Pontifícia Universidade Católica
do Rio de Janeiro

Emersson Duvan Torres Sánchez

Desenvolvimento de uma classe no contexto da POO para gerenciamento genérico de eventos de mouse em um canvas no ambiente MATLAB

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Engenharia Civil da PUC-Rio.

Orientador: Prof. Luiz Fernando Campos Ramos Martha

Rio de Janeiro
Setembro de 2017

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Emersson Duvan Torres Sánchez

Desenvolvimento de uma classe no contexto da POO para gerenciamento genérico de eventos de mouse em um canvas no ambiente MATLAB

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Engenharia Civil da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Luiz Fernando Campos Ramos Martha
Orientador
Departamento de Engenharia Civil e Ambiental – PUC-Rio

Profª. Elisa Dominguez Sotelino
Departamento de Engenharia Civil e Ambiental – PUC-Rio

Prof. Marcelo de Andrade Dreux
Departamento de Engenharia Mecânica – PUC-Rio

Prof. André Maués Brabo Pereira
Universidade Federal Fluminense

Prof. Márcio da Silveira Carvalho
Coordenador Setorial do
Centro Técnico Científico – PUC-Rio

Rio de Janeiro, 15 de setembro de 2017

Emersson Duvan Torres Sánchez

Graduou-se em Engenharia Civil, ênfase em Estruturas, pela UDENAR – Universidad de Nariño (Colômbia) em 2015. Desenvolveu seu trabalho de dissertação na área de computação gráfica aplicada.

Ficha Catalográfica

Torres Sánchez, Emersson Duvan

Desenvolvimento de uma classe no contexto da POO para gerenciamento genérico de eventos de mouse em um canvas no ambiente MATLAB / Emersson Duvan Torres Sánchez ; orientador: Luiz Fernando Campos Ramos Martha. – 2017.

109 f. : il. color. ; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Engenharia Civil e Ambiental, 2017.

Inclui bibliografia

1. Engenharia Civil – Teses. 2. Programação orientada a objetos. 3. Linguagem de modelagem unificada. 4. Classe de eventos de mouse. 5. MATLAB. 6. Círculo de Mohr. I. Martha, Luiz Fernando Campos Ramos. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Engenharia Civil e Ambiental. III. Título.

CDD: 624

À memoria do meu pai, César Augusto.

Agradecimentos

À minha mãe Sandra e minha avó Esperanza, pelo carinho e confiança.

À Elizabeth, pela companhia e amor incondicional.

À dona Clara, pelo carinho e apoio.

Ao meu irmão Gabriel, por me motivar a ser cada dia melhor.

Aos irmãos Verdugo, pelo apoio.

Ao meu orientador, Professor Luiz Fenando, pelo apoio na realização deste trabalho.

A todos os professores e funcionários do Departamento de Engenharia Civil pelos ensinamentos e pela ajuda.

À CAPES e à PUC-Rio, pelos auxílios concedidos.

Resumo

Torres Sánchez, Emersson Duvan; Martha, Luiz Fernando Campos Ramos. **Desenvolvimento de uma classe no contexto da POO para gerenciamento genérico de eventos de mouse em um canvas no ambiente MATLAB**. Rio de Janeiro, 2017. 109p. Dissertação de Mestrado – Departamento de Engenharia Civil e Ambiental, Pontifícia Universidade Católica do Rio de Janeiro.

O ensino de computação gráfica aplicada é de muita importância no processo de simulação computacional de problemas de engenharia. Atualmente, muitos programas de computador, de fácil utilização, têm melhorado este trabalho, como é o caso do MATLAB. A geração e manipulação de um modelo geométrico, que é a forma mais realista e apropriada de representar o problema a ser estudado, são etapas muito importantes na simulação computacional. O uso do mouse permite que estas etapas se tornem mais interativas e de fácil compreensão. Por este motivo, neste trabalho desenvolve-se uma classe genérica no contexto da programação orientada a objetos, no ambiente MATLAB, que permite gerenciar eventos de mouse em um canvas. O objetivo desta classe é ser utilizada no desenvolvimento de programas gráficos e interativos em MATLAB, principalmente para fins educacionais. Visando atender a essas expectativas, adotou-se a Orientação a Objetos, que possibilita a criação de códigos reutilizáveis. Aliada a essa técnica, utiliza-se a Unified Modeling Language, uma linguagem gráfica que permite a visualização, construção e documentação do desenvolvimento de um sistema computacional orientado a objetos. Para determinar o correto funcionamento e praticidade da classe desenvolvida, são implementadas duas aplicações interativas no software MATLAB; a primeira para desenhar pórticos planos em 2D e a segunda para demostrar o funcionamento do círculo de Mohr para estado plano de tensões.

Palavras-chave

Programação orientada a objetos; Linguagem de modelagem unificada; Classe de eventos de mouse; MATLAB; Círculo de Mohr.

Abstract

Torres Sánchez, Emersson Duvan; Martha, Luiz Fernando Campos Ramos (Advisor). **Development of a class in the context of OOP for generic management of mouse events in a canvas in the MATLAB environment**. Rio de Janeiro, 2017. 109p. Dissertação de Mestrado – Departamento de Engenharia Civil e Ambiental, Pontifícia Universidade Católica do Rio de Janeiro.

Teaching of applied computer graphics is of great importance in computational simulation of engineering problems. Currently, many user-friendly computer programs have improved this work, as is the case with MATLAB. The generation and manipulation of a geometric model, which is a more realistic and appropriate way to represent the problem to be studied, are very important steps in the computational simulation. The use of the mouse allows these steps to become more interactive and easy to understand. For this reason, in this work a generic class is developed in the context of object-oriented programming (OOP) in the MATLAB environment, which allows managing mouse events in a canvas. The goal of this OOP class is to be used as a base class in the development of graphics and interactive programs in MATLAB, mainly for educational purposes. In order to meet these expectations, an OOP paradigm was adopted, which enables the creation of reusable codes. Together to this technique, the Unified Modeling Language (UML) is used, a graphic language that allows the visualization, construction and documentation of the development of an object oriented computational system. To determine the correct functioning and practicality of the developed class, two interactive applications are implemented in MATLAB; the first to draw frame structures in 2D and the second to demonstrate the Mohr circle for stress state.

Keywords

Object-oriented programming; Unified modeling language; Mouse events class; MATLAB; Mohr circle.

Sumário

Lista de figuras

Lista de tabelas

Lista de Abreviaturas

POO – Programação Orientada a Objetos
UML – *Unified Modeling Language*
FCM – *Finite Cell Method*
GUI – *Graphical User Interface*
GUIDE – *ambiente de desenvolvimento de GUI*

1
Introdução

A crescente evolução da computação na atualidade tem facilitado a resolução de problemas de diversas áreas da engenharia, tornando-se indispensável seu uso. Antes da existência da computação, estes problemas eram de solução muito demorada, ou até impossível, devido à alta demanda de cálculos que envolvia sua análise.

Além dos cálculos relacionados aos problemas de engenharia, outra dificuldade que muitas vezes se enfrenta, é a modelagem destes problemas e suas soluções. Várias destas modelagens são muito complexas para serem desenvolvidas sem o auxílio da computação gráfica. Neste ponto o processo de modelagem computacional vem para facilitar a solução de problemas existentes. Desta forma, pode-se destacar a importância da geração e manipulação de um modelo geométrico. Estes processos se tornam mais confiáveis e fáceis de compreender mediante a visualização e interação com o mouse.

O ensino de computação gráfica aplicada é de muita importância no processo de simulação computacional de problemas de engenharia. Atualmente, muitos programas de computador, de fácil utilização, têm melhorado esse trabalho, como é o caso do MATLAB. A entrada de dados e manipulação de modelos neste ambiente, sempre se tornaram pouco interativos, pois geralmente são utilizados comandos ou botões para realizar estes processos. Embora exista no MATLAB tratamento formal de eventos de mouse em canvas, a maioria dos usuários não utiliza a interação com o mouse em seus programas. Neste trabalho pretende-se contribuir para a mudança dessa situação criando procedimentos para facilitar o uso do mouse para realizar essas atividades.

Para utilizar o mouse em aplicações feitas no ambiente MATLAB, neste trabalho se desenvolve uma classe genérica, neste ambiente, no contexto da Programação Orientada a Objetos, que permite o gerenciamento de eventos de mouse em canvas.

Para determinar o correto funcionamento e praticidade da classe que se desenvolve neste trabalho, são implementadas duas aplicações interativas no software MATLAB; a primeira, e-dles2D (*Draw Linear Elements Structure 2D*), para o desenho de pórticos planos; e a segunda, e-Mohr2 (*Mohr's circle for plane stress state*), para demostrar o funcionamento do círculo de Mohr.

1.1 Justificativa

A geração e manipulação de um modelo geométrico, que represente de forma mais realista e apropriada o problema a ser estudado, são etapas muito importantes na simulação computacional. O uso do mouse permite que estas etapas se tornem mais interativas e de fácil compreensão. Por esse motivo, neste trabalho desenvolve-se uma classe genérica no contexto da programação orientada a objetos, no ambiente MATLAB, que permite gerenciar eventos de mouse em canvas, com a finalidade de ser utilizada no desenvolvimento de programas gráficos e interativos neste ambiente, e no ensino de disciplinas afins.

É utilizado o ambiente MATLAB por ser uma ferramenta que, dentre outros fatores, possui alta flexibilidade e consistência, que permitem desenvolver aplicações de cálculo técnico complexas com rapidez. Além disso, sua linguagem simples e de fácil entendimento simplifica o estudo da engenharia.

1.2 Objetivos

O objetivo principal deste trabalho consiste no desenvolvimento de uma classe genérica no contexto da Programação Orientada a Objetos, que permite gerenciar eventos de mouse em canvas, e que possibilite a geração de ferramentas para ensino de engenharia no ambiente MATLAB. Para cumprir com este objetivo optou-se pela utilização da Programação Orientada a Objetos (POO), por ser uma técnica que permite a criação de códigos eficientes e reutilizáveis. Aliada a essa técnica, utiliza-se a Linguagem de Modelagem Unificada (UML), uma linguagem gráfica que permite a visualização, construção e documentação do desenvolvimento de um sistema computacional orientado a objetos.

Também tem-se como objetivo a implementação de duas aplicações educacionais para o ensino de engenharia, utilizando a classe desenvolvida, para corroborar seu correto funcionamento e fornecer exemplos de utilização.

1.3 Organização do trabalho

Este trabalho está dividido em seis capítulos. Neste primeiro capítulo, é apresentada a introdução, a justificativa, os objetivos e uma descrição resumida do conteúdo dos demais capítulos deste trabalho. Além disso, apresenta-se uma breve revisão bibliográfica de aplicações educacionais desenvolvidas no contexto da POO no ambiente MATLAB.

No segundo capítulo, são apresentados os principais conceitos teóricos utilizados no desenvolvimento do trabalho, tais como Programação Orientada a Objetos, Linguagem de Modelagem Unificada, MATLAB e eventos de mouse.

No terceiro capítulo, apresenta-se como foi desenvolvida a classe genérica Emouse, que permite o gerenciamento de eventos de mouse em canvas, com a descrição das funções utilizadas e como deve ser implementada na criação de uma aplicação.

No quarto capítulo, é apresentado o desenvolvimento de duas aplicações que determinam o correto funcionamento e praticidade da classe Emouse; a primeira para o desenho de pórticos planos e a segunda para demostrar o funcionamento do círculo de Mohr. Com estas duas aplicações também pretende-se exemplificar o uso da classe desenvolvida.

No quinto capítulo, são apresentadas as conclusões e as sugestões para trabalhos futuros.

Finalmente, no sexto capítulo estão os anexos do trabalho. Para melhorar a visibilidade e organização, os anexos apresentam os diagramas de sequência das aplicações desenvolvidas.

1.4 Revisão Bibliográfica

Nesta seção é apresentada uma breve revisão bibliográfica de trabalhos que desenvolvem aplicações, no ambiente MATLAB no contexto da Programação Orientada a Objetos (POO), para engenharia.

Em 2002, em [1] foi desenvolvida uma estrutura de software habilitada para a internet que facilita a utilização e o desenvolvimento colaborativo de um programa de análise estrutural de elementos finitos, aproveitando o contexto da POO, banco de dados e outras tecnologias de computação. O software foi concebido para fornecer aos usuários e desenvolvedores acesso fácil a uma plataforma de análise e visualização de resultados. Neste software, MATLAB é usado como um mecanismo para construir um serviço de pós-processamento simples, que leva um arquivo de dados como entrada e, em seguida, gera uma representação gráfica. Além disso, o pacote GUI (interface gráfica do usuário) de MATLAB e um navegador web padrão foram empregados para criar interfaces de usuário que permitem acessar os resultados da análise e informações relacionadas ao projeto.

O software colaborativo desenvolvido em [1], além de MATLAB, utilizou Java para representar dados numéricos, de forma conveniente para tratar e transmitir dados de tipo matriz. Para incorporar o MATLAB na estrutura do software é necessário lidar com a comunicação entre os ambientes colaborativos

e desenvolver um sistema orientado a objetos. MATLAB contém um pacote que permite interpretar a linguagem Java através de comandos próprios, e poder criar e executar programas que criem e acessem classes e objetos Java. Esta capacidade de MATLAB permitiu ao sistema desenvolvido em [1], trazer classes Java para o ambiente MATLAB, construir objetos dessas classes, chamar métodos nos objetos Java e salvar objetos Java para ser carregados quando sejam necessários. Tudo realizado com funções e comandos MATLAB. A POO fornece uma arquitetura de software em camadas que permite estabelecer o link entre o MATLAB e Java.

Em 2003, em [2] foi desenvolvido um software, no ambiente MATLAB no contexto da POO, para análise e pesquisa de métodos de projeto estrutural. onde diferentes métodos de análise estrutural e procedimentos de projeto são estudados. Esta aplicação foi desenvolvida para ser genérica para diferentes aplicações de análise e otimização estrutural, procedimentos de design, índices de desempenho e métodos de análise. Também foi implementada com o objetivo de atender diferentes tipos de elementos estruturais de propriedades não-lineares. Este software é implementado em MATLAB porque sua linguagem é de fácil entendimento, permite a utilização da POO e tem a capacidade integrada de processamento e visualização.

O software criado em [2], é composto por quatro módulos básicos: *structure*, *load*, *analysis*, e *design*. Estes módulos são descritos em um diagrama de sequência da Linguagem de Modelagem Unificada (UML). Cada módulo inclui um conjunto de interfaces e classes cooperantes, apresentados em diagramas de classe da UML. As interações entre esses módulos estão claramente definidas e ligeiramente acopladas. As classes isolam os dados e as interfaces de classe fornecem e restringem o acesso aos dados, portanto, mudanças em um determinado módulo não afetam outros módulos. Esta situação também permite a implementação de novas classes na criação de novas aplicações sem a necessidade de alterar a estrutura geral. Estas são as vantagens que a POO forneceu ao desenvolvimento desse trabalho.

Em 2007, em [3] é apresentada uma abordagem orientada a objetos de análise estrutural e sua implementação dentro de um programa de elementos finitos desenvolvido no ambiente MATLAB. O objetivo do trabalho foi simplificar o ciclo de preparação e análise de dados, e encontrar uma configuração estrutural robusta e econômica.

Em 2010, em [4] o *toolbox JWM SAOSYS v0.42* (Sistema de Análise e Otimização Estrutural), é apresentado como um protótipo experimental de *toolbox* para o ambiente MATLAB, desenvolvido no contexto da programação orientada a objetos. Abarcando um conjunto de dados, funções, objetos e

scripts o *toolbox* é destinado à pesquisa numérica, à análise e dimensionamento de estruturas de aço, utilizando o método de elementos finitos. Na apresentação deste *toolbox* são descritas as principais classes e métodos mediante um diagrama de classes da UML. Cada classe herda as características e comportamentos de uma classe principal que contém os métodos necessários para o funcionamento do sistema; estes métodos são implementados por cada classe de acordo a seus requerimentos.

Segundo [4], o MATLAB foi utilizado na criação do *toolbox JWM SAOSYS* por possuir inúmeras funções integradas que permitem o desenvolvimento eficaz de um sistema. A POO foi empregada pelos seus conceitos de encapsulamento e polimorfismo. Além disso, o sistema de modelagem do *toolbox* opera com elementos finitos orientados a objetos. Isso permite integrar novos elementos finitos no sistema para a resolução de diversos problemas.

A POO no *toolbox JWM SAOSYS* permitiu que fossem utilizadas suas ferramentas para o desenvolvimento do trabalho apresentado em [5]. Neste trabalho o *toolbox* foi atualizado e um novo módulo de análise (*EPSOp-timSD*) foi criado.

O Método das Células Finitas (FCM) é descrito em [6] como uma extensão do método dos elementos finitos que combina os benefícios dos elementos finitos de ordem superior com as ideias dos métodos de domínio fictício, visando simplificar o processo de geração de malha, separando a aproximação da solução analítica da representação geométrica do domínio físico. Em 2014, em [6] o *toolbox FCMLab*, desenvolvido no contexto da POO no ambiente MATLAB, é apresentado como uma ferramenta de aplicação e pesquisa do FCM, que permite o rápido desenvolvimento de novos métodos algorítmicos no contexto de métodos de domínio fictício de alta ordem. As partes mais importantes do desenvolvimento das classes do *toolbox* são apresentadas mediante diagramas de classes da UML. Também é descrito como o *FCMLab* deve ser utilizado por meio de três exemplos.

Segundo [6], novas ideias algorítmicas podem ser testadas facilmente. Portanto, o sistema foi estruturado de modo que diferentes componentes possam ser trocados facilmente sem afetar outras partes do código. Para desacoplar uma família de algoritmos do código, diferentes algoritmos foram implementados como classes separadas. Cada classe herda as propriedades e comportamentos da mesma classe base que define a interface comum. Estas classes e suas funções têm uma responsabilidade claramente definida. Isso permite que famílias de algoritmos possam ser estendidas simplesmente adicionando novas subclasses. Para resolver estas questões, no desenvolvimento deste *toolbox*, a POO foi utilizada, conseguindo que o programa seja decomposto hierarquica-

mente em subsistemas, dispostos como camadas, de modo que cada módulo dependa apenas de camadas inferiores. Essa organização permite considerar cada camada como um módulo individual com uma tarefa claramente definida, cuja implementação pode ser trocada sem afetar outros elementos.

Em [7] é apresentada a análise Isogeométrica como uma nova formulação do Método dos Elementos Finitos. O objetivo principal deste trabalho foi a implementação numérica deste método no ambiente MATLAB no contexto da programação orientada a objetos. A POO foi utilizada para produzir uma ferramenta mais genérica para futuras investigações.

Um exemplo prático de aplicação em MATLAB no contexto da POO desenvolve-se em [8]. O qual foi criado para o cálculo de placas pelo método dos elementos finitos.

A implementação de uma interface gráfica para um sofware de análise de elementos finitos, orientado a objetos em MATLAB é apresentada em [9].

Existem aplicações desenvolvidas no contexto da POO no ambiente MATLAB em outras áreas da engenharia, por exemplo o software para projeto de aviões apresentado em[10].

Algumas das aplicações apresentadas acima, poderiam ser mais interativas utilizando o mouse como ferramenta de desenho para a entrada de dados ou para modificar um modelo. Desta forma tornaria estes processos de fácil entendimento, pois seria possível visualizar os dados de entrada no sistema ou as modificações feitas no modelo, por exemplo, quando desenhamos um pórtico com determinadas coordenadas ou alongamos uma viga do modelo.

2
Fundamentos de Orientação a Objetos, UML e MATLAB

Neste capítulo são apresentados conceitos básicos sobre Orientação a Objetos, Linguagem de Modelagem Unificada (UML), MALTAB e eventos de mouse. A UML é linguagem gráfica que permite a visualização, construção e documentação do desenvolvimento de um sistema computacional orientado a objetos. Os tipos de diagramas da UML que são utilizados neste trabalho também são descritos neste capítulo.

2.1
Orientação a Objetos

A maioria dos métodos utilizados em ambientes de desenvolvimento de software se baseia em uma decomposição funcional e/ou controlada por dados dos sistemas. Estas abordagens se diferem em diversos aspectos das abordagens que adotam metodologias orientadas a objetos, onde dados e funções são altamente integrados.

O desenvolvimento de software com a abordagem orientada a objetos consiste na construção de módulos independentes ou objetos que podem ser facilmente substituídos, modificados e reutilizados. Ela retrata a visão do mundo real como um sistema de objetos cooperativos e colaborativos. Neste caso, o software é uma coleção de objetos discretos que encapsulam dados e operações executadas nesses dados para modelar objetos do mundo real. A classe descreve um grupo de objetos que têm estruturas semelhantes e operações similares.

Um sistema Orientado a Objetos é desenvolvido como um conjunto de elementos que interagem uns com os outros para realizar uma tarefa [11].

A abordagem orientada a objetos possibilita uma melhor organização, versatilidade e reutilização do código fonte, o que facilita atualizações e melhorias nos programas. A abordagem orientada a objetos é caracterizada pelo uso de classes e objetos, e de outros conceitos que serão apresentados a seguir.

2.1.1 Classes e Objetos

Um objeto representa uma entidade real, uma pessoa, um animal, uma casa, qualquer coisa que possa ser imaginada. Um objeto também possui características e comportamentos, por exemplo, um carro possui marca, modelo, tipo, ano de fabricação; assim como comportamentos, andar, parar, virar. Um objeto pode ser identificado a partir dos métodos e dos atributos que possui [12].

Na análise orientada a objetos, é simplificado o entendimento dos programas de computador decompondo sua estrutura em objetos, criando módulos com características e comportamentos que em conjunto permitem executar o processamento de dados do sistema.

Um objeto possui um estado, comportamento e identidade definidos. Tem como objetivo simplificar o entendimento das entidades que interatuam em um sistema. Segundo [13], os objetos envolvidos em um sistema são obtidos através da decomposição do sistema durante o processo de análise.

O estado de um objeto pode ser definido como seus atributos, pode ser estático ou não. Por exemplo, um carro pode ter o estado de "novo", mas depois de ser utilizado, seu estado pode ser modificado para "usado".

O comportamento de um objeto é uma propriedade dinâmica, uma resposta a uma ação executada no objeto. Por exemplo, um carro pode ter o estado de "desligado"e em resposta à ação "ligar", seu estado é modificado para "em funcionamento".

A identidade de um objeto é a capacidade que permite diferenciar uma instância dos demais objetos da mesma classe. Por exemplo, dois carros idênticos, podem ser diferenciados pelo seu número de placa, propriedade única de cada objeto capaz de diferenciá-lo dos demais.

Classes são espécies de montadoras de objetos, que definem suas características como, quais funções são capazes de realizar e quais os atributos que o objeto possui. Essa forma de programar permite ao usuário resolver problemas utilizando conceitos do mundo real. Objeto é uma instância gerada a partir de uma classe.

BOOCH em [13], define classe como um conjunto de objetos que possuem uma estrutura e comportamentos comuns. Além disso, também define que um único objeto pode ser chamado como instância de uma classe.

Segundo RUMBAUGH et al. em [14], um objeto é uma entidade discreta com uma fronteira definida e uma identidade que encapsula estado e comportamento. Também define classe como uma coleção de objetos que compartilham os mesmos atributos, operações, métodos, relacionamentos e comportamento.

Assim, podemos dizer que, um objeto é uma entidade abstrata, com atributos e métodos definidos pela sua classe e que tem significado para a aplicação. A instância de uma classe é um objeto e que uma classe representa uma abstração das caraterísticas semelhantes de um determinado conjunto de objetos [12].

2.1.2 Atributos e Métodos

Uma classe contém atributos e métodos que são, respectivamente, características e rotinas que podem ser executadas por um objeto desta classe. Uma instância pode ser diferenciada das demais pela combinação de valores de atributos e métodos, que definem sua identidade, seu estado e comportamento. Os métodos de uma instância podem ser distintos de outra quando se utiliza uma hierarquia de subclasses de uma classe. Os métodos e os atributos de uma classe são definidos durante o projeto do programa, procurando realizar uma transposição dos atributos reais para a aplicação. Por exemplo, com a classe *carro* pode ser criado um objeto *jeep*, que tem o atribuo *cor*, o método *transportar*.

Um atributo é a descrição de um repositório de um determinado tipo de dado em uma classe. Cada objeto pode possuir um determinado valor para o atributo. Método é a implementação de uma operação. Ele especifica o algoritmo ou procedimento que resulta da operação [14].

Assim, podemos dizer que, os atributos são as peculiaridades comuns que definem uma classe. Os valores dos atributos variam segundo a instância e permitem sua diferenciação. Por exemplo, cor, altura, peso [12].

Os atributos podem ser divididos em essenciais e derivados. Os atributos essenciais são variáveis locais que armazenam valores dos respectivos atributos. Os atributos derivados são operações (métodos) que retornam um cálculo a partir de um valor armazenado em um atributo essencial. Por exemplo, o volume de um cubo é um atributo derivado que pode ser calculado a partir de um atributo essencial comprimento da aresta do cubo.

Os métodos são as atividades ou ações realizadas por uma classe. Um método pode receber parâmetros e retornar valores. Os retornos podem indicar o sucesso da operação ou o valor resultante. Um método também pode ser definido como um conjunto de instruções para realizar uma determinada ação. Por exemplo, desenhar, pagar ou mover.

Métodos são as funções que um objeto pode realizar. Atributo é tudo que um objeto possui como variável.

2.1.3 Relacionamentos

Na vida real os objetos relacionam-se para realizar determinadas atividades, combinam suas características e ações para desempenhar tarefas cotidianas. Por exemplo, as peças de um carro se combinam para que o carro possa funcionar.

Na programação orientada a objetos, os objetos são combinados para produzir um determinado resultado, ou seja, um sistema é produzido pela combinação de vários elementos. Segundo [14], os relacionamentos são conexões semânticas entre os elementos de um sistema orientado a objetos. Estes relacionamentos podem ser classificados como:

1. Associação: são relacionamentos entre os objetos para o cumprimento de ações e objetivos comuns, sem relação de dependência. Estes relacionamentos podem ser genéricos ou ter duas subclassificações:

   (a) Agregação: indica uma independência dos objetos-parte do objeto-todo. Na destruição do objeto-todo, os objetos-parte ainda podem existir.

   (b) Composição: indica uma dependência dos objetos-parte do objeto-todo. Na destruição do objeto-todo, os objetos-parte não fazem mais sentido e deixam de existir.

2. Generalização/Especialização: indica uma relação entre um elemento mais genérico e um mais específico, com a transmissão de características do objeto mais genérico. Por exemplo, um objeto *mamífero* pode se especializar em *gato*, *cachorro*, *coelho*, etc. Todos os objetos especializados possuem características comuns definidas pela classe original.

3. Dependência: indica uma relação de dependência estrutural entre os objetos envolvidos, na qual, o componente dependente é impactado ao mudar o componente principal.

4. Realização: indica um relacionamento entre dois elementos de um sistema, no qual um elemento realiza um comportamento específico de outro elemento. Vários elementos podem realizar o comportamento de um único elemento.

Os relacionamentos envolvem multiplicidade e cardinalidade. A cardinalidade indica quantos objetos se relacionam naquele determinado relacionamento. Por exemplo, um relacionamento entre professor e aluno, onde um

professor pode ter n alunos. Estes dois conceitos são constantemente fundidos no conceito de multiplicidade.

Cardinalidade é definido como o número de objetos em um conjunto de elementos. A multiplicidade é uma especificação da variação possível de valores de cardinalidade que um conjunto de elementos pode assumir [14].

A Tabela 2.1, apresenta as possibilidades de multiplicidades em um relacionamento.

Tabela 2.1: Possibilidades de multiplicidades [12].

Multiplicidade	**Significado**
x	Indica que x elementos estão envolvidos no relacionamento.
*	Indica que n (muitos) elementos estão envolvidos no relacionamento.
0..1	Indica que no mínimo nenhum (0) elemento e no máximo um (1) elemento está envolvido no relacionamento.
0..*	Indica que no mínimo nenhum (0) elemento e no máximo n (muitos) elementos estão envolvidos no relacionamento.
1..1	Indica que um e somente um elemento está envolvido no relacionamento.
1..*	Indica no mínimo um (1) elemento e no máximo n (muitos) elementos estão envolvidos no relacionamento.
x..y	Indica que no mínimo x elementos e no máximo y elementos estão envolvidos no relacionamento.

2.1.4 Herança e Polimorfismo

Herança é uma característica que permite a determinada classe herdar as características de outra classe, chamada classe pai ou superclasse. A classe descendente, chamada classe filha, adquire todos os métodos e atributos da classe pai.

Segundo [15], a importância da herança na implementação se resume em:

- A herança suporta uma modelagem mais rica e poderosa. Beneficia toda a equipe de desenvolvimento, pois propicia um maior poder de reutilização de código.
- A herança possibilita definir informações e comportamentos em uma classe e compartilhar com outras. Isso resulta em menos código para escrever.
- A herança é natural. Este é um dos motivos mais importantes para a orientação a objetos ser o paradigma de programação mais utilizado.

A herança pode ser única ou múltipla. É única quando uma classe herda de apenas uma superclasse e é múltipla quando herda de diversas classes. Esta última, apesar de ser pouco difundida e de utilização complexa, sua implementação no MATLAB é possível, servindo como mais um benefício ao programador.

Após herdar os métodos da superclasse, estes nem sempre irão atender perfeitamente às necessidades da classe filha, muitas vezes precisam ser reescritos para adequar um comportamento ou definir seu funcionamento. Para atender esta demanda, é utilizado o recurso do polimorfismo, onde os métodos podem ser modificados de forma a atender melhor às necessidades da classe filha. O termo polimorfismo vem do grego *poli* (muitos) e *morfo* (formas), uma determinada característica pode assumir diversas formas, conseguindo atender as necessidades da nova classe.

O polimorfismo é uma característica da orientação a objetos que significa a possibilidade de modificar os métodos herdados de uma superclasse, mantendo um tratamento genérico dos objetos das subclasses. Ou seja, ao se herdar os atributos ou métodos de uma superclasse, estes podem ser redeclarados na nova classe ao mesmo tempo em que os objetos de todas as subclasses são tratados de maneira abstrata (polimorficamente) como objetos da superclasse.

Uma estratégia muito comum na POO quando se explora as características de Herança e Polimorfismo é a criação de superclasses abstratas. Uma classe é abstrata quando pelo menos um de seus métodos é abstrato (ou virtual). Um método abstrato não tem uma implementação concreta na classe abstrata. O método abstrato é declarado para definir um comportamento genérico para um objeto. Como o método abstrato não tem uma implementação concreta, não é possível instanciar (criar) um objeto de uma classe abstrata. Para criar um objeto é preciso definir uma subclasse por herança da superclasse abstrata. Essa subclasse vai implementar os métodos declarados como abstratos na superclasse. Dessa forma, a subclasse não só concretiza os métodos abstratos como também cria uma especialização da superclasse. Os objetos instanciados de uma subclasse concreta são tratados polimorficamente como objetos da superclasse. Dessa maneira, algoritmos que utilizam esses objetos os tratam genericamente, sem distinção de que subclasse eles pertencem na hierarquia de herança.

Durante o projeto de um sistema orientado a objetos, o programador pode desejar que os métodos e atributos possuam níveis de acesso diferente aos demais componentes do programa. Isto significa que se pode evitar que os componentes acessem os métodos e atributos de uma determinada classe. Este conceito é chamado visibilidade e pode ser classificado em 4 níveis [12] como

se apresenta na Tabela 2.2.

Tabela 2.2: Visibilidade.

Símbolo	Significado
+	Indica que o método ou atributo pode ser visível e utilizado por qualquer outra classe.
~	Indica que o método ou atributo pode ser visível e utilizado pelas classes no mesmo pacote*.
#	Indica que o método ou atributo pode ser visível e utilizado apenas pelas classes filhas ou pela superclasse.
-	Indica que o método ou atributo pode ser visível e utilizado apenas pela classe que o contém.

* Um Pacote é um conjunto de classes localizadas na mesma estrutura hierárquica de diretórios. Usualmente, são colocadas em um pacote classes relacionadas, construídas com um propósito comum.

Com a utilização da característica de visibilidade, a restrição de utilização de métodos por componentes externos é controlada, evitando que durante a execução do programa surjam resultados imprevistos.

O encapsulamento é o conceito utilizado na programação orientada a objetos para controlar a visibilidade dos atributos e métodos dos objetos de uma classe.

2.2 Linguagem de Modelagem Unificada UML

É uma linguagem gráfica para visualizar, especificar, construir e documentar os artefatos de um sistema computacional orientado a objetos. Esta linguagem, baseada no paradigma de orientação a objetos, possui em sua estrutura diversos componentes e mecanismos responsáveis por estruturar a linguagem e permitir sua adaptação para muitas necessidades, permitindo sua aplicação em vários tipos de projeto, dos mais simples até os mais complexos.

A UML pode ser definida com um "metamodelo"[16] porque sua linguagem é definida em um nível mais próximo de utilização do usuário. A linguagem gráfica UML permite uma especificação genérica da solução orientada a objetos de um problema que poderá ser traduzida diretamente para uma linguagem de programação.

2.2.1 Componentes da UML

Com o objetivo de representar diversas situações estruturais e comportamentais de um programa como uma abstração da realidade, segundo [14], a

UML apresenta 3 categorias de blocos de construção:

2.2.1.1 Itens

Os itens são os elementos mais básicos de um diagrama. Estão divididos em:

1. Itens Estruturais: são os blocos básicos de representação da estrutura estática de um programa. Os mais importantes são: classes, interfaces, casos de uso, componentes, artefatos, nós e de colaboração.

2. Itens Comportamentais: representam os itens dinâmicos e comportamentais de um programa. Os mais importantes são: de estado (por exemplo, ligado - desligado), de ação (por exemplo, ligar) e mensagens.

3. Itens de Agrupamento: são elementos organizacionais da estrutura do programa. São utilizados na decomposição dos elementos do diagrama. São representados por pacotes.

4. Itens Notacionais: são elementos auxiliares no detalhamento de uma determinada característica. São utilizados para realizar anotações sobre um elemento de um diagrama.

2.2.1.2 Relacionamento na UML

Conforme definido na seção 2.1.3, os relacionamentos são responsáveis por associar os itens de um diagrama. Sua representação gráfica na UML depende do tipo de relacionamento e é descrita na seção 2.3.2.

2.2.1.3 Diagramas

Os diagramas são representações gráficas dos elementos de um sistema orientado a objetos, comumente apresentados como gráficos de conexão (relacionamentos) e vértices (outros elementos do sistema) [14].

Um diagrama contém uma moldura que delimita sua área, com uma informação a respeito do diagrama, na extremidade superior esquerda, indicando o tipo de item representado, seu nome e os parâmetros utilizados em sua inicialização. Esta moldura pode ser suprimida quando o contexto do diagrama é claro e não precisa de informações em suas fronteiras.

Estes diagramas podem ser dos seguintes tipos:

1. Diagramas de estrutura: indica como as entidades de um projeto orientado a objetos devem ser no sistema. Os diagramas desta categoria se subdividem em:

   (a) Diagrama de classes: apresenta uma visão estática da estrutura de classes do sistema e seus relacionamentos.

   (b) Diagrama de objetos: apresenta uma visão estática dos objetos do sistema e seus relacionamentos.

   (c) Diagrama de componentes: apresenta uma visão de implementação dos componentes do sistema, com suas classes, interfaces, arquivos e dependências.

   (d) Diagrama de estrutura composta: semelhante ao digrama de componentes, apresenta as classes de um componente e suas colaborações para a realização do objetivo da estrutura.

   (e) Diagrama de implantação: apresenta uma visão de *hardware* do sistema, seus servidores e seus protocolos de comunicação.

   (f) Diagrama de pacotes: apresenta uma visão de agrupamento dos componentes do sistema e suas dependências.

   (g) Diagrama de perfil: apresenta uma visão de metamodelo, identificando as classes e os componentes pelo seu estereótipo e pelo seu perfil, respectivamente, com os relacionamentos resultantes.

2. Diagramas de comportamento: indica como as entidades de um projeto orientado a objetos devem se comportar no sistema. Os diagramas desta categoria se subdividem em:

   (a) Diagrama de atividade: apresenta as etapas ou passos para conclusão de uma determinada atividade no sistema.

   (b) Diagrama de caso de uso: apresenta os casos de uso ou funcionalidades e os atores que interagem com estas.

   (c) Diagrama de máquina de estado: apresenta o comportamento de determinado componente para a realização de uma atividade.

3. Diagramas de interação: indica o fluxo de controle e de dados entre os componentes do sistema. Os diagramas desta categoria se subdividem em:

   (a) Diagrama de sequência: apresenta a visão temporal das mensagens entre os objetos.

(b) Diagrama de comunicação: semelhante ao digrama de sequência, apresenta a organização estrutural dos objetos que enviam e recebem mensagens.

(c) Diagrama de visão geral de interação: apresenta a sequência de mensagens para a realização de uma atividade.

(d) Diagrama de temporização: apresenta em tempo real uma determinada interação.

Para ajudar o entendimento das classes e seus relacionamentos implementados nas aplicações desenvolvidas neste trabalho, foram criados diagramas de classes, de atividades e de sequência do sistema desenvolvido. Esses tipos de diagramas são estudados mais profundamente na sequência.

2.2.1.4 Estereótipos, valores atribuídos e restrições

Os estereótipos indicam condições especiais assumidas para um elemento, diferenciando-se dos demais. Um estereótipo é um novo tipo de elemento definido para determinado sistema, baseado em um elemento existente [14]. Seu formato é igual ao de outros elementos, mas deve ser diferenciado por possuir características diferentes dos demais. Sua finalidade é evitar a criação de um grande número de elementos para representação na UML.

Os valores atribuídos permitem que elementos importantes no projeto possam ter atribuições não previstas na UML. Por exemplo, informações de versão ou de autor.

As restrições definem condições especiais de classes, relacionamentos, métodos ou atributos. Devem ser definidas previamente, para o entendimento do sistema e devem ser explicadas no diagrama. As restrições são representadas como texto colocado entre chaves ou com notas, que devem estar próximos ao elemento que se refere.

2.3 Diagrama de Classes

Um digrama de classes apresenta uma visão estática da organização e relacionamento das classes de um programa, definindo sua estrutura lógica. Por isso, é utilizado no desenvolvimento deste trabalho.

Este diagrama é uma representação gráfica da visão estática que mostra uma coleção de elementos de um sistema orientado a objetos, como classes, tipos, seus conteúdos e relacionamentos. Um diagrama de classes contém

elementos comportamentais solidificados, como operações, mas sua dinâmica é representada em outros tipos de diagramas [14].

2.3.1 Classes

Uma classe é representa por um retângulo divido em 3 partes: identificador da classe, atributos e métodos. Pode ser realizada a supressão de seus atributos, métodos ou ambos, caso não sejam necessários no diagrama.

Na representação gráfica de uma classe, seus atributos ficam na segunda divisão do retângulo correspondente à classe. Os atributos são normalmente precedidos por um símbolo que indica sua visibilidade, depois está o nome do atributo e finalmente o tipo de dado.

Os métodos de uma classe ficam na terceira divisão do retângulo correspondente à classe. Assim como os atributos, são precedidos por um símbolo que indica sua visibilidade, seguido pelo nome do método e finalmente seu tipo de retorno. Um exemplo de classe é apresentado na Figura 2.1.

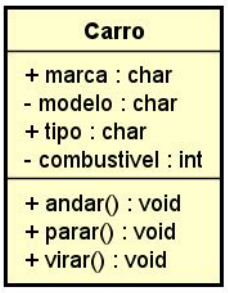

Figura 2.1: Exemplo de classe.

2.3.2 Relacionamentos

Conforme definido na seção 2.1.3, as classes se relacionam entre si, permitindo o compartilhamento de informações e a associação de métodos para cumprir tarefas específicas. De acordo com o objetivo do relacionamento eles podem ser classificados em: Associação, que pode ser genérica ou subclassificada como Agregação ou Composição, Generalização, Dependência e Realização.

2.3.2.1 Associações genéricas

As associações genéricas são representadas por retas ligando os elementos associados. Podem haver setas em suas terminações para indicar a navegabi-

lidade. As associações podem conter um identificador, que indica o vínculo direcionado da associação e pode haver em suas extremidades o indicador de multiplicidade. Na Figura 2.2 observam-se as formas mais comuns de representação de associação: (a) sem navegabilidade, sem identificador e sem multiplicidade; (b) com multiplicidade; (c) com navegabilidade e sem identificador; (d) com navegabilidade, com identificador e com multiplicidade.

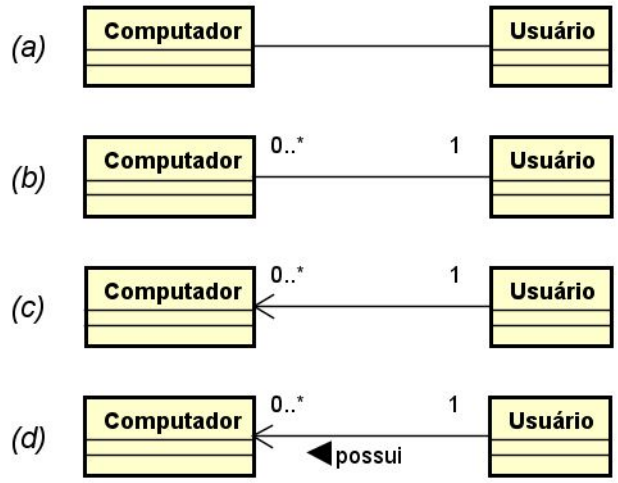

Figura 2.2: Exemplos de associação [12].

A associação pode indicar um relacionamento de uma classe com ela mesma. Este tipo de associação significa que seus objetos possuem um vínculo entre eles. Por exemplo, dentro de uma classe "funcionário", um determinado funcionário pode exercer o cargo de chefe sobre outros funcionários. Nesse caso, como todo chefe é um funcionário, a classe se relaciona com ela mesma (Figura 2.3) [12].

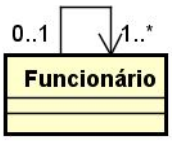

Figura 2.3: Associação realizada entre elementos da mesma classe [12].

Quando uma associação entre duas classes tem uma multiplicidade de elementos de muitos para muitos ("n para n"), neste caso, a multiplicidade "n", existe para os dois lados da associação. Esta situação pode gerar uma terceira classe, chamada classe associativa, responsável por armazenar estas referências das associações. A classe associativa também pode ter atributos próprios além dos gerados pelo relacionamento. Por exemplo (Figura 2.4), um "curso" pode

ter várias "disciplinas", assim como uma "disciplina" pode pertencer a vários "cursos". Neste relacionamento é gerada uma classe associativa, para indicar a associação entre esses objetos. Este elemento é representado por uma linha tracejada partindo da associação e chegando à classe associativa [12].

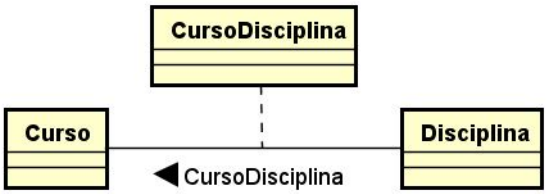

Figura 2.4: Classe associativa [12].

2.3.2.2 Agregação

Este tipo de associação indica que um objeto, chamado objeto-todo, tem um vínculo fraco com seus objetos relacionados, chamados objetos-parte. Nesta associação, quando o objeto-todo é destruído, os objetos-parte podem continuar existindo com sentido, e podem, inclusive, se associar a outros objetos-todo. Por exemplo, um objeto-todo "Time" pode ser destruído e os objetos-parte "Atleta" existem independentemente de o objeto-todo "Time" existir. Este relacionamento é representado por um losango branco, com origem no objeto-todo como se apresenta na Figura 2.5.

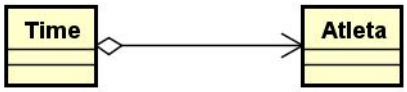

Figura 2.5: Agregação.

2.3.2.3 Composição

Este tipo de associação representa um vínculo mais forte entre o objeto-todo e os objetos-parte, estão intimante ligados, pois quando se destrói o objeto-todo, os objetos-parte perdem o sentido e deixam de existir. Por exemplo, um objeto-todo "Pedido" está composto pelos objetos "Item" (do pedido). Se o objeto todo "Pedido" for destruído, os objetos-parte "Item" também são destruídos, pois perdem o sentido. Este relacionamento é representado por um losango preto, originado no objeto-todo como se apresenta na Figura 2.6.

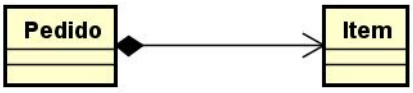

Figura 2.6: Composição.

2.3.2.4 Generalização

Este relacionamento é utilizado quando objetos possuem características comuns e se deseja realizar uma abstração, utilizando os atributos em comum. A generalização é uma relação taxonômica entre um objeto geral e objetos específicos. O objeto específico é complementado com informações do objeto geral e contém informações adicionais. Uma instância do objeto mais específico é uma instância indireta do objeto mais geral e herda suas características [14].

Na generalização é criado um objeto agregador de informações comuns, responsável por reunir os atributos comuns entre objetos específicos. Por exemplo (Figura 2.7), a classe "Veículo" pode se especializar nas classes "Moto", "Carro", "Caminhão".

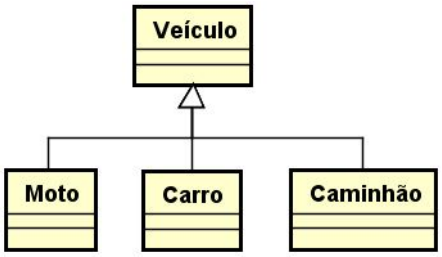

Figura 2.7: Generalização.

2.3.2.5 Dependência

Este relacionamento indica que uma modificação no elemento principal pode afetar informações ou operações no elemento dependente. É representado por uma seta tracejada entre os dois componentes, originada no elemento dependente e apontando para o elemento gerador de dependência.

2.3.2.6 Realização

Este relacionamento indica que uma classe utiliza os atributos e/ou métodos de outra classe. É representado por uma seta tracejada com ponta oca

entre os dois componentes, apontando para a classe que contém os atributos ou métodos utilizados.

2.4 Diagrama de Atividade

Na UML, um diagrama de atividade fornece uma visualização do comportamento de um sistema descrevendo o fluxo de ações em um processo; no entanto, os diagramas de atividades também podem mostrar fluxos paralelos ou simultâneos e fluxos alternativos.

Este diagrama tem maior ênfase no nível de fluxo da aplicação, sendo importante para a representação e o entendimento do funcionamento do sistema e a integração entre as ações e as mudanças de estado dos atores envolvidos. Por essa razão, é utilizado no desenvolvimento deste trabalho.

2.4.1 Nó de atividade

Uma atividade representa um comportamento organizado e estruturado com a coordenação sequencial de unidades subordinadas de elementos denominados ações [14].

Um comportamento é o resultado de ações executadas ordenadamente. Uma atividade é composta por ações, que podem ser executadas por esta, ou outras atividades.

Na UML, uma atividade é representada por um retângulo com cantos arredondados, semelhante ao nó de ação. Alguns sistemas diferenciam um nó de ação (Figura 2.8(a)) de um nó de atividade (Figura 2.8(b)) através de um indicativo em seu canto inferior direito, indicando sua decomposição em um diagrama de atividades.

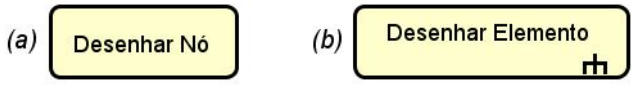

Figura 2.8: (a) Nó de ação; (b) nó de atividade

2.4.2 Nó de ação

Um nó de ação é o elemento mais básico de uma atividade e representa um passo ou uma ação executada que resulta na mudança de um objeto ou no retorno de um valor. O nó de ação não pode ser decomposto. Portanto, deve ser executado por completo.

2.4.3 Controle de fluxo

O controle de fluxo liga dois nós representando a sequência das ações para a conclusão da atividade.

Este elemento é representado por uma seta, que vai desde um nó para o seguinte na sequência como se apresenta na Figura 2.9.

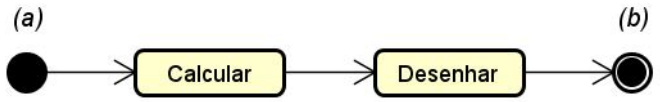

Figura 2.9: Controle de fluxo: (a) nó inicial; (b) nó final

2.4.4 Nós inicial e final

Os nós inicial e final representam o início e o fim da atividade. Somente é permitido um início para um diagrama, no entanto, pode possuir mais de um final.

O nó inicial é representado por um círculo preto (Figura 2.9(a)), enquanto o nó final é representado por um círculo branco com um círculo preto menor no centro (Figura 2.9(b)).

2.4.5 Ramificações

As ramificações indicam escolhas possíveis no fluxo (Figura 2.10(a)). Condições no sistema permitem a criação de escolhas no fluxo. As ramificações também podem ser utilizadas para unir fluxos bifurcados (Figura 2.10(b)) por nós de ramificações anteriores.

As ramificações são representadas por um losango como se apresenta na Figura 2.10.

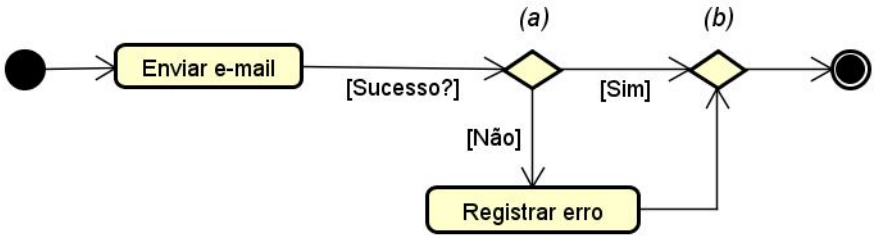

Figura 2.10: Ramificações: (a) nó de decisão; (b) nó de união [12].

2.4.6 Conectores

Os conectores são marcações utilizas quando se deseja ligar nós distantes dentro de um mesmo diagrama. Os nós de ação que se desejam ligar, são ligados diretamente aos conectores.

Um conector é representado por um círculo com um identificador, geralmente um letra ou um número como se pode observar na Figura 2.11.

Figura 2.11: Conectores.

2.4.7 Nó de bifurcação e junção

Estas entidades permitem dividir um fluxo em fluxos simultâneos. Estes terão sua execução paralelizada, indicando que poderão ter mais de uma saída para apenas uma entrada. Para unir os fluxos concorrentes são utilizados os nós de junção.

Estes nós são representados por uma barra preta que distribui ou recebe os fluxos como apresenta a Figura 2.12.

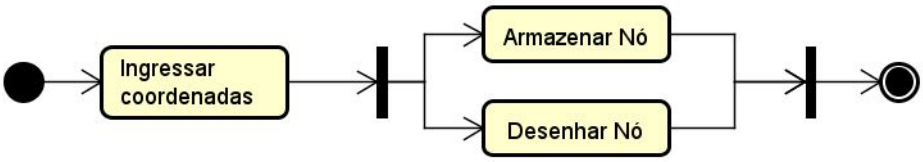

Figura 2.12: Nós de bifurcação e junção.

2.4.8 Final de fluxo

Este elemento representa o final de um fluxo, geralmente cíclico, mas não o final da atividade. Normalmente é utilizado quando se precisa indicar o final de um ciclo, e se deseja continuar com a atividade depois de obter os resultados do ciclo.

Este elemento é representado por um círculo branco com um X ao centro (Figura 2.13(a)).

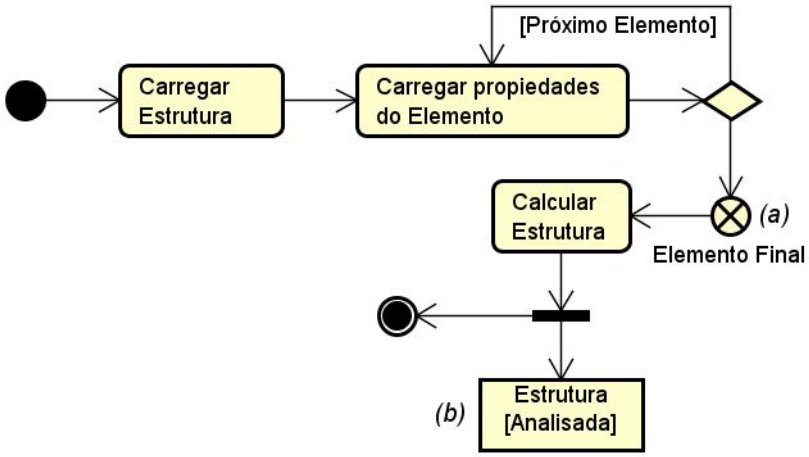

Figura 2.13: (a) Final de fluxo; (b) Nó de objeto.

2.4.9 Nós de objeto

O nó de objeto representa os objetos que estão envolvidos diretamente em um determinado ponto da atividade. Esta representação é feita quando se quer indicar uma modificação no estado de um objeto.

Este elemento é representado por um retângulo com seu nome e a indicação de seu estado abaixo do nome (Figura 2.13(b)).

2.4.10 Participação de atividade

Este recurso permite representar uma divisão de responsabilidades entres os responsáveis de uma atividade. As partições estão delimitadas por linhas horizontais ou verticais, que dividem o diagrama em áreas de responsabilidade. Por exemplo, em uma atividade de análise estrutura, o usuário é responsável pelas ações "desenhar estrutura"e "analisar", enquanto máquina é responsável por "validar os dados"e "analisar a estrutura"(Figura 2.14).

2.5 Diagrama de Sequência

Com este diagrama é possível representar a sequência de eventos e mensagens que são executados durante um processo. Além disso, apresenta o ciclo de vida dos objetos representado através de sua linha de vida, permitindo ter uma maior visualização da interação com um todo.

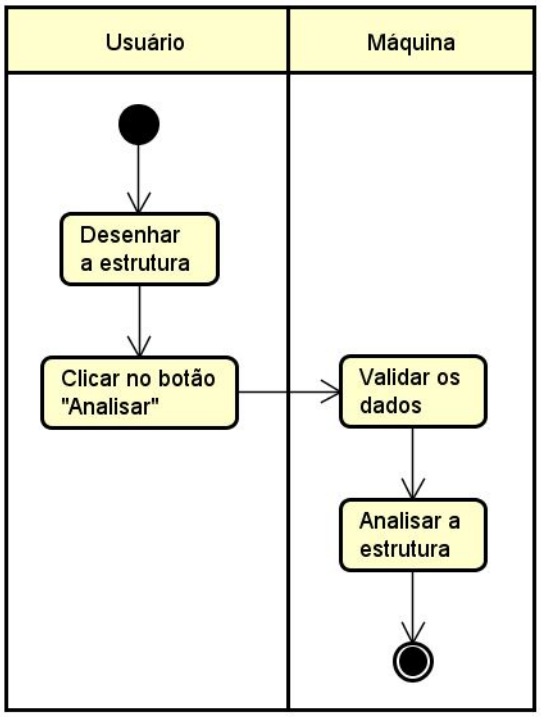

Figura 2.14: Participação de atividade.

O diagrama de sequência representa uma interação de chamadas entre os objetos ordenados pelo tempo de processamento. Diagrama este que representa a sequência de tempo de comunicação, mas não inclui os relacionamentos entre objetos [14].

2.5.1 Atores

Os atores são entidades externas que interagem com o sistema, normalmente são pessoas, mas também podem ser *hardwares* ou outros sistemas.

Um ator caracteriza e abstrai um usuário externo ou um conjunto relacionado de usuários, que interagem com o sistema. Um ator tem um propósito focado, que pode não corresponder exatamente a objetos físicos. Um mesmo objeto com diferentes propósitos pode ser modelado como diferentes atores. Objetos diferentes com um mesmo propósito podem ser modelados como apenas um ator [14].

Os atores possuem uma linha de vida e também podem ter em sua descrição uma instância de sua classe. Por ser este diagrama a representação do processo em um determinado momento, a instância às vezes é necessária.

Os atores podem ser representados por um *stickman* como pode se observar na Figura 2.15.

Figura 2.15: Ator.

2.5.2 Objetos

Os objetos neste diagrama são os que participam diretamente do fluxo do processo. Sua representação é semelhante a uma classe, indicando o nome do objeto e a classe a que pertence, mas sem atributos nem métodos; além disso, contém uma linha de vida tracejada como se apresenta na Figura 2.16.

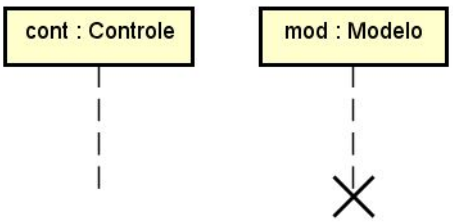

Figura 2.16: Objetos e linhas de vida.

2.5.3 Linhas de vida

Uma linha de vida representa o ciclo de vida de um objeto ou um ator durante toda a execução da interação. Esta é representada por uma linha vertical tracejada situada abaixo do objeto. No momento da destruição do objeto, é colocado "X" em cima desta linha como pode se observar na Figura 2.16.

Na linha de vida há a ocorrência do foco de controle ou ativação, que indica que o objeto está participando de um determinado evento. Este foco é representado pelo aumento da espessura da linha de vida como se apresenta na Figura 2.17 após a mensagem 1.

2.5.4 Mensagens ou Estímulos

As mensagens apresentam a ocorrência de comunicação entre os objetos envolvidos no processo. Estas mensagens podem envolver ou não a execução

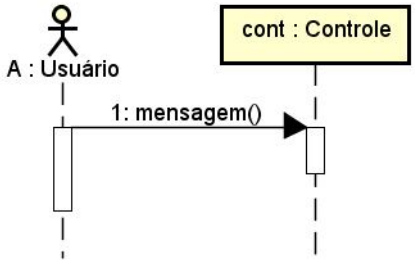

Figura 2.17: Foco de ativação.

de métodos. Geralmente, o estímulo inicial é realizado por um ator, e este dá início ao processo de comunicação dos diversos objetos. As mensagens podem envolver dois atores, um ator e um objeto, dois objetos ou um objeto (autochamada).

As mensagens são representadas por linhas horizontais, cruzando o diagrama entre os seus componentes, iniciando no componente que originou o estímulo e finalizando com uma seta preta no componente que recebeu o estímulo. A ordem entre as mensagens está dada pela posição horizontal ao contar de cima para baixo.

Acima da seta é colocado o método que foi chamado e seus parâmetros, se for necessário. É recomendável identificar a sequência de chamadas com uma numeração sequencial, separada do identificador do método por dois pontos (:). No caso de não haver a execução de um método, o texto conterá apenas a numeração. Na Figura 2.18 é apresentado um exemplo de mensagens entre os componentes de um sistema.

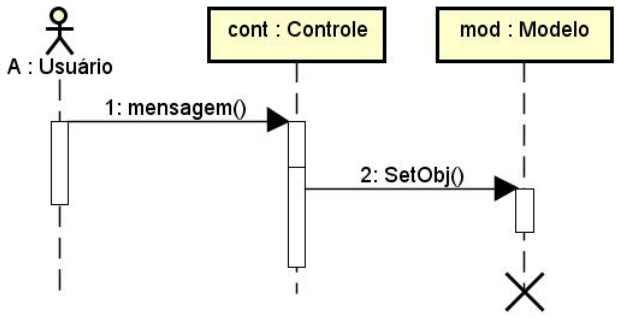

Figura 2.18: Mensagens ou estímulos.

Um objeto também pode ser criado durante a execução de um processo, mediante uma mensagem que execute a criação. Neste caso, o objeto será colocado na mesma altura da mensagem que determina sua criação, representada por uma seta aberta e tracejada, com o estereótipo «create» como se apresenta na Figura 2.19.

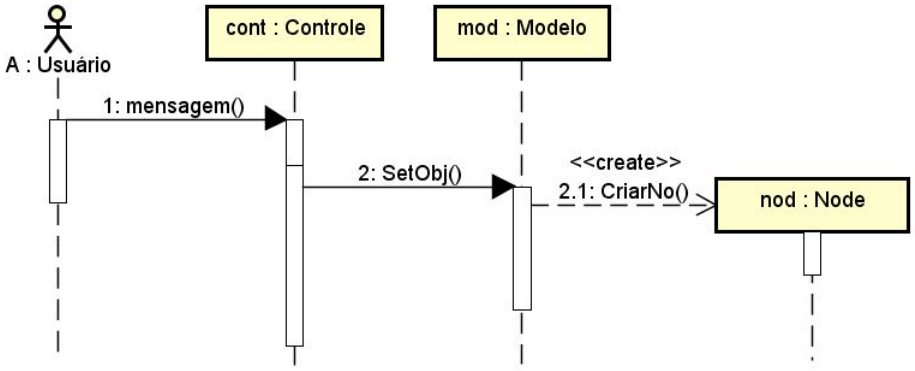

Figura 2.19: Mensagem de criação.

Um método destrutor também pode ser chamado por uma mensagem, para destruir a instância de um objeto. Esta mensagem é representada com uma seta continua com o estereótipo «destroy». Neste caso, a linha de vida do objeto será interrompida por um "X" como pode se observar na Figura 2.20.

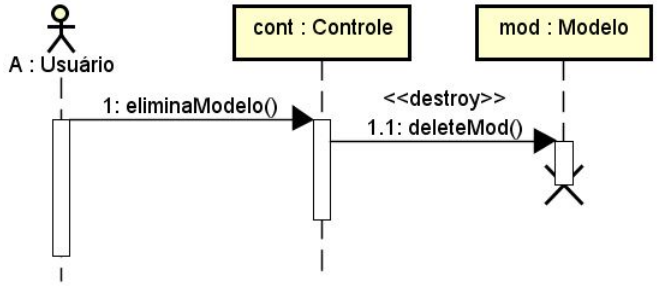

Figura 2.20: Mensagem de destruição.

As mensagens também podem ter como destino o mesmo objeto que a originou. Este caso é chamado de autochamada e a mensagem parte do objeto e chega a sua própria linha de vida como se apresenta na Figura 2.21.

As mensagens de retorno são respostas aos estímulos enviados para os objetos de destino. São representadas por setas tracejadas contendo o retorno esperado pelo processo. Para que o diagrama fique mais limpo, é recomendável apenas colocar os retornos mais importantes. Um exemplo deste tipo de mensagens é apresentado na Figura 2.22.

As mensagens assíncronas representam eventos que o objeto que a enviou não precisa aguardar uma resposta para continuar suas atividades. Por exemplo enviar um *e-mail*, onde a espera da resposta não impede continuar com o processo. Este tipo de mensagem é representado por uma seta vazada.

As mensagens encontradas representam eventos que são enviados por objetos desconhecidos, que estão fora do escopo do diagrama. Por exemplo

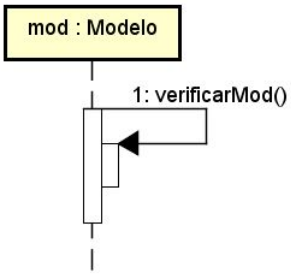

Figura 2.21: Automensagem.

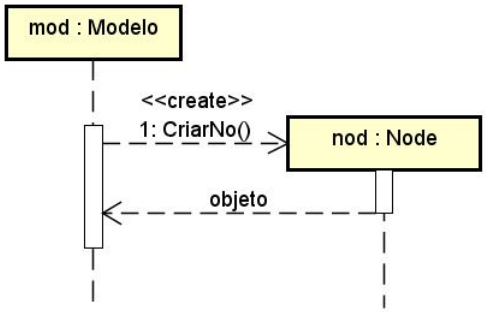

Figura 2.22: Mensagem de retorno.

receber um *e-mail* de diversos clientes externos, onde a origem da mensagem é desconhecida. Este tipo de mensagem é representada por um círculo preto na origem da seta da mensagem como se pode observar na Figura 2.23(a).

As mensagens perdidas representam eventos que são recebidos por objetos desconhecidos, que estão fora do escopo do diagrama. Por exemplo enviar um *e-mail* a um cliente não determinado. Este tipo de mensagem é representada por um círculo preto na ponta da seta da mensagem como se pode observar na Figura 2.23(b).

2.5.5 Fragmentos de interação

São trechos de um diagrama que podem ser detalhados ou reutilizados em outros diagramas. Estes trechos podem apresentar condições especiais de execução. Este tipo de estrutura é representado por um retângulo que envolve a interação, com uma aba que indica seu operador.

Para utilizar este recurso, podemos empregar os seguintes operadores apresentados na Tabela 2.3.

Um exemplo de fragmento de interação é apresentado na Figura 2.24.

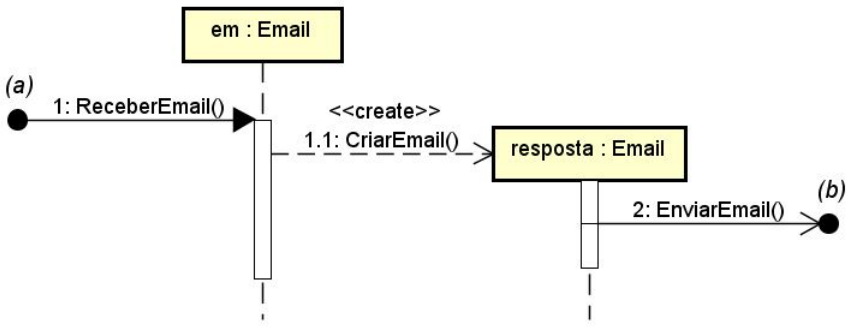

Figura 2.23: (a) Mensagem encontrada; (b) Mensagem perdida.

Tabela 2.3: Operadores de fragmentos de interação.

Operador	Significado
ref	Indica que este trecho do diagrama é uma referência de outro diagrama.
alt	Indica uma alternativa de execução entre mais de um comportamento.
opt	indica um comportamento que poderá ser executado caso sua condição de execução seja satisfeita.
par	Indica processamento paralelo ou chamadas concorrentes a métodos diferentes.
loop	Indica que este trecho entrará em loop de execução.

2.6 MATLAB

O MATLAB (*Matrix Laboratory*) [17] é um software de análise numérica e visualização de dados com capacidades gráficas e de programação. Embora seu nome signifique Laboratório de Matrizes, seus propósitos atualmente são mais amplos. Este software foi criado como um programa para operações matemáticas com matrizes, mas depois transformou-se em um sistema computacional bastante útil e flexível. Inclui um ambiente de desenvolvimento integrado, como construções de programação orientada a objeto.

Sua linguagem é baseada em uma linguagem matemática simples, tornando seu ambiente de trabalho fácil de ser utilizado. Assim, o MATLAB é uma ferramenta e uma linguagem de programação de alto nível, e tem como principais funções: construção de gráficos e compilação de funções, manipulação de funções específicas de cálculo e variáveis simbólicas. Além disso, este software possui funções integradas para executar muitas operações, e uma grande quantidade de bibliotecas auxiliares (*Toolboxes*) que podem ser adicionadas para aumentar essas funções.

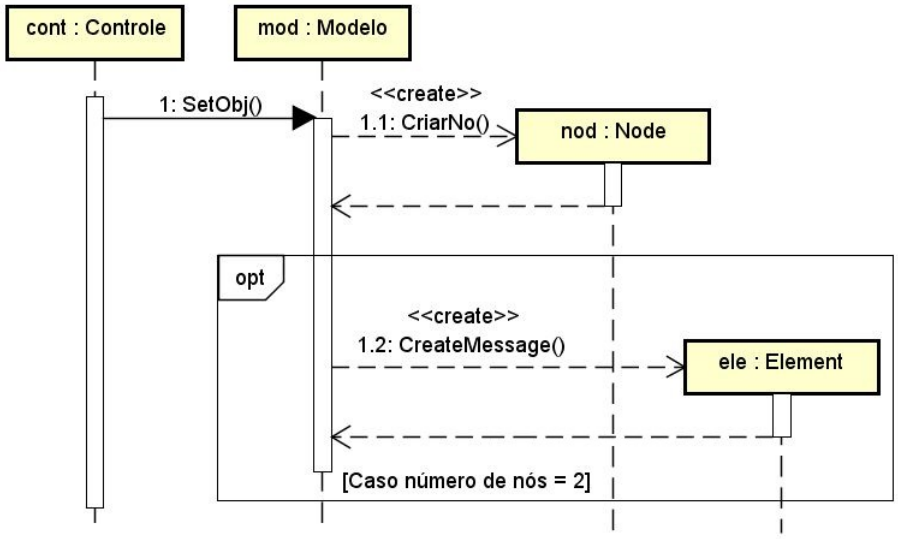

Figura 2.24: Fragmento de interação.

2.6.1 POO em MATLAB

As capacidades de programação orientada a objetos de MATLAB permitem desenvolver aplicações que precisam de processos de cálculo complexos com uma grande rapidez. É possível definir classes e aplicar conceitos de POO em MATLAB que permitem a reutilização de código, a herança, o encapsulamento e o comportamento de referência sem necessidade de prestar atenção nas tarefas habituais de baixo nível requeridas em outras linguagens [17].

A programação orientada a objetos em MATLAB implica o uso de:

- Arquivos de definição de classes, que permitem a definição de propriedades (atributos), métodos e eventos.
- Classes com comportamento de referência, que ajudam na criação de estruturas de dados como listas vinculadas.
- Emissores e receptores, que permitem monitorar as ações e trocas de informação das propriedades dos objetos.

2.6.2 GUI MATLAB

As GUI, também chamadas de interfaces gráficas de usuário ou interfaces de usuário, permitem um controle simples das aplicações de software.

As aplicações de MATLAB são programas autônomos de MATLAB com uma interface gráfica de usuário GUI que automatiza uma tarefa ou um cálculo.

Geralmente, uma GUI inclui controles como menus de ferramentas, botões e controles deslizantes.

O ambiente GUIDE do MATLAB (ambiente de desenvolvimento de apps GUI) proporciona ferramentas para desenhar interfaces de usuário para aplicações personalizadas. Mediante o editor de design do ambiente GUIDE, é possível desenhar graficamente a interface de usuário. Depois, o GUIDE gera automaticamente um código de MATLAB para construir a interface, este código pode ser modificado para programar o comportamento da aplicação [17].

Com a finalidade de ter um maior controle do design e desenvolvimento, também é possível criar um código de MATLAB que defina as propriedades e comportamentos de todos os componentes de uma GUI. MATLAB contém uma funcionalidade integrada que permite criar uma GUI de forma programática. Também tem a possibilidade de agregar quadros de diálogo, controles de interface de usuário, como botões e controles de deslizamento, e *containers*, como painéis e grupos de botões.

2.6.3 Funções Callback de MATLAB

Em um ambiente GUI, os componentes da interface gráfica (botões, menus etc.) possuem funcionalidades distintas. Cada tipo de componente possui um grupo de ações que um usuário pode exercer sobre ele, sendo que para cada uma dessas ações é possível associá-la a uma função *callback*.

Uma função *callback* é definida como a resposta a uma ação que um objeto GUI realizará quando o usuário o ativa. Por exemplo, suponha que em uma janela exista um botão que quando é pressionado executa uma série de tarefas. A execução das tarefas associadas a uma ação é feita pela função *callback*.

Quando uma GUI é criada no MATLAB, o ambiente GUIDE gera automaticamente um arquivo com modelos (*templates*) das funções *callback* relacionadas aos principais eventos que podem ocorrer com cada componente gráfico da interface. O programador deve apenas implementar o corpo das funções *callback* com os procedimentos adequados para executar as ações requeridas. Para aplicações simples, essas funções geradas automaticamente são suficientes, porém para aplicações com funcionalidades mais complexas pode ser necessário criar novas funções associadas aos eventos desejados.

Ao remover ou adicionar componentes da interface, o arquivo das funções *callback* é atualizado automaticamente as funções após salvar ou rodar a interface no ambiente GUIDE.

2.7 Eventos de mouse

Um evento de mouse em um canvas é uma ação realizada pelo usuário com o mouse em uma área de desenho (canvas) da interface gráfica. A programação de eventos de mouse é desenvolvida através de funções *callback* específicas que são executadas quando ocorre uma ação.

Os eventos de mouse em canvas utilizados no desenvolvimento deste trabalho são:

- *Mouse button pressed* (*down*): ocorre quando o usuário pressiona um botão do mouse.
- *Mouse move*: ocorre quando o usuário movimenta o mouse.
- *Mouse button released* (*up*): ocorre quando o usuário libera um botão do mouse que foi pressionado.

No próximo capítulo será detalhado o tratamento dos eventos de mouse em canvas realizado neste trabalho. As funções *callback* dos respectivos eventos de mouse no ambiente MATLAB serão explicadas.

3
Classe Emouse

O objetivo principal deste trabalho é o desenvolvimento de uma classe genérica (no contexto da POO), chamada de Emouse, que facilite o desenvolvimento de aplicações que lidam com eventos de mouse em canvas (área de desenho de uma aplicação GUI).

A classe abstrata *Emouse* apresenta, além do método construtor, quatro métodos concretos privados (implementados) e três métodos abstratos que devem ser implementados pelo usuário. Sua utilização se realiza mediante a criação de uma subclasse cliente que herda suas propriedades e implementa os três métodos abstratos apresentados na seção 3.3.2. A utilização da classe *Emouse* é detalhada na seção 3.4.

A vantagem dessa estratégia de implementação é clara: a maior parte da complexidade de lidar com eventos de mouse em canvas é tratada nos métodos concretos da superclasse abstrata *Emouse*. A subclasse cliente só precisa tratar do que é específico para sua aplicação. Esse tipo de reuso de código por herança é um dos poderes da POO.

3.1
Objetos figure e axes do MATLAB

No ambiente gráfico do MATLAB, um canvas é associado a uma entidade de interface gráfica. Quando uma figura é feita no MATLAB, dois objetos são criados *figure* e *axes*. O primeiro (*figure*) é a janela onde os comandos gráficos desenham seus resultados; o segundo objeto (*axes*) contém as propriedades do espaço de desenho (canvas) que está dentro da janela [17]. O eixo X do sistema de coordenadas destes objetos é crescente da esquerda para a direita e o eixo Y é crescente de baixo para cima.

Estes dois objetos são definidos como propriedades da classe *Emouse* e devem ser fornecidos pelo usuário na sua utilização.

O objeto ***figure*** tem atributos como cor, nome, posição etc. Desses atributos os mais importantes para o desenvolvimento da classe *Emouse* são:

- CurrentPoint: posição atual do mouse nos eixos do objeto *figure*. Este atributo é obtido do objeto *figure* mediante a função *get* do MATLAB, e é utilizado no método *eButtonDown* da classe *Emouse*.

• SelectionType: atributo que fornece informação sobre o último botão pressionado do mouse que ocorreu dentro da janela da figura. Esta informação indica o tipo de seleção feita: botão direito, esquerdo ou central, também determina se um botão foi pressionado duas vezes consecutivas. Este atributo pode ser obtido mediante a função *get* do MATLAB, e é utilizado no método *eButtonDown* da classe *Emouse*, conforme será detalhado na sequência.

A seguir são apresentados os métodos do objeto *figure* de maior importância para a classe *Emouse*:

• WindowButtonMotionFcn: função *callback* que é executada sempre que o usuário move o mouse dentro da janela da figura.

• WindowButtonDownFcn: função *callback* que é executada sempre que o usuário pressiona um botão do mouse dentro da janela da figura.

• WindowButtonUpFcn: função *callback* que é executada sempre que o usuário libera um botão do mouse.

Estas funções *callback* são implementadas no método construtor da classe *Emouse* e sua utilização é descrita na seção 3.3.

O objeto ***axes*** tem atributos como o limite dos eixos, o tamanho da fonte, a cor de fundo etc. Entre esses atributos, o mais importante para o desenvolvimento da classe *Emouse* é:

• CurrentPoint: posição atual do mouse nos eixos do objeto *axes*. Este atributo é obtido do objeto *axes* mediante a função *get* do MATLAB, e é utilizado nos métodos *eButtonDown*, *eMouseMove* e *eButtonUp* da classe *Emouse*.

3.2 Atributos da classe Emouse

A classe *Emouse* tem os atributos descritos a seguir, que armazenam a informação que se obtém com um evento de mouse:

• dialog: armazena um *handle* (referência) para o objeto *figure* associado aos eventos de mouse. Deve ser fornecido pelo cliente da classe *Emouse* na sua utilização.

• canvas: armazena um *handle* (referência) para o objeto *axes* corrente associado aos eventos de mouse. O cliente da classe *Emouse* deve fornecer um objeto *axes* inicial na utilização da classe *Emouse*. Um canvas inicial deve ser fornecido para que o evento *mouse move* (descrito na seção 2.7) possa ser inicializado, por exemplo, quando o usuário precise encontrar uma coordenada ou um ponto no canvas antes de pressionar um botão do mouse.

• mouseButtonMode: atributo definido como string, mas que funciona como um booleano, pois apresenta dois estados; *up*, quando os botões do mouse

não estão sendo pressionados, e *down*, quando um dos botões do mouse é pressionado. Este atributo é inicializado no estado *up*.

• whichMouseButton: atributo que determina qual dos botões do mouse foi pressionado ou liberado por última vez, é definido como um string, apresenta os estados a seguir: *left* para o botão esquerdo, *right* para o botão direito, *center* para o botão do centro, *none* quando nenhum botão é pressionado. Este atributo também define se o usuário pressiona um botão duas vezes consecutivas com o estado *double click*. É inicializado no estado *none*.

• currentPosition: este atributo contém a posição atual do mouse no canvas corrente; é definido como uma matriz de ordem 1x2, onde a variável 1x1 apresenta a coordenada X e a variável 1x2, a coordenada Y.

3.3 Métodos da classe Emouse

A classe *Emouse* apresenta os métodos concretos e abstratos descritos a seguir. Na Figura 3.1 apresenta-se a definição da classe *Emouse* com seus atributos e métodos.

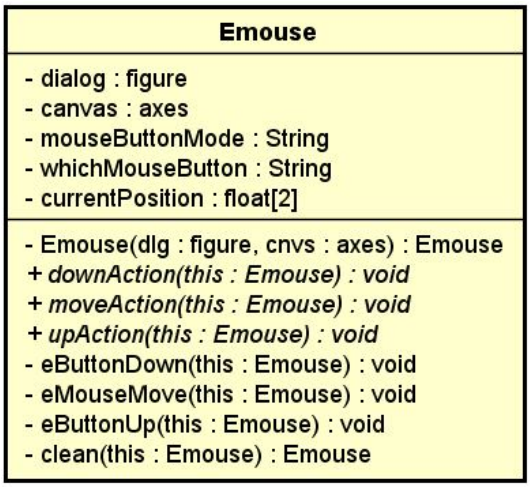

Figura 3.1: Definição da classe *Emouse*.

3.3.1 Métodos concretos da classe Emouse

Os métodos concretos atualizam os atributos descritos na seção anterior para gerenciar os eventos de mouse.

• Método construtor (Emouse), destinado a inicializar um objeto desta classe. Este método define a propriedade *units* da janela associada aos eventos de mouse (atributo *dialog*) como *pixels* e associa os eventos *mouse button*

pressed, *mouse move* e *mouse button released* (descritos na seção 2.7) na janela com os métodos concretos *eButtonDown*, *eMouseMove* e *eButtonUp*, respectivamente. Seus argumentos de entrada são a janela (*dialog*) e o canvas inicial associados aos eventos de mouse. Seu funcionamento é apresentado na sequência.

• eButtonDown: este método é uma função *callback* associada ao evento *mouse button pressed* na janela (*dialog*). O método encontra, na lista de *axes* (canvases) da janela, o objeto *axes* em que um botão do mouse foi pressionado e atualiza o atributo *dialog* com essa informação. O método também determina qual botão do mouse foi pressionado, atualiza a propriedade *whichMouseButton*, define a propriedade *mouseButtonMode* como *down*, determina a posição atual (*currentPosition*) onde um botão do mouse foi pressionado e chama o método abstrato *downAction*. Tem como argumento de entrada o objeto *this*, objeto corrente de uma subclasse cliente da classe *Emouse*.

• eMouseMove: este método é uma função *callback* associada ao evento *mouse move* na janela (*dialog*). Este método é encarregado de capturar as coordenadas do mouse no canvas, atualizar o atributo *currentPosition* com elas e chamar o método abstrato *moveAction*. Tem como argumento de entrada o objeto *this*.

• eButtonUp: este método é uma função *callback* associada ao evento *mouse button released* na janela (*dialog*). Este método define a propriedade *mouseButtonMode* como *up*, determina a posição atual (*currentPosition*) onde o botão do mouse foi liberado, chama o método abstrato *upAction* e redefine o atributo *whichMouseButton* como *none*. Tem como argumento de entrada o objeto *this*.

• clean: este método atribui os valores inicias às propriedades da classe.

O proposito do método construtor é ativar os eventos de mouse na janela que contém o canvas, em outras palavras, com este método o canvas fica atento a qualquer evento que possa ocorrer com o mouse. Para o desenvolvimento do método construtor foram considerados os três eventos de mouse apresentados na seção 2.7:

- Um botão do mouse é pressionado (*mouse button pressed*). A linha de código a seguir, permite implementar este evento:

set(this.dialog, 'WindowButtonDownFcn', @this.eButtonDown);

essa linha de código associa o método *eButtonDown* da classe *Emouse* com a função *callback WindowButtonDownFcn* do objeto *dialog*, que é ativada sempre que um botão do mouse é pressionado dentro da janela.

- O mouse está se movimentando (*mouse move*). Este evento é implementado mediante a linha de código a seguir:

set(this.dialog, 'WindowButtonMotionFcn', @this.eMouseMove);
essa linha de código associa o método *eMouseMove* da classe *Emouse* com a função *callback WindowButtonMotionFcn* do objeto *dialog*, que é ativada sempre que o mouse se movimenta dentro da janela.

- Um botão do mouse é liberado (mouse button released). Este evento é implementado mediante a linha de código a seguir:

set(this.dialog, 'WindowButtonUpFcn', @this.eButtonUp);
essa linha de código associa o método *eButtonUp* da classe *Emouse* com a função *callback WindowButtonUpFcn* do objeto *dialog*, que é ativada sempre que um botão do mouse é liberado.

3.3.2 Métodos abstratos da classe Emouse

A classe *Emouse* também tem os métodos abstratos descritos a seguir, que devem ser implementados na subclasse cliente:

- moveAction: este método deve ser implementado com as ações a serem realizadas quando o usuário mover o mouse.
- downAction: este método deve ser implementado com as ações a serem realizadas quando o usuário pressionar um botão do mouse.
- upAction: este método deve ser implementado com as ações a serem realizadas quando o usuário liberar o botão do mouse que foi pressionado.

3.4 Utilização da classe Emouse

Para utilizar a classe *Emouse* devem ser seguidos os passos abaixo:

1. Crie uma subclasse que herde os atributos e métodos da classe *Emouse*.

2. Implemente novos atributos nas propriedades desta subclasse, se forem necessários para o desenvolvimento da aplicação, por exemplo, um atributo que conte o número de vezes que um botão do mouse foi pressionado.

3. Implemente um método construtor para inicializar um objeto da subclasse criada e definir a herança da classe *Emouse*. Este método deve ter como argumentos de entrada um objeto *figure* e um objeto *axes* (canvas) inicial que deverão ser fornecidos à classe *Emouse* na herança.

4. Implemente os métodos abstratos da classe *Emouse*. Novos métodos podem ser implementados na subclasse, se forem necessários para a aplicação, por exemplo, um método que incorpore a utilização de um botão do teclado.

5. Quando for necessário utilizar os eventos do mouse, crie um objeto da nova subclasse fornecendo um objeto *figure* e um objeto *axes* incial que serão associados aos eventos de mouse.

4
Aplicações desenvolvidas

Como exemplos de utilização e para determinar o correto funcionamento da classe Emouse, foram desenvolvidas duas aplicações que precisam do gerenciamento de eventos de mouse. A primeira, denominada e-dles2D, para desenhar pórticos planos em 2D e a segunda, denominada e-Mohr2, para demostrar o comportamento do círculo de Mohr para estado plano de tensões.

Para ilustrar o funcionamento do sistema das aplicações, foram desenvolvidos, em UML, um diagrama de atividade, um de classe e um de sequência para cada uma delas.

Nestas duas aplicações é utilizada a função *inpolygon* do MATLAB como função auxiliar. Esta função determina se um ponto está localizado dentro ou na borda de uma região poligonal [17]. Tem como parâmetros de entrada as coordenadas do ponto, e dois vetores (x, y) com as coordenadas dos pontos que conformam a região poligonal. Como parâmetro de saída obtém-se um booleano, verdadeiro se o ponto está dentro do polígono e falso caso contrário.

4.1
e-dles2D

A aplicação e-dles2D (*Draw Linear Elements Structure 2D*) permite desenhar pórticos planos em 2D e tem dois tipos de entidades principais, nós e elementos lineares, estes últimos representam elementos de barra. Os modelos feitos nesta aplicação podem ser salvos em arquivos de texto (*txt*). Estos arquivos podem ser carregados por a aplicação posteriormente.

4.1.1
Interface Gráfica da aplicação e-dles2D

Uma interface GUI simples para esta aplicação foi implementada. A qual cria objetos das classes que inicializam a aplicação e controla a funcionalidade das entidades da interface gráfica.

Na Figura 4.1 é apresentada a interface gráfica desenvolvida para a aplicação e-dles2D. A seguir é descrito o funcionamento desta interface:

O botão *node* (1), permite criar e desenhar um nó no canvas nas coordenadas clicadas com o mouse.

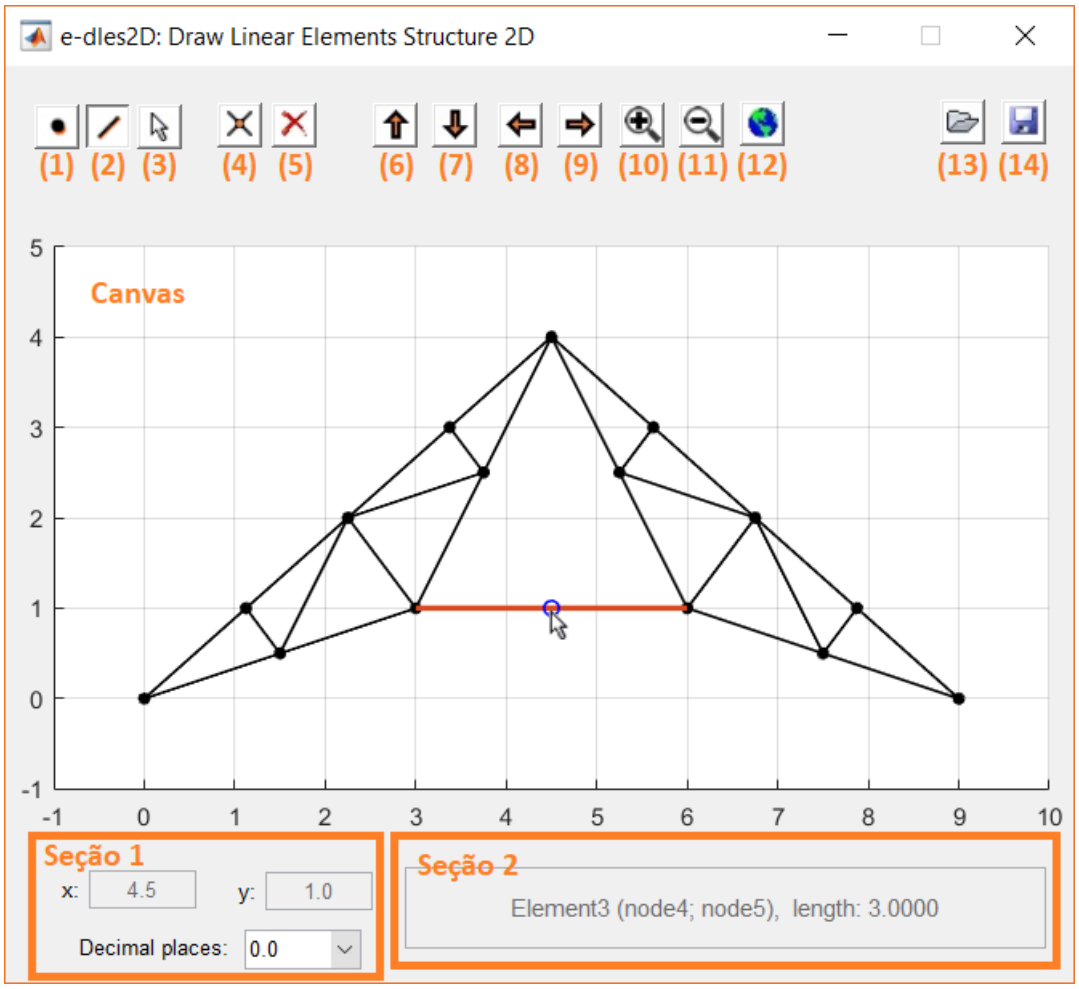

Figura 4.1: Interface Gráfica da aplicação e-dles2D.

O botão *element* (2), permite criar e desenhar um elemento de barra no canvas. Selecionando dois pontos no canvas com o mouse.

O botão *select* (3), permite selecionar uma ou mais entidades desenhadas no canvas com o mouse. É possível acumular entidades selecionadas mantendo o botão *shift*, do teclado, pressionado ou também selecionar várias entidades criando um retângulo com o mouse que envolva as entidades desejadas.

O botão *bar intersection* (4), permite encontrar um nó na interseção de dois elementos de barra selecionados que se cruzam.

O botão *delete* (5), permite deletar da memória e apagar do canvas as entidades selecionadas.

O botão *pan up* (6), permite trasladar a imagem no canvas para cima.

O botão *pan down* (7), permite trasladar a imagem no canvas para baixo.

O botão *pan left* (8), permite trasladar a imagem no canvas à esquerda.

O botão *pan right* (9), permite trasladar a imagem no canvas à direita.

O botão *zoom in* (10), permite diminuir a porção de visão do modelo no

canvas.

O botão *zoom out* (11), permite aumentar a porção de visão do modelo no canvas.

O botão *fit world* (12), permite ajustar a imagem no canvas para que o desenho feito seja visível em sua totalidade.

O botão *open file* (13), permite abrir um modelo feito nesta aplicação.

O botão *save file* (14), permite salvar um modelo desenhado no canvas.

A seção 1, apresenta as coordenadas x e y da posição atual do mouse no canvas. Além disso, contém o menu *decimal places*, que permite selecionar o número de casas decimais destas coordenadas. Por exemplo, na Figura 4.1 pode-se observar que o mouse esta próximo do ponto central do elemento 3, de coordenadas (4.5, 1.0) com uma casa decimal.

A seção 2, permite visualizar o comprimento de um elemento de barra que está sendo desenhado. Também apresenta o nome e as propriedades geométricas da entidade da qual o mouse está mais próximo. Por exemplo, na Figura 4.1 pode-se observar as propriedades do elemento 3, do qual o mouse está mais perto.

4.1.2 Diagrama de Atividade da aplicação e-dles2D

Na Figura 4.2 apresenta-se o diagrama de atividade da aplicação e-dles2D. Neste diagrama podem ser observadas as ações e a comunicação desenvolvida por quatro atores: o usuário, a interface gráfica (GUI), um módulo gerenciador de eventos de mouse e um módulo de análise. Estes dois últimos atores representam as classes descritas anteriormente. Este diagrama permite obter uma visão geral do funcionamento da aplicação e sua interação com o usuário.

4.1.3 Classes da aplicação e-dles2D

No desenvolvimento da aplicação e-dles2D foram criadas nove classes e uma subclasse. A seguir é apresentada uma descrição de cada uma das classes implementadas, além de sua funcionalidade na aplicação.

4.1.3.1 Subclasse dlesEm

A classe *dlesEM* é uma subclasse da classe *Emouse*. Esta subclasse herda todos os atributos e implementa os métodos abstratos da classe *Emouse*. Esta subclasse apresenta os seguintes métodos:

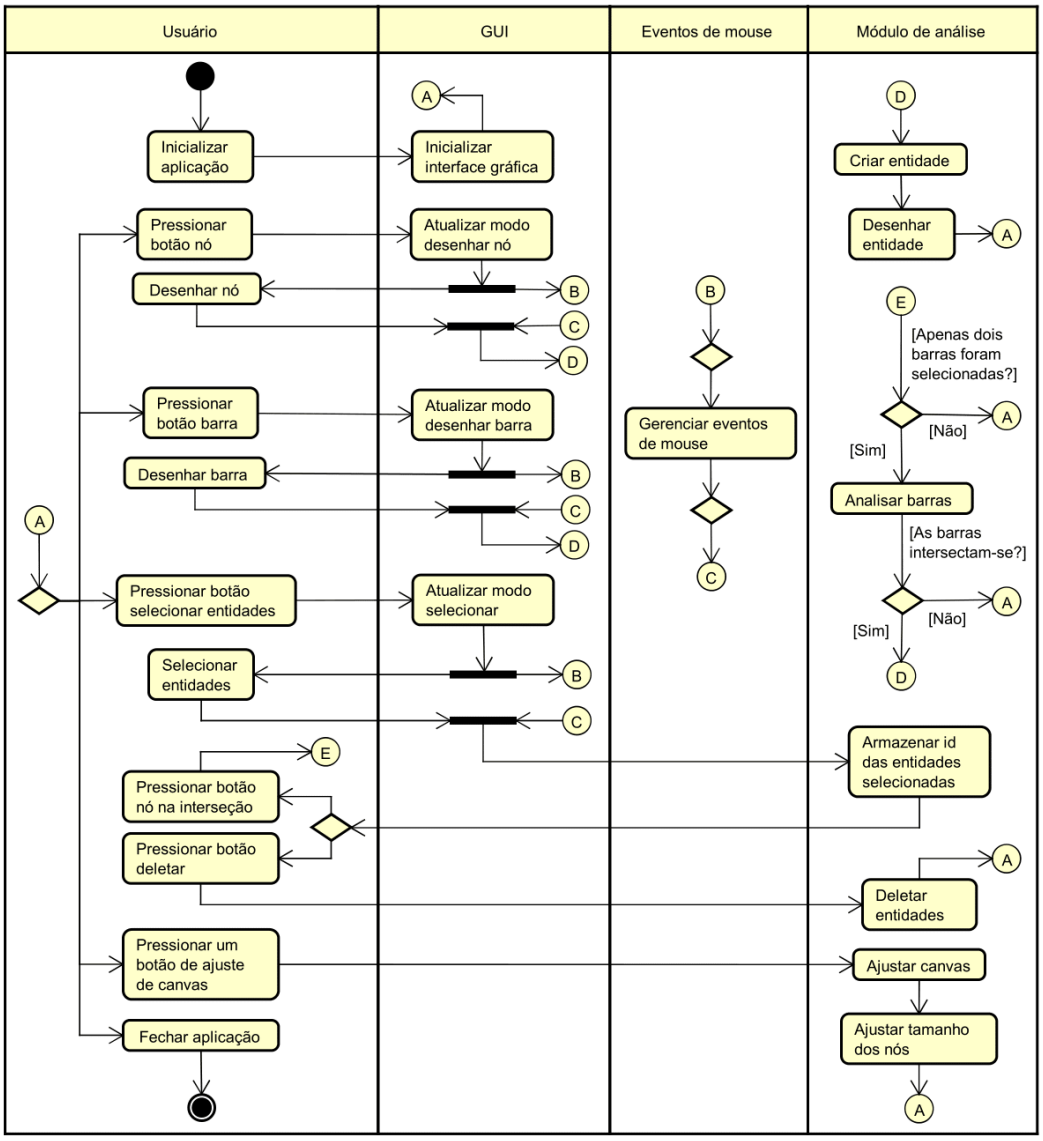

Figura 4.2: Diagrama de atividade e-dles2D.

• Método construtor (dlesEm), destinado a inicializar um objeto desta subclasse e a definir a herança da classe *Emouse*. Seus argumentos de entrada são um objeto *figure* e um objeto *axes* (canvas).

• moveAction: este método desenvolve as ações que ocorrem quando o usuário move o mouse. O método determina se a posição do mouse está dentro dos limites do canvas. Aproxima as coordenadas da posição do mouse ao número de caixas decimais segundo a escolha feita pelo usuário, e as apresenta na interface gráfica. Além disso, cria um objeto do tipo *eSelectObj* e executa seus métodos *node* e *element*, que permitem a seleção de entidades no canvas. Se o mouse está perto de uma entidade, atualiza as coordenadas de sua posição com as coordenadas dessa entidade. De acordo com o modo de desenho que o usuário precise, cria e executa o método necessário; se o programa está no modo

node (desenhe nós), cria um objeto do tipo *eDrawMode* e executa seu método *nodeMode_move*; se o programa está no modo *element* (desenhe elementos de barra), cria um objeto do tipo *eDrawMode* e executa seu método *elementMode_move*; se o programa está no modo *select* (selecione entidades), cria um objeto do tipo *eSelectMode*, e executa seu método *selectMode_move*, que permite a seleção de várias entidades. O método *moveAction* também chama os métodos, *keyPress* e *keyRelease* da subclasse *dlesmEm*, para determinar se as teclas SHIFT, DELETE ou ESC foram pressionadas. Por fim, detecta e apresenta na interface gráfica se um elemento está na posição atual do mouse, para mostrar suas propriedades.

• downAction: este método desenvolve as ações que ocorrem quando o usuário pressiona um botão do mouse. O método determina se o botão foi pressionado dentro dos limites do canvas, também se o botão foi o direito ou o esquerdo. Segundo o modo de desenho que o usuário precise, cria e executa o método necessário; se o programa está no modo *node* (desenhe nós), cria um objeto do tipo *eDrawMode* e executa seu método *nodeMode_down*; se o programa está no modo *element* (desenhe elementos de barra), cria um objeto do tipo *eDrawMode* e executa seu método *elementMode_down*; se o programa está no modo *select* (selecione entidades), cria um objeto do tipo *eSelectMode*, e executa seu método *selectMode_down*.

• upAction: este método desenvolve as ações que ocorrem quando o usuário libera o botão do mouse antes pressionado. O método determina se o botão foi liberado dentro dos limites do canvas, também se o botão foi o direito ou o esquerdo. Segundo o modo de desenho que o usuário precise, cria e executa o método necessário; se o programa está no modo *node* (desenhe nós), cria um objeto do tipo *eDrawMode* e executa seu método *nodeMode_up*; se o programa está no modo *element* (desenhe elementos de barra), cria um objeto do tipo *eDrawMode* e executa seu método *elementMode_up*; se o programa está no modo *select* (selecione entidades), cria um objeto do tipo *eSelectMode*, e executa seu método *selectMode_up*.

• keyPress: se um botão do teclado é pressionado, este método determina qual tecla foi pressionada; se for a tecla DELETE, cria um objeto da classe *eDeleteObj* e executa seu método *deleteObj*, que permite apagar uma ou várias entidades antes selecionadas no canvas; se for a tecla ESC, e uma entidade de elemento de barra está sendo desenhado, o programa cancela o desenho.

• keyRelease: este método determina se um botão do teclado antes pressionado é liberado.

Na Figura 4.3 é apresentada a definição da subclasse *dlesEm*.

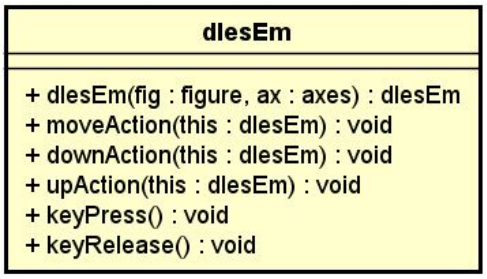

Figura 4.3: Definição da subclasse *dlesEm*.

4.1.3.2 Classe eNode

Para representar as entidades de nós, foi desenvolvida a classe *eNode*, que tem os atributos a seguir:

- id: atributo que armazena o identificador da entidade nó.
- coord_X: este atributo contém a coordenada no eixo global X da entidade nó.
- coord_Y: este atributo contém a coordenada no eixo global Y da entidade nó.

Os métodos da classe *eNode* são:

- Método construtor (eNode), seus argumentos de entrada são as coordenadas x e y de um nó.
- plot: este método recebe os atributos das coordenadas globais de um nó e o desenha no canvas de acordo com o fator de escala definido pela porção de visão do canvas.
- push: método que permite o armazenamento de uma entidade nó. Este armazenamento é feito em forma de pilha: quando um novo nó é criado, é armazenado no topo da pilha.
- pull: este método permite eliminar uma entidade nó. Como o armazenamento é feito em forma de pilha, quando um nó é eliminado, é removido da pilha e o atributo *id* dos nós que estão acima na pilha são atualizados.
- clean: este método limpa os atributos da classe.
- deletePlot: método encarregado de apagar o desenho de um nó no canvas. É utilizado antes do método *pull* pela modificação do atributo *id* que ocorre ao eliminar um nó da pilha.

Na Figura 4.4 é apresentada a definição da classe *eNode*.

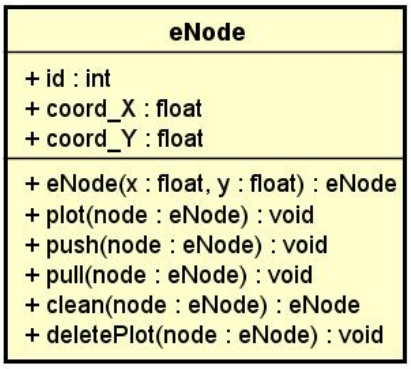

Figura 4.4: Definição da classe *eNode*.

4.1.3.3 Classe eElement

Para representar as entidades de elementos de barra, foi desenvolvida a classe *eElement*, que tem os atributos a seguir:

- id: atributo que armazena o identificador do objeto.
- type: atributo destinado a armazenar o tipo de elemento de barra, tipo Bernoulli ou Timoshenko, será utilizado em uma futura versão da aplicação.
- nodei: atributo definido como um objeto do tipo *eNode*, contém o nó inicial do elemento de barra.
- nodef: atributo definido como um objeto do tipo *eNode*, contém o nó final do elemento de barra.
- material: atributo destinado a armazenar as propriedades do material do elemento tipo barra. Será utilizado em uma futura versão da aplicação.
- section: atributo destinado a armazenar as propriedades da seção transversal do elemento tipo barra. Será utilizado em uma futura versão da aplicação.
- length: atributo que contém o comprimento do elemento tipo barra.
- cosine_X: atributo que contém o cosseno do ângulo de orientação do elemento barra com o eixo global *X*.
- cosine_Y: atributo que contém o cosseno do ângulo de orientação do elemento barra com o eixo global *Y*.

Os atributos length, cosine_X, e cosine_Y são calculados e atribuídos no método *set* do atributo *nodef*.

Os métodos da classe *eElement* são:

- Método construtor (eElement), seus argumentos de entrada mais relevantes são os nós inicial e final de um elemento de barra.
- plot: este método recebe os atributos *nodei* e *nodef* de um elemento de barra e o desenha no canvas de acordo com as coordenadas globais destes

nós.

• push: método que permite o armazenamento de uma entidade elemento de barra. Este armazenamento é feito em forma de pilha: quando um novo elemento é criado, é armazenado no topo da pilha.

• pull: este método permite eliminar uma entidade de elemento de barra. Como o armazenamento é feito em forma de pilha, quando um elemento é eliminado, é removido da pilha e o atributo *id* dos elementos que estão acima na pilha são atualizados.

• clean: este método limpa os atributos da classe.

• deletePlot: método encarregado de apagar o desenho de um elemento de barra no canvas. É utilizado antes do método *pull* pela modificação do atributo *id* que ocorre ao eliminar um elemento da pilha.

Na Figura 4.5 é apresentada a definição da classe *eElement*.

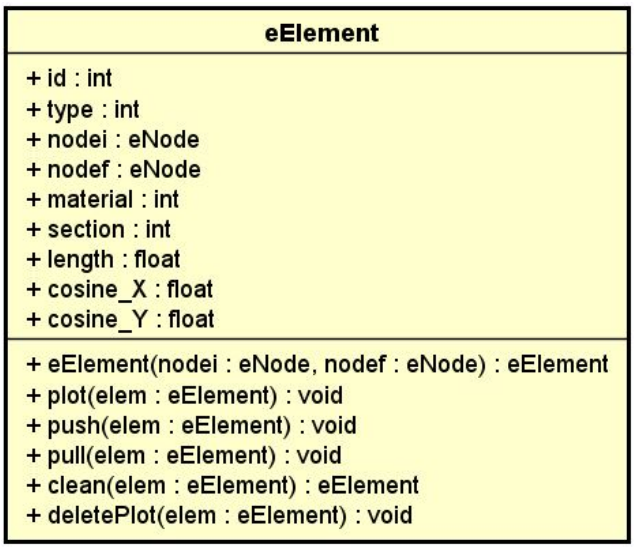

Figura 4.5: Definição da classe *eElement*.

4.1.3.4 Classe eDrawMode

A classe *eDrawMode* contém os modos de desenho no canvas. Esta classe apresenta os métodos que desenham as entidades segundo seu tipo e a ação executada com o mouse. Seus atributos são:

• x: este atributo armazena a coordenada no eixo X da posição do mouse no canvas.

• y: este atributo armazena a coordenada no eixo Y da posição do mouse no canvas.

Os métodos da classe *eDrawMode* são:

• Método construtor (eDrawMode), seus argumentos de entrada são as coordenadas x e y da posição do mouse no canvas.

• nodeMode_down: este método é executado quando a opção *desenhar nó* está ativa na interface gráfica e é pressionado um botão do mouse nas coordenadas x e y do canvas. Se o botão pressionado é o esquerdo, com as coordenadas selecionadas este método cria um nó, o desenha no canvas e o armazena; se o botão é pressionado próximo de um nó existente, o método não realiza ação nenhuma; se o botão é pressionado próximo de um elemento de barra, o método cria, desenha e armazena um novo nó com as coordenadas desejadas, e modifica o elemento de barra selecionado dividindo-o em dois elementos de barra de acordo com o novo nó.

• nodeMode_move: este método é executado quando a opção *desenhar nó* está ativa na interface gráfica e o mouse está se movendo. Para o desenho de nós este método não precisa realizar nenhuma ação.

• nodeMode_up: este método é executado quando a opção *desenhar nó* está ativa na interface gráfica e é liberado o botão do mouse antes pressionado. As ações de criação, desenho e armazenamento de um novo nó ocorrem quando o usuário pressiona o botão, então, quando o usuário o libera não é necessário realizar nenhuma ação. Por essa razão este método não foi utilizado, existe para ser implementado em uma possível futura versão da aplicação.

• elementMode_down: este método é executado quando a opção *desenhar elemento* está ativa na interface gráfica e é pressionado um botão do mouse nas coordenadas x e y do canvas. Se o botão pressionado é o esquerdo, e se o ponto inicial do elemento de barra ainda não foi selecionado, este método armazena os atributos x e y como as coordenadas do ponto inicial do elemento de barra; se o botão foi pressionado próximo de um nó, armazena o atributo *id* do nó selecionado; se o botão foi pressionado próximo de um elemento de barra, armazena o atributo *id* deste elemento. Quando o primeiro ponto do elemento de barra foi selecionado, e o botão do mouse é pressionado pela segunda vez em coordenadas diferentes às do ponto inicial, caso nessa posição não existam nós, este método cria e armazena os nós inicial e final do elemento barra utilizando as coordenadas destes pontos. Caso esses nós existam, não são criados nem armazenados. Com os nós inicial e final determinados, um objeto da classe *eElementComp* é criado e é executado seu método *compute*, e, se esses nós não existem, são desenhados no canvas. Por fim, o método limpa as variáveis utilizadas no armazenamento dos pontos do elemento barra e na determinação da etapa de desenho. Quando o botão direito do mouse é pressionado este método cancela o processo de desenho e limpa as variáveis utilizadas no processo.

• elementMode_move: este método é executado quando a opção *dese-*

nhar elemento está ativa na interface gráfica e o mouse está se movendo. Se o primeiro ponto do elemento de barra já foi selecionado, este método desenha uma linha desde este ponto até as coordenadas da posição atual do mouse, para visualizar o elemento de barra antes de selecionar seu ponto final. Além disso, o método calcula o comprimento desta linha e o apresenta na interface gráfica. Por fim, se o botão direito do mouse é pressionado o processo de desenho desta linha é cancelado.

• elementMode_up: este método é executado quando a opção *desenhar elemento* está ativa na interface gráfica e é liberado o botão do mouse antes pressionado. As ações de criação, desenho e armazenamento de um novo elemento de barra ocorrem quando o usuário pressiona o botão, então, quando o usuário o libera não é necessário realizar nenhuma ação. Por essa razão este método não foi utilizado, existe para ser implementado em uma possível futura versão da aplicação.

Na Figura 4.6 é apresentada a definição da classe *eDrawMode*.

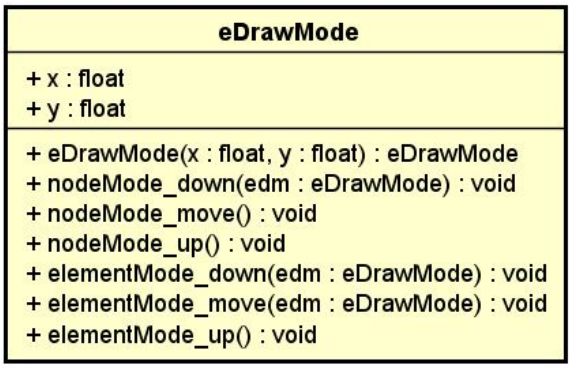

Figura 4.6: Definição da classe *eDrawMode*.

4.1.3.5 Classe eElementComp

Esta classe permite definir um objeto para a criação de elementos de barra, levando-se em conta as modificações que podem ocorrer com o elemento criado devido a nós muito próximos ou colineares ao comprimento deste elemento. Esta classe tem os atributos a seguir:

• nodei: atributo definido como um objeto do tipo *eNode*, contém o nó inicial do elemento de barra que se deseja criar.

• nodef: atributo definido como um objeto do tipo *eNode*, contém o nó final do elemento de barra que se deseja criar.

A classe *eElementComp* apresenta os seguintes métodos:

• Método construtor (eElementComp), seus argumentos de entrada são os nós inicial e final de um elemento de barra que se deseja criar.

• orient2d: este método tem como parâmetros de entrada três pontos, os pontos inicial e final de um elemento de barra e um outro ponto qualquer, o método calcula a área do triângulo conformado por estes pontos, e de acordo com a magnitude da área, determina se este terceiro ponto é colinear ou se está muito próximo do elemento de barra. Este método retorna uma variável do tipo float, se for zero, o ponto analisado está muito próximo ou é colinear ao elemento de barra desenhado.

• onExistingElement: quando um elemento de barra é desenhado próximo de um elemento de barra já existente, e um de seus nós inicial ou final está dentro do comprimento de um elemento de barra existente, este método divide o elemento existente em dois elementos de comprimento menor e armazena o elemento desenhado.

• verticalElement: este método encontra as entidades muito próximas ou colineares que existem entre os dois pontos do elemento de barra que se deseja criar. O método tem como parâmetros de entrada as coordenadas no eixo X, dos pontos inicial e final do elemento de barra que vai ser criado. O método obtém as coordenadas Y dos pontos mencionados mediante os atributos *nodei* e *nodef*. Com essas coordenadas, este método analisa cada nó dos elementos de barra existentes mediante o método *orient2D*, se os dois nós de um elemento de barra existente são colineares o estão muito próximos do elemento de barra que vai ser criado, o elemento analisado é armazenado como elemento de barra colinear ou próximo. Também, da mesma forma, utilizando as coordenadas do elemento que vai ser criado, este método analisa cada nó existente com o método *orient2D*, e se o nó analisado é colinear ou está muito próximo do elemento que vai ser criado, é armazenado como nó colinear ou próximo. O presente método tem como parâmetros de saída o número de elementos de barra próximos ou colineares, o número de nós próximos ou colineares, um vetor com os elementos de barra próximos ou colineares, um vetor com os nós próximos ou colineares e um vetor com a ordem dos nós mencionados, do mais perto até o mais afastado do ponto inicial do elemento que vai ser criado. Este método funciona só se o elemento que vai ser criado é vertical, porque os nós inicial e final desses elementos terão sempre as mesmas coordenadas no eixo X, por isso, são utilizadas as coordenadas no eixo Y para determinar a ordem dos elementos próximos ou colineares.

• nonVerticalElement: este método é igual ao método acima, mas funciona para qualquer orientação do elemento que vai ser criado, com exceção da orientação vertical, pois são utilizadas as coordenadas no eixo X para de-

terminar a ordem dos elementos próximos ou colineares, lembrando que as coordenadas no eixo X dos nós inicial e final de um elemento vertical são iguais.

• compute: se os nós do elemento que vai ser criado foram selecionados próximos de um mesmo elemento existente, este método utiliza o método *onExistingElement*, para processar e criar o elemento. Se o elemento que vai ser criado não é desenhado próximo de um elemento existente, e se sua orientação é vertical, este método determina as entidades próximas ou colineares mediante o método *verticalElement*, se o elemento a ser criado tem outra orientação, este método determina essas entidades mediante o método *nonVerticalElement*. Com os parâmetros de saída destes métodos, o elemento desenhado é processado e criado, se existem nós próximos ou colineares, são criados, desenhados e armazenados elementos de barra que unem estes nós, se dois destes nós formam um elemento de barra existente, entre estes dois nós não se realiza este processo, com a finalidade de não sobrescrever o elemento já existente. Condições de superposição de elementos, como criar um nó dentro de um elemento de barra, são levadas na conta, para dividir os elementos existentes se for necessário, e evitar armazenamentos repetidos.

Na Figura 4.7 é apresentada a definição da classe *eElementComp*.

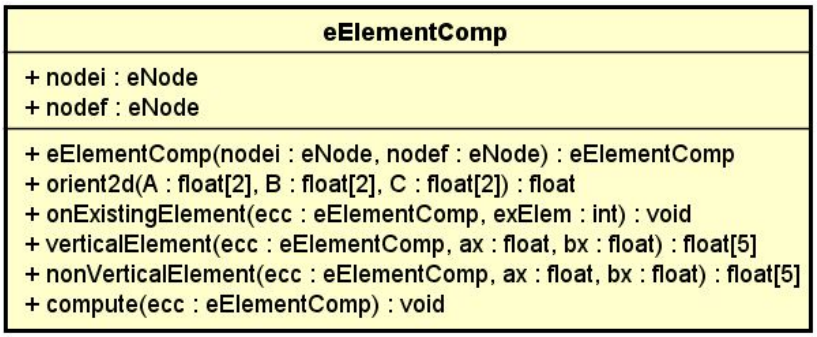

Figura 4.7: Definição da classe *eElementComp*.

4.1.3.6 Classe eSelectMode

A classe *eSelectMode* contém os modos de seleção de entidades no canvas. Esta classe apresenta os métodos que permitem selecionar as entidades existentes de acordo com o tipo de entidade e a ação executada com o mouse. Seus atributos são:

• x: este atributo armazena a coordenada no eixo X da posição do mouse no canvas.

• y: este atributo armazena a coordenada no eixo Y da posição do mouse no canvas.

Os métodos da classe *eSelectMode* são executados quando a opção *select* está ativa na interface gráfica, estes métodos são:

• Método construtor (eSelectMode), seus argumentos de entrada são as coordenadas x e y da posição do mouse no canvas.

• selectMode_down: este método é executado quando é pressionado um botão do mouse nas coordenadas x e y do canvas. Se o botão pressionado é o esquerdo e se foi pressionado próximo de um nó ou um elemento de barra, esta entidade é selecionada, se também, foi pressionada a tecla shift do teclado, este método acumula o atributo *id* das entidades clicadas em duas pilhas, uma de nós e uma de elementos de barra. Se é selecionada uma entidade que já tinha sido selecionada antes, é tirada da pilha correspondente. Este método redesenha as entidades selecionadas de uma outra cor para saber que foram selecionadas. Se o botão esquerdo do mouse é pressionado em uma coordenada x, y do canvas, sem clicar nenhuma entidade, são reinicializadas todas as variáveis utilizadas na seleção de entidades e é armazenado o ponto clicado.

• selectMode_move: se foi clicado um ponto do canvas, o botão esquerdo do mouse continua pressionado e o mouse é arrastrado, este método desenha um retângulo com ponto inicial igual ao ponto clicado, e como ponto oposto final, o ponto da posição atual do mouse.

• selectMode_up: este método é executado quando o botão esquerdo do mouse antes pressionado é liberado. O método *selectMode_up*, utilizando a função *inpolygon* do MATLAB, encontra todas as entidades que fiquem dentro do retângulo desenhado no método *selectMode_move*, e armazena o atributo *id* das entidades encontradas em duas pilhas, uma de nós e uma de elementos de barra. Este método também leva em conta se um ou mais elementos já foram selecionados com o método *selectMode_down* para evitar selecionar uma entidade duas vezes.

Na Figura 4.8 é apresentada a definição da classe *eSelectMode*.

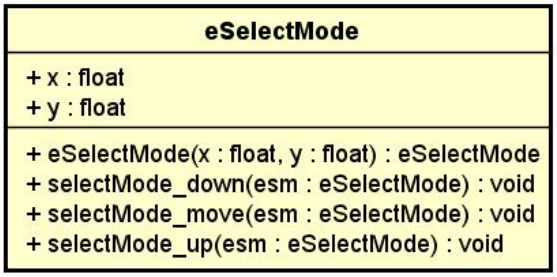

Figura 4.8: Definição da classe *eSelectMode*.

4.1.3.7 Classe eDeleteObj

A classe *eDeleteObj* não tem atributos e contém um único método, além do construtor, que permite eliminar as entidades selecionadas no canvas. Estes métodos são descritos abaixo:

- Método construtor (eDeleteObj).
- deleteObj: quando se tem entidades selecionadas, para evitar conflitos quando são eliminadas das pilhas, este método as ordena em forma descendente, de acordo com seu atributo *id*. Depois, executa, para cada entidade, seu método *deletePlot*, para apagá-las do canvas, e finalmente, executa, para cada entidade, seu método *pull*, para tirá-las das pilhas de armazenamento. Este processo é feito para cada tipo de entidade independentemente, elementos de barra e nós. Além disso, quando termina o processo de eliminação, as variáveis utilizadas na seleção de entidades são são reinicializadas, e os desenhos que indicam a seleção no canvas são apagados. Este método permite apagar um elemento de barra sem apagar seus nós, mas não permite apagar um nó que pertence a um elemento de barra não selecionado. Este método é ativado pressionando o botão *delete object* da interface gráfica ou pressionando a tecla DELETE do teclado.

Na Figura 4.9 é apresentada a definição da classe *eDeleteObj*.

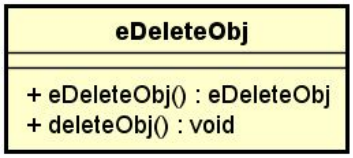

Figura 4.9: Definição da classe *eDeleteObj*.

4.1.3.8 Classe eSelectObj

A classe *eSelectObj* permite reconhecer se o mouse está próximo de uma entidade no canvas. Esta classe tem um único atributo, apresentado abaixo:

- currentPosition: este atributo contém a posição atual do mouse no canvas; está definido como uma matriz de ordem 1x2, onde a variável 1x1 apresenta a coordenada X e a variável 1x2 a coordenada Y.

A classe *eSelectObj* apresenta os métodos a seguir:

- Método construtor (eSelectObj), seu argumento de entrada é a posição atual do mouse no canvas.

• rotTras: este método permite rotacionar e trasladar um ponto em um plano, tem como parâmetros de entrada as coordenadas do ponto que vai ser transformado, as unidades em X e em Y que vai ser trasladado e os cossenos do ângulo, com relação ao eixo X e Y globais, que vai a ser rotacionado. Como parâmetro de saída se obtém uma matriz de ordem 1x2, com as coordenadas X e Y do ponto rotacionado e trasladado.

• node: se o mouse não está próximo de um elemento de barra, este método percorre cada nó e com suas coordenadas cria uma circunferência, invisível no canvas, ao redor de cada nó. Com a função *inpolygon* do MATLAB, o método *node* permite determinar se as coordenadas atuais do mouse estão dentro de uma das circunferências criadas. Quando o mouse está próximo de uma destas circunferências, seu contorno é desenhado no canvas e o id do nó correspondente é armazenado. Quando o mouse não está próximo de uma destas circunferências, o desenho do contorno da última circunferência ativa é apagado e a variável que armazena o id do nó é limpada.

• element: se o mouse não está próximo de um nó, este método percorre cada elemento de barra e cria um retângulo de largura muito pequena, invisível no canvas, ao redor de cada elemento de barra. O método *rotTras* permite trasladar e girar o retângulo para que fique ao redor de um elemento de barra. Com a função *inpolygon* do MATLAB, o método *element* permite determinar se as coordenadas atuais do mouse estão dentro de um dos retângulos criados. Quando o mouse está próximo de um destes retângulos, o elemento de barra dentro do retângulo é desenhado de uma outra cor no canvas e seu id é armazenado; também, este método encontra o ponto do elemento de barra mais próximo da posição do mouse e o armazena. Se o mouse está próximo do centro do elemento de barra, as coordenadas deste centro são armazenadas e uma pequena circunferência com centro nestas coordenadas é desenhada, para facilitar a seleção do centro de um elemento de barra. Quando o mouse não está próximo de um destes retângulos, o desenho do último elemento ativo é apagado e as variáveis que armazenam o id e as coordenadas do ponto mais próximo do elemento são limpadas.

Na Figura 4.10 é apresentada a definição da classe *eSelectObj*.

4.1.3.9 Classe eNode2CrossedElements

Esta classe permite criar um nó na interseção de dois elementos de barra, não apresenta atributos, e tem dois métodos além do construtor, descritos abaixo:

• Método construtor (eNode2CrossedElements), destinado à criação do

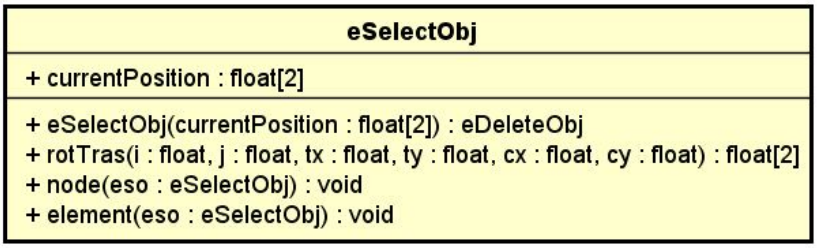

Figura 4.10: Definição da classe *eSelectObj*.

objeto.

• orient2d: este método tem como parâmetros de entrada três pontos, os pontos inicial e final de um elemento de barra e um outro ponto correspondente ao ponto inicial ou final de um outro elemento de barra. Este método calcula a área do triângulo formado por estes pontos. Esta área pode ser negativa ou positiva de acordo com a orientação do terceiro ponto com relação ao primeiro e segundo ponto. Em outras palavras, se uma linha que passa pelo primeiro e segundo ponto divide o plano de desenho em duas regiões, o sinal da área calculada permite determinar em qual destas regiões está o terceiro ponto.

• findNode: se foram selecionados dois elementos de barra no canvas e o botão *intersection* da interface gráfica é pressionado, este método utiliza o método *orient2d* para encontrar a orientação de cada ponto inicial e final de cada elemento de barra com relação ao outro elemento de barra; duas orientações com os pontos inicial e final do primeiro elemento de barra selecionado, uma com o ponto inicial do segundo elemento de barra selecionado e outra como o ponto final do segundo elemento selecionado; e duas com os pontos inicial e final do segundo elemento de barra selecionado, uma com o ponto inicial do primeiro elemento de barra selecionado e outra como o ponto final do primeiro elemento selecionado. De acordo com estas orientações é possível determinar se estes dois elementos de barra selecionados se cruzam, e se isto acontece, uma relação das áreas, encontradas com o método *orient2d*, é utilizada para encontrar o ponto de interseção. Um novo nó é criado, desenhado e armazenado utilizando as coordenadas do ponto de interseção. Os elementos de barra selecionados são divididos e armazenados de acordo com o novo nó.

Na Figura 4.11 é apresentada a definição da classe *eNode2CrossedElements*.

4.1.3.10 Classe eViewCanvas

Esta classe permite a manipulação da imagem no canvas, trasladá-la na direção horizontal ou vertical, aumentar ou diminuir o zoom e ajustá-lo para

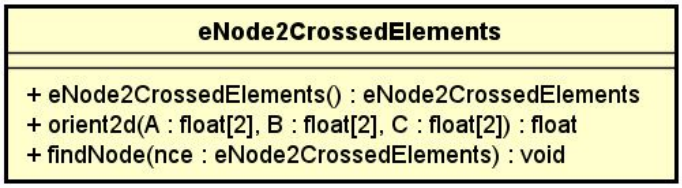

Figura 4.11: Definição da classe *eNode2CrossedElements*.

que o desenho seja visível em sua totalidade. A classe *eViewCanvas* não tem atributos e apresenta os métodos a seguir:

- Método construtor (eViewCanvas).
- fitWorldToViewport: este método ajusta o canvas para que o desenho feito seja visível em sua totalidade, sem alterar as propriedades geométricas do desenho. Se apenas um nó está desenhado, este método coloca o canvas com seu centro nas coordenadas deste único nó. Para realizar este processo são levados em conta apenas os nós, pois os extremos dos elementos de barra estão formados por nós. Este ajuste do canvas é feito de acordo com os nós com maiores e menores coordenadas nos eixos X e Y globais. Este método é ativado pressionando o botão *fit world* da interface gráfica.
- scaleWorldWindow: tem como parâmetro de entrada um fator de escala. Este método escala a imagem no canvas de acordo com o fator de escala. Em outras palavras, aumenta ou diminui a porção de visão do modelo no canvas, sempre mantendo o ponto centro atual. Este método também armazena uma variável de escala segundo o fator de escala dado, para manter uma proporção adequada no desenho dos nós. E por fim executa o método *updateNodeSize*.
- panWorldWindow: tem como parâmetros de entrada um fator de translação horizontal (no eixo X) e um vertical (no eixo Y). Este método traslada a imagem no canvas segundo os fatores de traslado, sempre mantendo o mesmo tamanho.
- zoomIn: este método diminui a porção de visão do espaço do modelo no canvas, dando a aparência de um aumento no tamanho do desenho, utilizando o método *scaleWorldWindow* com um fator de escala de 0,95. Este método é ativado pressionando o botão *zoom in* da interface gráfica.
- zoomOut: este método aumenta a porção de visão do espaço do modelo no canvas, dando a aparência de uma diminuição no tamanho do desenho, utilizando o método *scaleWorldWindow* com um fator de escala de 1,05. Este método é ativado pressionando o botão *zoom out* da interface gráfica.
- panLeft: este método traslada a imagem no canvas à esquerda, utilizando o método *panWorldWindow* com um fator de traslado no eixo X de

-0,05 unidades e um fator de traslado no eixo Y de 0,00 unidades. Este método é ativado pressionando o botão *pan left* da interface gráfica.

• panRight: este método traslada a imagem no canvas à direita, utilizando o método *panWorldWindow* com um fator de traslado no eixo X de 0,05 unidades e um fator de traslado no eixo Y de 0,00 unidades. Este método é ativado pressionando o botão *pan right* da interface gráfica.

• panUp: este método traslada a imagem no canvas para cima, utilizando o método *panWorldWindow* com um fator de traslado no eixo X de 0,00 unidades e um fator de traslado no eixo Y de 0,05 unidades. Este método é ativado pressionando o botão *pan up* da interface gráfica.

• panDown: este método traslada a imagem no canvas para baixo, utilizando o método *panWorldWindow* com um fator de traslado no eixo X de 0,00 unidades e um fator de traslado no eixo Y de -0,05 unidades. Este método é ativado pressionando o botão *pan down* da interface gráfica.

• updateNodeSize: este método é encarregado de apagar o desenho dos nós e redesenhá-los, para modificar seu tamanho de acordo com a porção de visão do canvas.

Na Figura 4.12 é apresentada a definição da classe *eViewCanvas*.

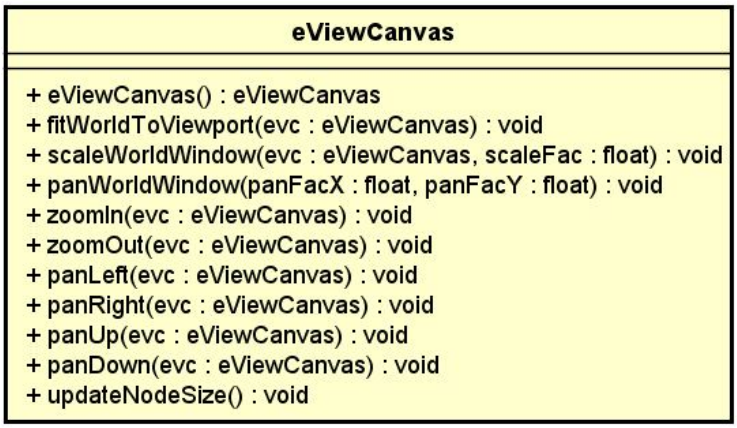

Figura 4.12: Definição da classe *eViewCanvas*.

4.1.3.11 Funções auxiliares

Duas funções auxiliares foram implementadas para permitir salvar e abrir um modelo feito na aplicação e-dles2D. Essas funções são apresentadas a seguir:

• saveFile: esta função auxiliar permite salvar o modelo desenhado no canvas em um arquivo de texto (*txt*) com o formato apresentado na Figura 4.13 que permite ser interpretado pela função *openFile*.

```
FILE: example.txt

NUMBER OF NODES: 4
NUMBER OF ELEMENTS: 3

----------------- NODES --------------------
id          Coord_X          Coord_Y
1            0.0000           0.0000
2            0.0000           4.0000
3            8.0000           4.0000
4            8.0000           0.0000
----------------- END NODES ----------------

----------------- ELEMENTS -----------------
id      Ini_Node        Fin_Node            Length
1              1               2            4.0000
2              2               3            8.0000
3              3               4            4.0000
----------------- END ELEMENTS -------------

END FILE
```

Figura 4.13: Arquivo de texto de um modelo salvado na aplicação e-dles2D.

• openFile: esta função auxiliar permite carregar e desenhar no canvas um modelo feito na aplicação e-dles2D, que foi salvo mediante a função *saveFile*.

4.1.4 Diagrama de Classes da aplicação e-dles2D

Na Figura 4.14 apresenta-se o diagrama de classes da aplicação e-dles2D. Neste diagrama pode-se observar a herança que se obtém da classe *Emouse* e os relacionamentos de dependência (uso) e agregação das classes envolvidas.

4.1.5 Diagrama de Sequência da aplicação e-dles2D

No diagrama de sequência da aplicação e-dles2D pode-se observar a representação da sequência de mensagens que são executadas durante o funcionamento da aplicação. Devido a seu tamanho e para melhorar sua visualização, este diagrama é apresentado na seção de anexos deste trabalho (Figuras 6.1, 6.2 e 6.3). Algumas partes deste diagrama são apresentadas em diagramas de fragmento de iteração de operador *ref*, com o objetivo de melhorar sua organização e visualização.

4.2 e-Mohr2

O círculo de Mohr é uma representação gráfica do estado de tensões (normais e de cisalhamento) que atuam em um ponto de um elemento.

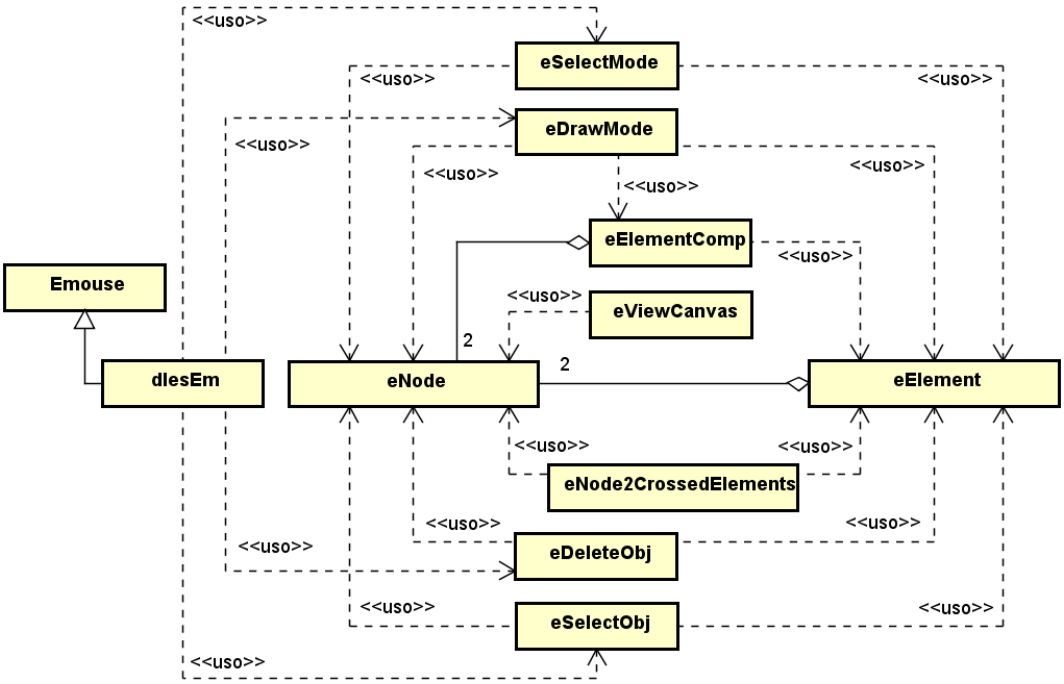

Figura 4.14: Diagrama de classes e-dles2D.

A aplicação e-Mohr2 (*Mohr's circle for plane stress state*), permite observar o comportamento do círculo de Mohr em estado plano de tensões. Neste programa, as componentes de tensão do estado corrente podem ser ajustadas através da manipulação interativa, com o mouse, de alguns pontos de controle. Esta aplicação é baseada no programa educacional [18] desenvolvido em 2004.

4.2.1 Interface Gráfica da aplicação e-Mohr2

Uma GUI simples para esta aplicação foi implementada. A qual cria objetos das classes que inicializam o processo, controla a funcionalidade das entidades da interface gráfica e ingressa dados de entrada preestabelecidos para o círculo de Mohr.

Na Figura 4.15 apresenta-se a interface gráfica desenvolvida para a aplicação e-Mohr2. A seguir é descrito o funcionamento desta interface:

A seção 1 permite a edição dos dados de entrada σx, σy e τxy. Além disso, contém a caixa de seleção *plane stress state* que permite apagar ou desenhar as tensões de estado plano no canvas. Nesta seção também são desenhadas as tensões de estado plano do círculo de Mohr atuantes em um elemento infinitesimal.

A seção 2 permite a edição do ângulo de inclinação do plano de ação das tensões θ, em graus, mediante a entrada de texto ou mediante uma barra

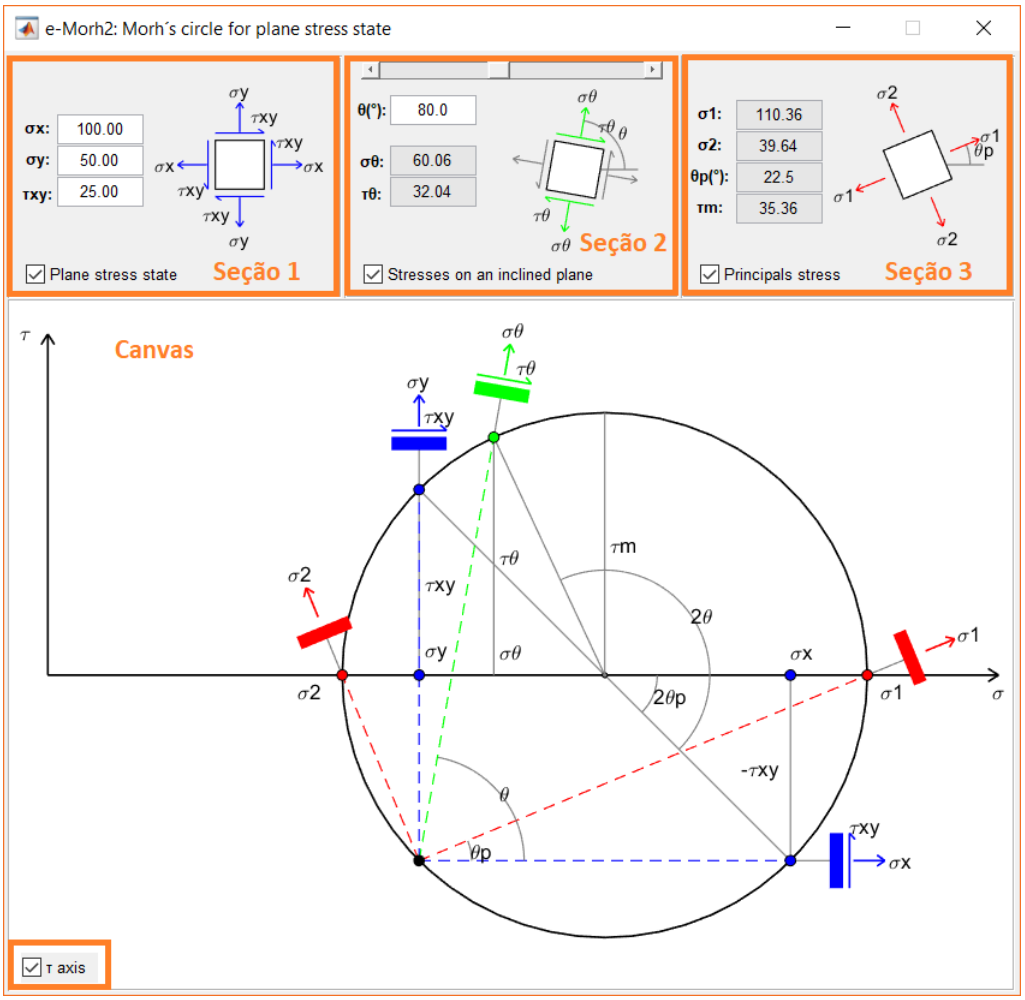

Figura 4.15: Interface Gráfica da aplicação e-Mohr2.

de rolagem. Apresenta também os dados de saída $\sigma\theta$ e $\tau\theta$. Contém a caixa de seleção *stresses on an inclined plane* que permite apagar ou desenhar as tensões que atuam no plano inclinado no canvas. Nesta seção também é desenhado um elemento infinitesimal com as tensões atuantes em um plano inclinado do círculo de Mohr.

A seção 3 apresenta os dados de saída $\sigma 1$, $\sigma 2$, θp, em graus e τm. Contém também a caixa de seleção *principals stress* que permite apagar ou desenhar as tensões principais no canvas. Nesta seção também são desenhadas as tensões principais do círculo de Mohr atuantes em um elemento infinitesimal.

No canto inferior esquerdo, a interface apresenta a caixa de seleção τ *axis* que permite apagar ou desenhar o eixo τ no canvas.

Na Figura 4.16 podem ser observados os pontos de controle do círculo de Mohr. Os pontos (σy, 0) e (σx, 0) (cor azul) podem ser arrastrados com o mouse para a direita ou para a esquerda. Os pontos (σy, τxy) e (σx, $-\tau xy$) (cor azul) podem ser arrastrados com o mouse para cima ou para baixo. E por

fim, o ponto ($\sigma\theta$, $\tau\theta$) (cor verde) pode ser arrastrado com o mouse em torno do círculo de Mohr. Quando um destes pontos é modificado, as mudanças que ocorrem por essa modificação são apresentadas no canvas e nas seções apresentadas acima.

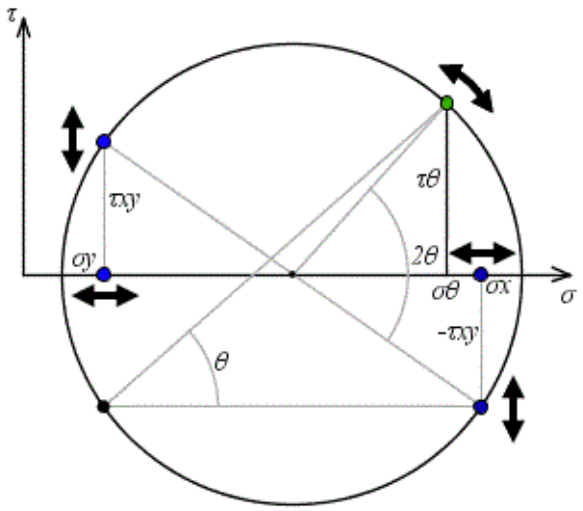

Figura 4.16: Pontos de controle da aplicação e-Mohr2 [18].

4.2.2 Diagrama de Atividade da aplicação e-Mohr2

Na Figura 4.17 apresenta-se o diagrama de atividade da aplicação e-Mohr2. Neste diagrama podem ser observadas as ações e a comunicação desenvolvida por cinco atores: o usuário, a interface gráfica (GUI), um módulo gerenciador de eventos de mouse, um módulo de cálculo e um módulo de desenho. Estes três últimos atores representam as classes descritas anteriormente. Este diagrama permite obter uma visão geral do funcionamento da aplicação e sua interação com o usuário.

4.2.3 Classes da aplicação e-Mohr2

No desenvolvimento desta aplicação foram criadas sete classes e quatro subclasses. A seguir é apresentada uma descrição de cada uma das classes e subclasses implementadas, além de sua funcionalidade na aplicação.

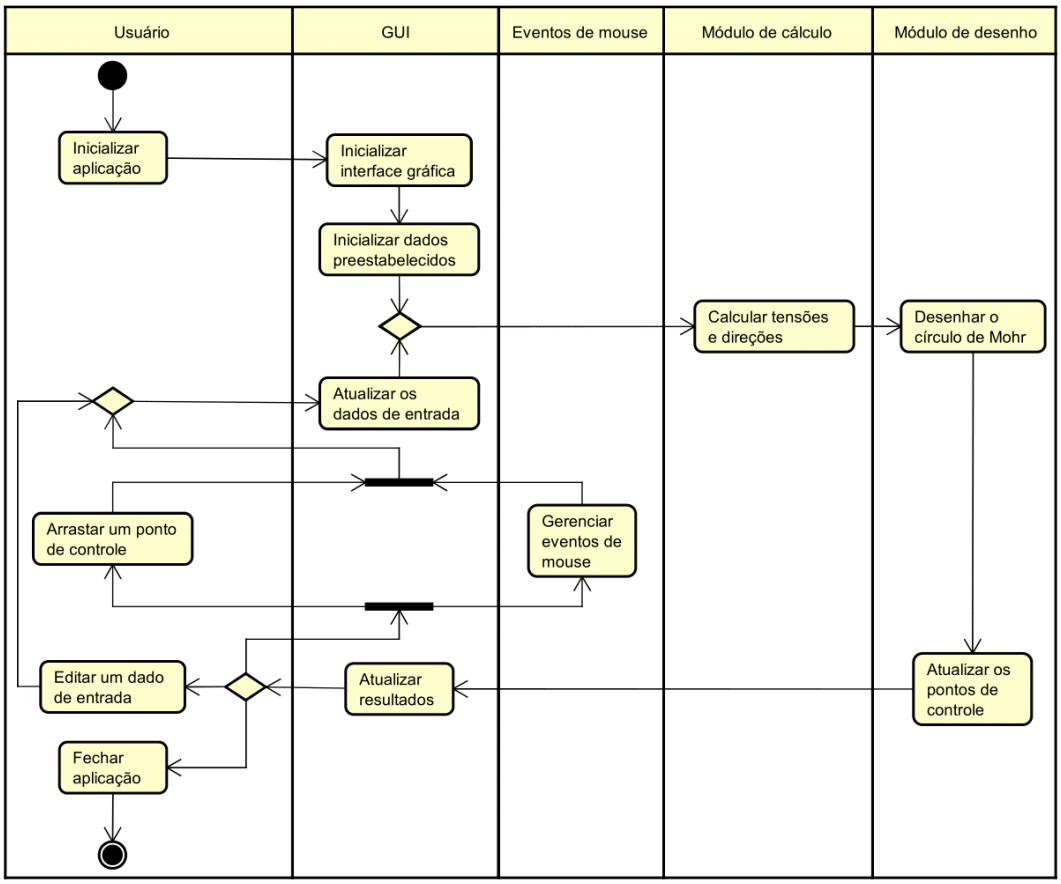

Figura 4.17: Diagrama de atividade e-Mohr2.

4.2.3.1 Subclasse MohrEm

A classe *MohrEm* é uma subclasse da classe *Emouse*. Esta subclasse herda todos os atributos e implementa os métodos abstratos da classe *Emouse*. Herda também os atributos da classe *MohrInput*, que contém os dados de entrada necessários para o desenho do círculo de Mohr. Esta subclasse apresenta o atributo abaixo:

- point_on: se é clicado um dos pontos de controle, o número inteiro que corresponde a esse ponto, é armazenado neste atributo.

Esta subclasse apresenta os métodos a seguir:

- Método construtor (MohrEm), destinado a inicializar um objeto desta subclasse e a definir a herança da classe *Emouse*. Seus argumentos de entrada são um objeto *figure* e um objeto *axes* (canvas).
- moveAction: este método executa as ações que ocorrem quando o usuário move o mouse. Cria um objeto do tipo *MohrControlPoints* e executa seu método *pointActivated*, para determinar se o mouse está próximo de um dos pontos de controle e trocar o ícone do seu ponteiro. Se o botão esquerdo

do mouse é pressionado próximo de um dos pontos de controle e o mouse é arrastrado, este método cria um objeto de tipo *MohrPlot*, segundo qual ponto de controle foi clicado, o ícone do mouse e os atributos desse objeto, que contém os dados necessários para o desenho do círculo de Mohr, são atualizados. Além disso, este método desenha o círculo de Mohr no canvas principal, mediante o método *plotCirMohr* do objeto tipo *MohrPlot* criado. Por fim, atualiza os dados apresentados na interface gráfica, mediante o método *setResults* desse objeto e desenha as tensões obtidas individualmente em três canvas menores, mediante o método *plotEQuad* desse objeto.

• downAction: este método executa as ações que ocorrem quando o usuário pressiona um botão do mouse. Se foi clicado um ponto de controle, mediante um objeto de tipo *MohrControlPoints* e seu método *pointActivated*, este método atualiza o atributo *point_on*.

• upAction: este método não é implementado nesta aplicação. Pode ser implementado em uma possível futura versão desta aplicação, se for necessário desenvolver ações que ocorram quando o usuário libera o botão do mouse que foi pressionado.

Na Figura 4.18 é apresentada a definição da subclasse *MohrEm*.

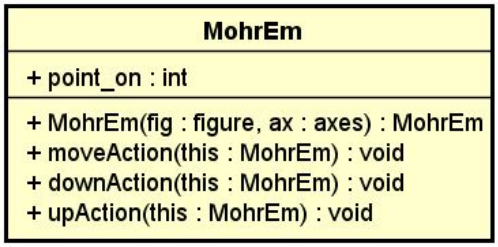

Figura 4.18: Definição da subclasse *MohrEm*.

4.2.3.2 Classe eLine

Esta classe permite desenhar uma linha no canvas, com determinada cor e espessura, definidos dois pontos. Seus atributos são apresentados a seguir:

• x1: este atributo contém a coordenada no eixo X do ponto inicial da linha, definido como uma variável de tipo float.

• y1: este atributo contém a coordenada no eixo Y do ponto inicial da linha, definido como uma variável de tipo float.

• x2: este atributo contém a coordenada no eixo X do ponto final da linha, definido como uma variável de tipo float.

• y2: este atributo contém a coordenada no eixo Y do ponto final da linha, definido como uma variável de tipo float.

• color: este atributo armazena a cor de desenho da linha. Está definido como uma variável de tipo string, que contém o nome de uma cor básica em inglês; ou também pode ser definido como uma matriz de ordem 1x3, que contém intensidades das três cores primárias, vermelho, verde e azul [R G B], capazes de reproduzir a maioria das cores vistas [19]. Está predefinido como 'black', preto em inglês.

• style: este atributo está definido como uma variável de tipo string que contém o estilo da linha; se a linha é continua este atributo é definido como '-', se é tracejada é definido como '- -', se é pontilhada é definido como ':' e se a linha está formada por traços e pontos é definido como '-.'. Este atributo está predefinido como linha contínua.

• thickness: atributo que armazena a espessura de desenho da linha, definido como uma variável de tipo float.

A classe *eLine* apresenta os métodos a seguir:

• Método construtor (eLine), seus argumentos de entrada são os atributos descritos.

• plotLine: este método é encarregado de desenhar a linha de acordo com as coordenadas dos pontos definidos, e os atributos de cor, estilo e espessura.

Na Figura 4.19 é apresentada a definição da classe *eLine*.

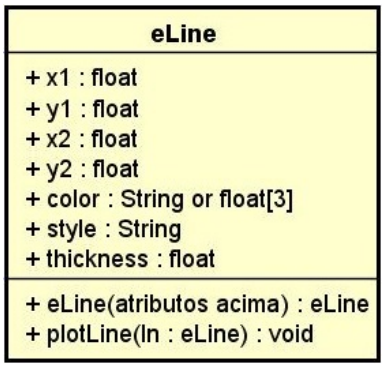

Figura 4.19: Definição da classe *eLine*.

4.2.3.3 Classe eCircle

Esta classe permite desenhar um círculo no canvas, com determinado raio, centro, preenchimento, espessura e cor de linha de desenho. Seus atributos são apresentados a seguir:

• radius: este atributo contém o raio do círculo, definido como uma variável de tipo float.

• centerx: este atributo contém a coordenada no eixo X do centro do círculo, definido como uma variável de tipo float.

• centery: este atributo contém a coordenada no eixo Y do centro do círculo, definido como uma variável de tipo float.

• color: este atributo é igual ao atributo *color* da classe *eLine*. Esta predefinido como 'black'.

• thickness: este atributo armazena a espessura da linha de desenho do círculo, definido como uma variável de tipo float.

• filling: este atributo é definido como uma variável de tipo booleano, que determina se o círculo tem enchimento. Esta predefinido como falso.

• filling_color: este atributo é igual ao atributo *color* da classe *eLine*. Esta predefinido como 'none', nenhum em inglês.

A classe *eCircle* apresenta os métodos a seguir:

• Método construtor (eCirle), seus argumentos de entrada são os atributos descritos.

• plotCircle: este método calcula um número determinado de pontos para desenhar um círculo, utilizando as funções seno e cosseno, para um raio e centro definidos. Com esses pontos, se o atributo *filling* for verdadeiro, o método preenche o círculo da cor definida no atributo *filling_color*, por fim, desenha o contorno do círculo no canvas, de acordo com os atributos *color* e *thickness*.

Na Figura 4.20 é apresentada a definição da classe *eCircle*.

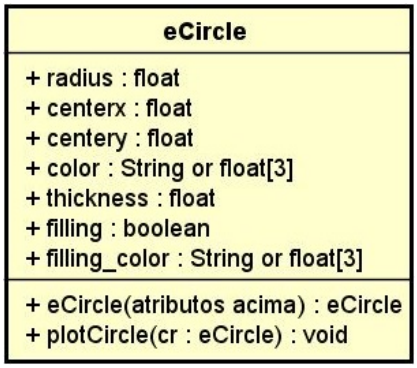

Figura 4.20: Definição da classe *eCircle*.

4.2.3.4 Classe eArc

Esta classe permite desenhar um arco de círculo com determinado raio, centro, ponto inicial, ponto final, espessura e cor. Apresenta os atributos a seguir:

• radius: este atributo contém o raio do arco, definido como uma variável de tipo float.

• centerx: este atributo contém a coordenada no eixo X do centro do arco, definido como uma variável de tipo float.

• centery: este atributo contém a coordenada no eixo Y do centro do arco, definido como uma variável de tipo float.

• sPointx: atributo que contém a coordenada no eixo X do ponto inicial do arco, definido como uma variável de tipo float.

• sPointy: atributo que contém a coordenada no eixo Y do ponto inicial do arco, definido como uma variável de tipo float.

• fPointx: atributo que contém a coordenada no eixo X do ponto final do arco, definido como uma variável de tipo float.

• fPointy: atributo que contém a coordenada no eixo Y do ponto final do arco, definido como uma variável de tipo float.

• color: este atributo é igual ao atributo *color* da classe *eLine*. Esta predefinido como 'black'.

• thickness: este atributo armazena a espessura da linha de desenho do arco, definido como uma variável de tipo float.

A classe *eArc* apresenta os métodos a seguir:

• Método construtor (eArc), seus argumentos de entrada são os atributos descritos.

• getAng: tem como parâmetros de entrada as coordenadas nos eixos X e Y de dois pontos. Este método permite obter o ângulo que existe entre um plano formado por estes pontos e o plano horizontal, em sentido anti-horário.

• plotArc1: este método tem como parâmetros de entrada o ponto inicial e o ponto final de um arco. Um plano formado pelo ponto central e pelo ponto inicial do arco tem um ângulo em relação ao plano horizontal, igualmente, um plano formado pelo ponto central e pelo ponto final do arco tem um ângulo em relação ao plano horizontal, estes ângulos são determinados por este método utilizando seus parâmetros de entrada e o método *getAng*. Com estes ângulos determinados, o método *plotArc1* executa o método *plotArc2*, para desenhar o arco e obter como parâmetro de saída o ponto meio do arco compreendido entre os ângulos determinados.

• plotArc2: um plano formado pelo ponto central e pelo ponto inicial do arco tem um ângulo em relação ao plano horizontal, igualmente, um plano formado pelo ponto central e pelo ponto final do arco tem um ângulo em relação ao plano horizontal, estes ângulos são os parâmetros de entrada da classe *plotArc2*. Este método calcula um número determinado de pontos, compreendidos entres os dois ângulos definidos, para desenhar um arco,

utilizando as funções seno e cosseno, para determinado raio e ponto central. Com esses pontos este método desenha o arco no canvas, de acordo com os atributos *color* e *thickness*. Como parâmetros de saída, o método calcula o ponto meio do arco compreendido entre os ângulos definidos. Este método foi implementado individualmente para ser executado diretamente quando deseja-se desenhar um arco com os ângulos inicial e final, sem a necessidade de definir os pontos inicial e final do arco.

Na Figura 4.21 é apresentada a definição da classe *eArc*.

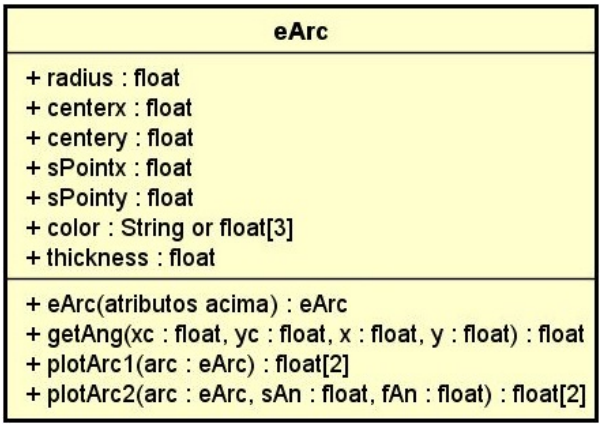

Figura 4.21: Definição da classe *eArc*.

4.2.3.5 Classe exLine

Esta classe permite desenhar uma linha a partir de um ponto inicial na direção de um ângulo definido. Seus atributos são apresentados a seguir:

- scale: este atributo define o comprimento da linha, definido como uma variável de tipo float. A maior valor maior comprimento.
- x: este atributo contém a coordenada no eixo X do ponto inicial da linha, definido como uma variável de tipo float.
- y: este atributo contém a coordenada no eixo Y do ponto inicial da linha, definido como uma variável de tipo float.

angle: atributo que armazena o ângulo de orientação, em graus, da linha, com relação ao plano horizontal.

- color: este atributo é igual ao atributo *color* da classe *eLine*. Esta predefinido como 'black'.
- style: este atributo é igual ao atributo *style* da classe *eLine*. Esta predefinido como linha continua.
- thickness: atributo que armazena a espessura de desenho da linha, definido como uma variável de tipo float.

A classe *exLine* apresenta os métodos a seguir:

• Método construtor(exLine), seus argumentos de entrada são os atributos descritos.

• rotTras: este método permite rotacionar segundo o atributo angle, e trasladar de acordo com os atributos x e y, um ponto. Tem como parâmetros de entrada as coordenadas do ponto que vai ser transformado. Como parâmetro de saída se obtém uma matriz de ordem 1x2, com as coordenadas nos eixos X e Y do ponto rotacionado e trasladado.

• plotEl: este método define os pontos inicial e final de uma linha vertical que passa pelo eixo Y com origem no ponto (0, 0). O método transforma estes pontos com o método *rotTras*, depois, com os pontos transformados cria um objeto de tipo *eLine*, e por fim, executa o método *plotLine* deste objeto.

Na Figura 4.22 é apresentada a definição da classe *exLine*.

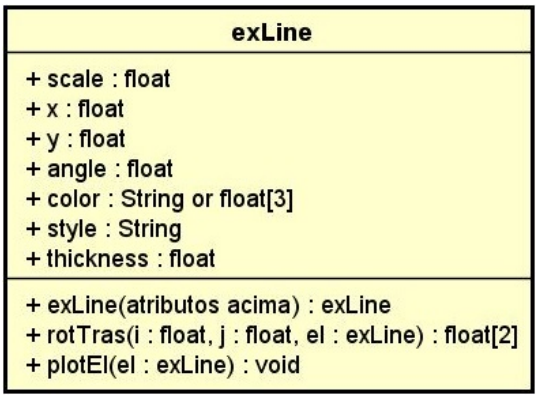

Figura 4.22: Definição da classe *exLine*.

4.2.3.6 Classe MohrInput

Esta classe não contém métodos, além do construtor. Foi implementada para inicializar, armazenar e atualizar os dados de entrada necessários para o círculo de Mohr. Estes dados de entrada são armazenados nos atributos a seguir:

• sigma_x: este atributo contém a tensão normal no sentido do eixo X, definido como uma variável de tipo float, inicializado com o valor de 100,00.

• sigma_y: este atributo contém a tensão normal no sentido do eixo Y, definido como uma variável de tipo float, inicializado com o valor de 50,00.

• tau_xy: este atributo contém a tensão de cisalhamento, definido como uma variável de tipo float, inicializado com o valor de 25,00.

• theta: este atributo contém o ângulo em graus, que define a direção da normal do plano inclinado, definido como uma variável de tipo float, inicializado com o valor de 52 graus.

Na Figura 4.23 é apresentada a definição da classe *MohrInput*.

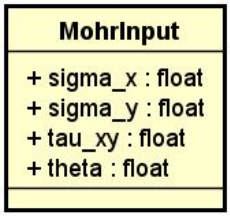

Figura 4.23: Definição da classe *MohrInput*.

4.2.3.7 Subclasse MohrStress

Esta subclasse herda os atributos da classe *MohrInput*. Está encarregada de calcular as tensões principais e as tensões atuantes em um plano inclinado do círculo de Mohr. A subclasse apresenta os métodos a seguir:

• Método construtor (MohrStress), destinado a inicializar um objeto desta subclasse e a definir a herança da classe *MohrInput*. Seus argumentos de entrada são os atributos da classe *MohrInput*.

• p_stress: este método calcula as tensões principais e o ângulo que define o plano da tensão principal máxima do círculo de Mohr.

• i_stress: este método calcula as tensões atuantes em um plano inclinado do círculo de Mohr.

Na Figura 4.24 é apresentada a definição da subclasse *MohrStress*.

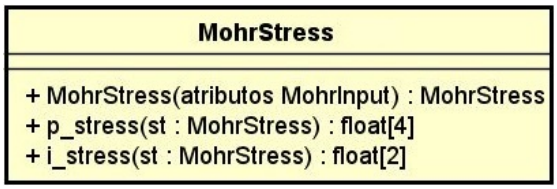

Figura 4.24: Definição da subclasse *MohrStress*.

4.2.3.8 Subclasse MohrControlPoints

Esta subclasse da classe *MohrInput*, permite determinar se o mouse está próximo de um dos pontos de controle do círculo de Mohr, que permitem ajustar o estado plano de tensões e o ângulo de inclinação do plano. Além dos

atributos herdados da classe *MohrInput*, esta subclasse apresenta um atributo descrito abaixo:

• position: armazena as coordenadas nos eixos X e Y da posição atual do mouse no canvas. Está definido como uma matriz de ordem 1x2, onde a variável 1x1 apresenta a coordenada X e a variável 1x2 a coordenada Y.

A subclasse *MohrControlPoints* apresenta os métodos a seguir:

• Método construtor (MohrControlPoints), seus argumentos de entrada são o atributo *position* descrito e os atributos herdados da classe *MohrInput*.

• pointsAtivation: este método obtém os pontos de controle atuais criando um objeto de tipo *MohrStress* e calculando os parâmetros necessários mediante os métodos *p_stress* e *i_stress* deste objeto. Se a opção *stresses on an inclined plane* na interface gráfica está selecionada, este método calcula um número determinado de pontos para criar um pequeno círculo em torno do ponto $(\sigma\theta, \tau\theta)$ e com a função *inpolygon* de MATLAB, determina se o mouse está próximo deste ponto. Se a opção *Plane stress state* na interface gráfica está selecionada, este método calcula um número determinado de pontos para criar pequenos círculos em torno dos pontos $(\sigma y, \tau xy)$, $(\sigma x, -\tau xy)$, $(\sigma x, 0)$ e $(\sigma y, 0)$, e com a função *inpolygon* de MATLAB, determina se o mouse está próximo de um destes pontos. Como parâmetro de saída este método determina próximo de qual ponto de controle está o mouse com uma variável de tipo inteiro, de acordo com o arquivo *include_constants*, um (1) para o ponto $(\sigma\theta, \tau\theta)$, 2 para o ponto $(\sigma y, \tau xy)$, 3 para o ponto $(\sigma x, -\tau xy)$, 4 para o ponto $(\sigma x, 0)$, 5 para o ponto $(\sigma y, 0)$, e zero (0) se o mouse não está próximo de nenhum ponto de controle.

Na Figura 4.25 é apresentada a definição da subclasse *MohrControlPoints*.

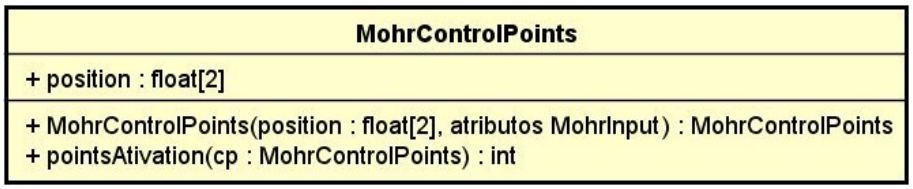

Figura 4.25: Definição da subclasse *MohrControlPoints*.

4.2.3.9 Subclasse MohrPlot

Esta subclasse permite desenhar o círculo de Mohr no canvas principal, atualizar os dados de entrada e saída na interface gráfica e desenhar as tensões de forma individual na representação de um ponto infinitesimal. A subclasse

MohrPlot herda os atributos da classe *MohrInput* e apresenta os métodos a seguir:

• Método construtor (MohrPlot), seus argumentos de entrada são os atributos da classe *MohrInput*.

• plotCirMohr: com os parâmetros herdados, este método calcula as tensões principais e as tensões atuantes em um plano inclinado do círculo de Mohr utilizando um objeto da classe *MohrStress*, com os dados obtidos cria todos os objetos necessários para o desenho do círculo de Mohr e seus eixos, mediante as classes *eLine*, *eCircle*, *eArc* e *exLine*. Depois, apaga o desenho atual no canvas principal. Segundo a orientação do plano inclinado, o tipo de tensão obtida e se é positiva ou negativa, o método *plotCirMohr* determina o sentido das tensões calculadas para criar um objeto da classe *dirPlot* por cada uma destas tensões, que serão utilizados para o desenho das setas que determinam seus sentidos. De acordo com as opções de desenho selecionadas na interface gráfica (*plane stress state*, *stresses on an inclined plane* e *principals stress*), o método *plotCirMohr* desenha os objetos correspondentes no canvas principal. Além disto, este método desenha o eixo τ das tensões cisalhantes no círculo de Mohr, se a opção τ *axis* é selecionada na interface gráfica.

• setResults: com os parâmetros herdados, este método calcula as tensões principais e as tensões atuantes em um plano inclinado do círculo de Mohr utilizando um objeto da classe *MohrStress*, para atualizar os dados apresentados na interface gráfica com as mudanças nos dados de entrada.

• plotEQuad: com os parâmetros herdados, este método calcula as tensões principais e as tensões atuantes em um plano inclinado do círculo de Mohr utilizando um objeto da classe *MohrStress*. Segundo a orientação do plano inclinado, o tipo de tensão obtida e se é positiva ou negativa, o método *plotEQuad* determina o sentido das tensões calculadas para criar três objetos da classe *eQuad*; um para o estado plano de tensões (cor azul), outro para as tensões atuantes em um plano inclinado (cor verde) e o último para as tensões principais (cor vermelha). Estes objetos são desenhados na interface gráfica em três canvas individuais e são atualizados com as mudanças nos dados de entrada.

Na Figura 4.26 é apresentada a definição da subclasse *MohrPlot*.

4.2.3.10 Classe dirPlot

Esta classe está encarregada de desenhar as setas que definem o sentido de determinada tensão atuante no círculo de Mohr, apresenta os atributos a seguir:

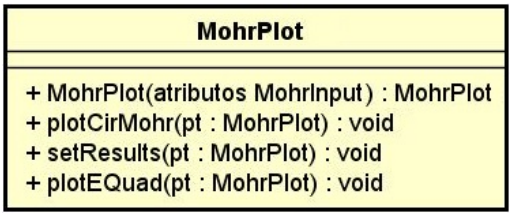

Figura 4.26: Definição da subclasse *MohrPlot*.

• scale: atributo que contém uma variável, de tipo float, que determina o tamanho do desenho, esta predefinido com o valor de um (1,00).

• x: este atributo é definido como uma variável de tipo float, que contém a coordenada no eixo X do ponto onde será desenhada a seta para definir o sentido de determinada tensão atuante no círculo de Mohr.

• y: este atributo é definido como uma variável de tipo float, que contém a coordenada no eixo Y do ponto onde será desenhada a seta para definir o sentido de determinada tensão atuante no círculo de Mohr.

• angle: este atributo é definido como uma variável de tipo float, que contém o angulo, em graus, da direção do plano em que atua determinada tensão do círculo de Mohr.

• addL: este atributo é definido como uma variável de tipo float, que contém o comprimento de um segmento de reta, que é adicionado à seta que define a direção do plano em que atua determinada tensão do círculo de Mohr. Este atributo é predefinido com o valor de zero (0,00), e é utilizado apenas se for necessário para melhorar a visibilidade do desenho.

• tranv_arrow: este atributo é definido como uma variável de tipo string, que é utilizada como uma variável de tipo booleano, quando deseja-se desenhar uma seta transversal à direção do plano em que atua determinada tensão do círculo de Mohr, é definida como 'on', caso contrario recebe o valor 'off'.

• sense1: este atributo é definido como uma variável de tipo string. Se o atributo *tranv_arrow* está definido como 'on', o atributo *sense1* determina o sentido da seta transversal, se é definido como '->' o sentido da seta é desenhado a -90 graus com relação ao sentido de determinada tensão do círculo de Mohr, e se é definido como '<-' o sentido da seta é desenhado a 90 graus com ralação ao sentido desta tensão.

• sense2: este atributo é definido como uma variável de tipo string. O atributo *sense2* define o sentido de determinada tensão atuante no círculo de Mohr, se é definido como '->' o sentido é de tensão, e se é definido como '<-' o sentido é de compressão.

• color: este atributo é igual ao atributo *color* da classe *eLine*. Esta

predefinido como 'black'.

• tau_: este atributo é definido como uma variável de tipo string. Se o atributo *tranv_arrow* está definido como 'on', o atributo *tau_* armazena o texto que vai ser escrito no canvas referente à seta transversal, nesta aplicação corresponde à tensão de cisalhamento τ.

• sigma_: este atributo é definido como uma variável de tipo string, armazena o texto que vai ser escrito no canvas referente a determinada tensão atuante no círculo de Mohr, nesta aplicação corresponde à tensão normal σ.

A classe *dirPlot* possui os métodos a seguir:

• Método construtor (dirPlot), seus argumentos de entrada são os atributos descritos.

• rotTras: este método é igual ao método *rotTras* da classe *exLine*.

• plotC: a partir do ponto (0, 0), este método define os pontos para o desenho de um retângulo e duas setas orientadas de acordo com os atributos *sense1* e *sense2*. Estes pontos são rotacionados e trasladados utilizando o método *rotTras*. Com estes pontos transformados, o método *plotC* desenha o retângulo com seu lado maior perpendicular à direção de atuação de determinada tensão do círculo de Mohr, e por fim, desenha as setas de acordo com o sentido desta tensão. As linhas necessárias para o desenho são criadas como objetos da classe *eLine* e desenhadas com o método *plotLine* dessa classe. O método *plotC* também é encarregado de escrever o texto com o nome das tensões de acordo com os atributos *tau_* e *sigma_*.

Na Figura 4.27 é apresentada a definição da classe *dirPlot*.

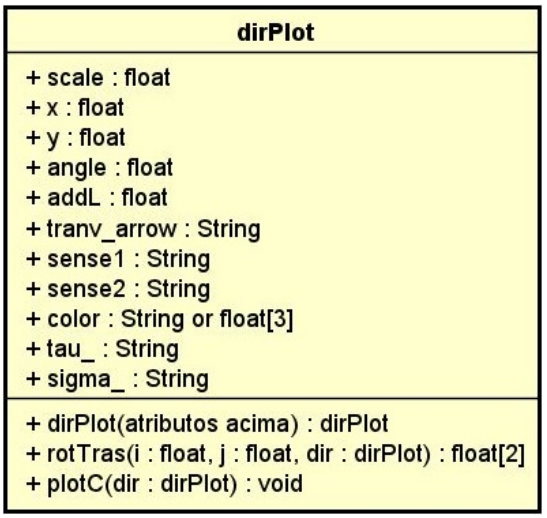

Figura 4.27: Definição da classe *dirPlot*.

4.2.3.11 Classe eQuad

Esta classe permite desenhar as tensões do círculo de Mohr atuantes em um elemento infinitesimal segundo um plano de inclinação (Figura 4.15 seções 1, 2 e 3). Possui os atributos a seguir:

• x: este atributo é definido como uma variável de tipo float, contém a coordenada no eixo X do ponto de origem do desenho.

• y: este atributo é definido como uma variável de tipo float, contém a coordenada no eixo Y do ponto de origem do desenho.

• dx: este atributo é definido como uma variável de tipo float, contém o comprimento do lado do quadrado que representa o elemento infinitesimal, em sentido do eixo X.

• dy: este atributo é definido como uma variável de tipo float, contém o comprimento do lado do quadrado que representa o elemento infinitesimal, em sentido do eixo Y.

• angle: este atributo é definido como uma variável de tipo float, contém o ângulo, em graus, da direção do plano em que atua a tensão principal normal ao elemento infinitesimal.

• sensePaArrows: este atributo é definido como uma variável de tipo string. O atributo *sensePaArrows* define o sentido das tensões cisalhantes atuantes no elemento infinitesimal, pode ser definido como '->' ou '<-' segundo o tipo de tensão e sua orientação no círculo de Mohr.

• sensePeArrows1: este atributo é definido como uma variável de tipo string. O atributo *sensePeArrows1* define o sentido das tensões normais atuantes na orientação normal do elemento infinitesimal, pode ser definido como '->' ou '<-' segundo o tipo de tensão e sua orientação no círculo de Mohr.

• sensePeArrows2: este atributo é definido como uma variável de tipo string. O atributo *sensePeArrows2* define o sentido das tensões normais atuantes na orientação transversal do elemento infinitesimal, pode ser definido como '->' ou '<-' segundo o tipo de tensão e sua orientação no círculo de Mohr.

• areaColor: este atributo é igual ao atributo *color* da classe *eLine*. Esta predefinido como 'white'.

• edgeColor: este atributo é igual ao atributo *color* da classe *eLine*. Esta predefinido como 'black'.

• lineWidth: atributo que armazena a espessura das linhas de desenho, definido como uma variável de tipo float.

• aColor1: este atributo é igual ao atributo *color* da classe *eLine*. Esta predefinido como 'black'.

• aColor2: este atributo é igual ao atributo *color* da classe *eLine*. Esta predefinido como 'black'.

• textPaA1: este atributo é definido como uma variável de tipo string, que armazena o texto que vai ser escrito no canvas referente ás tensões cisalhantes atuantes na direção normal do elemento infinitesimal.

• textPaA2: este atributo é definido como uma variável de tipo string, que armazena o texto que vai ser escrito no canvas referente ás tensões cisalhantes atuantes na direção transversal do elemento infinitesimal.

• textPeA1: este atributo é definido como uma variável de tipo string, que armazena o texto que vai ser escrito no canvas referente ás tensões normais atuantes na direção normal do elemento infinitesimal.

• textPeA2: este atributo é definido como uma variável de tipo string, que armazena o texto que vai ser escrito no canvas referente ás tensões normais atuantes na direção transversal do elemento infinitesimal.

• textAngle: este atributo é definido como uma variável de tipo string, que armazena o texto que vai ser escrito no canvas referente ao ângulo da direção do plano em que atua a tensão principal normal ao elemento infinitesimal.

A classe *eQuad* possui os métodos a seguir:

• Método construtor (eQuad), seus argumentos de entrada são os atributos descritos.

• rotTras: este método é igual ao método *rotTras* da classe *exLine*.

• plotRectangle: a partir do ponto (x, y), este método define os pontos para o desenho de um retângulo de acordo com os atributos *dx* e *dy*. Estes pontos são rotacionados e trasladados utilizando o método *rotTras*; com estes pontos transformados, o método *plotRectangle* desenha o retângulo que representa o elemento infinitesimal, de acordo com os atributos *areaColor* e *edgeColor*.

• plotParallelArrows: a partir do ponto (x, y), e de acordo com o atributo *sensePaArrows*, este método define os pontos para o desenho de quatro setas que representam as tensões cisalhantes atuantes em torno do elemento infinitesimal. Estes pontos são rotacionados e trasladados utilizando o método *rotTras*; com estes pontos transformados, o método *plotParallelArrows* desenha as setas das cores definidas nos atributos *aColor1* e *aColor2*. As linhas necessárias para o desenho são criadas como objetos da classe *eLine* e desenhadas com o método *plotLine* dessa classe. O método *plotParallelArrows*

também é encarregado de escrever o texto com o nome das tensões segundo os atributos *textPaA1* e *textPaA2*.

• plotPerpendicularArrows: a partir do ponto (x, y), e de acordo com os atributos *sensePeArrows1* e *sensePeArrows2*, este método define os pontos para o desenho de quatro setas que representam as tensões normais atuantes em torno do elemento infinitesimal. Estes pontos são rotacionados e trasladados utilizando o método *rotTras*; com estes pontos transformados, o método *plotPerpendicularArrows* desenha as setas das cores definidas nos atributos *aColor1* e *aColor2*. As linhas necessárias para o desenho são criadas como objetos da classe *eLine* e desenhadas com o método *plotLine* dessa classe. O método *plotPerpendicularArrows* também é encarregado de escrever o texto com o nome das tensões segundo os atributos *textPeA1* e *textPeA2*.

• plotAngle: se o atributo angle é maior que 0 graus e menor que 180 graus, este método desenha o ângulo da direção do plano em que atua a principal tensão normal ao elemento infinitesimal, para isso, este método utiliza objetos das classes *eLine* e *eArc*, com seus correspondentes métodos de desenho. O método *plotAngle* também é encarregado de escrever o texto com o nome do ângulo de acordo com o atributo *textAngle*.

Na Figura 4.28 é apresentada a definição da classe *eQuad*.

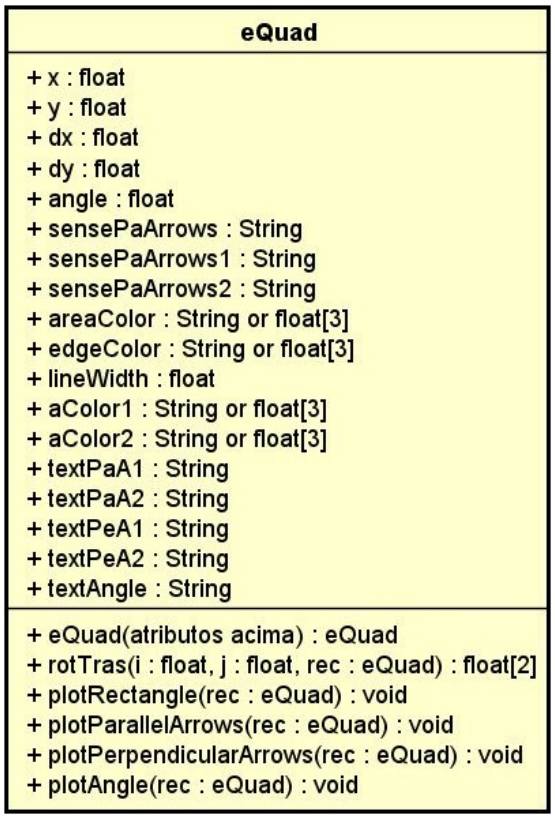

Figura 4.28: Definição da classe *eQuad*.

4.2.4
Diagrama de Classes da aplicação e-Mohr2

Na Figura 4.29 apresenta-se o diagrama de classes da aplicação e-Mohr2. Neste diagrama pode-se observar as heranças que se obtêm das classes *Emouse* e *MohrInput*, e os relacionamentos de dependência (uso), agregação e composição das classes envolvidas.

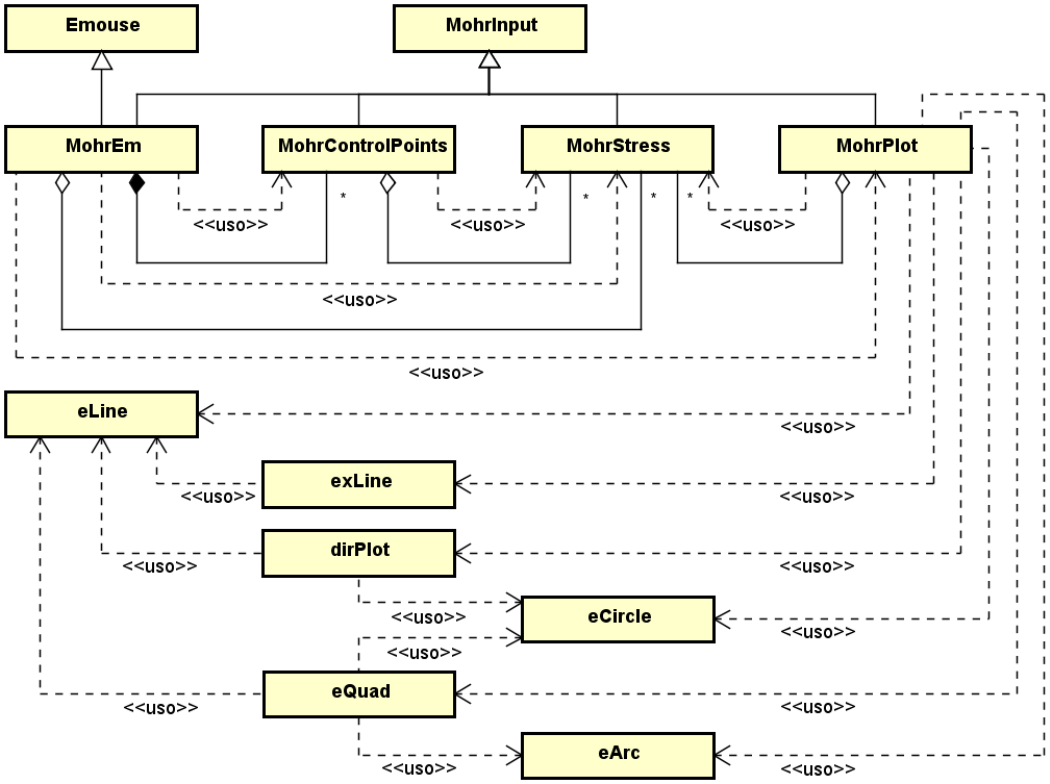

Figura 4.29: Diagrama de classes e-Mohr2.

4.2.5
Diagrama de Sequência da aplicação e-Mohr2

No diagrama de sequência da aplicação e-Mohr2 pode-se observar a representação da sequência de mensagens que são executadas durante o funcionamento da aplicação. Devido a seu tamanho e para melhorar sua visualização, este diagrama é apresentado na seção de anexos deste trabalho (Figura 6.10). Algumas partes deste diagrama são apresentadas em diagramas de fragmento de iteração de operador *ref*, com o objetivo de melhorar sua organização e visualização.

5
Conclusões e Trabalho Futuro

Uma classe no contexto da POO para gerenciamento genérico de eventos de mouse em um canvas no ambiente MATLAB foi desenvolvida.

Duas aplicações de engenharia que precisam do gerenciamento de eventos de mouse em um canvas no ambiente MATLAB, foram desenvolvidas no contexto da POO mediante a utilização da classe obtida. O desenvolvimento destas aplicações permite destacar a propriedade genérica da classe.

O contexto da POO foi empregado porque fornece um código de simples manutenção, reutilizável, que favorece a criação de novas funcionalidades.

Quando um sistema orientado a objetos é desenvolvido, a UML permite melhorar seu entendimento fornecendo uma documentação gráfica sobre o comportamento do sistema.

A classe desenvolvida pode ser usada na implementação de aplicativos de MATLAB, no contexto da POO, que, além de visualização, criação e manipulação de modelos, tenham o objetivo de ser ferramentas educacionais gráfico-interativas. Como resultado da interação com o mouse, esses processos se tornam de melhor entendimento.

Assim, como proposta para trabalho futuro se inclui: implementar os eventos que ocorrem com o *scroll* do mouse; implementar eventos de mouse que considerem três dimensões; criar uma versão mais completa da aplicação e-dles2D, onde seja possível botar condições de contorno e propriedades geométricas e constitutivas dos elementos; implementar uma versão da aplicação e-Mohr2 que permita a interação com o mouse em três dimensões.

Além disso, o trabalho futuro tem uma visão mais ampla, pois a classe obtida é uma potencial ferramenta para o desenvolvimento, no ambiente MATLAB, de inúmeros softwares educacionais gráfico-interativos.

Referências bibliográficas

[1] PENG, J. **An Internet-enabled software framework for the collaborative development of a structural analysis program**. Ph.D. Thesis, Stanford University, California (USA), 2002.

[2] LIU, W.; TONG, M.; WU, X.; LEE, G. C. **Object-oriented modeling of structural analysis and design with application to damping device configuration**. Journal of computing in civil engineering, 17 (2): 113-122, 2003.

[3] HARAHAP, I. S.; JR, V. R. M.; MUSTAPHA, M. **A new approach of structural analysis in conjunction with structural design and optimization**. Mission-Oriented Research: PETROCHEMICAL CATALYSIS TECHNOLOGY, 5 (2): 32-42, 2007.

[4] JANKOVSKI, V.; ATKOČIŪNAS, J. **Saosys toolbox as Matlab implementation in the elastic-plastic analysis and optimal design of steel frame structures**. Journal of Civil Engineering and Management, 16 (1): 103-121, 2010.

[5] JANKOVSKI, V.; ATKOČIŪNAS, J. **Biparametric shakedown design of steel frame structures**. Mechanics, 17 (1): 5-12, 2011.

[6] ZANDER, N.; BOG, T.; ELHADDAD, M.; ESPINOZA, R.; HU, H.; JOLY, A.; ...; PARVIZIAN, J. **FCMLab: A finite cell research toolbox for MATLAB**. Advances in engineering software, 74: 49-63, 2014.

[7] KACPRZYK, Z.; OSTAPSKA-ŁUCZKOWSKA, K. **Isogeometric Analysis as a New FEM Formulation-Simple Problems of Steady State Thermal Analysis**. Procedia Engineering, 91: 87-92, 2014.

[8] GUARDIA, J. M. **Desarrollo en Matlab de software de cálculo de placas por MEF**. Universitat Politècnica de Catalunya, Barcelona (Espanha), 2015.

[9] GUERRERO, L. F.; PIZANO, D. G.; THOMSON, P. **Desarrollo e implementación de una interfaz gráfica para el programa de elementos finitos FEM**. In: 22th Jornadasaie, 2012.

[10] ELMENDORP, R. J. M.; VOS, R.; LA ROCCA, G. **A conceptual design and analysis method for conventional and unconventional airplanes**. In: ICAS 2014, Proceedings of the 29th Congress of the International Council of the Aeronautical Sciences, St. Petersburg (Russia), 2014.

[11] NATH, R. KUMAR, D. **Lecture Notes on Object-Oriented Methodology**. 2014. Disponível em <http://www.ddegjust.ac.in/studymaterial/mca-5/mca-503.pdf>

[12] SEABRA, J. **UML - Unified Modeling Language: Uma ferramenta para o Desing de Software**. 2da Ed. Rio de Janeiro: Editora Ciência Moderna Ltda., 2013.

[13] BOOCH, G. **Object-oriented Analysis and Design with Applications**. 2da Ed. Massachusetts, USA: Addison Wesley, 1994.

[14] RUMBAUGH, J.; BOOCH, G.; JACOBSON, I. **The Unified Modeling Language Reference Manual**. 2nd Ed. Boston, USA: Pearson Education, 2005.

[15] O'DOCHERTY, M. **Object-oriented analysis and design: Understanding system development with UML 2.0.**. West Sussex, UK: John Wiley Sons Ltda, 2005.

[16] WEILKIENS, T.; OESTEREICH, B. **UML 2 Certification Guide - Fundamental Intermediate Exams**. San Francisco, USA: Morgan Kaufmann Publishers, 2006.

[17] MATHWORKS. **MATLAB documentation**. 2017. Disponível em <https://www.mathworks.com/help/>

[18] MARTHA, L. F.; CARBONO, A. J.; PEREIRA, A. R.; RAMIRES, F. B.; DALCANAL, P. R.; RODRIGUES, R. **e-Mohr: Ferramenta educacional para círculo de Mohr**, 2004. [software, v. 1.0]. Projeto Final CIV2802, Departamento de Engenharia Civil, Pontifícia Universidade Católica do Rio de Janeiro. Acesso em: Julho de 2016.

[19] JOBLOVE, G. H.; GREENBERG, D. **Color spaces for computer graphics**. In: ACM siggraph computer graphics, 12 (3): 20-25, 1978.

[20] ATTAWAY, S. **MATLAB: a practical introduction to programming and problem solving**. 4th Ed. Butterworth-Heinemann, 2017.

[21] MATHWORKS. **MATLAB**, 2016. [software, V. 9.1 (R2016b)]. Acesso em: Setembro de 2016.

6
Anexos

6.1
Diagrama de sequência e-dles2D

Este diagrama é apresentado nas Figuras 6.1, 6.2 e 6.3.

6.1.1
Fragmento de iteração (ref) SD Delete Object

Este fragmento é apresentado na Figura 6.4.

6.1.2
Fragmento de iteração (ref) SD Node Mode (down)

Este fragmento é apresentado na Figura 6.5.

6.1.3
Fragmento de iteração (ref) SD Element Mode (down)

Este fragmento é apresentado na Figura 6.6.

6.1.3.1
Fragmento de iteração (ref) SD Compute Bar Element

Este fragmento é apresentado na Figura 6.7.

6.1.4
Fragmento de iteração (ref) SD Two Bars Intersected

Este fragmento é apresentado na Figura 6.8.

6.1.5
Fragmento de iteração (ref) SD View Canvas

Este fragmento é apresentado na Figura 6.9.

6.2
Diagrama de sequência e-Mohr2

Este diagrama é apresentado na Figura 6.10.

6.2.1
Fragmento de iteração (ref) SD Draw Morh's Circle

Este fragmento é apresentado nas Figuras 6.11 e 6.12.

6.2.2
Fragmento de iteração (ref) SD Mouse Events MC

Este fragmento é apresentado na Figura 6.13.

6.2.3
Fragmento de iteração (ref) SD Draw Infinitesimal B

Este fragmento é apresentado na Figura 6.14.

6.2.4
Fragmento de iteração (ref) SD Set Results GUI

Este fragmento é apresentado na Figura 6.15.

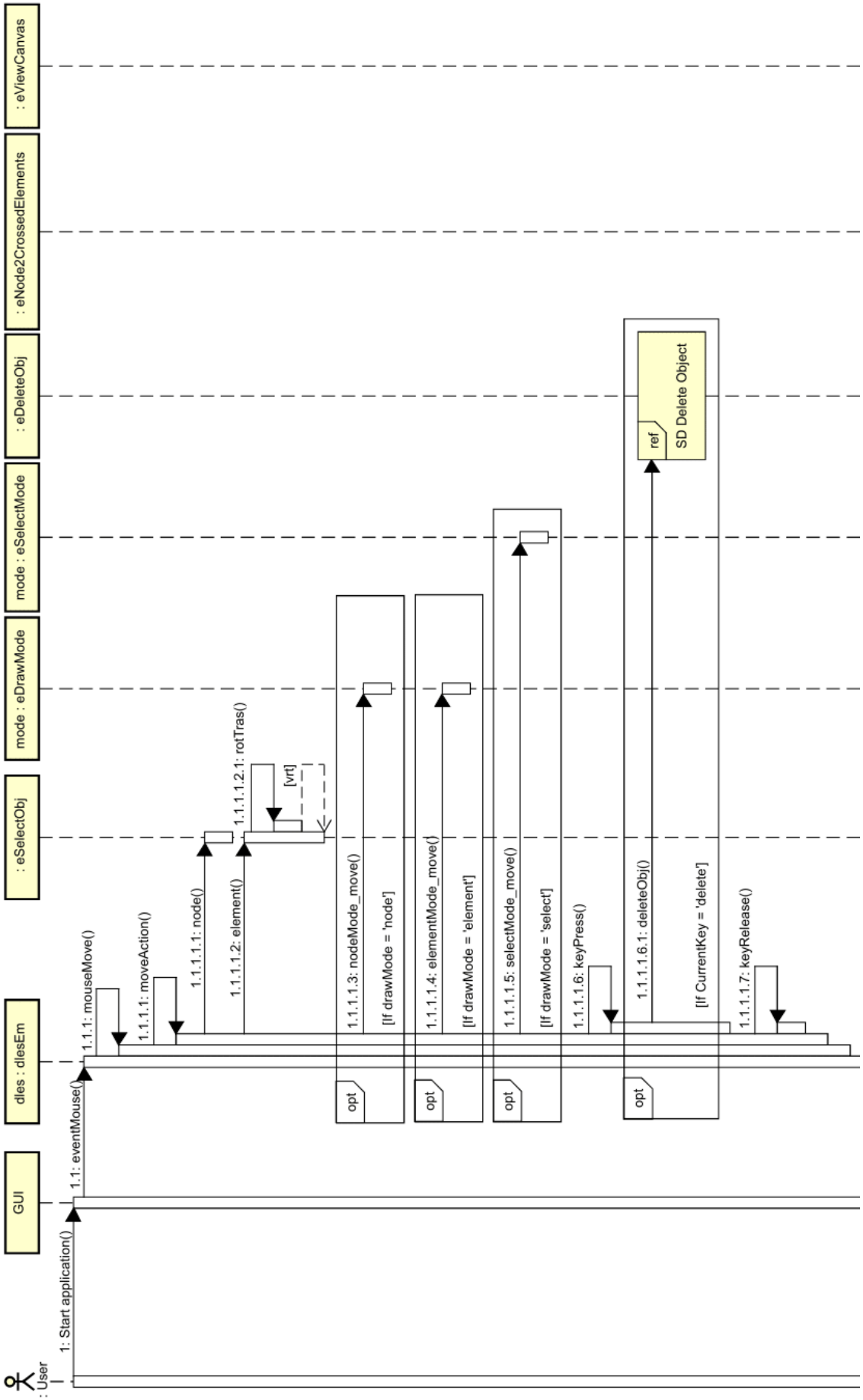

Figura 6.1: Diagrama de sequência e-dles2D (Parte 1).

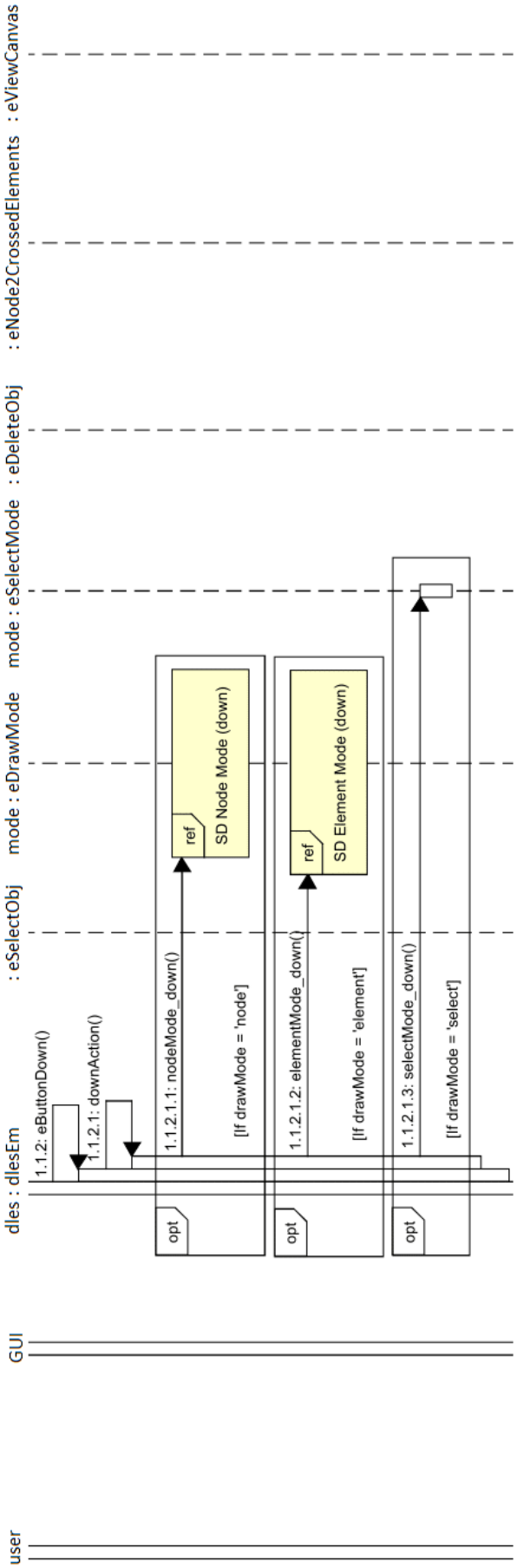

Figura 6.2: Diagrama de sequência e-dles2D (Parte 2).

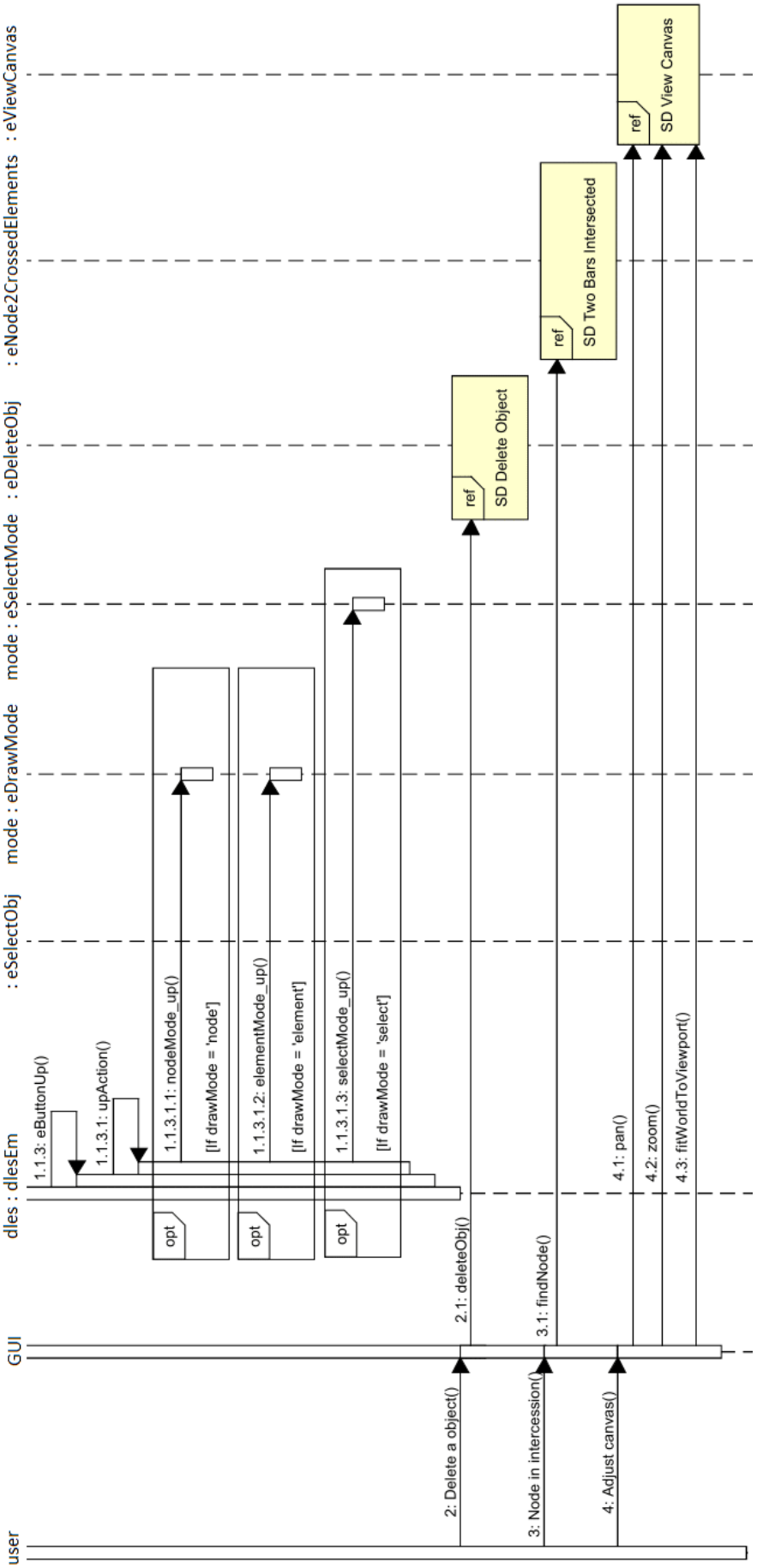

Figura 6.3: Diagrama de sequência e-dles2D (Parte 3).

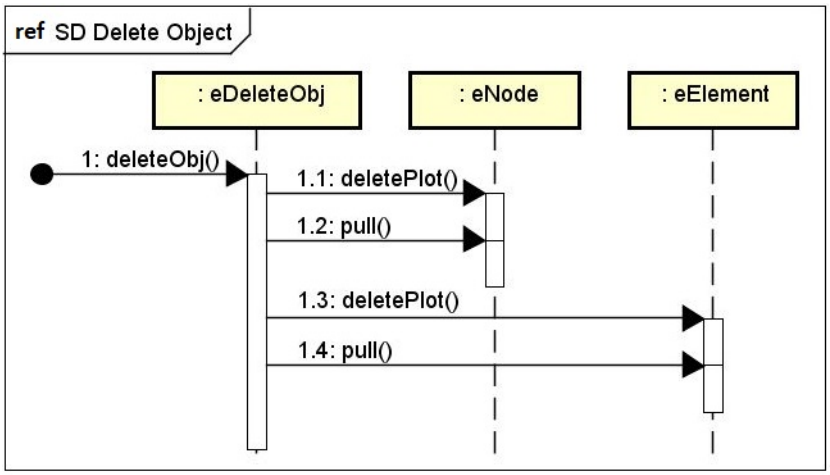

Figura 6.4: Fragmento de iteração: *SD Delete Object*.

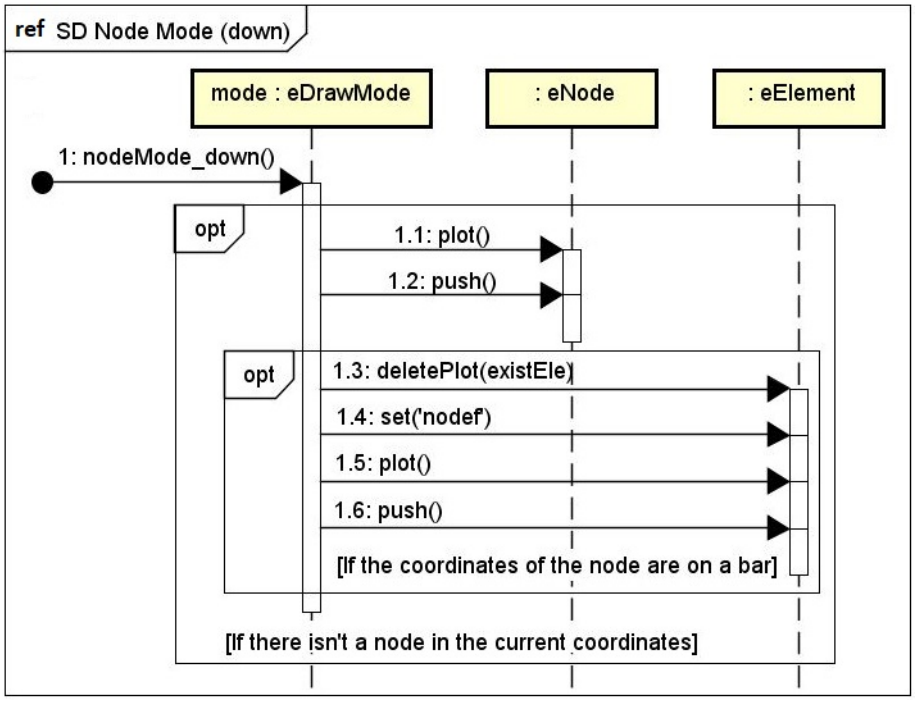

Figura 6.5: Fragmento de iteração: *SD Node Mode (down)*.

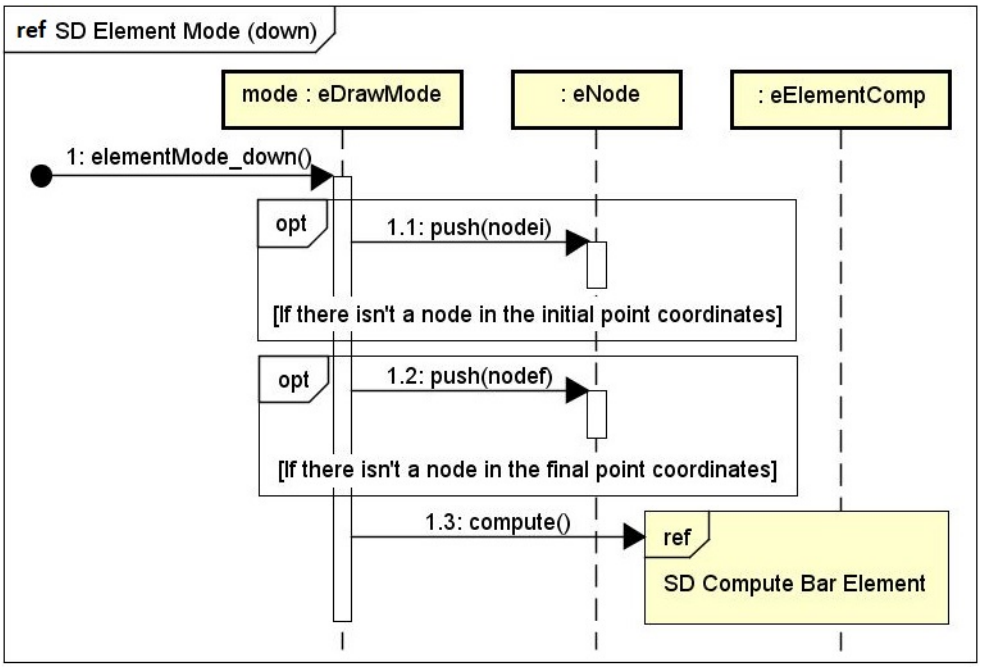

Figura 6.6: Fragmento de iteração: *SD Element Mode (down).*

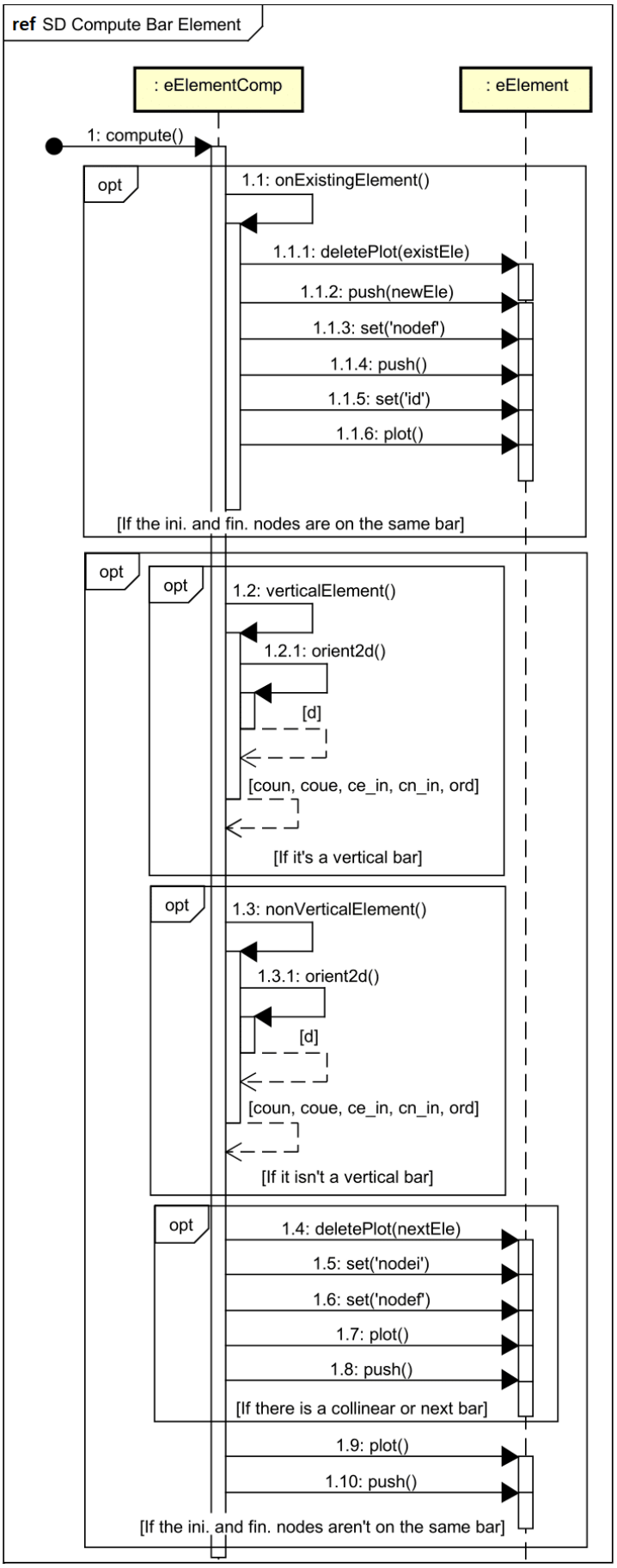

Figura 6.7: Fragmento de iteração: *SD Compute Bar Element*.

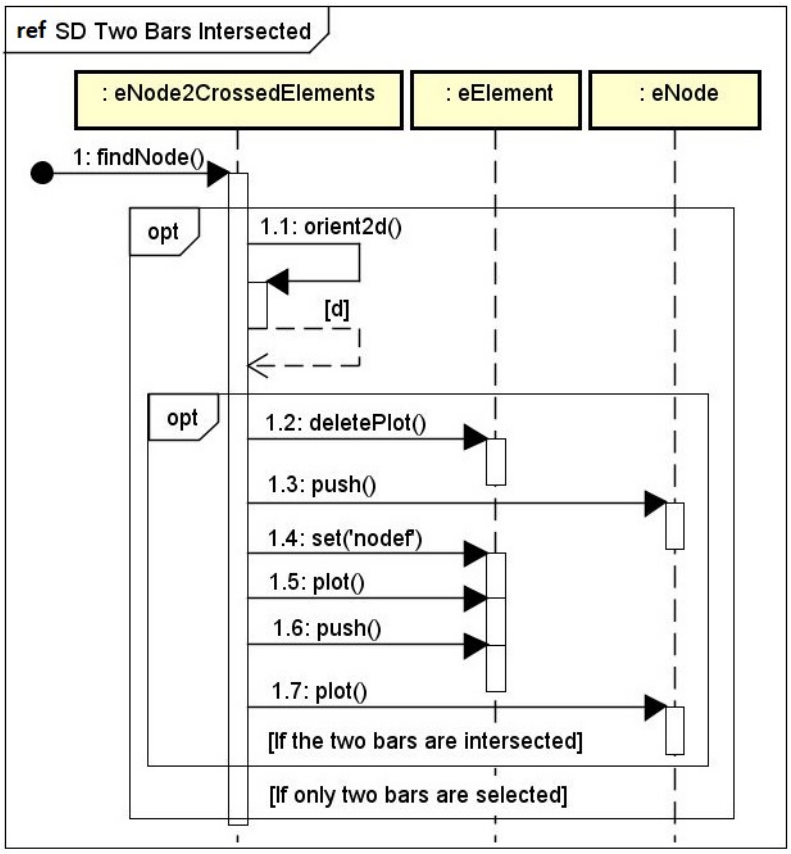

Figura 6.8: Fragmento de iteração: *SD Two Bars Intersected*.

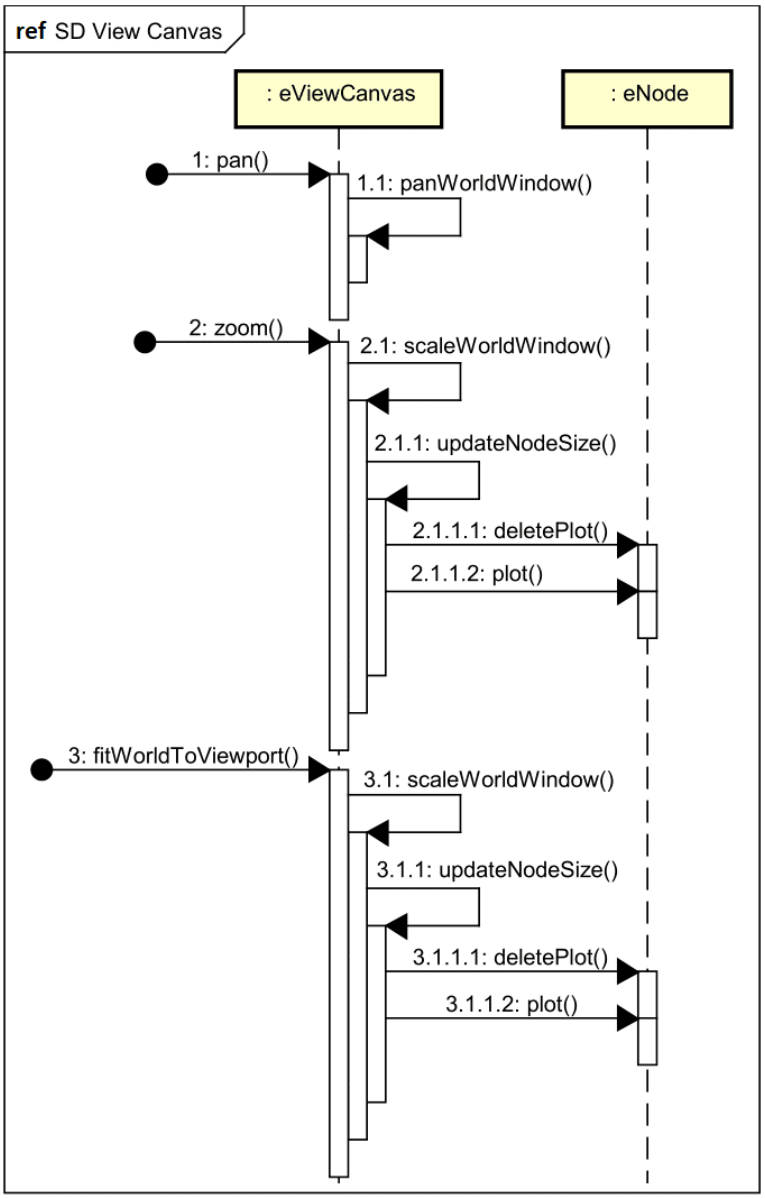

Figura 6.9: Fragmento de iteração: *SD View Canvas*.

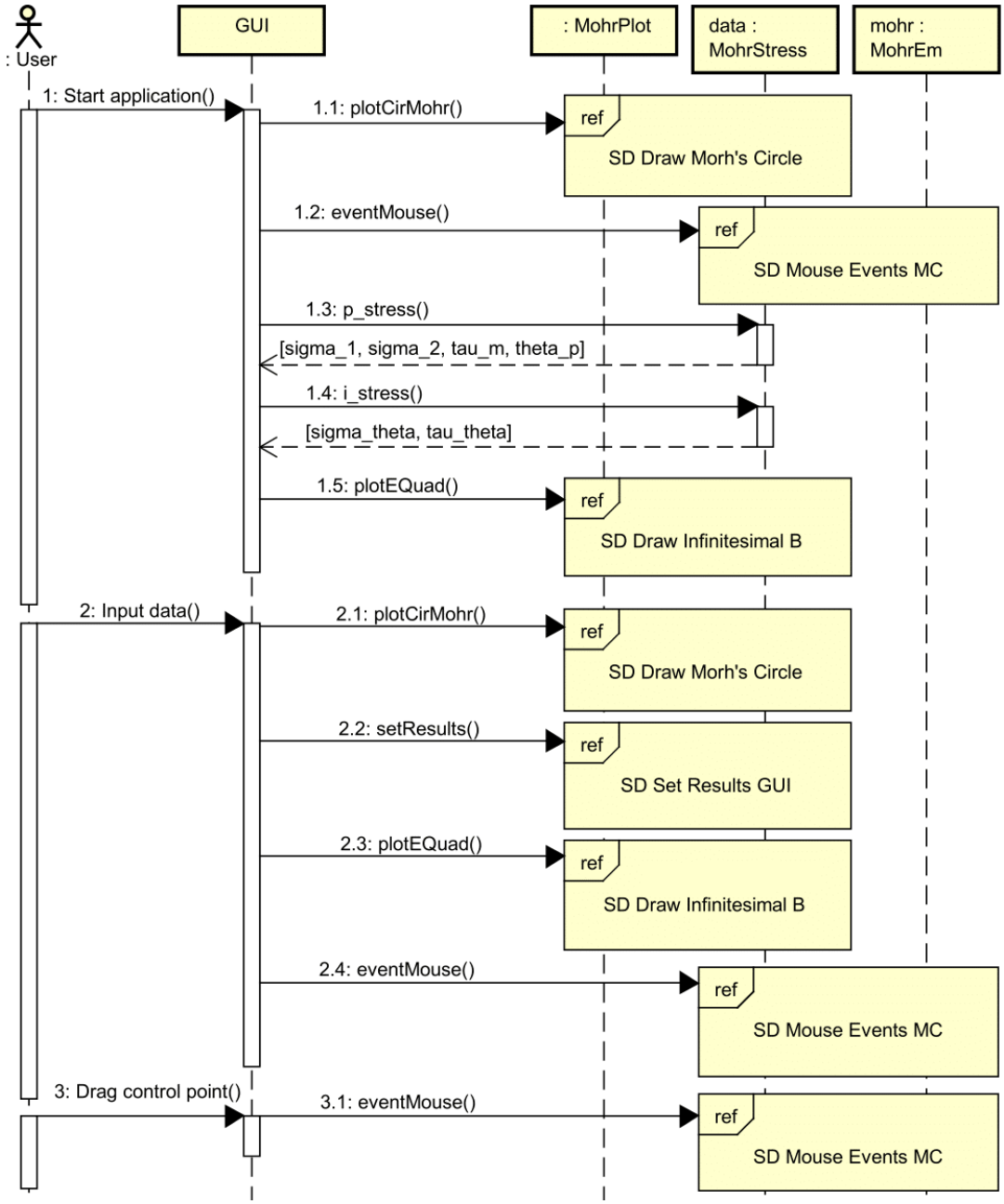

Figura 6.10: Diagrama de sequência e-Mohr2.

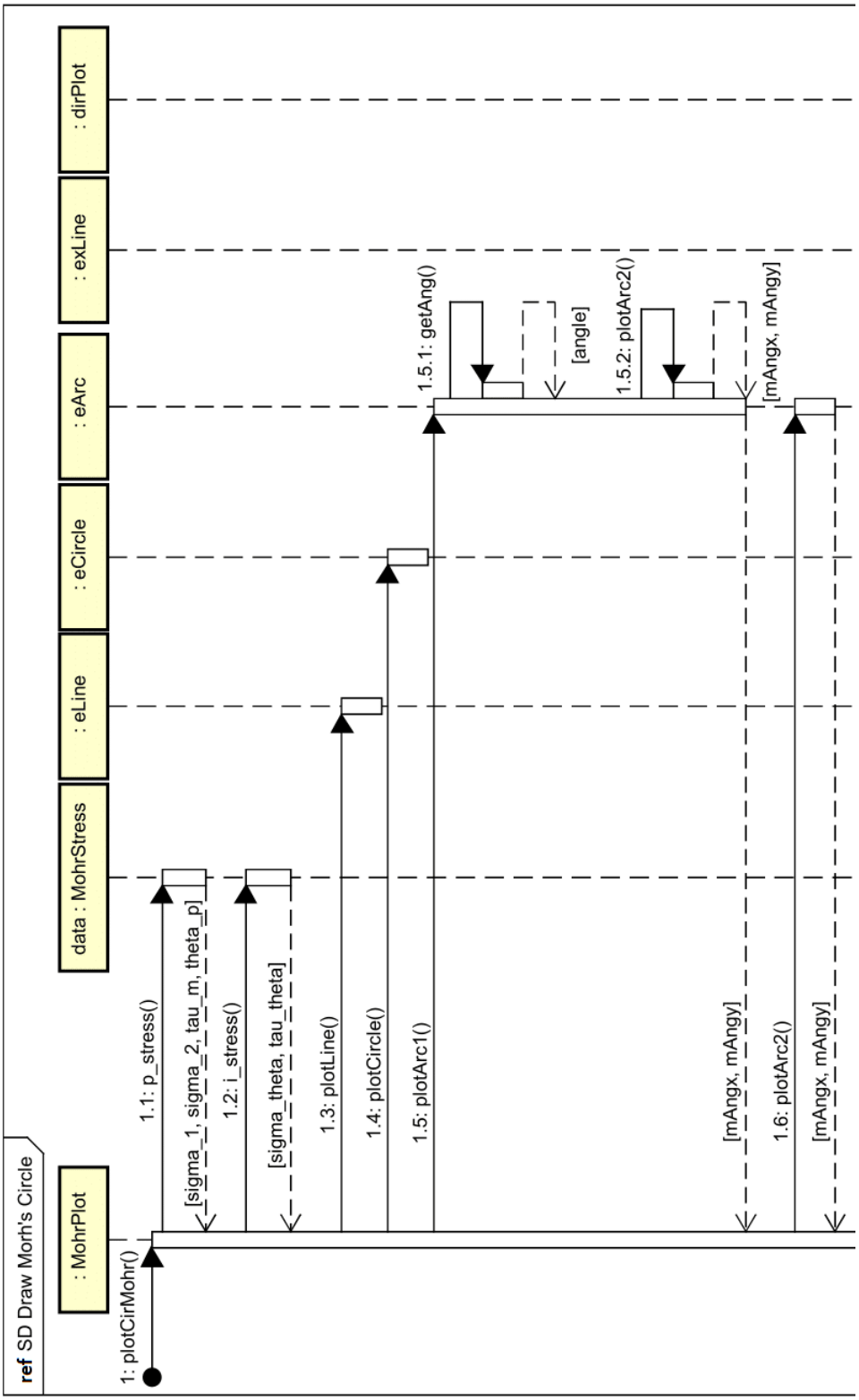

Figura 6.11: Fragmento de iteração: *SD Draw Morh's Circle* (Parte 1).

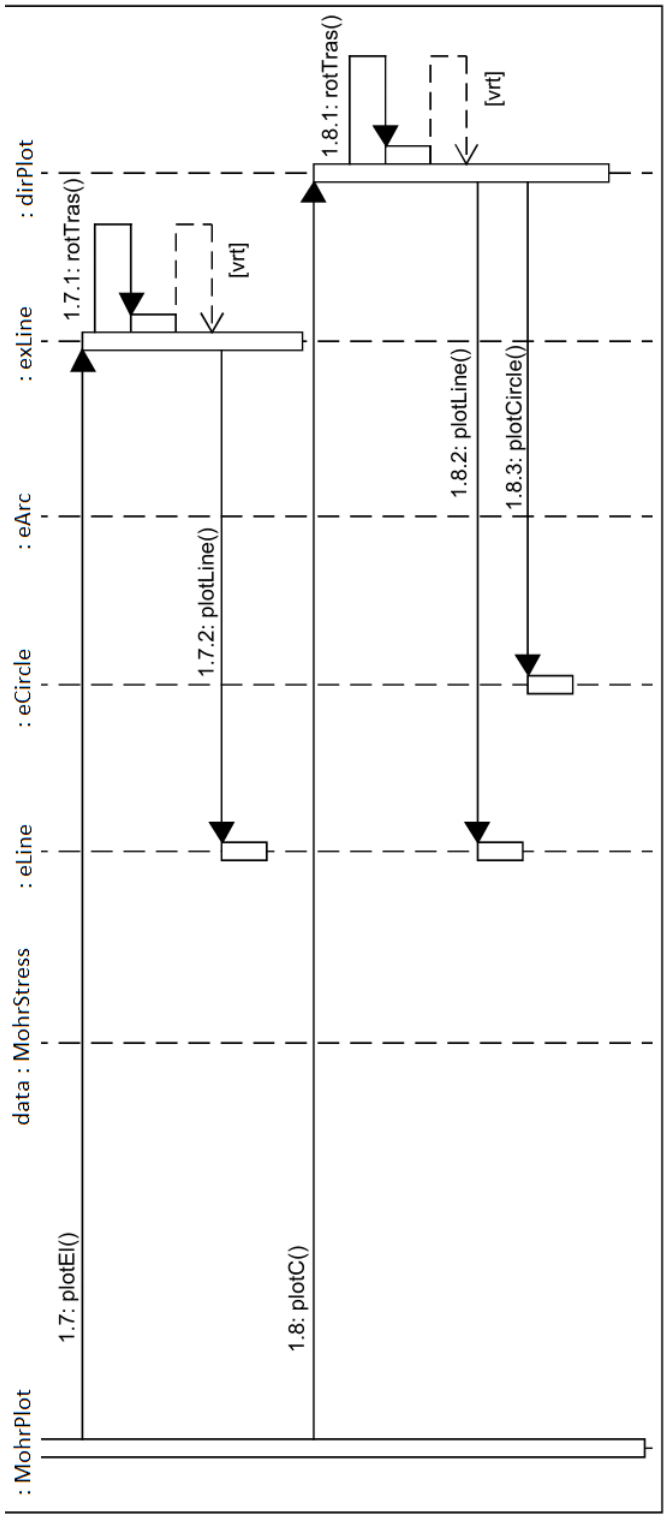

Figura 6.12: Fragmento de iteração: *SD Draw Morh's Circle* (Parte 2).

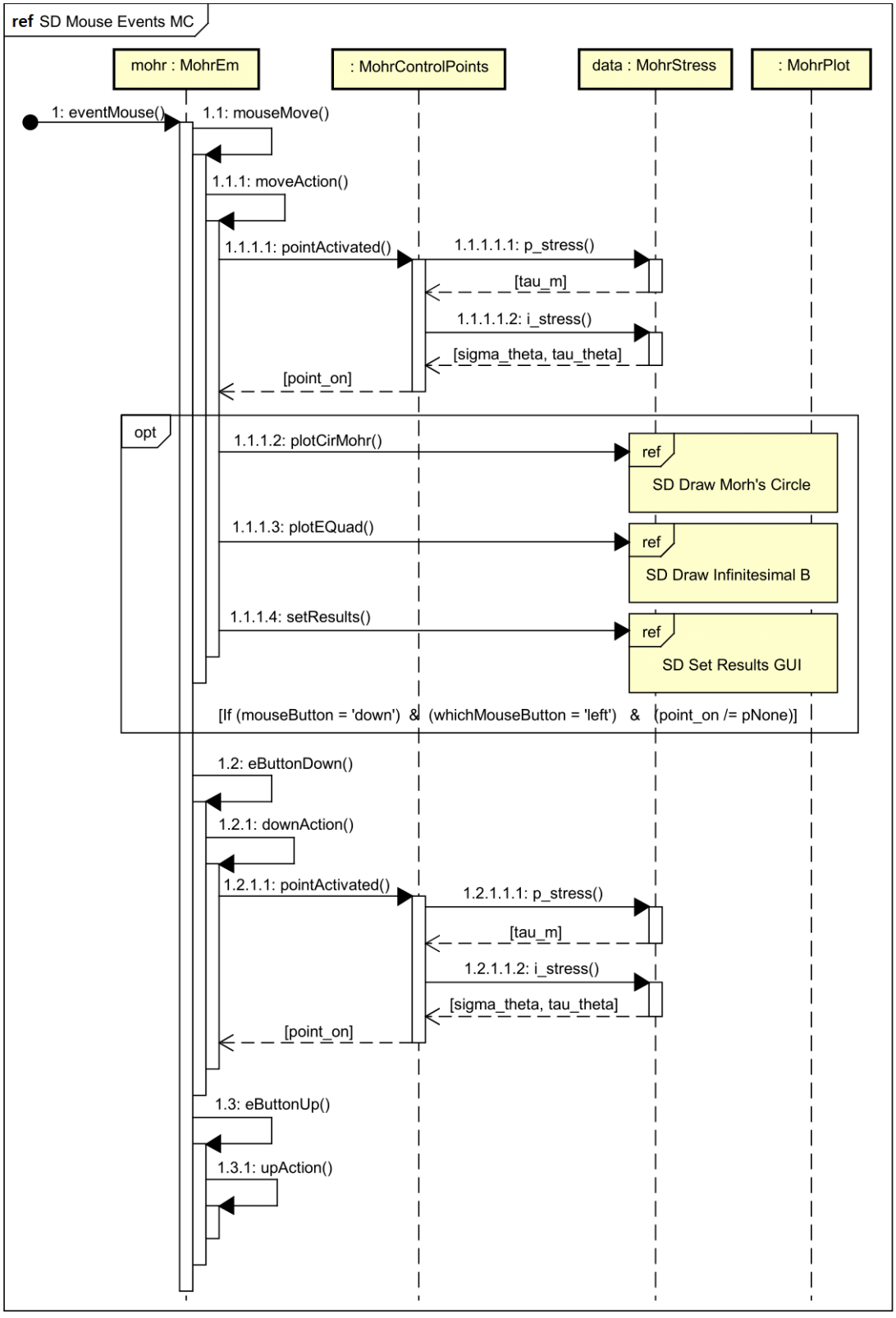

Figura 6.13: Fragmento de iteração: *SD Mouse Events MC*.

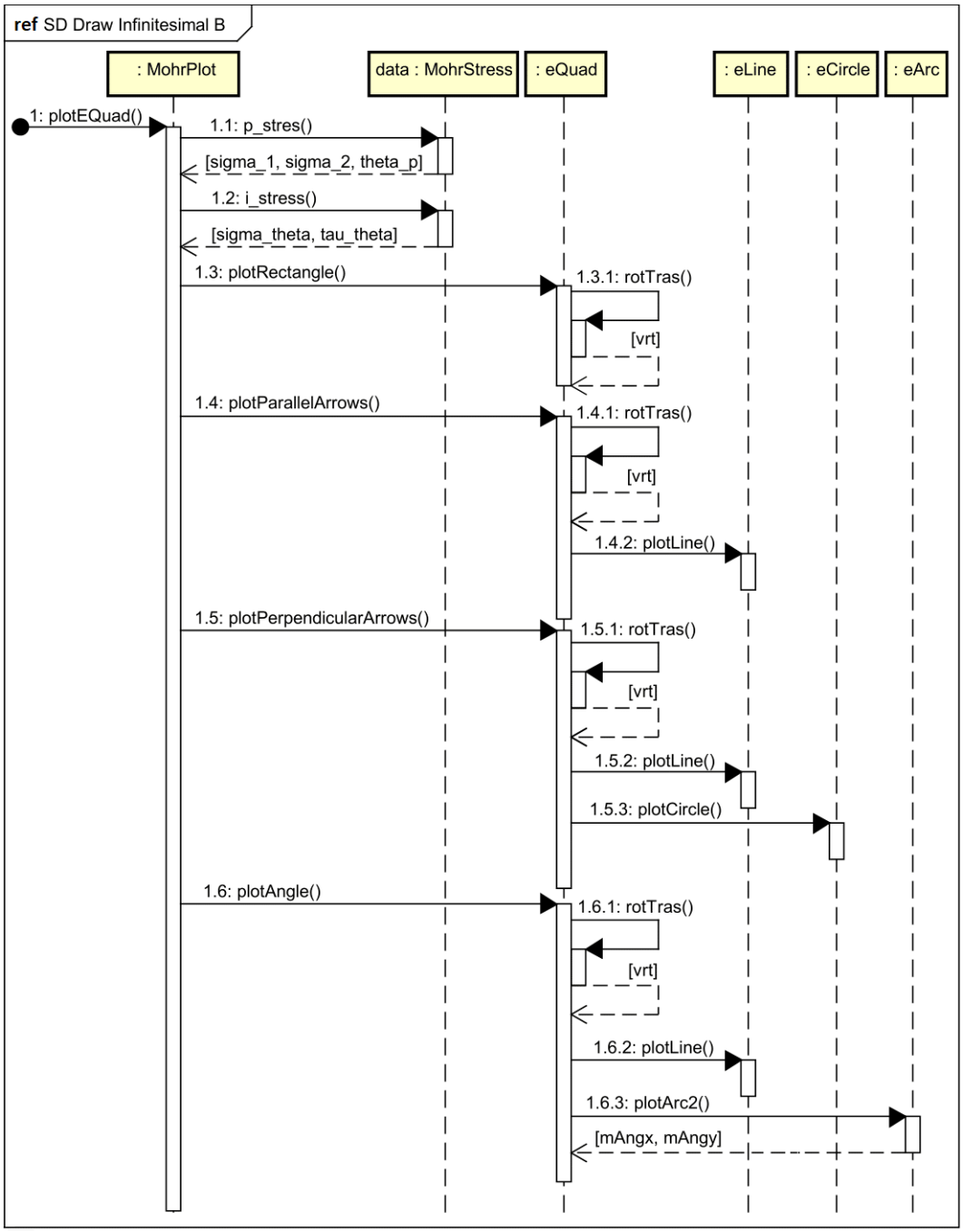

Figura 6.14: Fragmento de iteração: *SD Draw Infinitesimal B*.

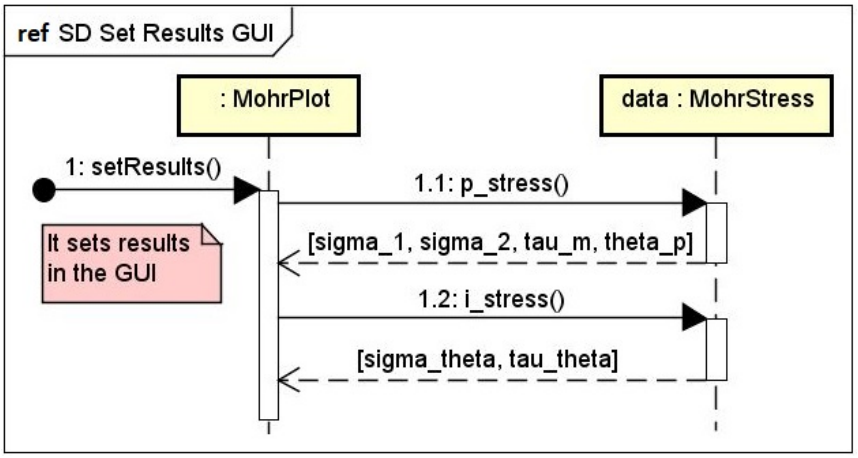

Figura 6.15: Fragmento de iteração: *SD Set Results GUI*.