a responsabilidade na execução das atividades e são representadas graficamente por retângulos que definem o escopo das atividades em relação aos objetos. Cada atividade pertence a uma única raia, mas as transições podem cruzá-las.

Figura 11: Elementos do diagrama de atividades.



Fonte: [5].

Diagrama de Tempo: diagrama comportamental que deriva do Diagrama de Interação. Ele apresenta o comportamento do objeto e sua interação em uma escala temporal, com foco nas condições de mudança no decorrer do tempo.

Diagrama de Sequência: diagrama comportamental que deriva do Diagrama de Interação. Ele visualiza a interação entre objetos participantes em termos de suas linhas de vida e mensagens trocadas no decorrer do tempo. Eixos verticais, representados por linhas pontilhadas, são denominados "linhas de vida". Eles determinam o tempo de vida dos objetos, da parte superior para a inferior. Os objetos envolvidos na realização da atividade retratada são exibidos na horizontal.

Objetos interagem por meio de mensagens, que podem conter números para explicitar a sequência do diagrama. Condições para envio das mensagens denominam-se condições de guarda e são expressas entre colchetes. Mensagens especificam o emissor, o receptor, e definem o tipo de comunicação entre as linhas de vida. Elas representam ações de chamada de operação, retorno de valor para objeto solicitante, e ainda criação ou destruição de objeto. Mensagens classificam-se de síncrona, assíncrona ou de retorno, conforme exibido na Figura 12.

Figura 12: Tipos de mensagem no Diagrama de Sequência.

Símbolo	Significado
	Mensagem síncrona
	Mensagem assíncrona
	Mensagem de retorno

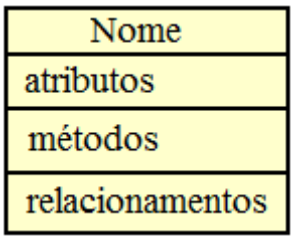
Chamadas síncronas associadas a uma operação possuem mensagem de envio e de recebimento. A linha de vida de origem fica bloqueada para outras operações até receber a mensagem de retorno da linha de vida de destino. Chamadas assíncronas contêm mensagem de envio, e possivelmente de resposta. A linha vida de origem fica desbloqueada para outras operações. Mensagens de recebimento e de resposta são classificadas de mensagens de retorno, cuja exibição é opcional.

Diagrama de Comunicação: diagrama comportamental que deriva do Diagrama de Interação. Ele visualiza a interação entre objetos com ênfase no contexto do sistema.

Diagrama de Visão Geral de Interação: diagrama comportamental que deriva do Diagrama de Interação. Engloba diversos diagramas de interação, podendo ser de tipos distintos, para representar um processo geral.

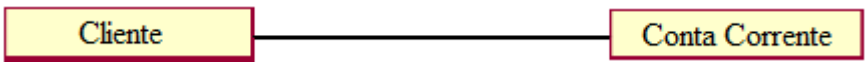
Diagrama de Classes: diagrama estrutural que exibe um conjunto de classes e seus relacionamentos. Um sistema costuma ser representado por alguns Diagramas de Classe, podendo a mesma classe estar presente em mais de um deles. Classes são representadas graficamente por retângulos, que incluem nome, atributos, métodos e relacionamentos. Elas devem receber nomes significativos com inicial maiúscula. A Figura 13 apresenta modelo para representação de classe.

Figura 13: Representação gráfica genérica de classe.



Relacionamentos classificam-se de “Associação”, “Generalização” e “Dependência”, podendo conter indicadores de multiplicidade e sentido de leitura. Associação, representada graficamente por uma linha sólida, indica o vínculo entre objetos de classes distintas. A Figura 14 exemplifica esse relacionamento. Nela o objeto da classe “Cliente” possui objeto da classe “Conta Corrente”.

Figura 14: Exemplo de relacionamento do tipo Associação.



Agregação é um tipo especial de associação que indica uma classe estar contida ou ser parte de outra classe. A Figura 15 exemplifica esse relacionamento. Nela, um objeto da classe “Marinha” contém vários objetos da classe “Navio”.

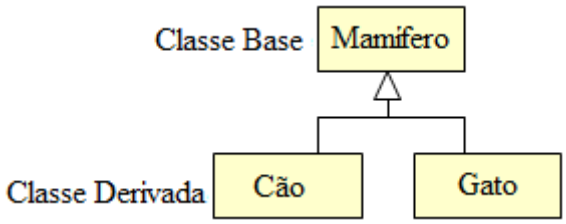
Figura 15: Exemplo de relacionamento do tipo Agregação.

Composição é uma variação de agregação. Nela os objetos "parte" só podem pertencer a um único objeto "todo" e seus tempos de vida coincidem com o dele. A Figura 16 exemplifica esse relacionamento.

Figura 16: Exemplo de relacionamento do tipo Composição.

Relacionamento do tipo generalização, exemplificado na Figura 17, refere-se à classe derivada herdar comportamento e significado da classe base.

Figura 17: Exemplo de relacionamento do tipo Generalização.



Relacionamento de dependência indica que alteração no objeto independente pode afetar o objeto dependente. A Figura 18 exemplifica esse relacionamento. Nela a classe "Cliente" depende de algum serviço da classe "Fornecedor", logo a mudança de estado de um objeto dessa afeta o objeto daquela.

Figura 18: Exemplo de relacionamento do tipo Dependência.

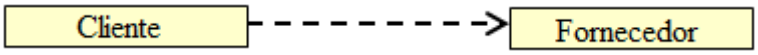


Diagrama de Objetos: diagrama estrutural que exibe o estado e os vínculos entre os objetos em determinado momento da execução. A representação gráfica do objeto contém o nome seguido por ":" e pelo nome da classe referente na parte superior. Na Figura 19 (a), o objeto foi nomeado como "p1". Na Figura 18 (b), o nome do objeto foi omitido, sendo classificado de objeto anônimo. Na parte inferior, insere-se os atributos e seus valores correspondentes em um determinado momento da execução.

Figura 19: (a) Objeto nomeado. (b) Objeto anônimo.

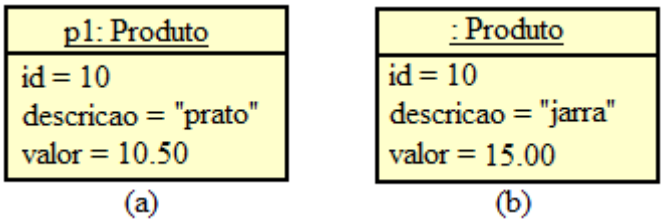


Diagrama de Componentes: diagrama estrutural que descreve os componentes de *software* e suas dependências. Componente de *software* é uma parte física do sistema que corresponde à realização de um conjunto de classes e/ou interfaces. Componente é representado graficamente por uma das notações da Figura 20, sendo a notação mais atualizada a presente na lateral direita dessa figura. Código fonte, biblioteca e arquivos binários, executáveis e de textos são exemplos de componentes.

Dependência entre componentes é representada por meio de uma linha tracejada com uma seta na ponta. Componentes podem definir interfaces visíveis aos demais, devendo os relacionamentos ser independentes das implementações nesses casos. A interface é representada por uma linha contendo um círculo na extremidade, com o nome junto ao círculo.

Figura 20: Notação de componentes em UML.

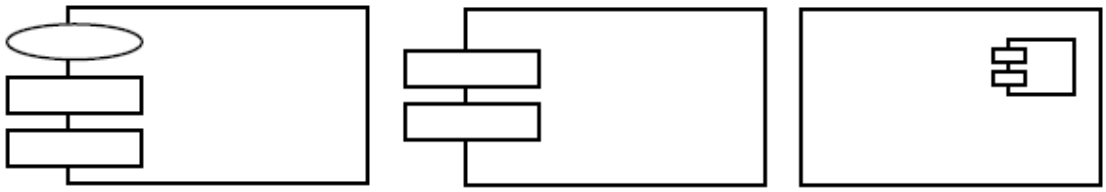


Diagrama de Estruturas Compostas: diagrama estrutural que modela a estrutura interna de um classificador por meio de peças, portas e conectores. Classificador é um mecanismo que descreve características comportamentais e de estrutura na UML. Ele pode representar classes, interfaces, tipos de dados, sinais, nós, casos de uso, subsistemas, colaborações. Peças, representadas graficamente por retângulos, constituem conjuntos de entidades cooperativas (instâncias) contidas no classificador. Portas, representadas graficamente por quadrados, definem pontos de interação entre classificador e o ambiente, entre classificador e peças internas, ou ainda entre peças internas do classificador. Conectores, representados graficamente por linhas sólidas, expressam os relacionamentos no modelo.

Diagrama de Perfil: diagrama estrutural que define novos modelos UML. Ele permite estender os diagramas existentes por meio da inclusão de estruturas customizadas para atender necessidades específicas. Esse diagrama possibilita a UML se adaptar a plataformas, domínios e metodologias de desenvolvimento de *software* para os quais não foi projetada originalmente.

Diagrama de Implantação: diagrama estrutural que exibe a arquitetura do *hardware* e do *software* do sistema. Ele pode conter componentes, nós e conexões. Componentes constituem partes do sistema referentes à realização de um conjunto de classes e/ou interfaces. Nós, representados graficamente por cubos, constituem elementos físicos que compõem o sistema. Computador cliente, computador servidor, impressora e roteador são exemplos de nós. Conexões, representadas graficamente por linhas, interligam nós e componentes.

Diagrama de Pacotes: diagrama estrutural que agrupa classes em pacotes; decompõe sistema em subsistemas; decompõe sistema em camadas. Esse diagrama é composto por pacotes e relacionamentos. Os relacionamentos entre pacotes costumam ser de dependência, expressos graficamente por uma linha pontilhada com uma seta. A notação gráfica de pacote é um diretório contendo um nome. O critério para definição de pacotes é subjetivo e depende da visão e necessidades do projetista. Um pacote deve agrupar elementos similares que tendem a ser modificados em conjunto.

No processo de documentação do *software* ADDEP foram implementados Diagramas de Casos de Uso, para exibir as funcionalidades do sistema de acordo com as necessidades do usuário; Diagrama de Pacotes e Diagramas de Classes, para documentar a estrutura estática do sistema; Diagrama de Atividades e Diagrama de Sequência, para apresentar o comportamento dinâmico do sistema.

3.8. **Ferramentas de Diagramação**

Os instrumentos de modelagem utilizados para elaboração dos diagramas foram o programa Microsoft Visio e a ferramenta Gliffy, apresentados com mais detalhes em Anexo. Eles foram escolhidos por serem simples de serem utilizados, possuírem interface amigável e intuitiva, possibilitarem a elaboração de diversos diagramas UML, e contarem com ampla gama de elementos gráficos. Apesar das vantagens apresentadas, ambas as ferramentas possuem como desvantagens não serem gratuitas.

Microsoft Visio é um *software* que permite elaborar fluxogramas; mapear redes de TI; criar organogramas; documentar processos empresarias, *software*, etc. Ele contém modelos atualizados e uma grande variedade de formas que atendem a padrões como *Unified Modeling Language* (UML), *Business Process Model* (BPMN) e em conformidade com o *Institute of Eletrical and Electronics Engineers* (IEEE) [10].

Gliffy é uma ferramenta de diagramação que permite a colaboração em tempo real de diversos usuários. Essa ferramenta foi desenvolvida em tecnologia Flash, permitindo a modelagem de diversos tipos de diagramas [11].

A documentação do ADDEP foi inicialmente elaborada utilizando o programa *Microsoft* Visio. No entanto, devido à limitação de disponibilidade desse *software*, adotou-se o Gliffy para dar prosseguimento às atividades.

4. Documentação e Reestruturação do ADDEP

4.1. Definição de Requisitos da Documentação

As atividades apresentadas neste relatório foram aplicadas no software ADDEP quando ele já havia completado a etapa de prototipação e estava passando pelos processos de expansão e consolidação.

Até essa fase de vida do *software*, ele contava com funções para leitura de arquivos COMTRADE dos tipos configuração e dados, estimação fasorial, detecção, classificação e localização de faltas, rotinas para estimação do tempo de abertura dos disjuntores e funções de proteção de distância (elementos mho e quadrilateral).

No entanto, até essa fase de desenvolvimento, o ADDEP estava sendo elaborado sem um projeto prévio e não havia sido documentado, surgindo assim a necessidade de realizar-se a documentação e reestruturação do mesmo. Para atender essa necessidade de documentar e reestruturar o *software* ADDEP, este trabalho de conclusão de curso foi realizado.

Para a realização dos processos de documentação e reestruturação foi disponibilizado apenas o código fonte do *software*, elaborado na linguagem Matlab. Após o estudo do código, realizou-se uma documentação inicial do ADDEP com o objetivo de compreender o sistema. Para isso, foram elaborados Diagrama de Blocos, para apresentar a estrutura estática do sistema; Diagrama de Casos de Uso, para ilustrar os requisitos do sistema; e um documento de descrição detalhado de Caso de Uso.

Em seguida, realizou-se a análise do *software* e reestruturação do mesmo, aplicando-se técnicas de refatoração e padrões de projeto.

Por fim, realizou-se a documentação do *software* reestruturado, composto pelo Diagrama de Casos de Uso, o mesmo elaborado antes da reestruturação uma vez que os requisitos do sistema se mantiveram os mesmos, e por novos diagramas elaborados para documentar o *software* reestruturado. Esses consistem em: Diagramas de Classes e Diagrama de Pacotes, para apresentar a estrutura estática do sistema; Diagrama de Sequência e Diagrama de Atividades, para apresentar o comportamento Dinâmico do sistema.

A documentação visa informar tanto a equipe de projeto quanto o cliente sobre o *software* implementado e prover suporte ao reuso. A reestruturação realizada, aplicando-se técnicas de refatoração e padrões de projeto, visa tornar o *software* mais robusto e reutilizável, de forma a facilitar sua expansão.

A fim de auxiliar a equipe do projeto no entendimento do trabalho realizado, foi elaborado um tutorial, apresentado em Apêndice, contendo informações relativas a Padrões de Projeto, Refatoração, UML e MDS.

Ao término dessas atividades, a documentação do *software* reestruturado foi apresentada a integrantes do projeto para validação. Após a leitura do tutorial, eles compreenderam os diagramas elaborados e certificaram que os modelos estavam de acordo com as funcionalidades e princípio de funcionamento do *software* ADDEP.

Para se determinar as funcionalidades do sistema de acordo com as necessidades do usuário, elaborou-se o Diagrama de Casos de Uso, exibido na Figura 41, e o documento de descrição detalhado apresentado a seguir:

UTILIZAR ADDEP – CASO DE USO

NOME: Utilizar ADDEP.

DESCRIÇÃO SUCINTA: Usuário utiliza o sistema ADDEP.

ATOR

1. Usuário.

PRÉ-CONDIÇÕES

1. Usuário possuir dados requeridos pelo sistema.

ESTRUTURA DE DADOS

(ED1) Dados de Entrada

1.1 Parâmetros da linha de transmissão.
1.2 Parâmetros do relé.
1.3 Nome do arquivo Comtrade.
1.4 Canais analógicos.
1.5 Canais digitais.

(ED2) Dados de Saída

2.1 Dados extraídos do registro.
2.2 Dados estimados.
2.3 Dados do relé.
2.4 Gráficos.

FLUXO BÁSICO

1. Usuário limpa dados defaults.
2. Sistema
3. Usuário informa dados ao sistema.
4. Sistema obtém dados.

5. Sistema calcula parâmetros.
6. Sistema gera resultados.
7. Sistema exibe resultados.
8. Usuário seleciona gráfico que deseja visualizar.
9. Sistema exibe gráfico selecionado.

FLUXO ALTERNATIVO

(A1) Alternativa ao Passo 3 – Dados inseridos são inválidos

1.a) Sistema não calcula parâmetros.
1.b) Sistema informa ao usuário que os dados são inválidos.
1.c) Sistema retorna ao passo 2.

(A2) Alternativa ao Passo 3 – A regra RN1 não é atendida

2.a) Sistema não calcula parâmetros.
2.b) Sistema informa ao usuário erro no arquivo Comtrade.
2.c) Sistema retorna ao passo 2.

REGRAS DE NEGÓCIO

(RN1) O arquivo Comtrade não deve estar corrompido.

4.2. **Análise do *Software***

Para a compreensão da estrutura estática inicial do *software*, elaborou-se o Diagrama de Blocos exibido na Figura 24. A escolha desse tipo de diagrama ocorreu devido ao código fonte disponibilizado não ter sido elaborado utilizando-se o paradigma de orientação a objetos. Esse estava segmentado em blocos de códigos, presentes em um mesmo escopo, e em métodos. Devido a isso, para a representação do código recebido, não foram elaborados Diagramas de Classes, nem Diagrama de Objetos.

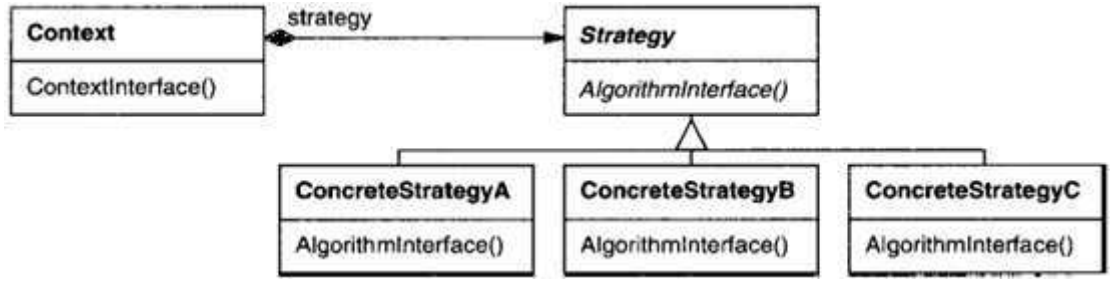
Após a documentação do sistema inicial, realizou-se o processo de reestruturação. Na análise do *software*, utilizou-se o paradigma de orientação a objetos. Dessa forma, modelou-se o mundo real e, na fase de projeto, traduziram-se os objetos encontrados.

No processo de refatoração, reestruturou-se o sistema em classes. Blocos de código extensos foram segmentados em mais de uma classe. Nas classes relativas à leitura de arquivos Comtrade, aplicou-se o padrão de projeto *Strategy*, cujo modelo é exibido na Figura 21.

O padrão *Strategy* é um padrão comportamental de objetos que define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. Segundo [6] deve-se utilizá-lo quando:

- Muitas classes relacionadas diferem somente no comportamento;
- Necessita-se de variantes de um algoritmo;
- Deseja-se encapsular algoritmo dos clientes. Dessa forma, evita-se a exposição das estruturas de dados complexas específicas do algoritmo;
- Uma classe define muitos comportamentos, e estes aparecem em suas operações como múltiplos comandos condicionais da linguagem. Movem-se os ramos condicionais para suas próprias classes concretas “Strategy”.

Figura 21: Diagrama contendo formato do Padrão de Projeto *Strategy*.



Fonte: [6].

O padrão *Strategy* foi aplicado na leitura do arquivo Comtrade “.dat” por esse poder ser do tipo “*Binary*” ou “ASCII”. As implementações da leitura desses arquivos diferem, e apareciam como múltiplos condicionais no código fonte. Com a aplicação desse padrão projeto, moveram-se os ramos condicionais relacionados para suas próprias classes Strategy.

O formato do padrão *Strategy* aplicado na leitura do arquivo Comtrade “.dat” é exibido no Diagrama de Classes presente na Figura 32. A classe abstrata “*StrategyDat*” se relaciona com a classe concreta “Comtrade” por meio de composição e com as classes concretas derivadas “StrategyBinary” e “StrategyAscii” por meio de generalização.

A classe base “*StrategyDat*” define uma interface comum a todos os algoritmos suportados. As classes derivadas “StrategyBinary” e “StrategyAscii” implementam o algoritmo de leitura do arquivo “.dat”, utilizando a interface de “*StrategyDat*”. A classe “Comtrade” mantém uma referência para o objeto da classe “*StrategyDat*”, e define uma interface que permite “*StrategyDat*” acessar seus dados.

O uso do padrão *Strategy* teve como principal benefício a eliminação do uso de comandos condicionais para seleção de comportamentos desejados, e teve como principal custo o aumento do número de objetos na aplicação.

Após o processo de reestruturação do *software* ADDEP, realizou-se então o projeto do *software* reestruturado por meio de diagramas UML contendo modelos estáticos e dinâmicos do sistema.

4.3. **Documentação do *Software***

A estrutura dos modelos elaborados para documentar o *software* ADDEP antes da reestruturação é apresentada na Figura 22. Ela é composta pelo Diagrama de Blocos e pelo Diagrama de Casos de Uso.

A estrutura dos modelos elaborados para documentar o *software* ADDEP após a reestruturação é apresentada na Figura 23. Ela é composta pelo Diagrama de Pacotes, Diagrama de Classes, Diagrama de Casos de Uso, Diagrama de Atividades e pelo Diagrama de Sequência.

Figura 22: Estrutura de modelos do software ADDEP antes da reestruturação.

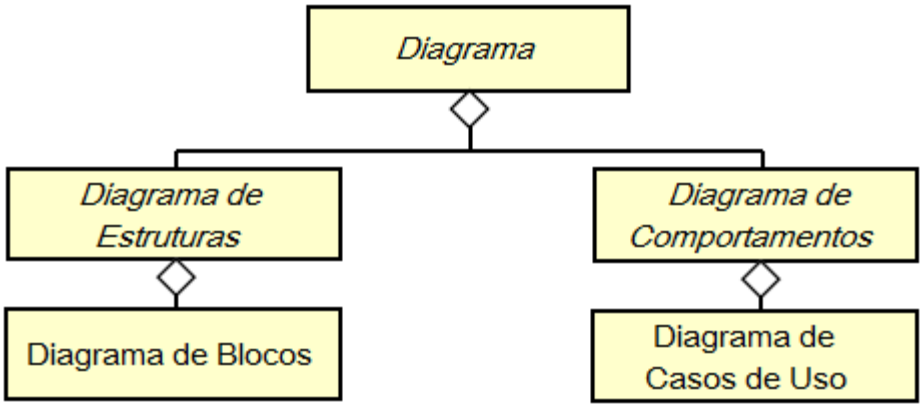
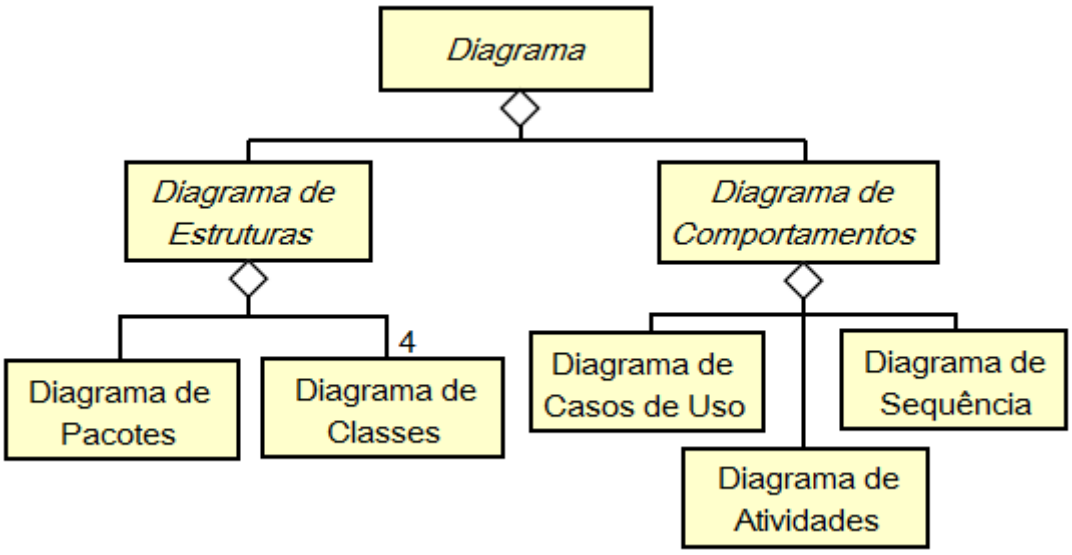


Figura 23: Estrutura de modelos do software ADDEP após a reestruturação.



Os modelos presentes nas estruturas apresentadas na Figura 22 e na Figura 23 serão explicadas em mais detalhes na seção 4.3.1. Modelos Estáticos e na seção 4.3.2. Modelos Dinâmicos.

4.3.1. Modelos Estáticos

Os modelos estáticos elaborados para documentar os aspectos estáticos do sistema ADDEP consistem no Diagrama de Blocos, que representa o sistema antes da reestruturação; e no Diagrama de Pacotes e nos Diagramas de Classes, que representam o sistema após a reestruturação.

4.3.1.1. DIAGRAMA DE BLOCOS

A representação do sistema, antes de ser reestruturado, em Diagrama de Blocos é exibida na Figura 24, e a legenda do mesmo é exibida na Figura 25. Esse diagrama consiste em dezesseis blocos de atividade, que estavam presentes em um mesmo escopo no código fonte; em dezesseis blocos de função; em dois blocos que indicam o início e fim do diagrama.

Figura 24: Diagrama de Blocos.

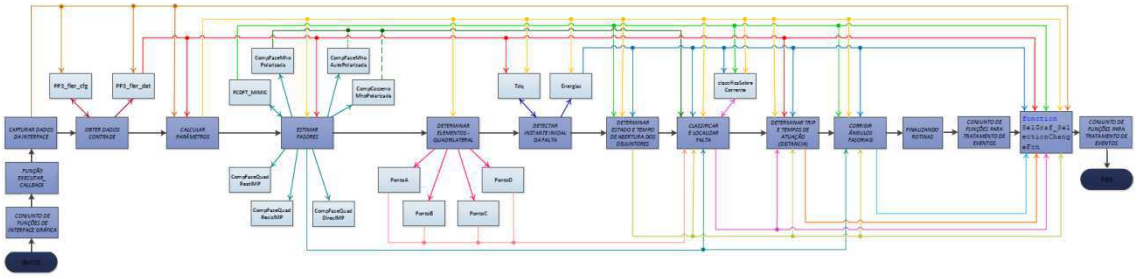
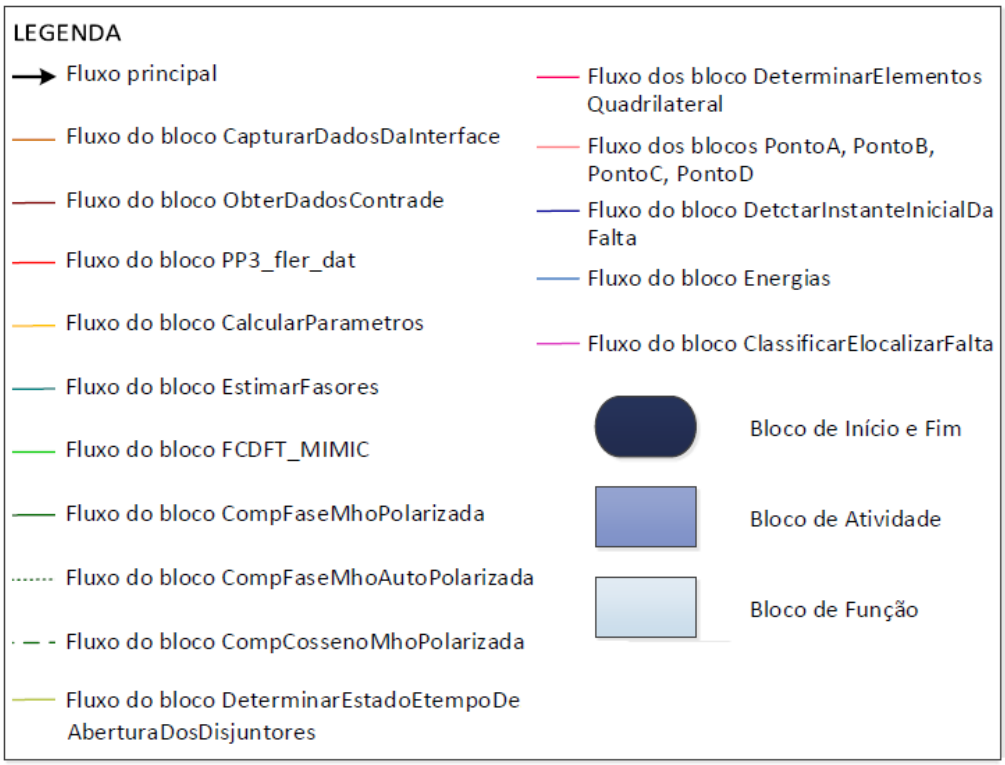
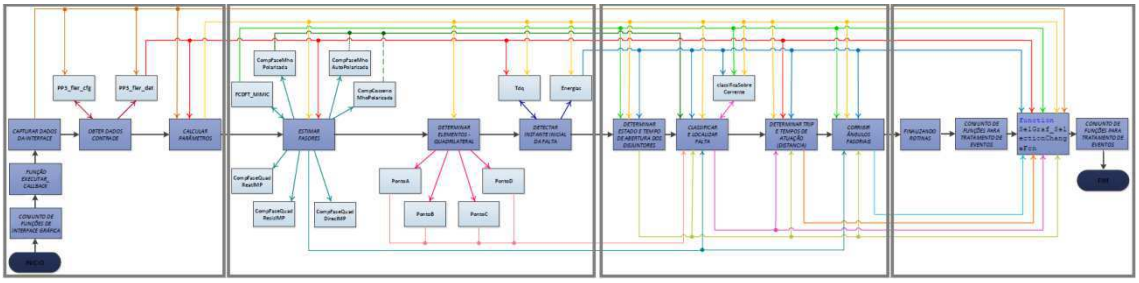


Figura 25: Legenda do Diagrama de Blocos da Figura 25.



Para facilitar a visualização, a Figura 24 foi dividida em quatro seguimentos, conforme exibido na Figura 26. Esses seguimentos são explicados a seguir.

Figura 26: Diagrama de Blocos segmentado.



No primeiro segmento, exibido na Figura 27, tem-se início a aplicação. Os dados são inseridos pelo usuário, realiza-se a leitura e obtenção de dados do arquivo Comtrade e calculam-se os parâmetros.

No segundo segmento, exibido na Figura 28, realiza-se a estimação fasorial, determinam-se os elementos quadrilaterais e detecta-se o instante inicial da falta.

No terceiro seguimento, exibido na Figura 29, determina-se o estado e tempo de abertura dos disjuntores, classifica-se e localiza-se a falta, determinam-se o trip e o tempo de atuação da proteção de distância e corrigem-se os ângulos fasoriais.

No quarto segmento, exibido na Figura 30, finalizam-se as rotinas e implementa-se o tratamento de eventos relativos à interface gráfica.

Figura 27: Segmento 1 - Diagrama de Blocos: obtenção de dados e cálculo dos parâmetros.

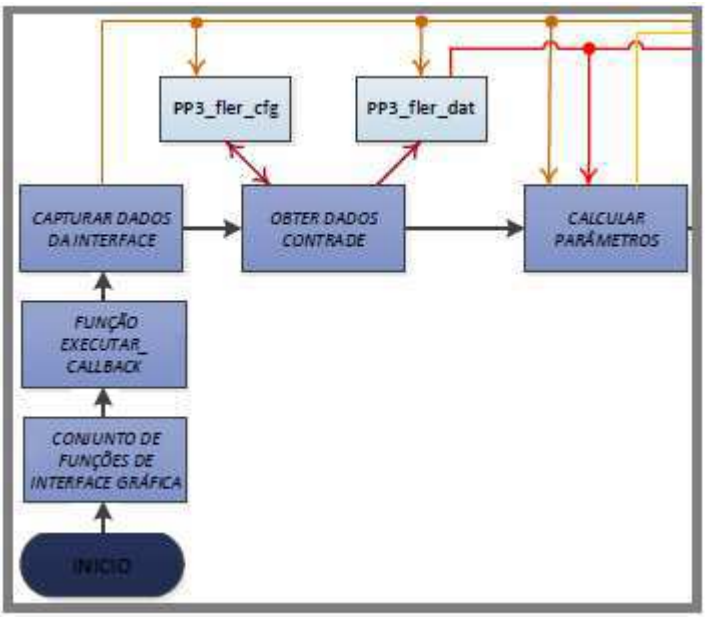


Figura 28: Segmento 2 - Diagrama de Blocos: estimação fasorial, determinação dos elementos quadrilaterais e detecção da falta.

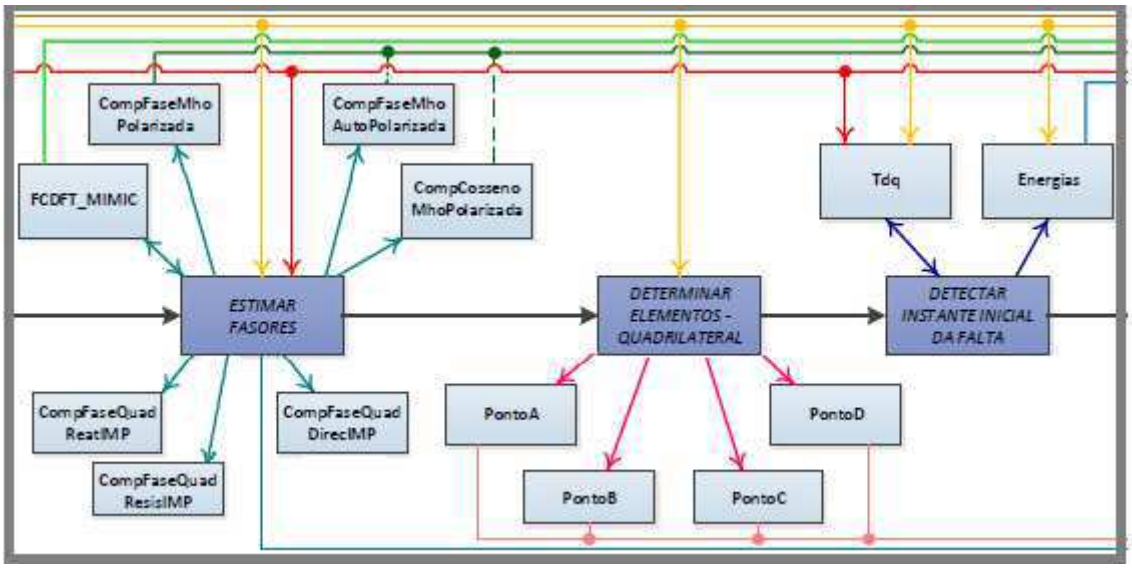


Figura 29: Segmento 3 - Diagrama de Blocos: estado dos disjuntores, classificação da falta, determinação do trip e correção fasorial.

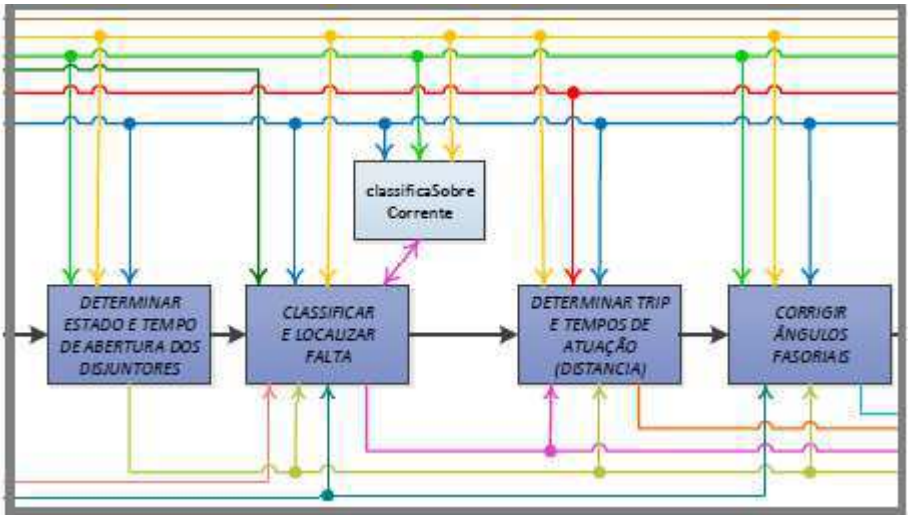


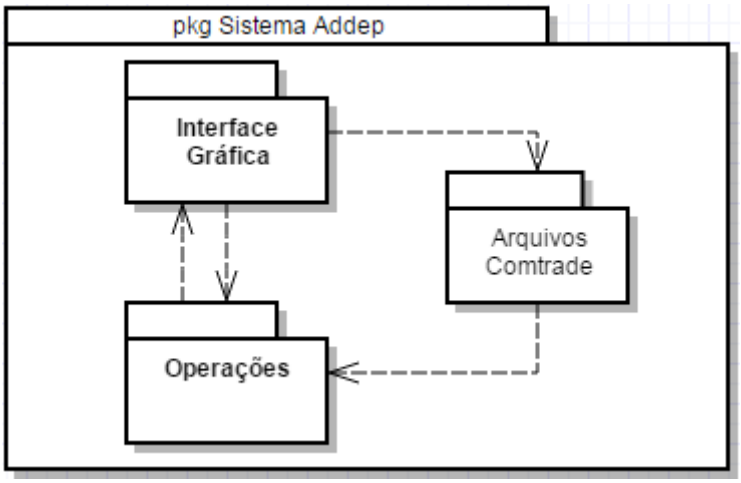
Figura 30: Segmento 4 - Diagrama de Blocos: tratamento dos eventos da interface.

4.3.1.2. DIAGRAMA DE PACOTES

O Diagrama de Pacotes elaborado refere-se ao sistema após ser reestruturado. Ele agrupa classes em pacotes a fim de decompor o sistema ADDEP em subsistemas. Cada pacote agrupa elementos similares, que tendem a ser modificados em conjunto.

A Figura 31 exibe o Diagrama de Pacotes implementado para modelar o sistema ADDEP. Os pacotes presentes nesse diagrama são: "Sistema ADDEP", que contém os demais pacotes; "Interface Gráfica", que contém classes referentes à interface gráfica do *software*; "Arquivos Comtrade", que contém as classes envolvidas na leitura e aquisição de dados dos arquivos Comtrade; e "Operações", que contém classes relativas às operações realizadas no cálculo dos resultados.

Figura 31: Diagrama de Pacotes.



4.3.1.3. DIAGRAMA DE CLASSES

Os Diagramas de Classes elaborados referem-se ao sistema após ser reestruturado. O sistema ADDEP foi representado por meio de quatro Diagramas de Classes. O Diagrama de Classes referente ao pacote "Sistema ADDEP" é exibido na Figura 32. A documentação dessas classes é apresentada na Figura 33.

Figura 32: Diagrama de Classes referente ao pacote "Sistema ADDEP".

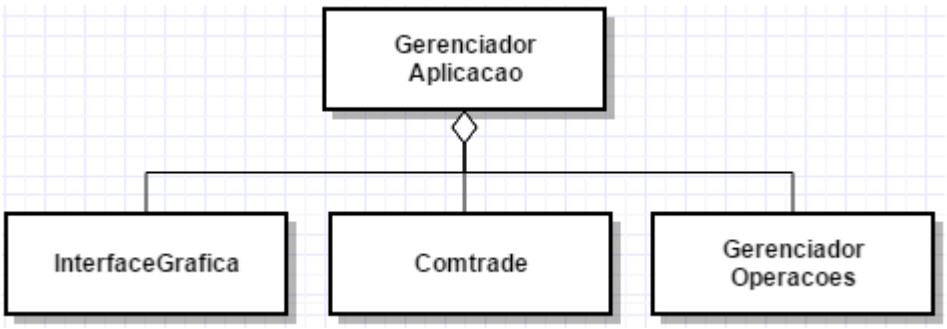
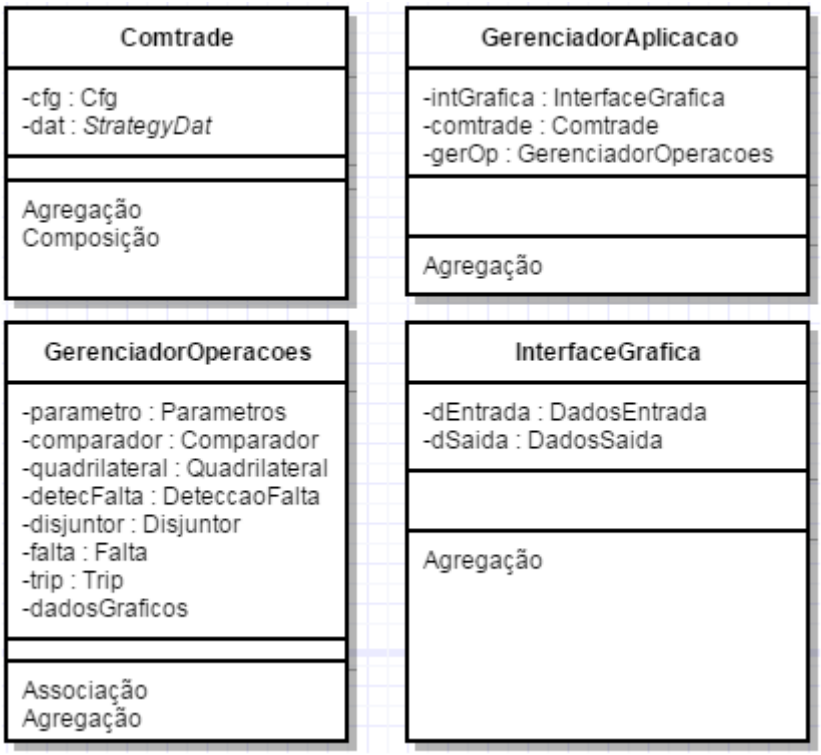


Figura 33: Documentação das classes referente à Figura 32.



O Diagrama de Classes referente ao pacote “Interface Gráfica” é exibido na Figura 34. A documentação dessas classes é apresentada na Figura 35.

Figura 34: Diagrama de Classes referente ao pacote “Interface Gráfica”.

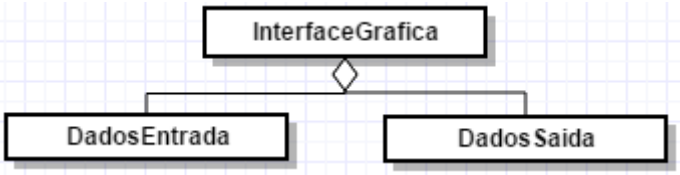


Figura 35: Documentação das classes referente à Figura 34.

O Diagrama de Classes referente ao pacote “Arquivos Comtrade” é exibido na Figura 36. A documentação dessas classes é apresentada na Figura 37.

Esse Diagrama de Classes contém o padrão de projeto *Strategy* aplicado na leitura dos arquivos Comtrade, conforme explicado na seção 4.2. Análise de *Software*.

Figura 36: Diagrama de Classes referente ao pacote “Arquivos Comtrade”.

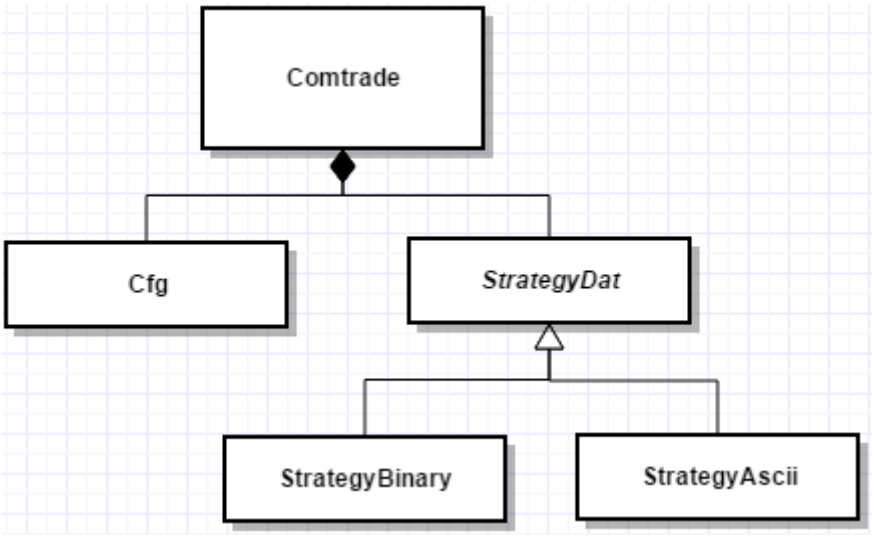
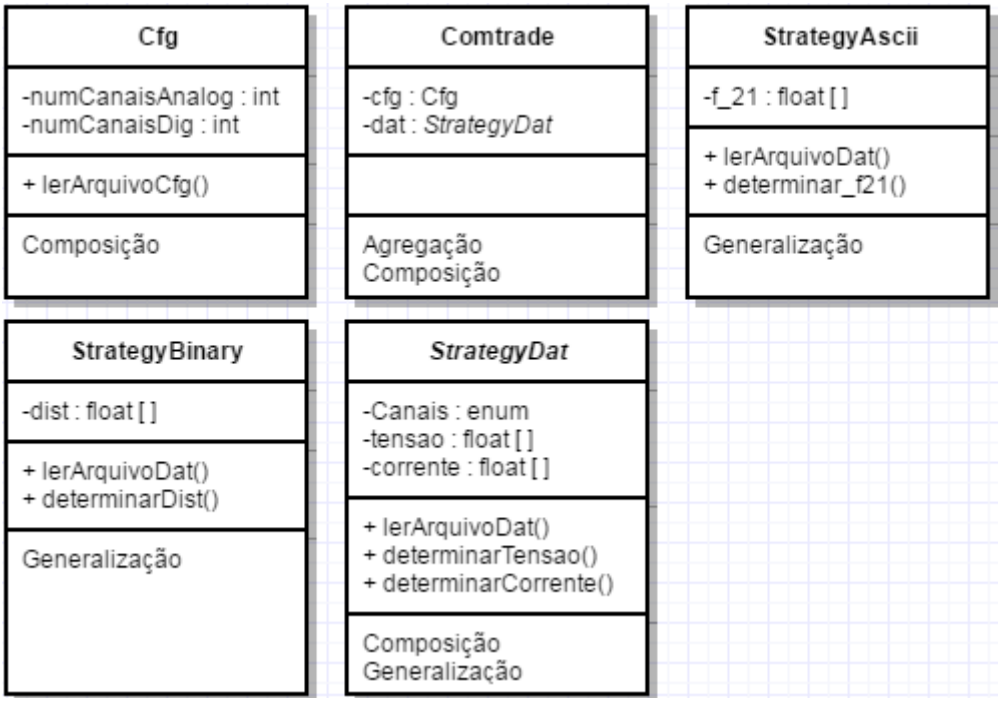


Figura 37: Documentação das classes referente à Figura 36.



O Diagrama de Classes referente ao pacote “Operações” é exibido na Figura 38. A documentação dessas classes é apresentada na Figura 39 e Figura 40.

Figura 38: Diagrama de Classes referente ao pacote “Operações”.

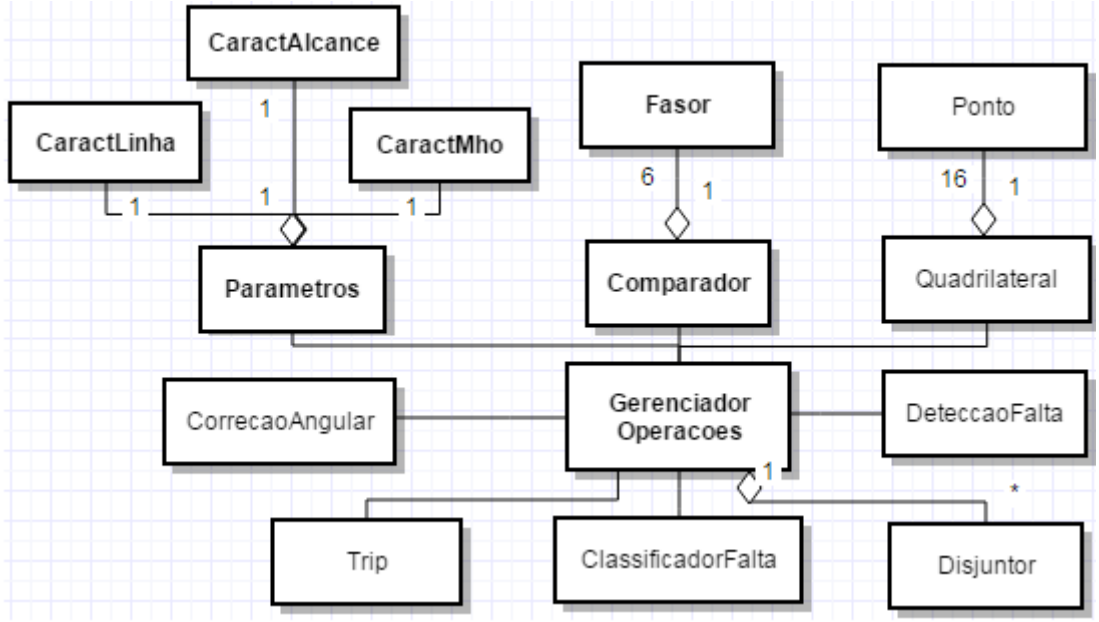


Figura 39: Documentação das classes referente à Figura 38 – Parte 1.

CarctAlcance
-X : float -R : float
+ setCaractQuadrilateral()
Agregação

CarctLinha
-L : float -RL : float -XL : float -YL : float -ZL : float
+ setParametrosLinha()
Agregação

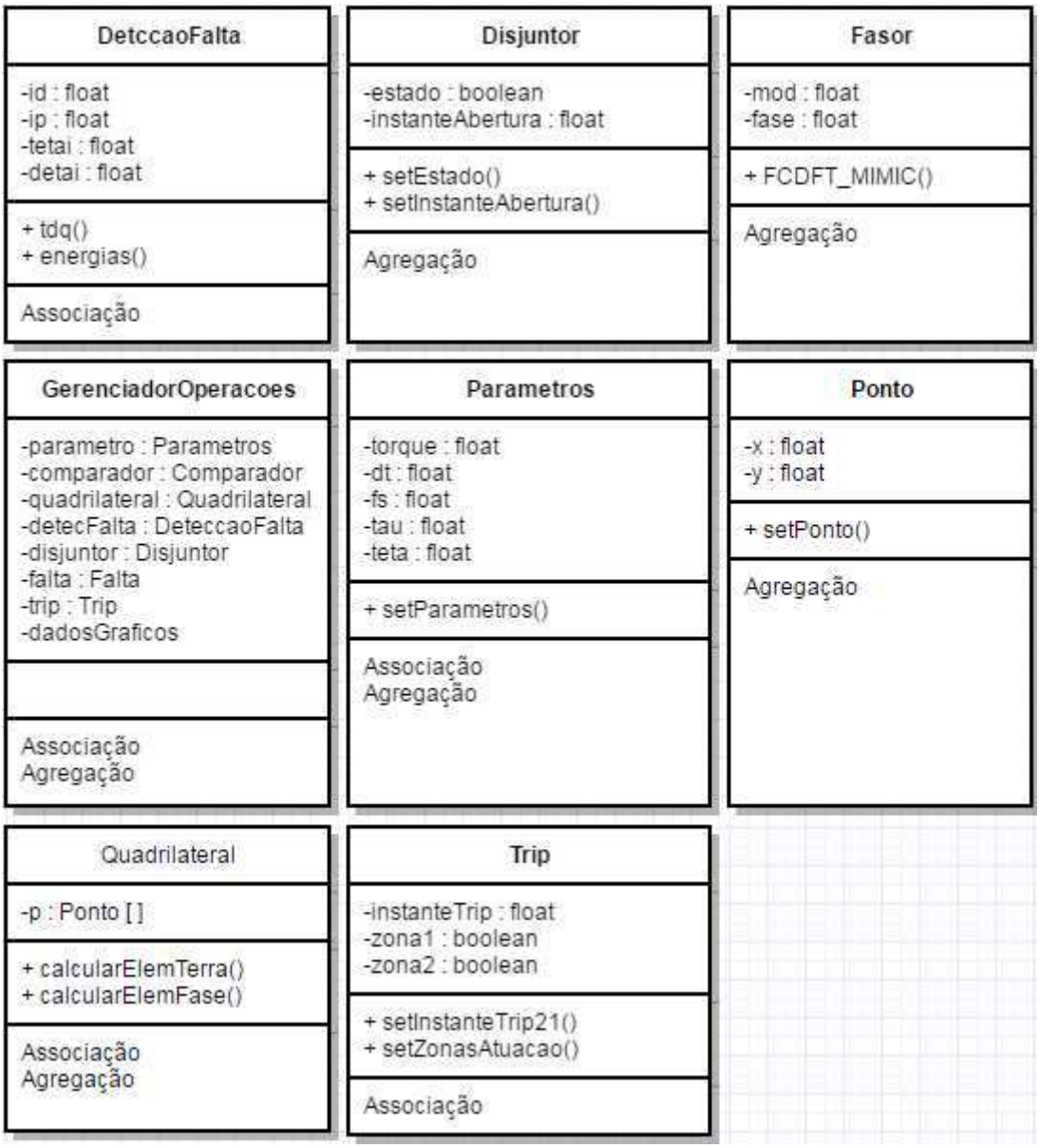
CarctMho
-ZA : float -reMho : float -imMho : float
+ setCaractMho()
Agregação

ClassificadorFalta
-TiposDeFalta : enum -sobrecorrente : int -falta : int -localizacao : float
+ classificarSobrecorrete() + classificarFalta() + localizarFalta()
Associação

Comparador
-cf_pol : float -cf_Apol : float -cCos : float -cReat : float -cResist : float -cDirecional : float
+ compFaseMhoPol() + compFaseMhoAutoPol() + compCosMhoPol() + compFaseQuadReatImp() + compFaseQuadResImp() + compFaseQuadDirecImp()
Associação Agregação

CorrecaoAngular
-angPreFalta : float -angPreFaltaCorrig : float -angFalta : float -angFaltaCorrig : float
+ corrigirAngPreFalta() + corrigirAngFalta()
Associação

Figura 40: Documentação das classes referente à Figura 38 – Parte 2.



4.3.2. Modelos Dinâmicos

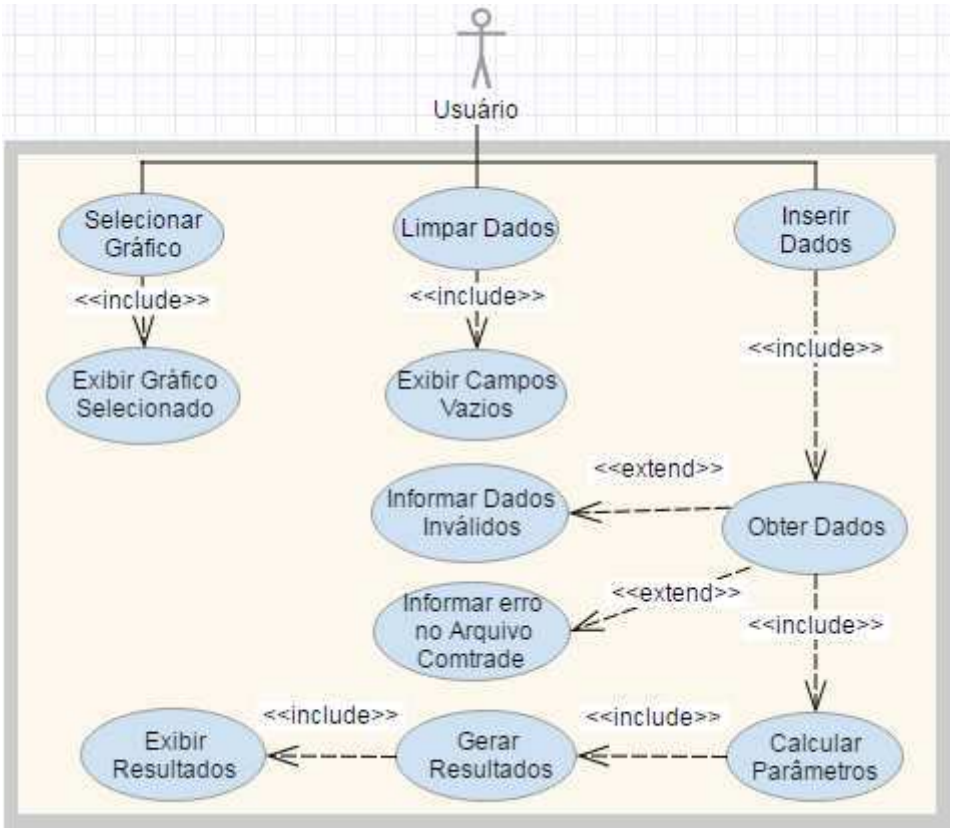
Visando obter melhor compreensão e documentação do sistema, elaboraram-se diagramas UML contendo modelos dinâmicos. Esses modelos representam o sistema antes e após a reestruturação, tendo em vista que os requisitos do *software*, a sequência de atividades e as interações do sistema não sofreram alterações.

Os modelos dinâmicos elaborados para documentar o comportamento do sistema ADDEP consistem no Diagrama de Casos de Uso, para exibir os requisitos funcionais do *software*; no Diagrama de Atividades, para modelar o fluxo sequencial das atividades; e no Diagrama de Sequência, para representar as interações do sistema com ênfase no decorrer do tempo.

4.3.2.1. DIAGRAMA DE CASOS DE USO

Na especificação dos requisitos do *software*, determinam-se as funcionalidades do sistema de acordo com as necessidades do usuário. Para isso, elaborou-se o Diagrama de Casos de Uso, exibido na Figura 41, além o documento de descrição detalhada, como técnica de modelagem de requisitos.

Figura 41: Diagrama de Casos de Uso.



No Diagrama de Casos de Uso apresentado, o "Usuário" é o único ator do sistema. Ele se relaciona com os casos de uso "Selecionar Gráfico", "Limpar Dados" e "Inserir Dados". Ao selecionar o caso de uso "Limpar Dados", os campos de dados da interface gráfica ficarão em branco. Após informar os dados, os casos de uso "Obter Dados", "Calcular Parâmetros", "Gerar Resultados" e "Exibir Resultados" correspondem ao fluxo básico do sistema, por isso são conectados pelo relacionamento de dependência "*include*".

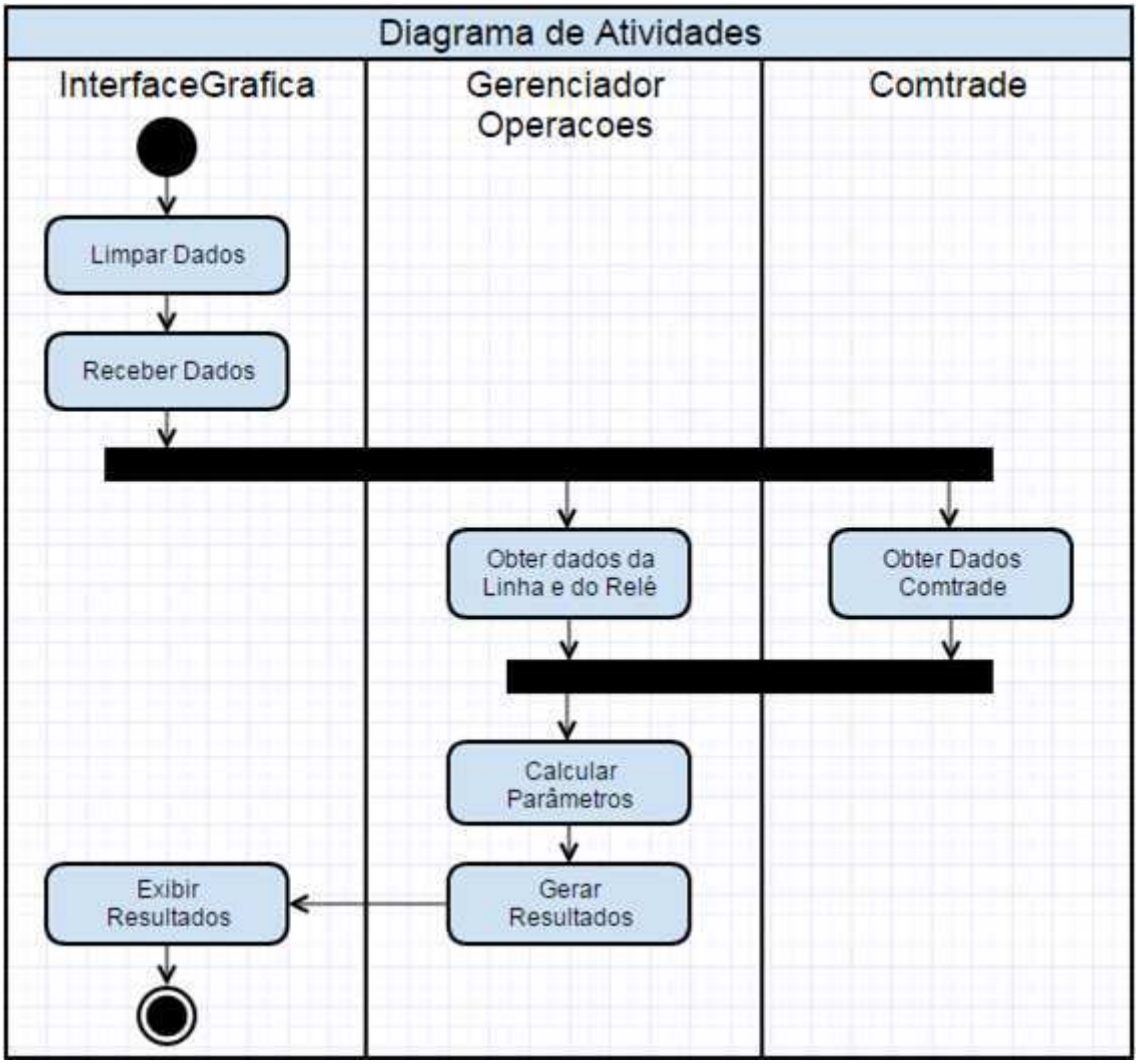
Caso seja identificado algum problema no caso de uso "Obter Dados", o sistema segue o fluxo alternativo "Informar Dados Inválidos" ou "Informar erro no Arquivo Comtrade", a depender do problema que ocorra. Por isso, esses casos de uso são conectados pelo relacionamento de dependência "*extend*". Após os resultados do sistema terem sido exibidos, quando o usuário selecionar um gráfico, o caso de uso "Exibir Gráfico Selecionado" corresponde ao fluxo básico do sistema, conectando-se por meio do relacionamento de dependência "*include*".

4.3.2.2. DIAGRAMA DE ATIVIDADES

A Figura 42 exibe o Diagrama de Atividades do *software* ADDEP, a fim de apresentar o fluxo de atividades realizadas. Ele é composto por três objetos: “InterfaceGrafica”, “GerenciadorOperacoes” e “Comtrade”.

Cada objeto é representado dentro de uma raia, que define a responsabilidade na execução das atividades. As atividades têm início no escopo do objeto “InterfaceGrafica”. A primeira a ser realizada é “Limpar Dados”, seguida de “Receber Dados”. Os dados recebidos são encaminhados para os escopos dos objetos “GerenciadorOperacoes” e “Comtrade” de forma paralela. No escopo do objeto “Comtrade”, realiza-se a atividade “Obter Dados Comtrade”. No escopo do objeto “GerenciadorOperacoes” realizam-se as atividades “Obter dados da Linha e do Relé” e, em conjunto com os dados Comtrade, “Calcular Parâmetros” e “Gerar Resultados”. Os resultados gerados são enviados para o escopo do objeto “InterfaceGrafica”, onde a atividade “Exibir Resultados” é realizada e o fluxo é finalizado.

Figura 42: Diagrama de Atividades.

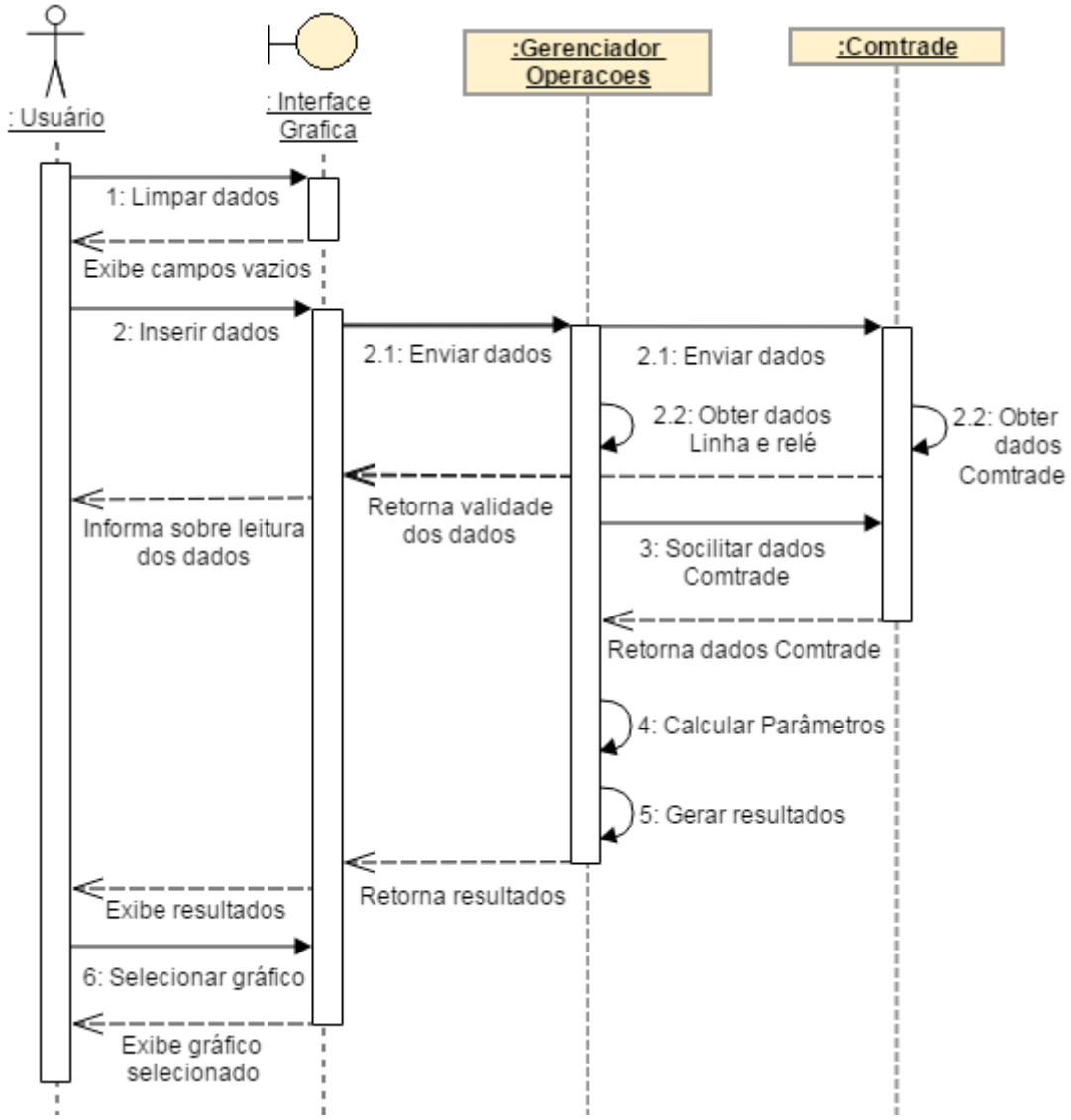


4.3.2.3. DIAGRAMA DE SEQUÊNCIA

A Figura 43 exibe o Diagrama de Sequência do *software* ADDEP, a fim de visualizar a interação entre objetos participantes em termos de suas linhas de vida e mensagens trocadas no decorrer do tempo.

Nesse diagrama, pode-se visualizar que "Usuário" troca mensagens com "InterfaceGráfica", e essa se comunica com "GerenciadorOperacoes" e "Comtrade". As mensagens enviadas são do tipo síncronas, logo a linha de vida de origem fica bloqueada para outras operações até o recebimento da mensagem de retorno, e são numeradas para indicar a ordem.

Figura 43: Diagrama de Sequência.



A reestruturação realizada e os modelos documentados por meio de diagramas UML tornam o sistema ADDEP mais robusto para que possa continuar o processo de expansão de forma facilitada.

5. Considerações Finais

As atividades de documentação e reestruturação do aplicativo ADDEP resultaram em modelos UML que visam documentar e reestruturar o *software*, a fim de atender seus requisitos, servir como aspecto técnico para o sistema e prover suporte ao reuso.

Os processos envolvidos nessas atividades proporcionaram a aprendizagem sobre técnicas de modelagem e métodos de desenvolvimento, refatoração e padrões de projeto; além disso, contribuíram para o desenvolvimento de habilidades como capacidade de abstração, uso de tecnologias de diagramação, elaboração de modelos UML, e realização de análise e projeto de *software*.

A realização das atividades mostrou-se não trivial, visto demandar grau elevado de abstração. No entanto, complementou de forma imprescindível a formação acadêmica, uma vez que a necessidade do mercado por profissionais com conhecimento em desenvolvimento de *software* tem aumentado e se mostrado indispensável para atuação em diversos segmentos.

Uma sugestão para continuidade deste trabalho consiste na refatoração do *software* ADDEP, tendo como base a documentação do *software* reestruturado fornecido neste relatório, a fim de que a qualidade do produto final satisfaça as expectativas do cliente.

6. Referências Bibliográficas

[1] FILHO, A. M. S. Sobre a Importância da Arquitetura de *Software* no Desenvolvimento de Sistemas de *Software*. União dos Institutos Brasileiros de Tecnologia. Recife, PE.

[2] Lopes, F.; Barros, D.; Reis, R.; Costa, C.; Nascimento, J.; Brito, N.; Neves, W.; Moraes, S. Avaliação Preliminar de Aplicativo para Diagnóstico de Perturbações no Sitema CHESF de Transmissão. XVI Encontro Ibero-Americano do Cigre. Misiones, Argentina, 2015.

[3] VERAS, B. P. M. S. Estudo de Registros Oscilográficos. Relatório de Projeto de Pesquisa e Desenvolvimento. Campina Grande, PB, 2014.

[4] Lopes, F.; Barros, D.; Reis, R.; Costa, C.; Nascimento, J.; Brito, N.; Neves, W.; Moraes, S. Influência de Métodos de Estimação Fasorial no Processo de Análise de Faltas no Sistema CHESF. XI Conferência Brasilelira sobre Qualidade da Energia Elétrica. Campina Grande, PB, 2015.

[5] SANTOS, R. F. UML - Linguagem de Modelagem Unificada, 2009.

[6] GAMMA, E. et al. Padrões de Projeto: Soluções Reutilizáveis de *Software* Orientado a Objetos. Porto Alegre, RS: Bookman, 2007.

[7] FOWLER, M. et al. Refactoring: Improving the Design of Existing Code. New Jersey, United States: Pearson Education, 1999.

[8] Disponível em < http://www.moniro.com/products/services/software-refactoring.html> Acessado em: 19/05/2016.

[9] Disponível em <http://www.etelg.com.br/paginaete/downloads/informatica/apostila_uml.pdf> Acessado em: 19/05/2016.

[10] Disponível em <http://office.microsoft.com/pt-br/visio/> Acessado em: 19/05/2016.

[11] Oliveira, J. A.; Souza, P.P.; Figueiredo, E. Uma Avaliação de Ferramentas de Modelagem de *Software*. Trabalho de Iniciação Científica. Universidade Federal de Minas Gerais, Belo Horizonte, MG.

[12] Disponível em <https://blogs.office.com/2012/08/01/visios-new-modern-interface/> Acessado em: 19/05/2016.

7. Bibliografias Consultadas

GUEDES, T. A. G. et al. UML 2: Uma abordagem prática. São Paulo, SP: Novatec, 2011.

MELO, A. C. et al. Exercitando modelagem em UML. São Paulo, SP: Brasport, 2006.

GATTO, R. A. Estratégias para Reestruturação de Código Legado Visando à Utilização de Aspectos. 2007. 112 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Ciência da Computação, Universidade Estadual de Maringá, Maringá, PR.

BUENO, A. D. Programação Orientada a Objeto em C++, 2003.

Documents Associated With Unified Modeling Language (UML) Version 2.5. Disponível em <http://www.omg.org/spec/UML/2.5/> Acessado em: 19/05/2016.

User Manual for Gliffy Online. Disponível em <https://www.gliffy.com/user-manual/> Acessado em: 19/05/2016.

The Unified Modeling Language. Disponível em <http://www.uml-diagrams.org/> Acessado em: 19/05/2016.

Anexo

UTILIZAÇÃO DO *SOFTWARE MICROSOFT* VISIO

A utilização do *software* Visio ocorre de forma intuitiva, uma vez que sua interface gráfica segue o padrão dos produtos da Microsoft. A interface gráfica do Visio (Figura 44) é composta por uma barra de *menu* na parte superior, que contém as funcionalidades do *software*; por uma região contendo as categorias de diagramas e os componentes gráficos correspondentes na lateral esquerda; e, ao lado dessa região, por uma área para elaboração do diagrama.

Para iniciar um projeto, seleciona-se o *menu* "*File*" e a opção "*New*". O usuário conta com as opções de iniciar um novo projeto a partir de outro recentemente utilizado, de um modelo, dos diagramas de exemplo, ou iniciar um projeto em branco. Os elementos disponíveis na área de componentes gráficos variam de acordo com a categoria de diagrama selecionada.

Para elaborar um modelo, deve-se selecionar a categoria de diagrama desejado, clicar sobre os elementos gráficos, arrastá-los até a área de elaboração do diagrama, e dispô-los conforme desejado. Para conectar os elementos inseridos, deve-se acessar a *menu* "*Home*", selecionar sobre a opção "*Connector*" e, em seguida, clicar sobre os blocos que deseja-se conectar. Os componentes gráficos inseridos podem ser configurados clicando-se com o botão direito do *mouse* sobre eles e sobre a opção desejada, conforme exibido na Figura 45.

Figura 44: Interface gráfica do *software* Visio.

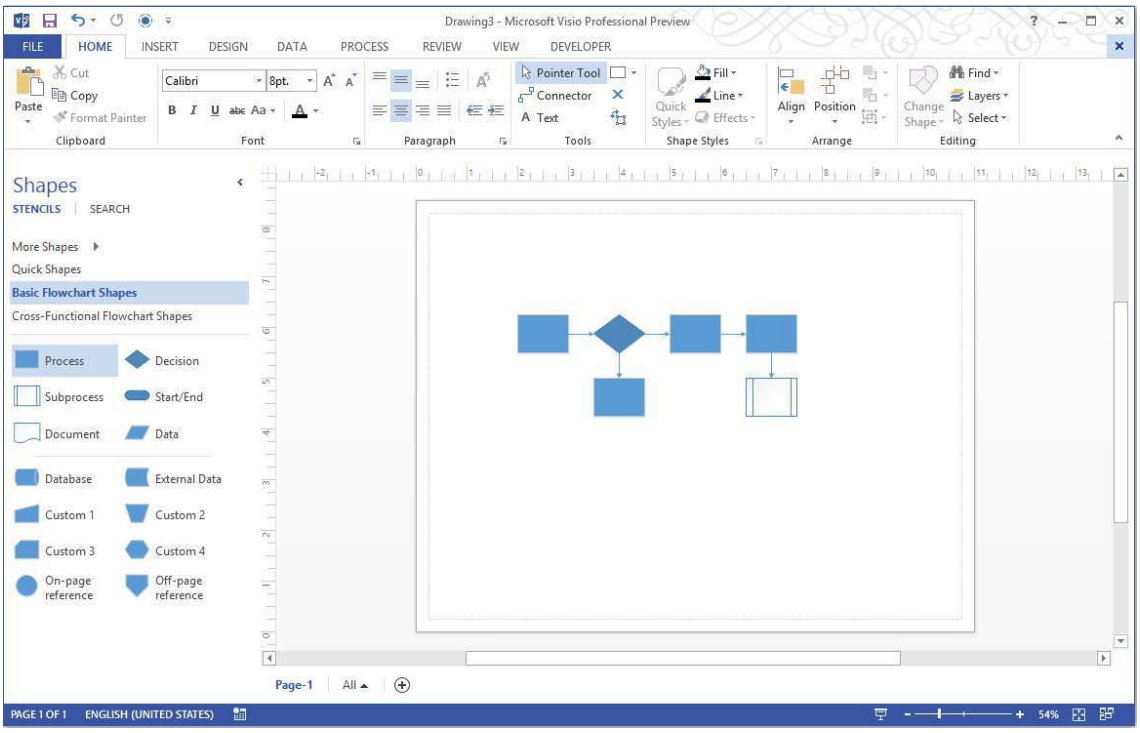
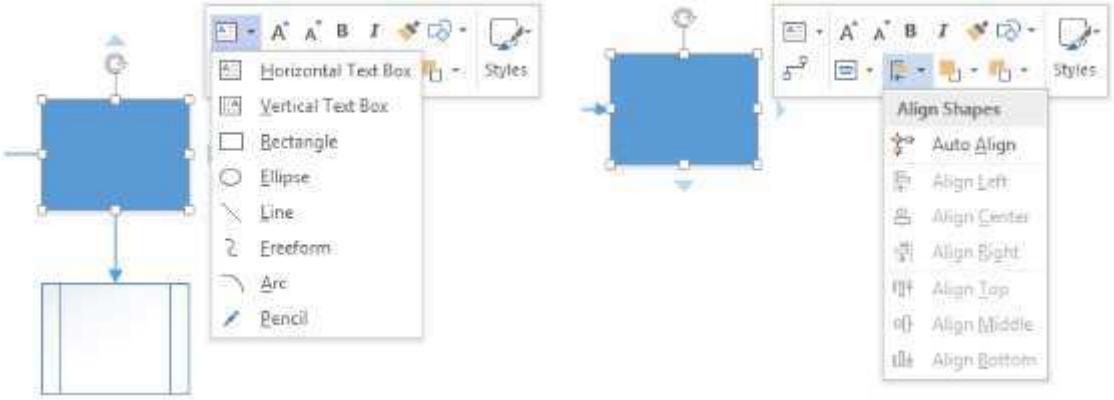


Figura 45: Configuração de elementos gráficos no *software* Visio.

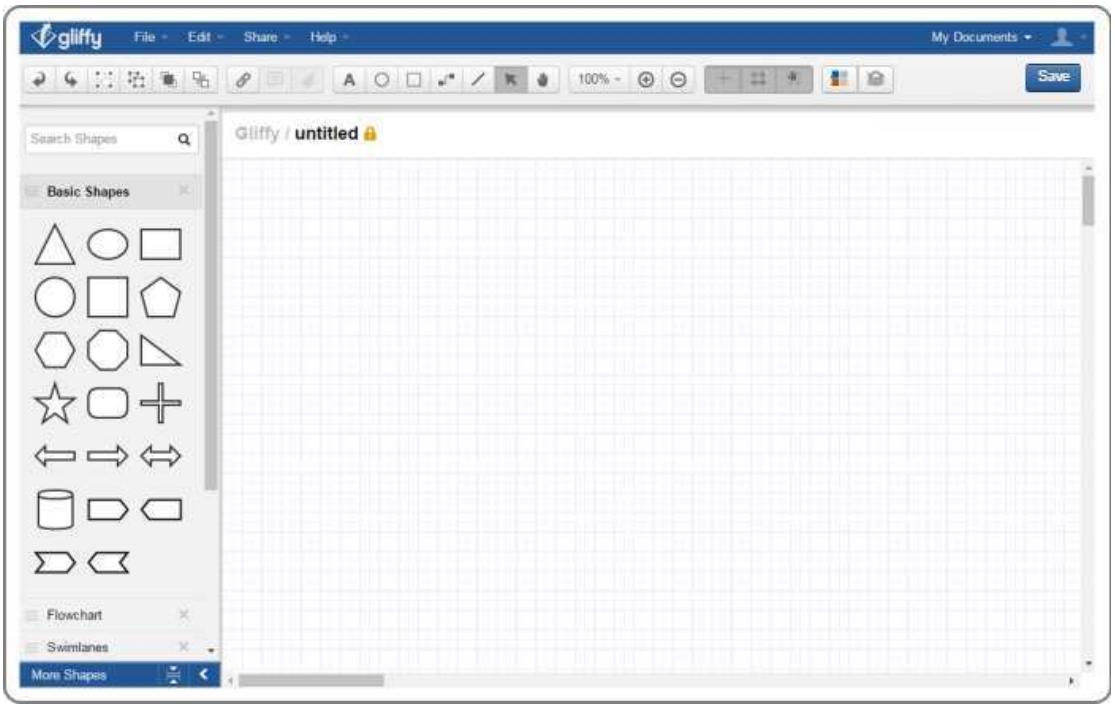


Fonte: [12].

UTILIZAÇÃO DA FERRAMENTA GLIFFY

Para a utilização do Gliffy, cuja interface gráfica é exibida na Figura 46, necessita-se que o usuário crie uma conta. Após realizar o *login*, as opções abrir um diagrama recente, criar um novo diagrama, importar um diagrama e criar diagrama a partir de um *template* serão apresentadas.

Figura 46: Interface gráfica da ferramenta Gliffy.



Para iniciar um novo projeto, seleciona-se a opção “Criar um novo diagrama” e escolhe-se o tipo de diagrama desejado. A interface gráfica do Gliffy é composta por uma barra de *menu* na parte superior, seguida por uma barra de ferramentas.

Na lateral esquerda, apresenta-se um *menu* expansível, referente às categorias de diagramas, contendo os elementos gráficos. Ao lado dessa região, tem-se a área desenho para elaboração do diagrama.

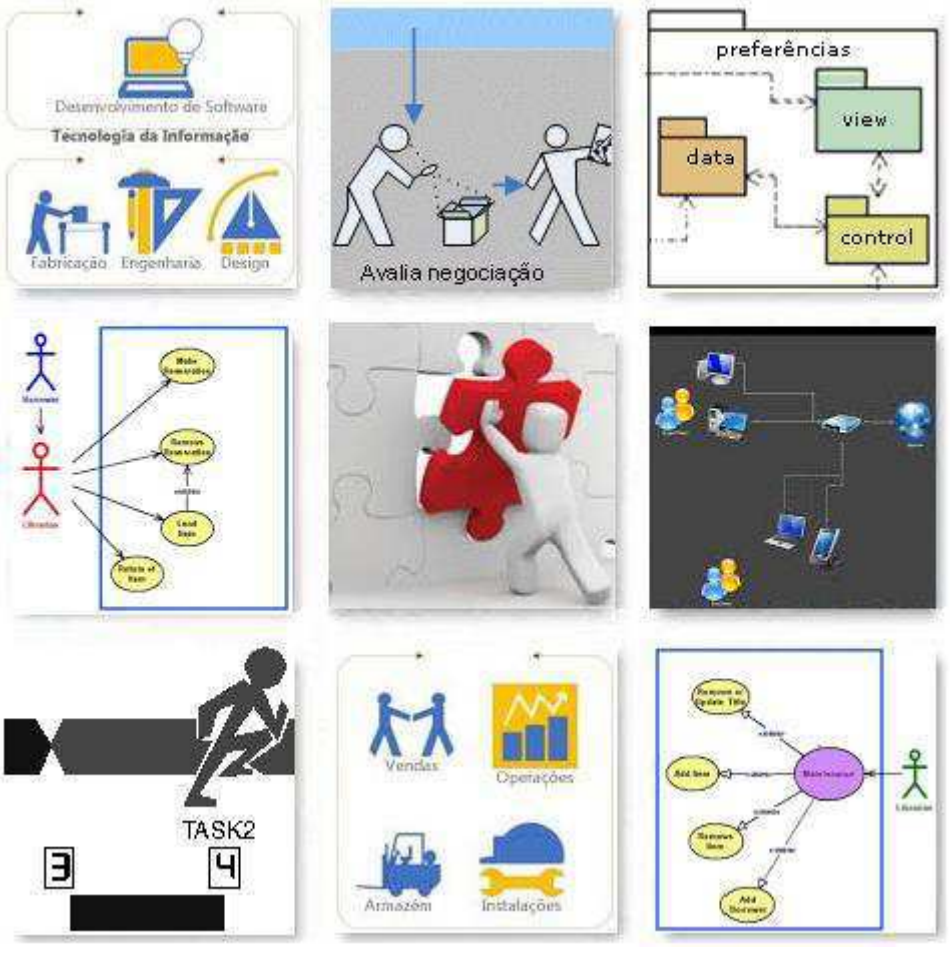
Para elaborar um modelo, deve-se selecionar a categoria de diagrama desejado, clicar sobre os elementos gráficos, arrastá-los até a área de desenho, e dispô-los conforme desejado. Para conectar os elementos, deve-se selecionar a opção "*Connector Tool*" na barra de ferramentas e, em seguida, clicar sobre os blocos que deseja-se conectar. Componentes gráficos inseridos podem ser configurados selecionando-se as funcionalidades presentes na barra de ferramentas.

O Gliffy, que inicialmente funcionava apenas *online*, agora possibilita o uso *offline*. Constitui uma ferramenta intuitiva e conta com três versões: *Business Team*, *Business* e *Standard*. Essas diferem no preço, no número de diagramas suportados, nas categorias de diagrama disponíveis, dentre outras coisas. A desvantagem do Gliggy é o uso gratuito ser limitado a um determinado número de dias, sendo necessário após esse período um pagamento mensal.

Apêndice

Relatório para modelagem e construção de *Software*

Desenvolvimento de *Software*



Neste Relatório:

Padrões de Projeto

Refatoramento

UML

MDS

Autora:
Polyana B. Marcelino
Supervisão:
Prof. Benemar A. Souza
Profª. Núbia D. Brito

Sumário

1. **Metodologia de Desenvolvimento de *Software***

Um programa computacional busca resolver um problema que define ou está contido em um domínio. Esse domínio está sujeito às leis da física, da matemática, do direito do mercado financeiro, etc.

Busca-se no desenvolvimento de um sistema computacional a construção de um modelo que coloque em termos de algoritmos o domínio da aplicação. Uma tecnologia utilizada na construção de modelos é a Orientação a Objetos. Modelos orientados a objetos são implementados por meio de linguagem de programação orientada a objetos.

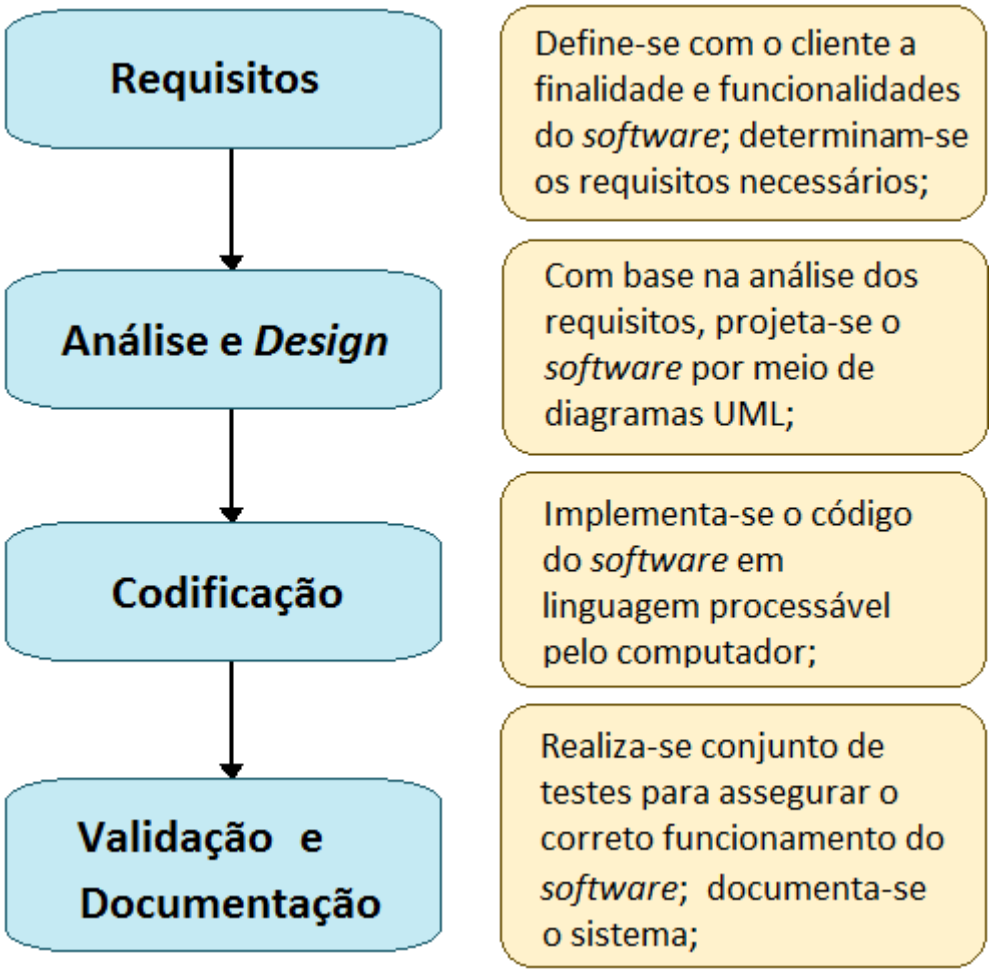
Sistemas orientados a objetos, quando projetados corretamente, são flexíveis a mudanças, possuem estruturas bem conhecidas e proporcionam a implementação de componentes reutilizáveis.

Os métodos de desenvolvimento de *software* anteriores ao surgimento do paradigma Orientação a Objetos organizavam a especificação de um sistema de acordo com suas funções ou com dados manipulados. Esses métodos costumavam apresentar dificuldades na transição entre as etapas do processo de desenvolvimento.

Metodologia de Desenvolvimento de *Software* (MDS) pode ser baseada em diversos modelos e fazer uso de diferentes processos reconhecidos. Ciclo de vida, cascata e prototipação são exemplos de modelos de MDS. Análise Estruturada, Orientação a Objetos, Orientação a Eventos e *Agile* são exemplos de processos de MDS.

No desenvolvimento de *software*, deve-se buscar adotar práticas como código padronizado, refatoração, código coletivo, *feedback,* comunicação eficiente e rotinas de testes. MDS envolve comunicação, planejamento, modelagem, construção e implantação. Em relação aos processos de modelagem e construção, as principais etapas e as respectivas atividades são exibidas na Figura 1.

Figura 1: Etapas presentes no desenvolvimento de *Software*.



2. Conceitos

2.1 Método

Elemento que descreve comportamento. Seu nome deve remeter a uma ação e ter inicial minúscula. O método possui a seguinte estrutura:

```
<modificador> <tipo de retorno> <nome> (<lista de argumentos>) {<bloco>}
```

Modificadores definem o tipo de acesso ao método, isto é, tipo de encapsulamento. Podem ser *public*, *protected* ou *private*. *Public* permite classes de qualquer escopo acessar o método. *Protected* permite classes do mesmo escopo acessarem o método. *Private* permite apenas a própria classe acessar o método. O conceito de modificador se estende a atributos. Tipo de retorno indica o tipo de variável retornada pelo método. O nome identifica o método. A lista de argumentos informa os tipos de

valores que deverão ser passados ao método. O Bloco contém a implementação do método.

2.2 Classe

Elemento que contém as propriedades e métodos que descrevem um conjunto de objetos com interfaces semelhantes. Ela deve ser nomeada por um substantivo e seu nome deve ter inicial maiúscula. A classe pode ser de dois tipos: Concreta ou Abstrata. As classes costumam trabalhar em conjunto com outras classes. A comunicação entre elas é denominada relacionamento.

Classe concreta contém a assinatura e implementação de todos os métodos. Classe abstrata é aquela cuja finalidade principal é definir uma interface comum para suas subclasses. Ela posterga parte de, ou toda, sua implementação para operações definidas nas subclasses. Portanto uma classe abstrata deve possuir pelo menos um método abstrato e não pode ser instanciada. Nomes de classes abstratas devem ter notação em itálico.

Métodos abstratos são aqueles que apenas determinam a existência de um comportamento, não definindo uma implementação. O tipo em itálico também é usado para denotar métodos abstratos. Uma classe que possui apenas métodos abstratos é denominada classe virtual pura.

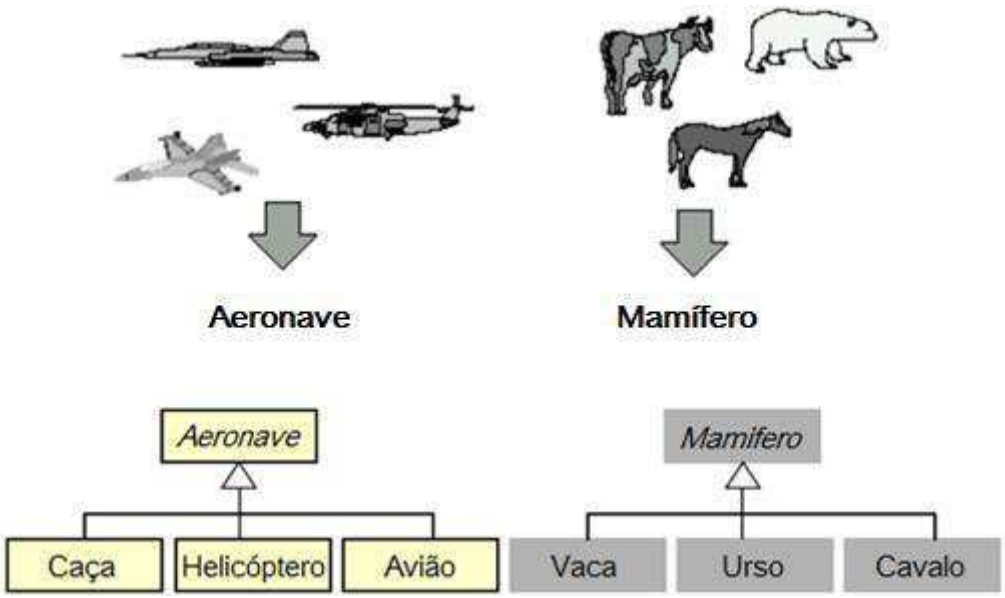
Classes abstratas capturam as propriedades e comportamentos essenciais dos objetos. A Figura 2 exibe as classes abstratas *Aeronave* e *Mamífero*. Elas representam a essência das classes Caça, Helicóptero, Avião, referentes à classe *Aeronave*; Vaca, Urso, Cavalo, referentes à classe *Mamífero*.

Na Figura 2, as classes *Aeronave* e *Mamífero* são classificadas de Classe Base, Classe Mãe ou SuperClasse. As demais classes são classificadas de Classe Derivada, Classe Filha ou SubClasse. Elas se relacionam por meio de Herança.

2.3 Herança

Processo no qual classes derivadas incorporam as propriedades e métodos de suas classes bases respectivas. Classes derivadas podem adicionar novas propriedades e métodos, e redefinir a implementação dos métodos herdados.

Figura 2: Exemplo de abstração expressa em classes.



Fonte: Adaptado de Santos (2009, p.12).

2.4 Polimorfismo

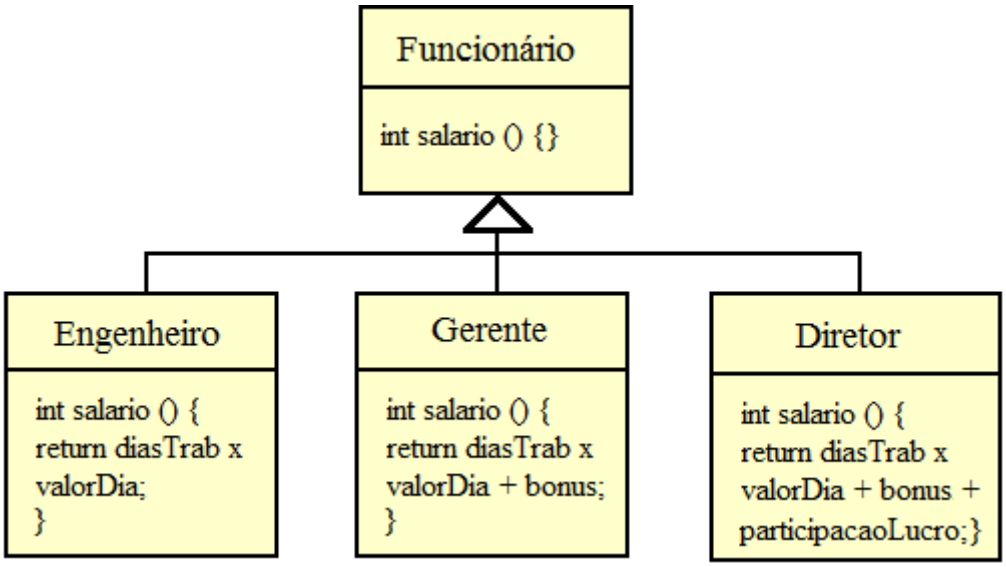
Mecanismo pelo qual um método assume diferentes interfaces para realizar uma mesma implementação ou assume diferentes implementações para uma mesma interface. O primeiro caso é denominado *Overloading*, e o segundo *Overriding*.

A Figura 3 exibe exemplo de *Overloading*. A implementação do método *subtrair* é a mesma, o nome do método é mantido, mas os tipos dos argumentos são distintos, o que caracteriza mudança na interface.

A Figura 4 exibe exemplo de *Overriding*. A interface do método "salario" é a mesma em todas as classes, mas a sua implementação difere.

Figura 3: Exemplo de Polimorfismo por meio de *Overloading*.

Figura 4: Exemplo de Polimorfismo por meio de *Overriding*.



Fonte: Adaptado de Santos (2009, p.23).

3. Tipos de *Software*

S*oftwares* podem ser classificados de Programas de Aplicação, *Toolkits* e *Frameworks*.

Programas de Aplicação constituem aplicativos para a solução de um problema específico, como editores de documentos ou planilhas. Nesse tipo de *software*, as prioridades são reusabilidade interna e facilidade de manutenção e extensão.

Frequentemente um Programa de Aplicação incorpora classes de uma ou mais bibliotecas de classes pré-definidas. Essas são denominadas *Toolkits* ou Bibliotecas de Classes.

Toolkits fornecem funcionalidades para auxiliar a execução de um Programa de Aplicação. Dessa forma, evita-se que o implementador recodifique funcionalidades comuns. Eles constituem conjuntos de classes relacionadas e reutilizáveis, projetados para fornecer funcionalidades específicas de uso genérico. Dessa forma, sua prioridade é a reusabilidade.

Uma biblioteca matemática é um exemplo de *Toolkit*. Ela deve conter funcionalidades específicas, realizar operações matemáticas, e ser implementada de forma a poder ser empregada em diferentes Programas de Aplicação.

Framework, também denominado Arcabouço de Classes, constitui um conjunto de classes cooperantes que constroem um projeto reutilizável para uma determinada categoria de *software*.

Um *Framework* é orientado à criação de Programas de Aplicação de natureza específica, como por exemplo, editores gráficos. A aplicação é customizada por meio da criação de subclasses específicas, derivadas das classes abstratas do *Framework*.

Frameworks determinam a arquitetura da aplicação. Eles definem a estrutura geral do Programa de Aplicação, sua divisão em classes e objetos, os relacionamentos dos componentes e o fluxo de controle. Dessa forma, o projetista/implementador concentra-se apenas nos aspectos específicos da aplicação.

O *Framework* deve poder ser aplicado em diversos Programas de Aplicação no domínio para o qual foi projetado. Dessa forma, suas prioridades são flexibilidade e extensibilidade.

4. Ciclo de vida de um *software*

O ciclo de vida de um *software* orientado a objetos possui várias fases: prototipação, expansão e consolidação, conforme indicado na Figura 5.

Figura 5: Ciclo de vida de um *software*.

Fonte: Gamma, Helm, Johnson e Vlissides (2007, p. 326).

A fase de prototipação é a fase inicial de um *software*. Nela surgem as primeiras implementações com a finalidade de atender um conjunto inicial de requisitos. Dessa forma, o *software* reflete as entidades no domínio do problema inicial e é colocado em funcionamento.

Uma vez que isso ocorre, sua evolução é governada por duas necessidades conflitantes: (1) o *software* deve satisfazer mais requisitos, e (2) o *software* deve ser mais reutilizável.

Comumente novos requisitos acrescentam novas classes e operações. O *software* passa por uma fase de expansão para atender novos requisitos, contudo eventualmente tornar-se muito inflexível e rígido para permitir mais mudanças. Assim, para continuar a evoluir, ele deve ser reorganizado por meio de um processo conhecido como refatoração.

Na fase de consolidação, novos objetos são produzidos, frequentemente pela decomposição de objetos existentes. A necessidade contínua de satisfazer mais requisitos, juntamente com a necessidade de maior reutilização, faz o *software* passar por repetidas fases de expansão e consolidação à medida que ele se torna mais genérico.

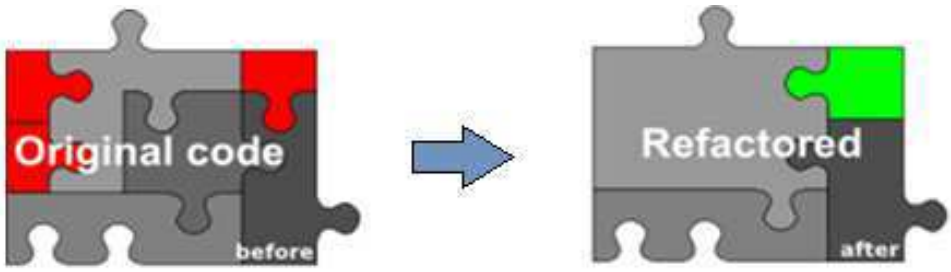
Algumas estruturas de classes e objetos tornam o projeto mais robusto frente às mudanças de requisitos, e diminuem as refatorações. Um sistema bem projetado facilita a identificação de mudanças necessárias, diminuindo o risco de erros durante esse processo.

5. **Refatoração**

Segundo Martin Fowler, refatoração é o processo de otimização do *software*, melhorando sua estrutura interna, sem modificar sua funcionalidade para o usuário. Recomenda-se aplicar esse processo quando deseja-se expandir um *software*, facilitar sua compreensão ou identificar e solucionar *bugs*.

A Figura 6 faz uma analogia ao processo de refatoração utilizando peças de quebra-cabeça como exemplo. Observe que a funcionalidade da peça vista externamente não foi alterada, no entanto, realizaram-se mudanças estruturais internas.

Figura 6: Analogia ao processo de Refatoração.



Fonte: Página da internet[1].

Esse aperfeiçoamento proporciona as seguintes vantagens:

- Melhora o *design* do *software*;
- Facilita o entendimento do código;
- Facilita a detecção de *bugs*;
- Ajuda a desenvolver o código mais rapidamente.

As quatro etapas principais aplicadas no processo de refatoração consistem em:

- Construir um conjunto sólido de testes para cada seção de código. Os testes são essenciais para detectar erros ocorridos durante esse processo. Refatore o programa em pequenas etapas para facilitar a detecção de erros, caso eles ocorram.
- Decompor os métodos longos. Peças menores de código facilitam o manuseio, sendo mais fácil utilizá-las e reutilizá-las. Portanto decomponha métodos grandes em métodos menores, dessa forma evita-se duplicação de código, otimizando a reutilização.
- Remover variáveis temporárias. Elas são úteis apenas em suas próprias rotinas. Substitua-as por métodos de pergunta, pois esses são acessíveis aos demais membros da classe, proporcionando um *design* limpo sem métodos complexos e longos.
- Substituir operações com múltiplos comandos condicionais por estrutura de classe baseada em polimorfismo. Padrões de Projeto como *State* e *Strategy* podem ser aplicados nessa situação.

1- Disponível em: < http://www.moniro.com/products/services/software-refactoring.html> Acesso em maio 2016.

6. **Projetando para mudanças**

A chave para maximizar a reutilização consiste em antecipar novos requisitos e em projetar sistemas de modo a possibilitar sua evolução.

Para projetar-se um sistema robusto, deve-se analisar possíveis necessidades de mudança ao longo de sua vida. Um projeto que desconsidera essa possibilidade está sujeito ao risco de uma grande reformulação futura.

Mudanças podem envolver redefinições e reimplementações de classes, modificação de clientes e retestagem do sistema. Mudanças não previstas devem ser evitadas, pois geram custos adicionais ao projeto.

7. **Padrões de Projeto**

Padrões de Projeto ajudam a otimizar processos ao garantirem que as mudanças ocorram segundo maneiras específicas. Cada padrão permite algum aspecto da estrutura do sistema variar independentemente de outros, tornando-o mais robusto em relação a um tipo particular de mudança.

Classes fortemente acopladas são difíceis de reutilizar isoladamente, uma vez que dependem umas das outras. O acoplamento forte leva a sistemas monolíticos, nos quais não se pode mudar ou remover uma classe sem compreender e mudar muitas outras.

Acoplamento fraco entre os componentes possibilita às classes serem modificadas e estendidas mais facilmente. Alguns padrões de projeto usam técnicas como acoplamento abstrato e projeto em camadas para obter sistemas fracamente acoplados. O grau de flexibilidade do *software* varia de acordo com o tipo de *software* que se está desenvolvendo.

Há três categorias de Padrões de Projeto: Padrões de Criação, Padrões Estruturais e Padrões Comportamentais. A Tabela 1 exibe a classificação de Padrões de Projeto de acordo com escopo e propósito.

Padrões de Criação abstraem o processo de instanciação. Eles ajudam a tornar um sistema independente de como seus objetos são criados, compostos e representados. Padrão de criação de classe usa herança para variar a classe

instanciada, enquanto que Padrão de Criação de objeto delega a instanciação para outro objeto.

Padrões Estruturais se preocupam com a maneira como classes e objetos são compostos para formar estruturas maiores. Padrões Estruturais de classes utilizam herança para compor interfaces ou implementações. Padrões Estruturais de objetos descrevem maneiras de compor objetos para obter novas funcionalidades.

Padrões Comportamentais se preocupam com algoritmos e atribuições de responsabilidade entre classes e objetos, e descrevem a comunicação entre eles. Padrões Comportamentais de classe utilizam herança para distribuir o comportamento entre classes. Padrões Comportamentais de objetos utilizam composição de objetos em vez de herança.

Tabela 1: Classificação de Padrões de Projeto.

		Propósito		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Fonte: Adaptado de Gamma, Helm, Johnson e Vlissides (2007, p. 26).

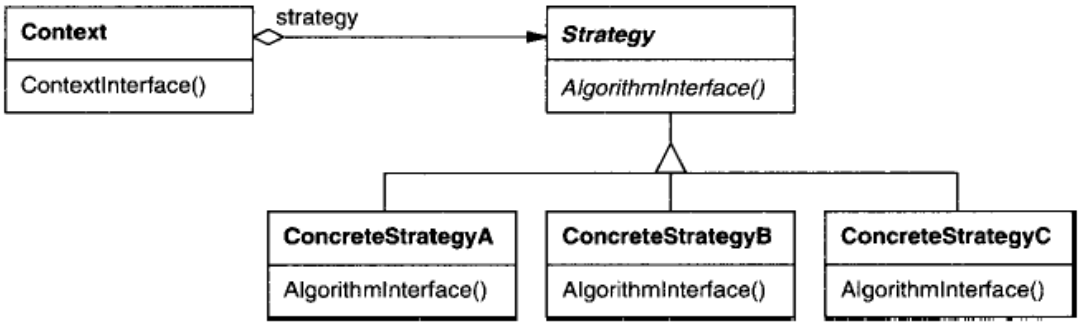
Para exemplificar, a Figura 7 exibe um diagrama contendo o formato do Padrão de Projeto comportamental *Strategy*.

Esse padrão define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. Recomenda-se utilizá-lo quando:

- Muitas classes relacionadas diferem somente no seu comportamento;
- Necessita-se de variantes de um algoritmo;

- Deseja-se encapsular algoritmo dos clientes. Dessa forma, evita-se a exposição das estruturas de dados complexas específicas do algoritmo;
- Uma classe define muitos comportamentos, e estes aparecem em suas operações como múltiplos comandos condicionais da linguagem. Mova os ramos condicionais para suas próprias classes concretas “Strategy”.

Figura 7: Diagrama contendo formato do Padrão de Projeto *Strategy*.



Fonte: Gamma, Helm, Johnson e Vlissides (2007, p. 294).

Padrões de Projeto devem ser utilizados apenas quando a flexibilidade que oferecem é realmente necessária. Antes de aplicá-los, devem-se observar as consequências e avaliar o custo-benefício do uso.

O autor Erich Gamma em seu livro “Padrões de Projeto: Soluções reutilizáveis de *software* orientado a objetos” elaborou um catálogo de Padrões de Projeto e criou o seguinte algoritmo para selecionar o padrão adequado ao problema a ser resolvido:

- Considere como Padrões de Projeto solucionam problemas de projeto;
- Examine a intenção dos padrões;
- Estude como os padrões se inter-relacionam;
- Estude padrões de finalidades semelhantes;
- Examine possíveis causas de reformulação do projeto;
- Considere o que deveria ser modificado futuramente em seu projeto.

Ao escolher um Padrão de Projeto, Erich Gamma sugere a seguinte metodologia para aplicá-lo:

- Leia o padrão por inteiro para obter uma visão geral;
- Estude sua estrutura, seus participantes e suas colaborações;

- Analise um exemplo concreto do padrão codificado;
- Escolha nomes significativos para os participantes do padrão no contexto da aplicação;
- Defina as classes;
- Implemente as operações para suportar as responsabilidades e colaborações presentes no padrão.

Padrões de Projeto constituem uma ferramenta importante nos projetos orientados a objetos. Eles facilitam a reutilização de arquiteturas bem-sucedidas; otimizam a implementação, documentação e manutenção do sistema; reduzem a quantidade de refatorações futuras.

8. **Análise de *Software***

A análise de *software* orientado a objetos possibilita muitas abordagens diferentes. Pode-se escrever a descrição de um problema, separar os substantivos e verbos para criar as classes e operações correspondentes; concentrar-se sobre as colaborações e responsabilidades do sistema; ou ainda, modelar o mundo real e, na fase de projeto, traduzir os objetos encontrados durante a análise.

As duas técnicas mais comuns para a reutilização de funcionalidade em sistemas orientados a objetos são Composição de Objetos e Herança de Classe.

Na Composição de Objetos, funcionalidades complexas são obtidas pela montagem e /ou composição de objetos. Ela requer que os objetos que estão sendo compostos tenham interfaces bem definidas. Como os detalhes internos dos objetos não são visíveis aos demais, não há violação do princípio de encapsulamento. Devido a isso, essa técnica é chamada “reutilização de caixa preta”.

O encapsulamento é importante, em termos de segurança, para proteger os atributos dos objetos de terem seus valores corrompidos por outros objetos; em termos de independência, é importante para proteger outros objetos de complicações de dependência de sua estrutura interna.

Composição de Objetos é definida dinamicamente em tempo de execução pela obtenção de referências a outros objetos. Ela requer que os objetos respeitem as interfaces dos demais, o que exige interfaces cuidadosamente projetadas. Como os

objetos são acessados exclusivamente por meio de suas interfaces, não há violação do princípio do encapsulamento. Qualquer objeto pode ser substituído por outro do mesmo tipo, isto é, com a mesma interface, em tempo de execução.

As principais vantagens e desvantagens da Composição de Objetos são exibidas na Figura 8.

Figura 8: Principais vantagens e desvantagens da Composição de Objetos.

Herança de Classe permite definir classes pela extensão de outras. Ela constitui um mecanismo para compartilhamento de código e de representação. Reutilização por meio de herança permite que o interior da classe base seja visível às classes derivadas. Devido a isso, essa técnica é chamada “reutilização de caixa branca”.

Devido as classes derivadas herdarem a implementação dos métodos da classe base, há limitação da flexibilidade e reusabilidade do código. Uma solução para essa questão é a classe base ser abstrata, de preferência virtual pura.

Herança de Classe é definida estaticamente em tempo de compilação e é simples de usar. Ela facilita modificar a implementação durante o reuso. No entanto, não é possível mudar implementações herdadas das classes ancestrais em tempo de execução.

As principais vantagens e desvantagens da Herança de Classe são exibidas na Figura 9.

Figura 9: Principais vantagens e desvantagens da Herança de Classe.

Idealmente deve-se obter toda a funcionalidade de que se necessita reutilizando-se componentes já existentes por meio da Composição de Objetos. Mas este raramente é o caso, porque o conjunto de componentes disponíveis dificilmente é diversificado o bastante na prática.

A reutilização por Herança de Classe torna mais fácil criar novos componentes que podem ser obtidos pela Composição de outros já existentes. Assim essas duas técnicas trabalham em conjunto na implementação de novas funcionalidades.

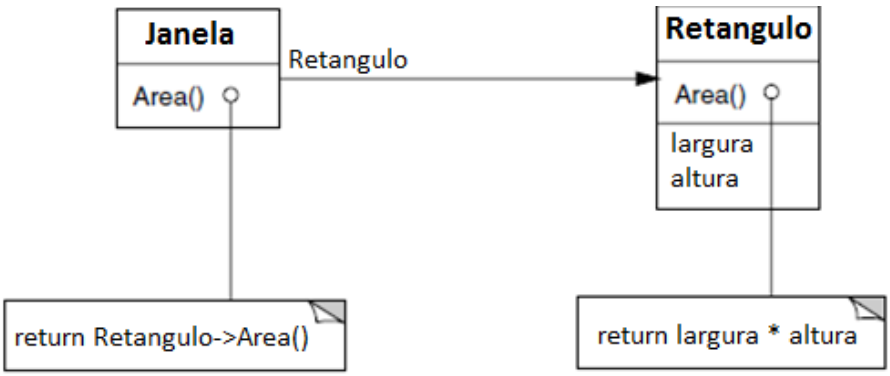
Delegação é uma maneira de tornar a Composição de Objetos tão eficaz para fins de reutilização quanto a Herança de Classes. Na delegação, dois objetos são envolvidos no tratamento de uma solicitação. O objeto receptor delega operações para seu delegado. Isto é análogo à postergação de solicitações enviadas às classes derivadas pela classe base.

A Herança possibilita uma operação herdada se referir ao objeto receptor por meio da variável membro *this*. Na Delegação, o objeto receptor passa a si mesmo ao delegado para permitir a operação referenciar o receptor.

Um exemplo de Delegação é explanado a seguir. Deseja-se implementar o modelo de uma janela retangular e calcular sua área. Em vez de fazer da classe *Janela* uma subclasse de *Retangulo*, a classe *Janela* deve reutilizar o comportamento de

Retangulo, conservando uma variável de instância de *Retangulo* e delegando a ela o comportamento específico, no caso o cálculo da área. Dessa forma, *Janela* deve encaminhar as solicitações para sua instância *Retangulo* explicitamente, conforme indicado na Figura 10. Caso se tivesse trabalhado com herança de classe, a classe *Janela* teria herdado essa operação.

Figura 10: Diagrama contendo lógica da delegação.



Fonte: Adaptado de Gamma, Helm, Johnson e Vlissides (2007, p. 36).

A Delegação apresenta a desvantagem das técnicas que tornam o *software* mais flexível por meio da Composição de Objetos: o *software* dinâmico, altamente parametrizado, é mais difícil de ser compreendido do que o *software* estático baseado em Herança de Classe.

A aplicação da Delegação é aconselhável de acordo com as condições do contexto e com a experiência do programador com seu uso. Recomenda-se utilizá-la em formas altamente estilizadas, isto é, por meio de Padrões de Projeto catalogados.

Outra técnica para reutilização de funcionalidade, não estritamente orientada a objetos, são os Tipos Parametrizados, conhecido em C++ como *Templates*. Essa técnica permite definir um tipo genérico, que suporta todos os demais. Os tipos não especificados são fornecidos como parâmetros no ponto de utilização.

Tipos Parametrizados constituem uma terceira maneira de efetuar a reutilização de comportamentos em sistemas orientados a objetos (além da Herança de Classe e da Composição de Objetos). Os projetos costumam ser implementados usando uma dessas três técnicas ou combinações delas.

A decisão das técnicas a se empregar deve levar em consideração a natureza do *software*, as restrições de implementação e a experiência do programador.

9. *Unified Modeling Language* (UML)

UML é uma linguagem visual que visa modelar sistemas por meio de diagramas, em termos do paradigma de orientação a objeto.

Até o ano de 1996, não havia método padrão para a modelagem de sistemas. Três metodologias principais eram utilizadas pelos desenvolvedores da época: *Object Modeling Technique* (OMT), *Object Oriented Software Engineering* (OOSE) e o Método de Booch.

Visando unificar essas metodologias e definir um padrão, em 1996, foi desenvolvida a primeira versão da UML. Atualmente essa linguagem encontra-se na versão 2.5 e constitui o padrão de modelagem de *software*. Ela é mantida pelo *Object Management Group*.

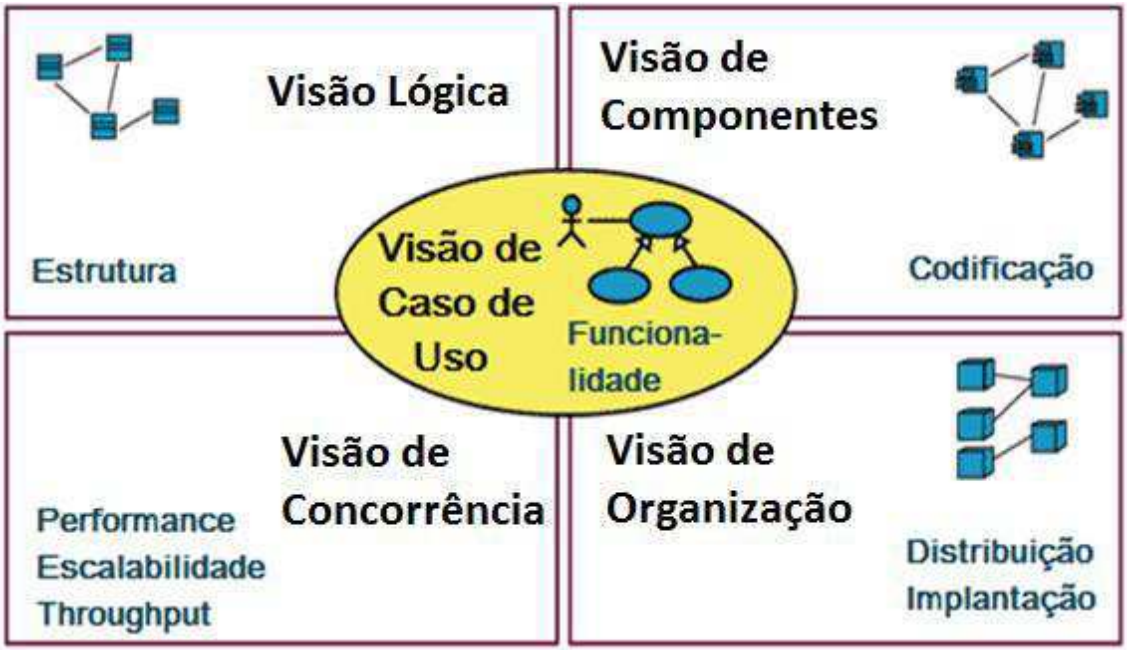
A construção de um *software* engloba requisitos, arquitetura, projeto, código-fonte, planos de projeto, testes, protótipos e versões. Os modelos UML especificam a estrutura e comportamento do sistema, servem como mapa para sua construção e documentação, e atuam no gerenciamento de versões.

Um sistema é composto por aspectos funcionais, não funcionais e organizacionais. Aspectos funcionais representam a estrutura estática e suas interações dinâmicas. Aspectos não funcionais representam requisitos de tempo, confiabilidade e desenvolvimento. Aspectos organizacionais englobam a organização do trabalho e o mapeamento dos módulos de código.

9.1 Visões UML

O esquema de visões tem o objetivo de aumentar a produtividade no processo de desenvolvimento ao fornecer técnicas a serem seguidas pelos membros da equipe. As visões que compõem um sistema são: Visão de Componentes, Visão de Caso de Uso, Visão Lógica, Visão de Organização e Visão de Concorrência. A Figura 11 apresenta esse esquema de visões.

Figura 11: Esquema de Visões.



Fonte: Adaptado de Santos (2009, p. 26).

Para se obter visões de um sistema sob diferentes perspectivas, utilizam-se diagramas. Eles constituem representações gráficas de um conjunto de elementos e definem características, dinâmica, comportamento e estrutura lógica do processo de *software*. Cada diagrama analisa o sistema ou parte dele sob um determinado ponto de vista.

9.1.1 Visão de Casos de Uso

Descreve o comportamento do sistema pelo ponto de vista do usuário final, analista e equipe de teste. Os aspectos estáticos dessa visão são representados pelo Diagrama de Casos de Uso; os aspectos dinâmicos pelo Diagrama de Interação, Diagrama de Máquina de Estados e pelo Diagrama de Atividades.

9.1.2 Visão Lógica

Também denominada Visão de Projeto, descreve as classes e colaborações que compõem o vocabulário do problema e de sua solução. Ela proporciona suporte aos requisitos funcionais que o sistema deverá fornecer aos usuários finais. Os aspectos estáticos dessa visão são representados pelo Diagrama de Classes e Diagrama de Objetos; os aspectos dinâmicos pelo Diagrama de Máquina de Estados, Diagrama de Sequência, Diagrama de Comunicação e pelo Diagrama de Atividades.

9.1.3 Visão de Componentes

Descreve a implementação dos módulos e dependências do sistema; gerencia a configuração das versões do sistema. Os aspectos estáticos dessa visão são representados pelo Diagrama de Componentes.

9.1.4 Visão de Concorrência

Descreve os processos, comunicação e concorrência das linhas de execução de processos paralelos (*threads*) que compõem os mecanismos de concorrência e de sincronização do sistema. Ela engloba questões relativas ao desempenho, escalabilidade e *throughput*.

Escalabilidade de *software* diz respeito à capacidade do sistema assimilar uma carga crescente de conexões ou processos. *Throughput* é uma medida de quantas unidades de informação um sistema pode processar em uma dada quantidade de tempo.

Os aspectos estáticos dessa visão são representados pelos Diagramas de Componentes e pelo Diagrama de Implantação; os aspectos dinâmicos são representados pelo Diagramas de Máquina de Estados, Diagrama de Sequência, Diagrama de Comunicação e pelo Diagrama de Atividades.

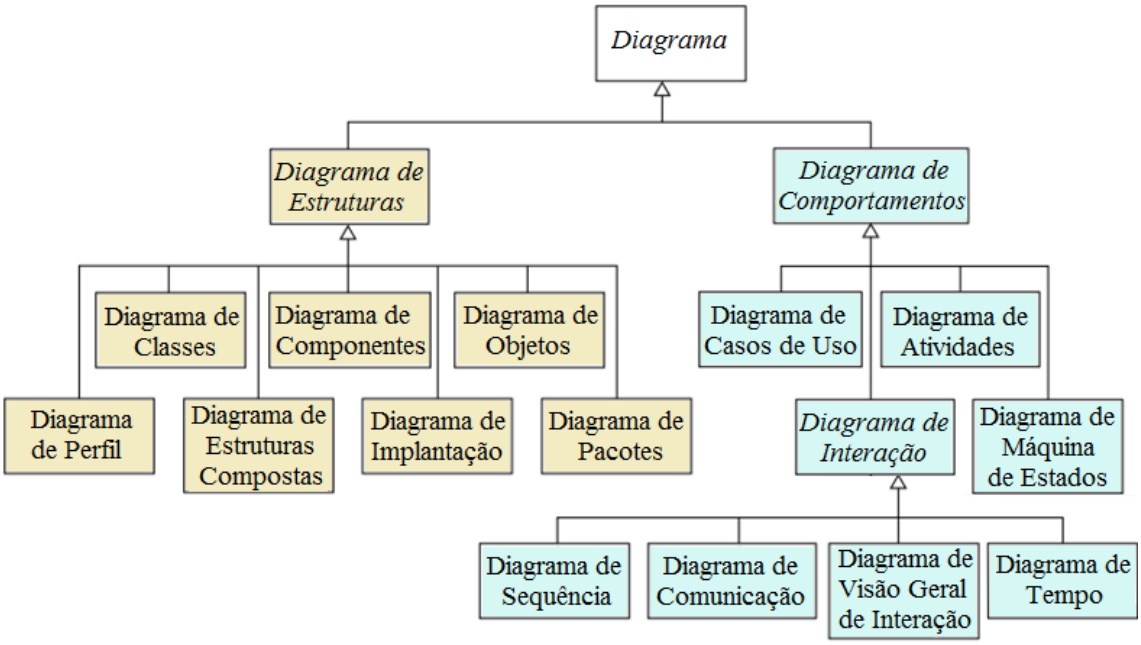
9.1.5 Visão de Organização

Descreve a organização física do sistema, isto é, a topologia do *hardware* em que o sistema é executado. Os aspectos estáticos dessa visão são representados pelos Diagramas de Implantação; os aspectos dinâmicos pelo Diagramas de Máquina de Estado, Diagrama de Interação e pelo Diagrama de Atividades.

9.2 Diagramas UML

A Figura 12 exibe os diagramas UML classificados de estruturais e comportamentais. Diagramas estruturais são estáticos e tem por objetivo modelar a estrutura do sistema. Diagramas comportamentais são dinâmicos e visam modelar o comportamento do sistema.

Figura 12: Diagramas UML classificados de Estruturais e Comportamentais.



9.2.1 Diagrama de Casos de Uso

Diagrama comportamental que visa descrever requisitos funcionais do sistema em termos de atores, casos de uso, relacionamentos e fronteira. Ele descreve o que o sistema faz sem especificar como deve ser feito.

Atores representam papéis desempenhados por elementos externos ao sistema que interagem com ele. Usuário, *software* e *hardware* são exemplos de atores. Eles são modelados graficamente por meio do boneco palito.

Casos de uso representam funcionalidades do sistema. A modelagem gráfica desses elementos é uma elipse e seus nomes devem ser iniciados por verbos.

Elementos do Diagrama de Caso de Uso interagem por meio de relacionamentos. Esses se classificam de associação; generalização; dependência, que pode ser por extensão (*extend*) ou inclusão (*include*).

Relacionamento do tipo associação conecta um ator a um caso de uso. Associações não representam fluxo de informação.

Relacionamento do tipo generalização refere-se ao elemento filho herdar comportamento e significado do elemento pai, válido para atores e casos de uso. O elemento filho pode sobrescrever o comportamento do pai e substituí-lo em

qualquer lugar em que esse apareça. Recomenda-se localizar graficamente o elemento filho abaixo do elemento pai.

Relacionamento de dependência "*extend*" representa uma variação/extensão do comportamento do caso de uso base. O caso de uso estendido só é executado sob certas circunstâncias. Dessa forma, ações *defaults* devem ser associadas ao caso de uso base e as esporádicas ao caso de uso estendido.

Relacionamento de dependência "*include*" evita repetição ao fatorar uma atividade comum a dois ou mais casos de uso.

Fronteira do sistema é um elemento opcional utilizado para definir o escopo do diagrama. Sua notação é feita por meio de um retângulo circunscrito aos casos de uso, mas sem englobar os atores.

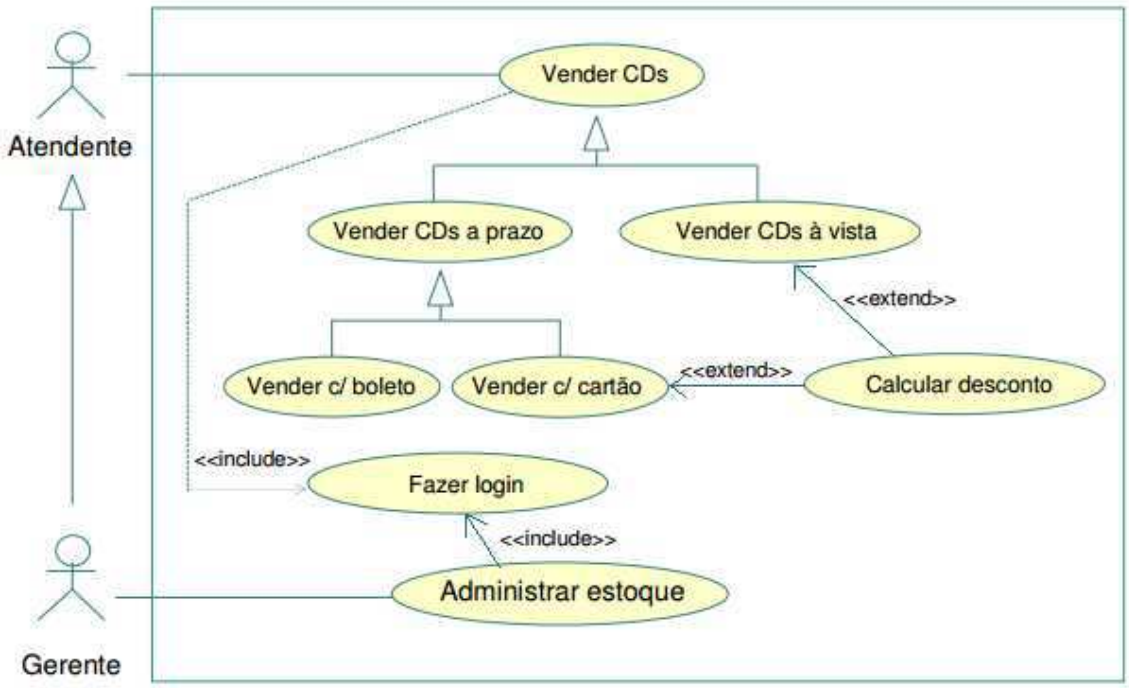
A Figura 13 apresenta um Diagrama de Caso de Uso. Observe que "Atendente" é o ator pai, menos especializado, e "Gerente" é o ator filho, mais especializado. A função do "Atendente" é apenas "Vender CDs", enquanto que o "Gerente" está apto a "Vender CDs", por herança do ator pai, e "Administrar estoque".

O caso de uso "Fazer login" é uma atividade obrigatória para os casos de uso "Vender CDs" e "Administrar estoque", o que justifica o uso do relacionamento de dependência "*include*".

O caso de uso "Calcular desconto" possui duas opções de prosseguimento, ou pelo caso de uso "Vender CDs à vista" ou por "Vender c/ cartão", o que justifica o uso do relacionamento de dependência "*extend*".

Os casos de uso "Vender CDs a prazo" e "Vender CDs à vista" são especializações do caso de uso "Vender CDs", o que justifica o uso do relacionamento do tipo generalização. Processo análogo ocorre com os casos de uso "Vender c/ boleto" e "Vender c/ cartão" em relação ao caso de uso "Vender CDs a prazo".

Figura 13: Diagrama Casos de Uso.



Fonte: Laboratório de Engenharia de *Software*/PUC-Rio, 2008.

Casos de Uso podem ser descritos por meio de um documento. Uma descrição típica deve conter:

- Identificação do ator que iniciou o caso de uso;
- Pré-requisitos (se houver) do caso de uso;
- Descrição textual do fluxo normal e dos fluxos alternativos (se houver);

9.2.2 Diagrama de Máquina de Estados

Diagrama comportamental que visa capturar o ciclo de vida do objeto, seus estados durante sua existência e a transição de estados em resposta a eventos de ativação ou devido a ações internas do código.

Estado é a condição do objeto em um determinado momento. Sua representação gráfica é um retângulo com bordas arredondadas. O estado inicial é representado por um círculo, e o estado final por um círculo dentro de uma circunferência. O diagrama deve possuir um estado inicial e um ou mais estados finais.

Transição é o relacionamento que indica mudança de estado. É representada graficamente por uma linha contínua com uma seta, que parte do estado inicial e

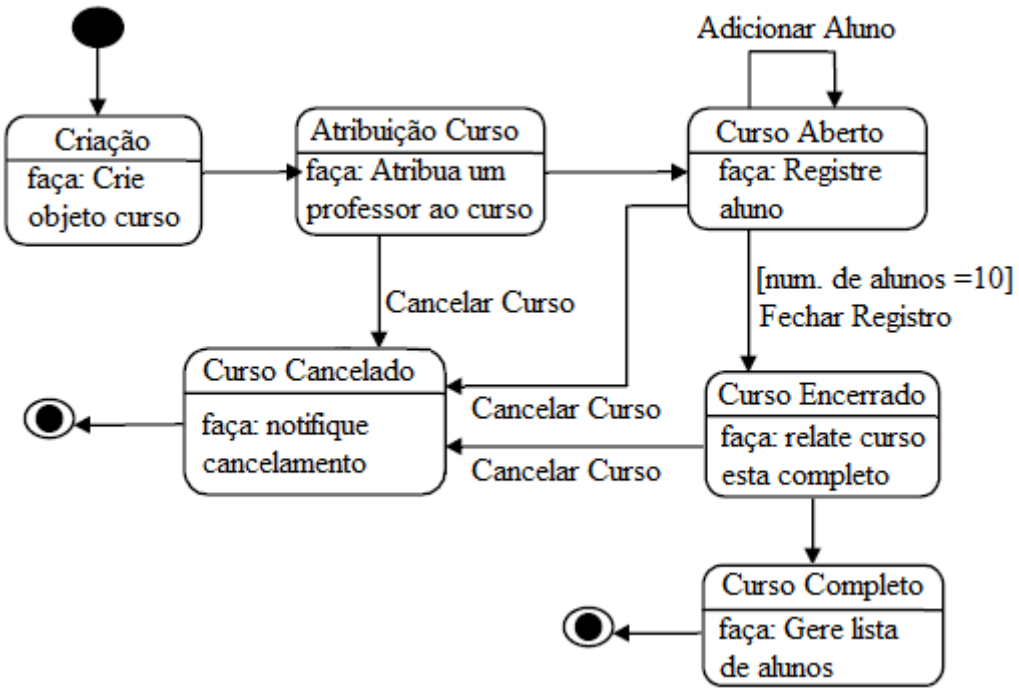
aponta para o estado final. Transições podem ser acionadas por eventos ou por ações internas do código referente ao estado de origem.

Evento de ativação é um estímulo capaz de disparar transição de estado em um objeto. Condição de guarda é a expressão booleana, expressa entre colchetes, que define o disparo do evento de ativação.

É desnecessário elaborar esse diagrama para todas as classes do sistema. Elabora-se para classes que possuem um número definido de estados conhecidos e que se deseja analisar o comportamento relativo a estados e eventos.

A Figura 14 apresenta o Diagrama de Máquina de Estados referente ao objeto "curso". Ele conte um estado inicial e dois estados finais. Os nomes das transições indicam os eventos de disparo. Transições não nomeadas indicam mudança de estado por ações internas do código.

Figura 14: Diagrama de Máquina de Estados.

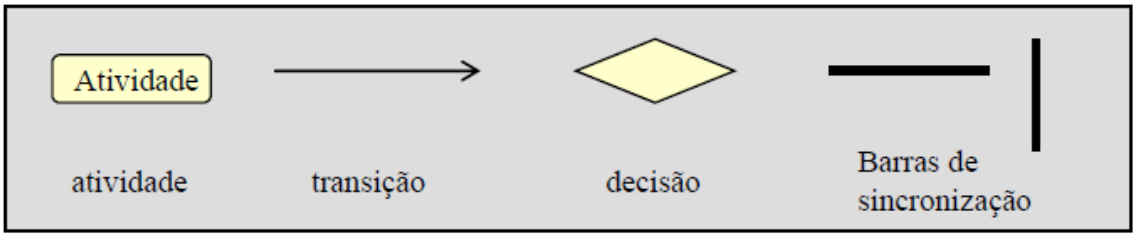


Fonte: Adaptado de Santos (2009, p. 69).

9.2.3 Diagrama de Atividades

Diagrama comportamental que visa modelar o fluxo sequencial das atividades. Elementos gráficos desse diagrama são exibidos na Figura 15. Losango representa notação de decisão. Barras de sincronização representam a execução de fluxos concorrentes ou paralelos. Elas podem ser barras de bifurcação, um fluxo de entrada e dois ou mais de saída; ou de união, dois ou mais fluxos de entrada e um de saída.

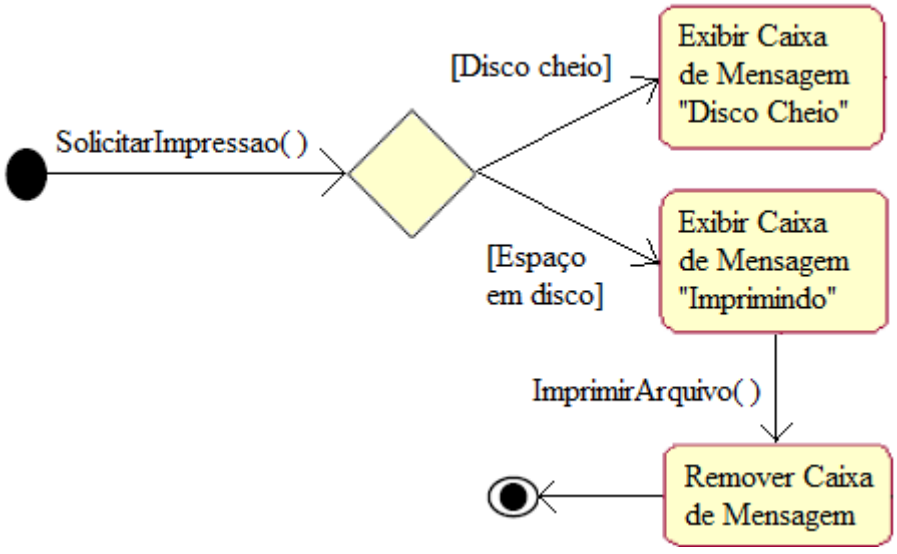
Figura 15: Elementos do diagrama de atividades.



Fonte: Santos (2009, p. 76).

A Figura 16 exibe um Diagrama de Atividade aplicado à impressão de um arquivo. Ele possui um estado inicial e dois estados finais. Os nomes presentes nas transições entre colchetes indicam condições para realização das atividades (condições de guarda), e os demais indicam ações para realização das atividades.

Figura 16: Diagrama de Máquina de Estados aplicado à impressão de arquivo.

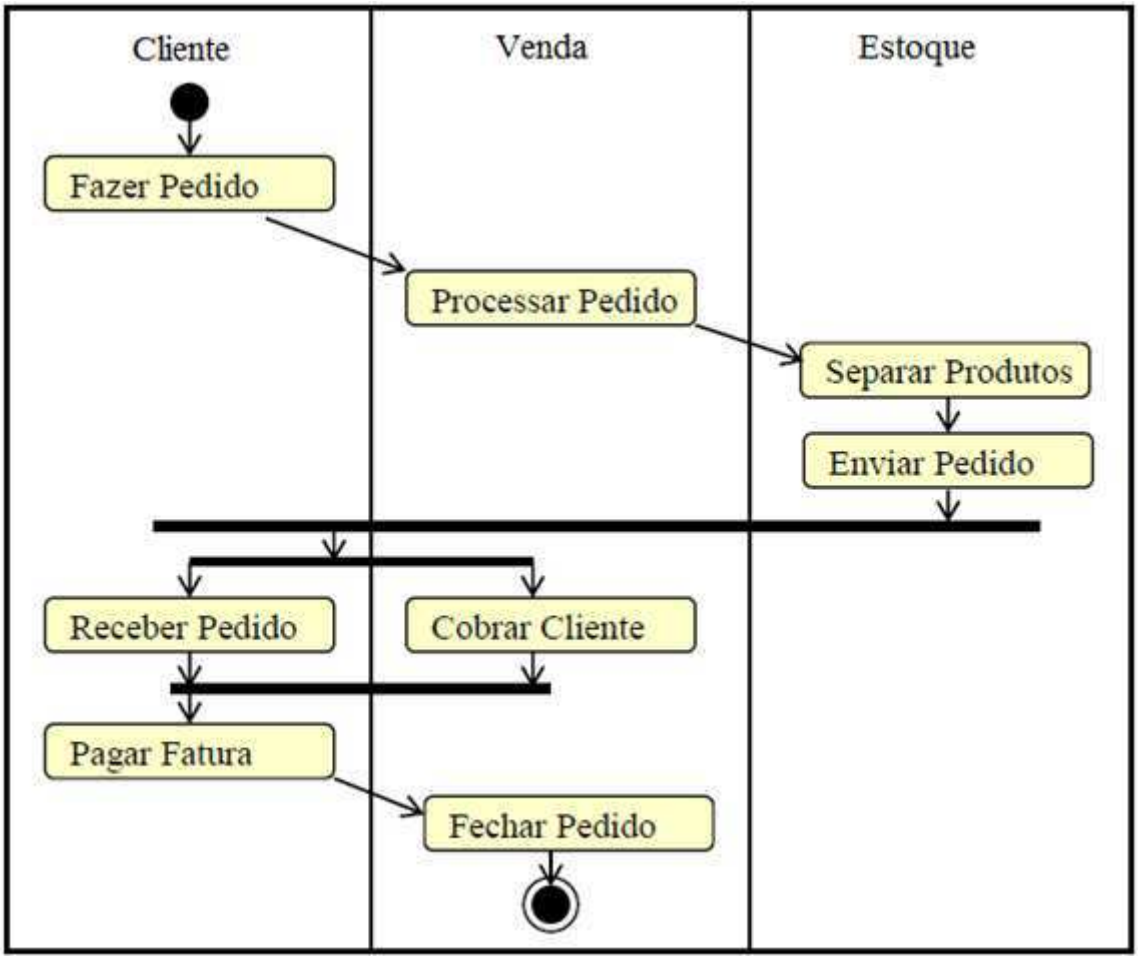


Fonte: Adaptado de Santos (2009, p. 75).

O Diagrama de Atividades também pode ser representado utilizando o elemento raia (*Swin Lane*), conforme exibido na Figura 17. Raias definem a responsabilidade na

execução das atividades e são representadas graficamente por retângulos que definem o escopo das atividades em relação aos objetos. Cada atividade pertence a uma única raia, mas as transições podem cruzá-las.

Figura 17: Diagrama de Atividades aplicado a processo de venda.



Fonte: Adaptado de Santos (2009, p. 76).

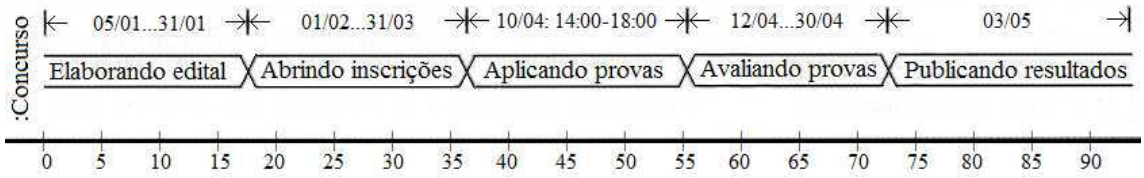
9.2.4 Diagrama de Tempo

Diagrama comportamental que deriva do Diagrama de Interação. Ele apresenta o comportamento do objeto e sua interação em uma escala temporal, com foco nas condições de mudança no decorrer desse período.

A Figura 18 exibe um exemplo desse diagrama aplicado em um processo de concurso. As atividades relativas ao objeto “Concurso” são descritas numa escala

temporal com suas respectivas durações. Os pontos de interseção indicam mudança de estado/atividade.

Figura 18: Diagrama de Tempo.



Fonte: Adaptado de Guedes (2011, p. 352).

9.2.5 Diagrama de Sequência

Diagrama comportamental que deriva do Diagrama de Interação. Ele visualiza a interação com ênfase no decorrer do tempo e exibe os objetos participantes em termos de suas linhas de vida e mensagens trocadas.

Os eixos verticais, representados por linhas pontilhadas, são denominados “linhas de vida”. Eles determinam o tempo de vida dos objetos, da parte superior para a inferior. Os objetos envolvidos na realização da atividade retratada são exibidos na horizontal.

Objetos interagem por meio de mensagens, que podem conter números para explicitar a sequência do diagrama. Condições para envio das mensagens denominam-se condições de guarda e são expressas entre colchetes.

Mensagens especificam o emissor e o receptor, e definem o tipo de comunicação que ocorre entre as linhas de vida. Elas representam ações de chamada de operação, retorno de valor para objeto solicitante, e ainda criação ou destruição de objeto.

Mensagens podem ser classificadas de síncrona, assíncrona ou de retorno. A Figura 19 exibe a classificação e os referentes símbolos.

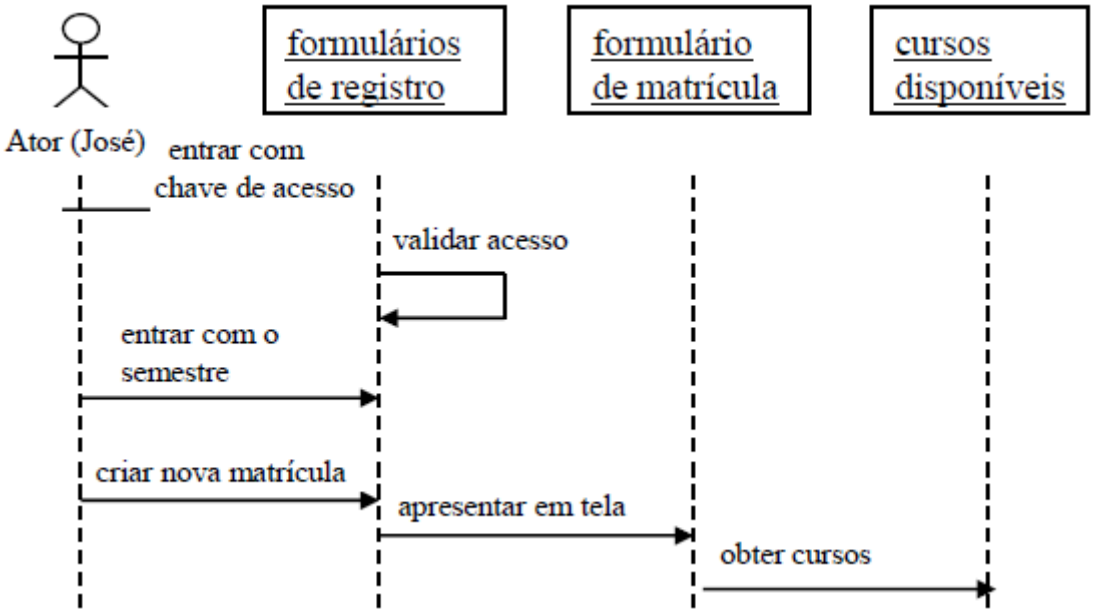
Figura 19: Tipos de mensagem no Diagrama de Sequência.

Símbolo	Significado
	Mensagem síncrona
	Mensagem assíncrona
	Mensagem de retorno (opcional)

Chamadas síncronas associadas a uma operação possuem mensagem de envio e de recebimento. A linha de vida de origem que envia a mensagem fica bloqueada para outras operações até receber a mensagem de retorno da linha de vida de destino.

No diagrama da Figura 20, por exemplo, a operação “criar nova matrícula” só poderá ser realizada após a operação “entrar com o semestre” ser finalizada, pois a linha vida referente ao objeto José estará bloqueada até o término dessa operação.

Figura 20: Diagrama de Sequência.

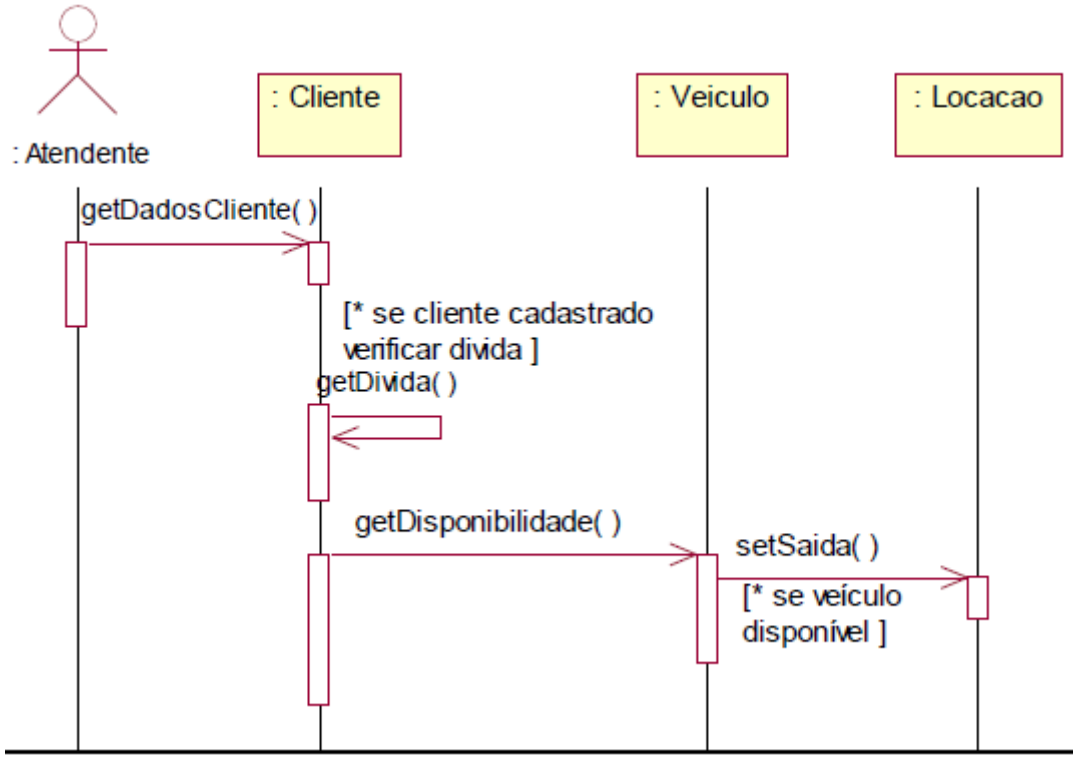


Fonte: Santos (2009, p. 63).

Chamadas assíncronas costumam conter apenas a mensagem de envio, mas é possível haver mensagens de resposta. Nessas chamadas, a linha vida de origem fica desbloqueada para outras operações.

No diagrama da Figura 21, por exemplo, a operação “getDisponibilidade()”, que verifica se o veículo solicitado está disponível para aluguel, pode ser realizada antes do término da operação “getDivida()”, que verifica se o cliente possui dívida com a empresa.

Figura 21: Diagrama de Sequência.



Fonte: Santos (2009, p. 65).

Mensagens de recebimento e de resposta são classificadas de mensagens de retorno, cuja exibição é opcional. Essas mensagens foram ocultadas nos diagramas da Figura 20 e da Figura 21.

Para construir um Diagrama de Sequência, as seguintes etapas devem ser seguidas:

- Escolha um caso de uso;
- Identifique os objetos participantes da interação;
- Identifique o objeto que inicia interação;

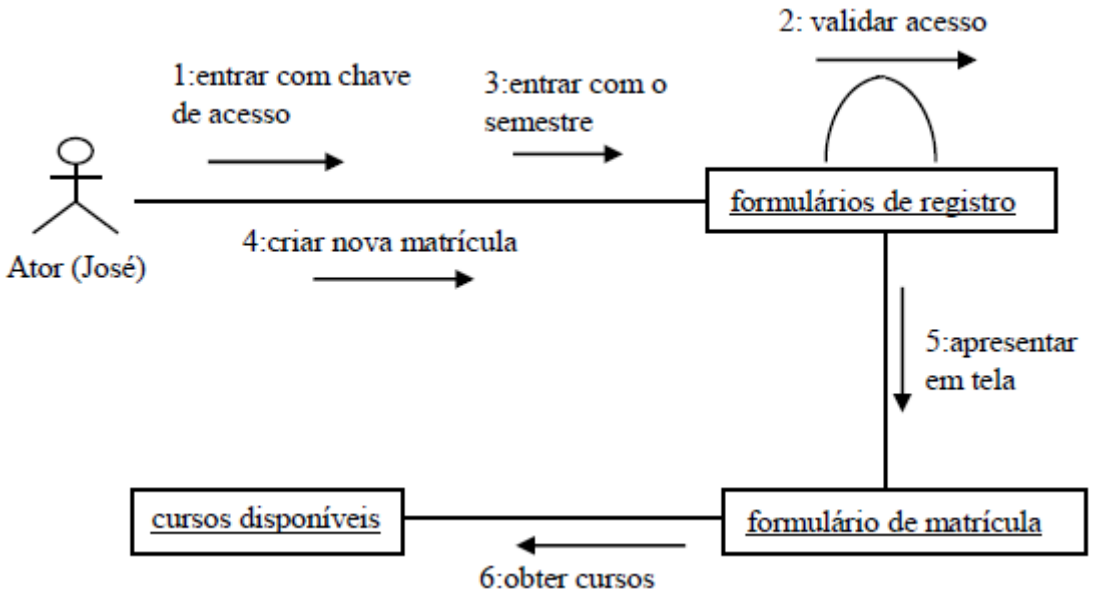
- Identifique as mensagens trocadas entre os objetos;
- Identifique a sequência das mensagens trocadas;

9.2.6 Diagrama de Comunicação

Diagrama comportamental que deriva do Diagrama de Interação. Ele visualiza a interação entre objetos com ênfase no contexto do sistema.

A Figura 22 exibe esse diagrama no contexto de realização de matrícula. A numeração indica a sequência do fluxo de mensagem. “Ator”, “formulários de registro”, “formulário de matrícula” e “cursos disponíveis” representam os objetos presentes. As setas representam as mensagens de comunicação entre eles.

Figura 22: Diagrama de Comunicação.



Fonte: Santos (2009, p. 67).

9.2.7 Diagrama de Visão Geral de Interação

Diagrama comportamental que deriva do Diagrama de Interação. Costuma englobar diversos diagramas de interação para representar um processo geral, podendo ser de tipos distintos.

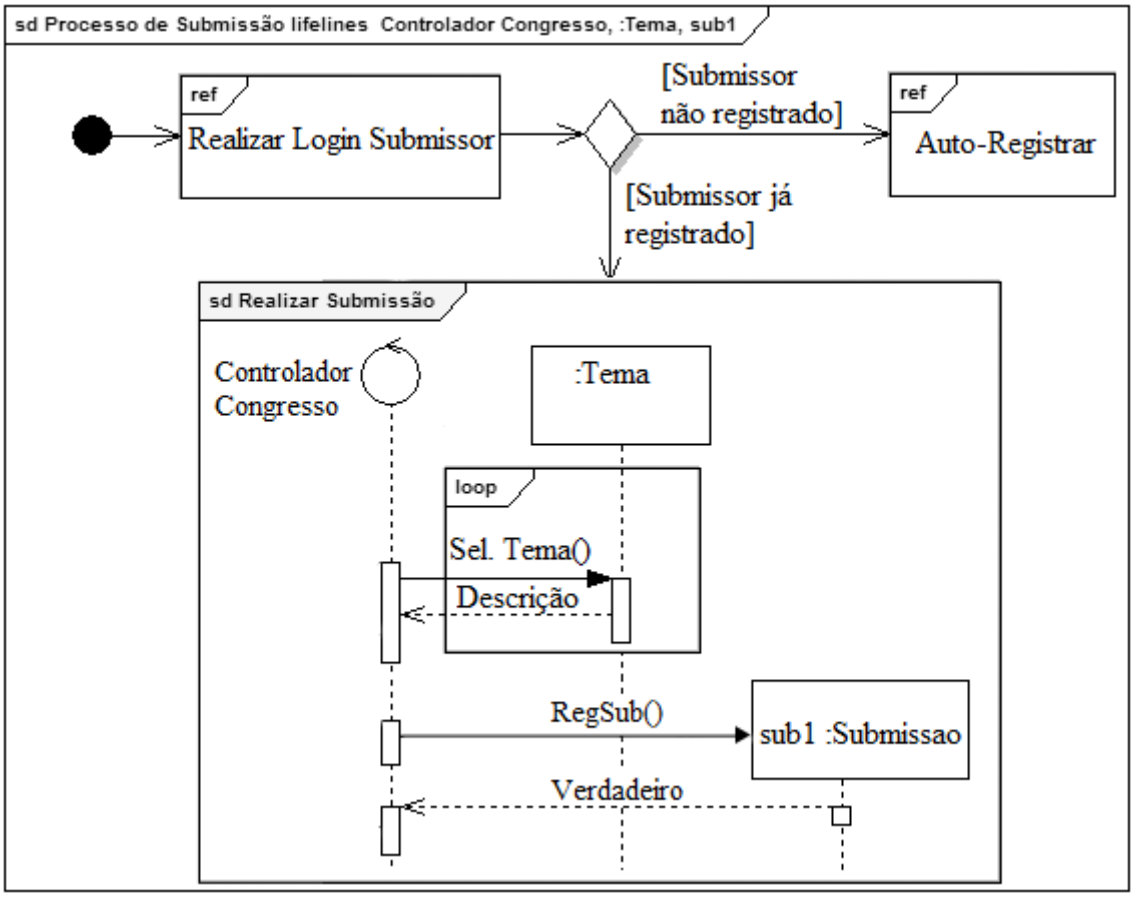
A Figura 23 apresenta esse diagrama aplicado a um processo de submissão. O elemento “*label*” circunscreve os demais elementos. Na parte superior, tem-se a

identificação composta pela sigla “sd”, que representa o diagrama do tipo “*Sequence Diagram*”, pelo nome do diagrama “Processo de Submissão” e pelas linhas de vida referentes às classes “Controlador Congresso”, “Tema” e “Submissao”.

O símbolo utilizado no elemento “Controlador Congresso” tem a finalidade de validação e controle. Ele é responsável por coordenar e sequenciar outros objetos, no caso do diagrama da Figura 23 os objetos das classes “Tema” e “Submissao”.

Observe que o Diagrama de Visão Geral de Interação exibido na Figura 23 contém os diagramas de sequência “Processo de Submissão”, “Realizar submissão”, “Realizar Login Submissor” e “Auto-Registrar”. A sigla “ref” presente no *label* desses dois últimos refere-se ao nome do diagrama de sequência inserido na área de conteúdo. Os diagramas são conectados por meio de um diagrama de atividades.

Figura 23: Diagrama de Visão Geral de Interação.



Fonte: Adaptado de Guedes (2014, p. 26).

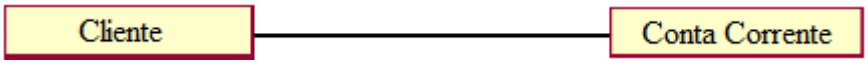
9.2.8 Diagrama de Classes

Diagrama estrutural que exibe um conjunto de classes e seus relacionamentos. Um sistema costuma ser representado por alguns Diagramas de Classe. A mesma classe pode estar presente em mais de um desses diagramas. Classes são representadas graficamente por retângulos, que incluem nome, atributos e métodos. Elas devem receber nomes significativos, de preferência no singular com inicial maiúscula.

Relacionamentos classificam-se de “Associação”, “Generalização” e “Dependência”. Eles podem possuir indicadores de multiplicidade, nome, sentido de leitura e navegabilidade (seta no fim do relacionamento).

Associação indica o vínculo entre objetos de classes distintas e é representada graficamente por uma linha sólida que conecta duas classes. A Figura 24 exemplifica esse relacionamento. Nela o objeto da classe “Cliente” possui objeto da classe “Conta Corrente”.

Figura 24: Exemplo de relacionamento do tipo Associação.



Agregação é um tipo especial de associação que indica uma classe estar contida ou ser parte de outra classe. A Figura 25 exemplifica esse relacionamento. Nela, um objeto da classe “Marinha” contém vários objetos da classe “Navio”.

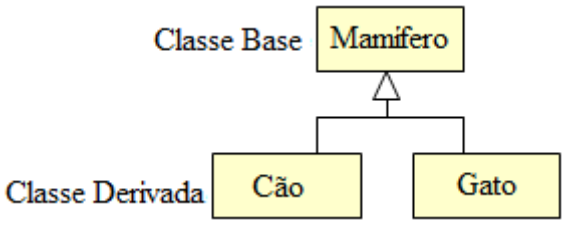
Figura 25: Exemplo de relacionamento do tipo Agregação.

Composição é uma variação de agregação. Nela os objetos “parte” só podem pertencer a um único objeto “todo” e seus tempos de vida coincidem com o dele. A Figura 26 exemplifica esse relacionamento.

Figura 26: Exemplo de relacionamento do tipo Composição.

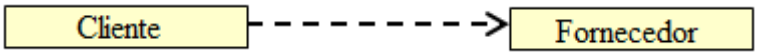
Relacionamento do tipo generalização refere-se à classe derivada herdar comportamento e significado da classe base. Ele é exemplificado na Figura 27.

Figura 27: Exemplo de relacionamento do tipo Generalização.



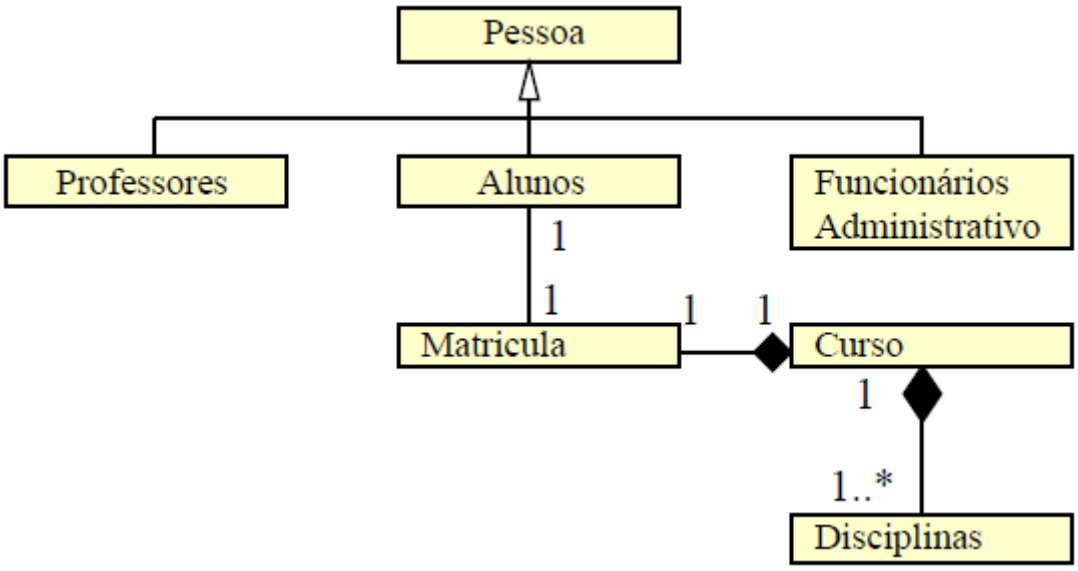
Relacionamento de dependência indica que alteração no objeto independente pode afetar o objeto dependente. A Figura 28 exemplifica esse relacionamento. Nela a classe “Cliente” depende de algum serviço da classe “Fornecedor”, logo a mudança de estado de um objeto dessa afeta o objeto daquela.

Figura 28: Exemplo de relacionamento do tipo Dependência.



A Figura 29 apresenta exemplo desse diagrama contendo relacionamentos de Generalização, Composição e Associação.

Figura 29: Diagrama de Classes.



Fonte: Santos (2009, p. 80).

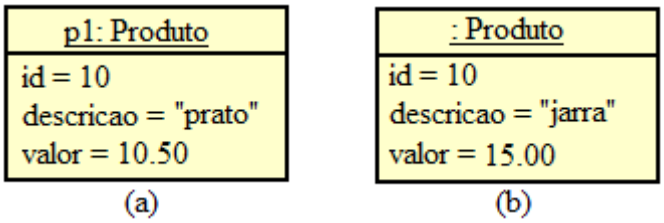
9.2.9 Diagrama de Objetos

Diagrama estrutural que exibe o estado e os vínculos entre os objetos em determinado momento da execução. A Figura 30 apresenta dois objetos da classe “Produto”.

Na parte superior, insere-se o nome do objeto seguido por “:” e pelo nome da classe referente. Na Figura 30 (a), o objeto foi nomeado como “p1”. Na Figura 30 (b), o nome do objeto foi omitido, sendo classificado de objeto anônimo.

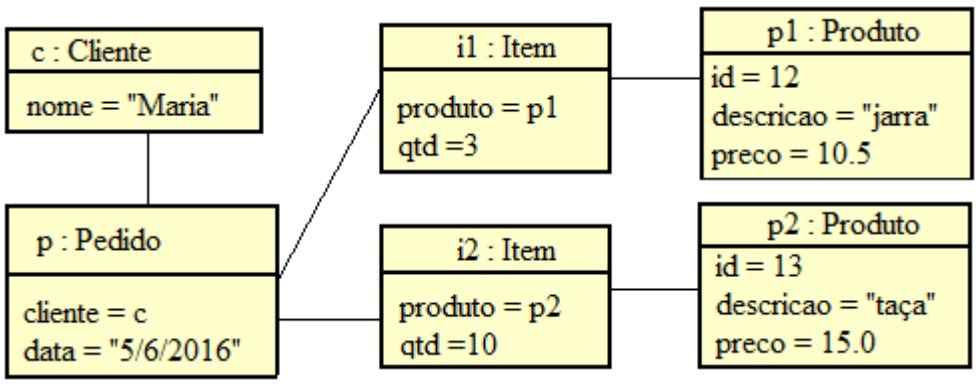
Na parte inferior, insere-se os atributos e seus valores correspondentes em um determinado momento da execução.

Figura 30: (a) Objeto nomeado. (b) Objeto anônimo.



A Figura 31 apresenta um exemplo desse diagrama. Ele exibe como objetos das classes “Cliente”, “Pedido”, “Item” e “Produto” se relacionam, bem como os valores de seus atributos em determinado instante da execução. Observe que o Diagrama de Objetos exibe os vínculos entre objetos sem especificar os tipos de relacionamento entre eles.

Figura 31: Diagrama de Objetos.



Fonte: Adaptado de Carvalho (2010, p.17).

9.2.10 Diagrama de Componentes

Diagrama estrutural que descreve os componentes de *software* e suas dependências. Componente de *software* é uma parte física do sistema que corresponde à realização de um conjunto de classes e/ou interfaces.

Componente é representado graficamente por uma das notações da Figura 32, sendo a notação mais atualizada a presente na lateral direita dessa figura. Código fonte, biblioteca e arquivos binários, executáveis e de textos são exemplos de componentes. Dependência entre componentes é representada por meio de uma linha tracejada com uma seta na ponta.

Componentes podem definir interfaces visíveis aos demais. Componentes que se relacionam com outros por meio de interfaces devem depender apenas das interfaces, e não das implementações. A interface é representada por uma linha contendo um círculo na extremidade, com o nome junto ao círculo. A Figura 33 apresenta um exemplo desse diagrama contendo componentes, dependências e interface.

Figura 32: Notação de componentes em UML.

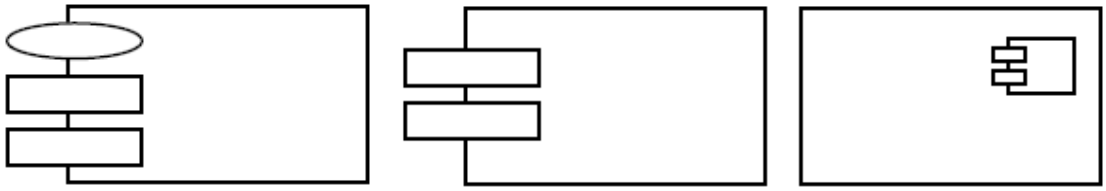
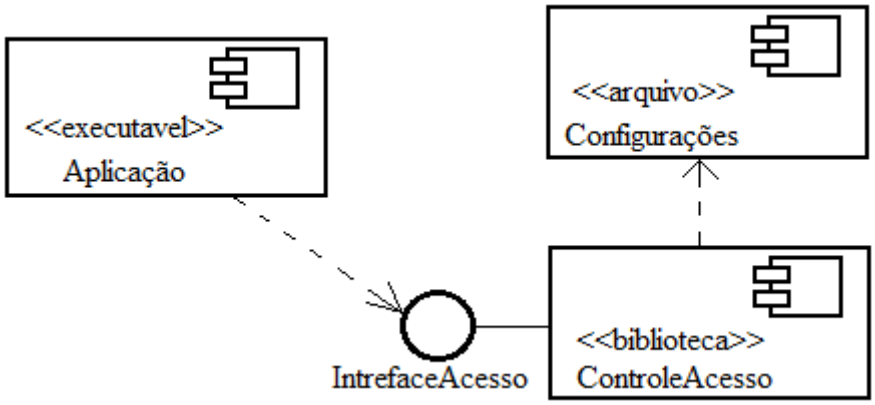


Figura 33: Diagrama de Componentes.



Fonte: Página da internet[2].

2- Disponível em: <http://www.alvarofpinheiro.eti.br/tags/%C3%81lvaro%20Pinheiro/> Acesso maio de 2016.

9.2.11 Diagrama de Estruturas Compostas

Diagrama estrutural que visa modelar a estrutura interna de um classificador por meio de peças, portas e conectores.

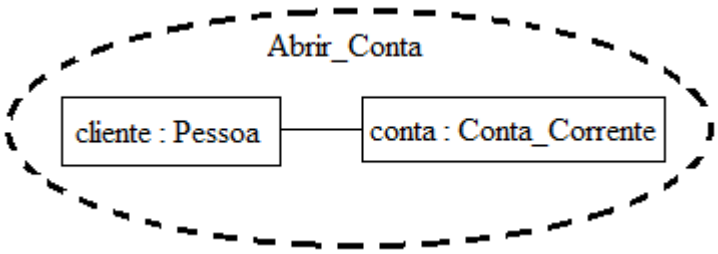
Classificador é um mecanismo que descreve características comportamentais e de estrutura na UML. Ele pode representar classes, interfaces, tipos de dados, sinais, nós, casos de uso, subsistemas, colaborações.

Peças constituem conjuntos de entidades cooperativas (instâncias) contidas no classificador. Peças são representadas graficamente por retângulos. Portas definem pontos de interação entre classificador e o ambiente, entre classificador e peças internas, ou ainda entre peças internas do classificador. Portas são representadas graficamente por quadrados. Conectores expressam os relacionamentos no modelo. Conectores são representados graficamente por linhas sólidas.

Colaboração constitui um tipo de classificador que define um conjunto de peças e conexões necessários para executar uma tarefa. A representação gráfica desse elemento é uma elipse pontilhada contendo um nome. Uma colaboração pode conter outras dentro de si.

A Figura 34 apresenta um Diagrama de Estruturas Compostas contendo colaboração, peças e conector. Nesse exemplo, “cliente”, instância da classe “Pessoa”, interage com “conta”, instância da classe “Conta_Corrente” para abrir uma nova conta.

Figura 34: Diagrama de Estruturas Compostas contendo Colaboração.

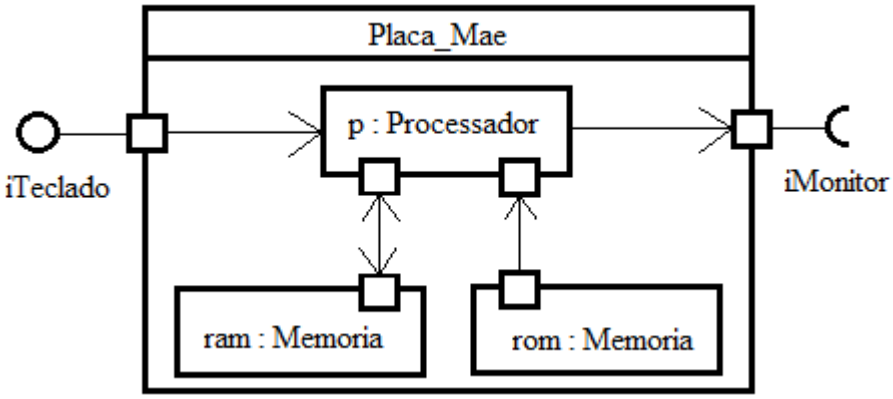


Fonte: Adaptado de Guedes (2011, p.40).

Propriedade constitui um conjunto de peças contidas na instância de um classificador contêiner. A Figura 35 apresenta um Diagrama de Estruturas Compostas contendo peças, conectores, interfaces e propriedade.

Nesse exemplo a estrutura “Placa_Mae” é composta pela peça “p”, instância da classe “Processador”, e pelas peças “ram” e “rom”, instâncias da classe “Memoria”. As peças se comunicam entre si e com o classificador contêiner por meio de conexões entre as portas. O classificador contêiner se relaciona com dispositivos presentes no ambiente por meio de interfaces, “iTeclado” e “iMonitor”. A notação circunferência indica interface fornecida ao classificador, e a notação meia-circunferência indica interface requerida pelo classificador.

Figura 35: Diagrama de Estruturas Compostas contendo Propriedade.



Fonte: Adaptado de Guedes (2011, p.350).

9.2.12 Diagrama de Perfil

Diagrama estrutural que visa definir novos modelos UML. Ele permite estender os diagramas existentes com a inclusão de estruturas customizadas para atender uma determinada necessidade. Esse diagrama permite que a UML se adapte a plataformas, domínios e metodologias de desenvolvimento de *software* para os quais não foi projetada originalmente.

Diagrama de Perfil é composto por um ou mais perfis UML. Perfil UML constitui mecanismo de extensão do padrão UML. Ele pode definir classes, estereótipos, tipos de dados, tipos primitivos, enumerações; e pode reutilizar outro perfil, ou partes desse, para estender perfis já existentes. A notação utilizada para perfil é

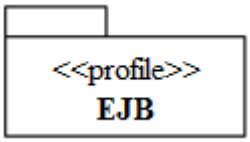
composta pela palavra chave <<profile>> e por um nome. Um exemplo de notação é exibido na Figura 36.

Meta Classe é uma classe definida por um perfil e encapsulada, de forma a poder ser estendida por um estereótipo. Estender refere-se ao relacionamento extensão, que deriva do relacionamento associação. Ele conecta um estereótipo a uma Meta Classe e é representado graficamente por uma seta com ponta preenchida.

Estereótipo é uma classe definida por um perfil que define como uma Meta Classe pode ser estendida. Ele não pode ser utilizado sozinho, deve-se utilizá-lo com uma ou mais Meta Classes por ele estendidas. Estereótipos podem alterar a aparência gráfica de elementos estendidos por meio de ícones personalizados, cujo relacionamento ocorre por meio de composição.

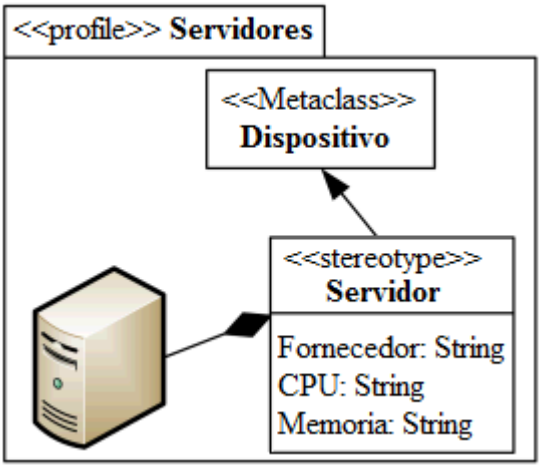
A Figura 37 exibe um Diagrama de Perfil. Nele, a meta classe "Dispositivo" é estendida pelo estereótipo "Servidor", que se relaciona com ícone personalizado.

Figura 36: Notação do elemento Perfil.



Fonte: Página da internet[3].

Figura 37: Diagrama de Perfil.



Fonte: Página da internet[3].

3- Disponível em: <http://www.uml-diagrams.org/profile-diagrams.html> Acesso em maio de 2016.

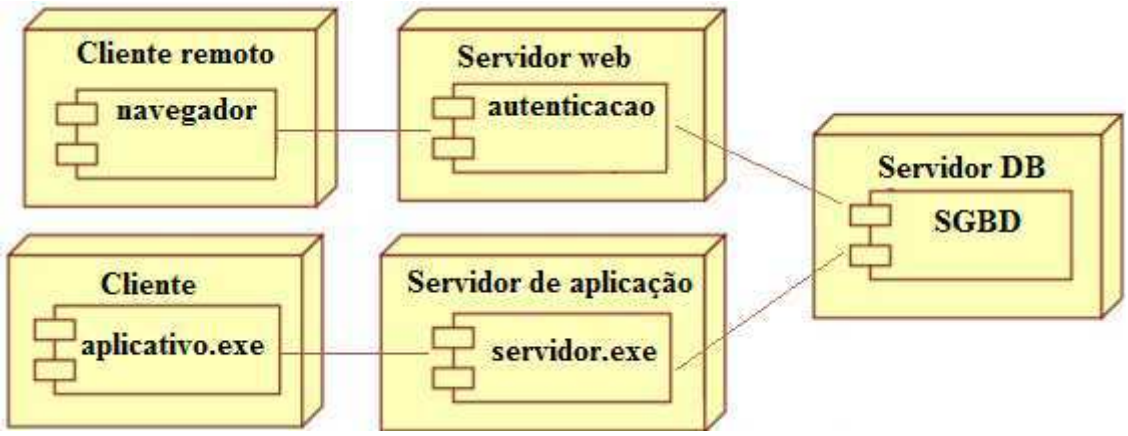
9.2.13 Diagrama de Implantação

Diagrama estrutural que visa exibir a arquitetura física do *hardware* e do *software* do sistema. Ele pode conter componentes, nós e conexões. Componentes representam partes físicas do sistema que correspondem à realização de um conjunto de classes e/ou interfaces. Representações gráficas de componentes são exibidas na Figura 32.

Nós representam elementos físicos que compõem o sistema. Computador cliente, computador servidor, impressora e roteador são exemplos de nós. Eles são graficamente representados por cubos. Conexões interligam nós e componentes e são graficamente representadas por linhas.

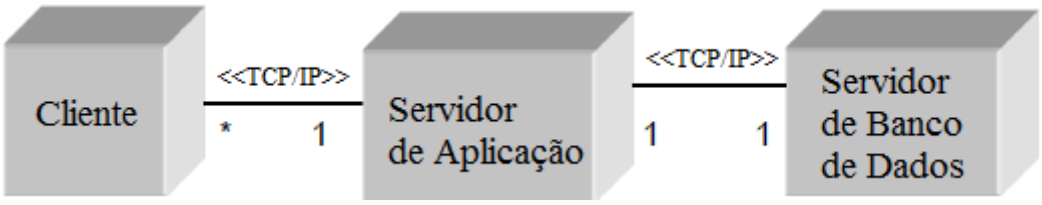
A Figura 38 apresenta um Diagrama de Implantação contendo nós, componentes e conexões. A Figura 39 apresenta um Diagrama de Implantação contendo apenas nós e conexões. Acima das conexões, indica-se os protocolos utilizados.

Figura 38: Diagrama de Implantação.



Fonte: Site SlideShare[3].

Figura 39: Diagrama de Implantação.



Fonte: Santos (2009, p.88).

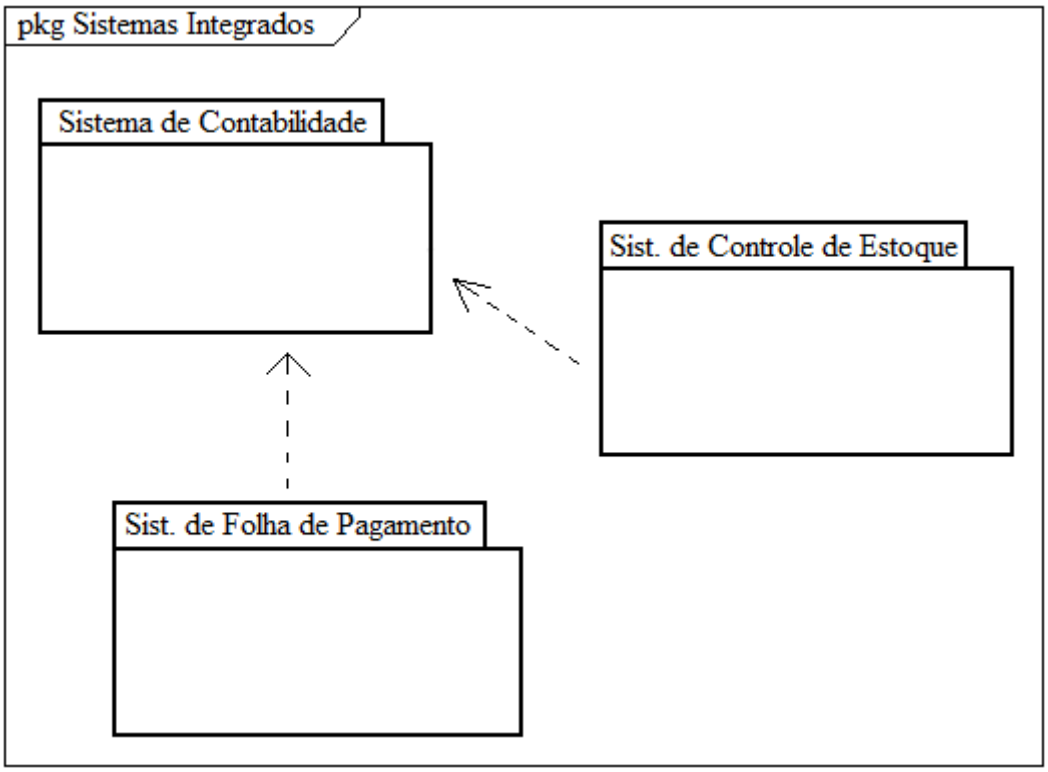
9.2.14 Diagrama de Pacotes

Diagrama estrutural que visa agrupar classes em pacotes; decompor sistema em subsistemas; decompor sistema em camadas. Esse diagrama é composto por pacotes e relacionamentos. A notação gráfica de pacote é um diretório contendo um nome. Os relacionamentos entre pacotes costumam ser de dependência, expressos graficamente por uma linha pontilhada com uma seta.

O critério para definição de pacotes é subjetivo e depende da visão e necessidades do projetista. Um pacote deve agrupar elementos similares que tendem a ser modificados em conjunto.

A Figura 40 exibe um Diagrama de Pacotes. Nele há três sistemas integrados: Sistema de contabilidade, sistema de folha de pagamento e sistema de controle de estoque. Os relacionamentos entre os pacotes indicam que “Sist. de Controle de Estoque” e “Sist. de Folha de Pagamento” necessitam do “Sistema de Contabilidade”, por exemplo, para lançar suas operações financeiras.

Figura 40: Diagrama de Pacotes.



Fonte: Guedes (2011, p.189).

10. Considerações Finais

Numa situação ideal, o *software* é projetado antes de ser implementado. Nesses casos, deve-se inicialmente realizar o projeto, que inclui a possível aplicação de padrões de projeto, e documentar o *software* por meio de diagramas UML. Posteriormente, implementa-se o código aplicando-se boas práticas de programação, visando mitigar futuros refatoramentos, e documenta-se o mesmo. Documentação de código refere-se a inserir comentários para explicar parâmetros, retornos e funcionalidades dos elementos, tais como classes, métodos, *structs*, etc. Algumas linguagens, como por exemplo Java, possuem mecanismo automático para gerar essa documentação a partir do código-fonte.

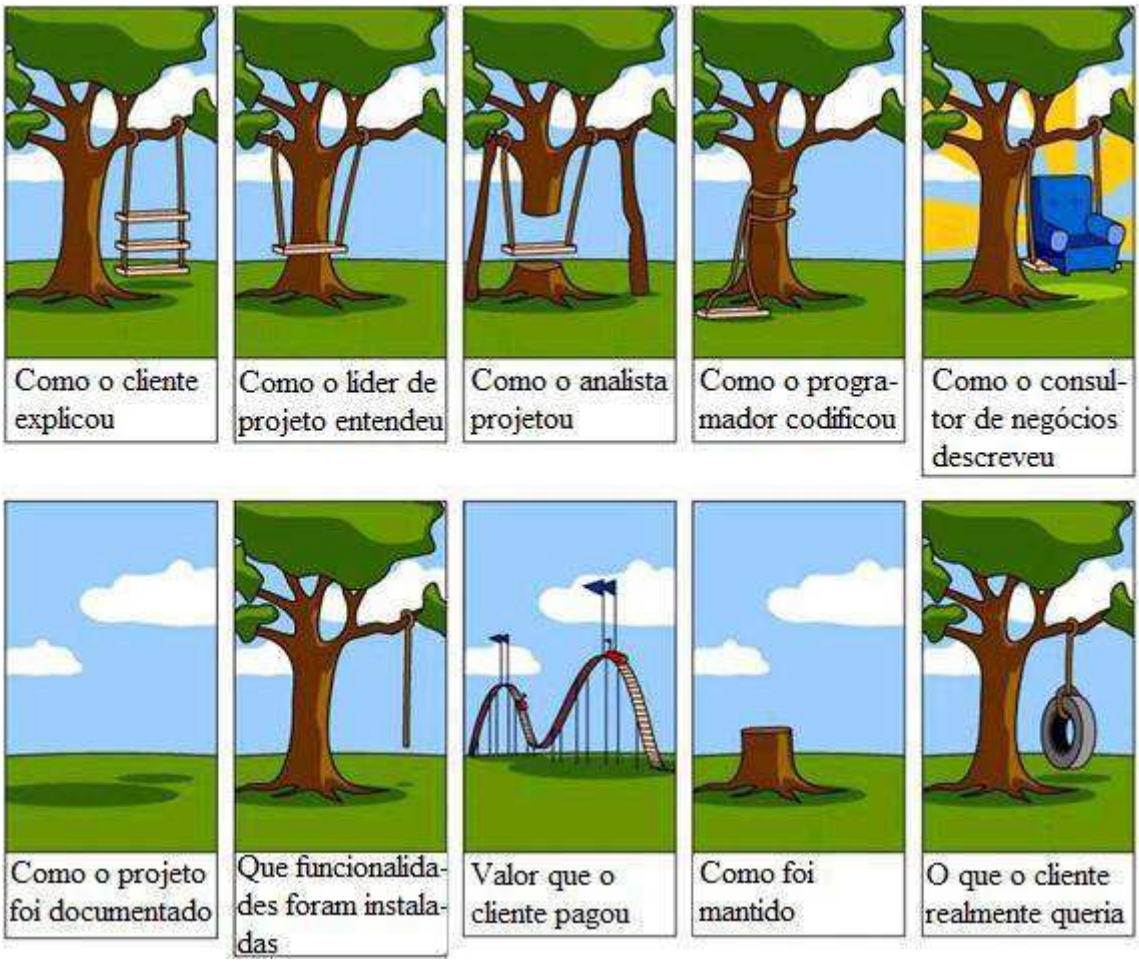
No entanto, o código as vezes é implementado sem ter sido projetado. Ao se deparar esses casos, deve-se primeiramente documentar, por meio de diagramas UML, o *design* atual do código. Recomenda-se utilizar nessa atividade diagramas de pacotes, de classes, de objetos ou simplesmente um diagrama de blocos caso o código não seja baseado em classes ou objetos. Em seguida, deve-se projetar o Diagrama de Casos de Uso de acordo com as funcionalidades que o sistema deve possuir. Com base nessa documentação realizada, verifica-se quais elementos devem ser refatorados e analisa-se o custo-benefício no uso de padrões de projeto. Deve-se então projetar o *software*, aplicando os refatoramentos e padrões adotados. O projeto deve conter diagramas estruturais para embasar a recodificação, por exemplo Diagrama de Classes, Diagrama de Objetos, Diagrama de Pacotes; e diagramas comportamentais para facilitar a compreensão do sistema, por exemplo Diagrama de Atividades, Diagrama de Sequência, Diagrama de Máquina de Estados. Ao final, deve-se recodificar o *software* baseado no projeto realizado e, ao longo desse processo, documentar o código.

Os diagramas estruturais e comportamentais a serem implementados variam de acordo com as necessidades do projetista e com grau de documentação desejado. No processo de codificação, devem-se utilizar nomes significativos. Caso se esteja trabalhando com várias classes, recomenda-se criar uma classe gerente para se ter maior controle dos objetos instanciados. Em *softwares* codificados em conjunto,

recomenda-se utilizar repositório para evitar o desenvolvimento de versões em paralelo e eliminar a atividade de junção de código, que costuma ser onerosa.

A qualidade do *software* pode ser mensurada pelo grau de facilidade de expansão e compreensão por outros programadores. Finalidades e funcionalidades do sistema devem ser definidas com o cliente de forma clara e exata, para que o produto entregue corresponda às expectativas do mesmo. Alterações no projeto devido a falhas de comunicação com o cliente, ou mesmo entre os desenvolvedores, geram custos adicionais, financeiros e de tempo, principalmente após o término da fase de implementação. A Figura 41 exibe uma crítica humorística de falhas recorrentes em projetos de *software*. Espera-se que você, caro leitor, ao término desse manual esteja preparado para evitá-las.

Figura 41: Falhas recorrentes em projeto de *software*.



Fonte: Página da internet[4].

4- Disponível em: < http://www.alvarofpinheiro.eti.br/tags/%C3%81lvaro%20Pinheiro/> Acesso maio de 2016.

11. Referências Bibliográficas

GAMMA, E. et al. Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos. Porto Alegre, RS: Bookman, 2007.

FOWLER, M. et al. Refactoring: Improving the Design of Existing Code. New Jersey, United States: Pearson Education, 1999.

SANTOS, R. F. UML - Linguagem de Modelagem Unificada, 2009.

GUEDES, T. A. G. et al. UML 2: Uma abordagem prática. São Paulo, SP: Novatec, 2011.