

CIV2802 – Sistemas Gráficos para Engenharia
2024.1

Modelagem de Software Orientada a Objetos com UML



Luiz Fernando Martha

André Pereira



Conteúdo

- Conceitos Básicos de Orientação a Objetos
- UML (*Unified Modeling Language*)
- Modelagem de Software Orientada a Objetos
Calculadora RPN
- Introdução aos Padrões de Projeto

Abordagem Orientada a Objetos

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 à 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.

A filosofia Orientada a Objetos é **muito similar ao mundo real** e, portanto, vem ganhando popularidade pois os sistemas aqui são vistos como um conjunto de objetos que interagem assim como no mundo real. Para implementar este conceito, a programação estruturada baseada em processos não é utilizada; em vez disso, os **objetos são criados usando estruturas de dados**. Assim como toda linguagem de programação oferece vários tipos de dados, da forma similar, no caso dos objetos certos tipos de dados são pré-definidos. (Nath, 2014 – [Lecture Notes on Object-Oriented Methodology](#))

Orientação a Objetos

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 esclarecidos a seguir.

- **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 instancia gerada a partir de uma classe. Um objeto é identificado a partir dos métodos e dos atributos que possui.
- **Encapsulamento** é o ato de esconder do usuário os processos internos de um objeto, classe ou método.
- **Herança (e Polimorfismo)** é uma característica que permite a determinada classe herdar as características de outra classe. Ou seja, a classe descendente adquire todos os métodos e atributos da classe pai.

Métodos são as funções que objeto pode realizar.

Atributo é tudo que um objeto possui como variável.

Orientação a Objetos

Classe, Objeto e Encapsulamento

```
import numpy as np

class Stack:          Versão utilizando "array"
    def __init__(self):
        self.top = -1
        self.elem = np.zeros(50, dtype=int)

    def push(self, _n):
        if type(_n) == int:
            self.top += 1
            self.elem[self.top] = _n

    def pop(self):
        if self.top < 0:
            return None
        n = self.elem[self.top]
        self.top -= 1
        return n

    def isEmpty(self):
        return self.top < 0

    def dump(self):
        text = ""
        for i in range(self.top, -1, -1):
            text += f"{self.elem[i]}\n"
        return text
```

Orientação a Objetos

Classe, Objeto e Encapsulamento

```
import numpy as np

class Stack: Versão utilizando "array"
    def __init__(self):
        self.top = -1
        self.elem = np.zeros(50, dtype=int)

    def push(self, _n):
        if type(_n) == int:
            self.top += 1
            self.elem[self.top] = _n

    def pop(self):
        if self.top < 0:
            return None
        n = self.elem[self.top]
        self.top -= 1
        return n

    def isEmpty(self):
        return self.top < 0

    def dump(self):
        text = ""
        for i in range(self.top, -1, -1):
            text += f"{self.elem[i]}\n"
        return text
```

```
class Stack: Versão utilizando "list"
    def __init__(self):
        self.elem = []

    def push(self, _n):
        if type(_n) == int:
            self.elem.insert(0, _n)

    def pop(self):
        return self.elem.pop(0)

    def isEmpty(self):
        return len(self.elem) == 0

    def dump(self):
        text = ""
        for item in self.elem:
            text += f"{item}\n"
        return text
```

As estruturas de dados das duas implementações da classe Stack (utilizando "array" e utilizando "list") são encapsuladas dentro da classe. Clientes da classe não precisam saber da diferença de implementação, e utilizam a classe através dos métodos que têm chamadas iguais nas duas versões.

Orientação a Objetos

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 esclarecidos a seguir.

- **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 instancia gerada a partir de uma classe. Um objeto é identificado a partir dos métodos e dos atributos que possui.
- **Encapsulamento** é o ato de esconder do usuário os processos internos de um objeto, classe ou método.
- **Herança (e Polimorfismo)** é uma característica que permite a determinada classe herdar as características de outra classe. Ou seja, a classe descendente adquire todos os métodos e atributos da classe pai.

Métodos são as funções que objeto pode realizar.

Atributo é tudo que um objeto possui como variável.

Orientação a Objetos: Herança

Classe RPNcalc **sem herança**, i.e., com relacionamento do tipo **agregação**:
um objeto da classe RPNcalc **tem** uma propriedade que é um objeto da classe Stack.

```
from stack import *

class RPNcalc:
    def __init__(self):
        self.stack = Stack()

    def getOperands(self):
        if self.stack.isEmpty():
            print("Empty stack!\n")
            return False, 0, 0

        n1 = self.stack.pop()
        if self.stack.isEmpty():
            print("Two operands needed!\n")
            self.stack.push(n1)
            return False, 0, 0

        n2 = self.stack.pop()
        return True, int(n1), int(n2)

    def enterNumber(self, n):
        self.stack.push(n)
```

```
    def selectOp_sum(self):
        check, n1, n2 = self.getOperands()
        if not check:
            return
        self.stack.push(n2+n1)

    def selectOp_sub(self):
        check, n1, n2 = self.getOperands()
        if not check:
            return
        self.stack.push(n2-n1)

    def selectOp_mul(self):
        check, n1, n2 = self.getOperands()
        if not check:
            return
        self.stack.push(n2*n1)

    def selectOp_div(self):
        check, n1, n2 = self.getOperands()
        if not check:
            return
        if n1 != 0:
            self.stack.push(int(n2/n1))
        else:
            print("Error: division by zero!\n")
```

Orientação a Objetos: Herança

Classe RPNcalc com relacionamento do tipo **herança**: a classe RPNcalc **herda** propriedades e métodos da classe Stack, e adiciona outras propriedades e métodos (**especialização**). Um objeto da classe RPNcalc **também é** um objeto da classe Stack.

```
from stack import *  
  
class RPNcalc(Stack):  
    def __init__(self):  
        super(RPNcalc, self).__init__()  
  
    def getOperands(self):  
        if self.isEmpty():  
            print("Empty stack!\n")  
            return False, 0, 0  
  
        n1 = self.pop()  
        if self.isEmpty():  
            print("Two operands needed!\n")  
            self.push(n1)  
            return False, 0, 0  
  
        n2 = self.pop()  
        return True, int(n1), int(n2)  
  
    def enterNumber(self, _n):  
        self.push(_n)
```

```
    def selectOp_sum(self):  
        check, n1, n2 = self.getOperands()  
        if not check:  
            return  
        self.push(n2+n1)  
  
    def selectOp_sub(self):  
        check, n1, n2 = self.getOperands()  
        if not check:  
            return  
        self.push(n2-n1)  
  
    def selectOp_mul(self):  
        check, n1, n2 = self.getOperands()  
        if not check:  
            return  
        self.push(n2*n1)  
  
    def selectOp_div(self):  
        check, n1, n2 = self.getOperands()  
        if not check:  
            return  
        if n1 != 0:  
            self.push(int(n2/n1))  
        else:  
            print("Error: division by zero!\n")
```

UML

Unified Modeling Language

(Linguagem de Modelagem Unificada)

UML



Linguagem de Modelagem Unificada

UML é uma linguagem padrão da indústria para:



Specifying
(Especificação)



Visualizing
(Visualização)



Constructing
(Construção)



Documenting
(Documentação)



Business Modeling
(Modelagem de negócios)



Communications
(Comunicações)

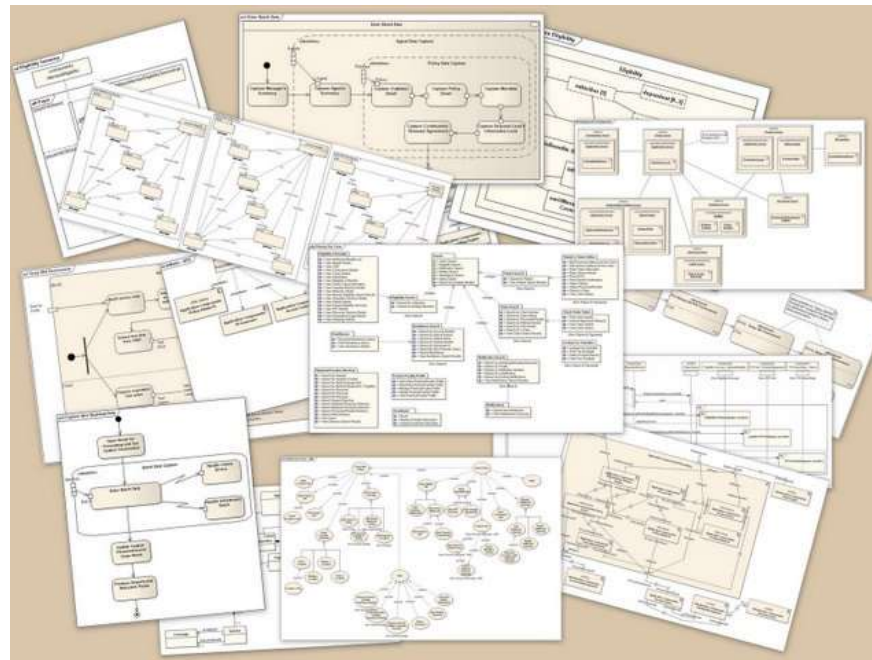
Componentes de um Software

UML

Linguagem de Modelagem Unificada

Definição:

É uma linguagem gráfica para visualizar, especificar, construir e documentar os artefatos de um sistema computacional orientado a objetos



UML



Linguagem de Modelagem Unificada

Definição:

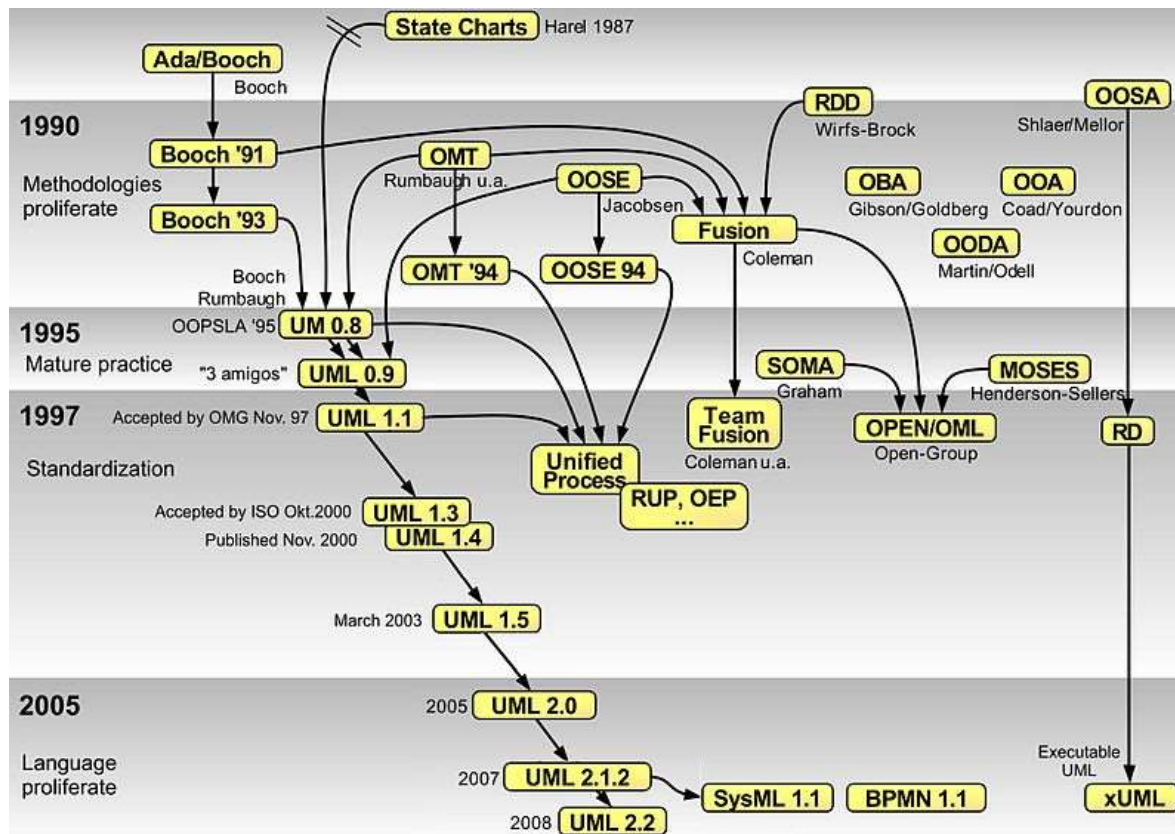
É uma linguagem gráfica para visualizar, especificar, construir e documentar os artefatos de um sistema computacional orientado a objetos

Vantagens:

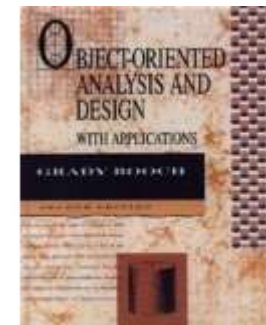
- Desenvolvimento de programas de forma rápida, eficiente e efetiva;
- Revela a estrutura desejada e o comportamento do sistema;
- Permite a visualização e controle da arquitetura do sistema;
- Melhor entendimento do sistema que está sendo construído e gerenciamento de riscos.

UML

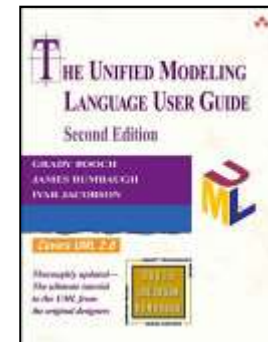
Breve Histórico



1993

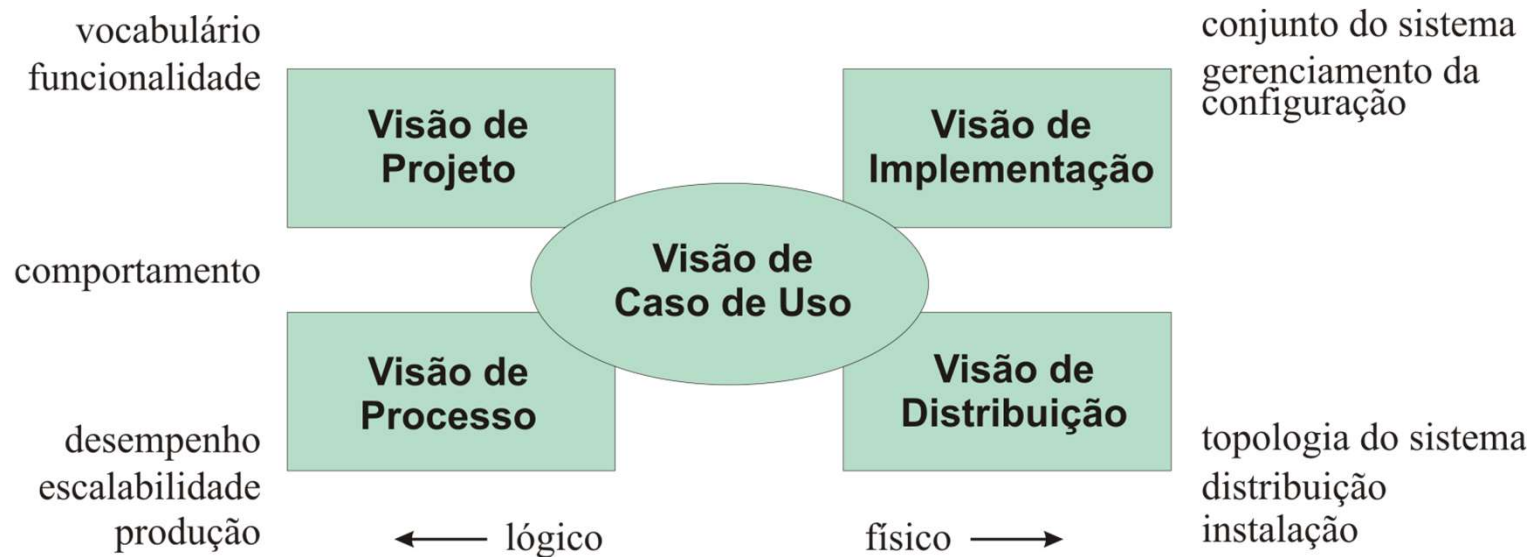


2005



UML

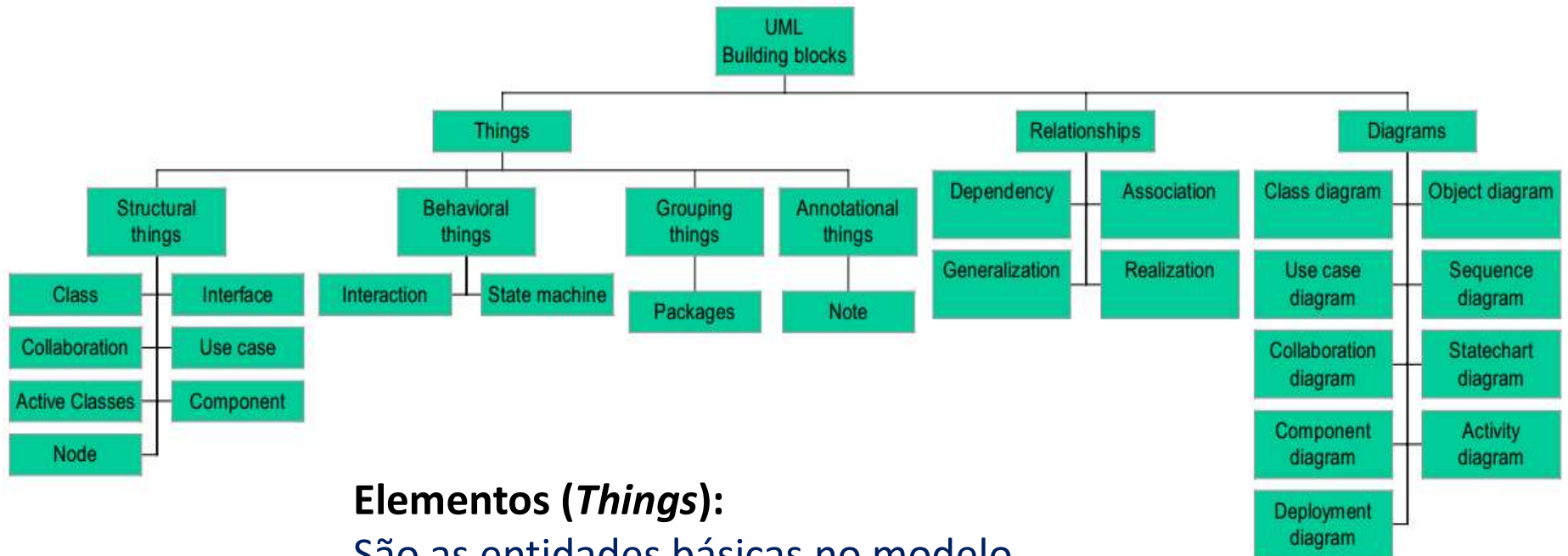
Visões (Arquitetura de um Sistema OO)



De acordo com a UML, deve-se ter uma **visão de casos de uso**, expondo as exigências do sistema; uma **visão de projeto**, capturando o vocabulário do espaço do problema e do espaço da solução; uma **visão do processo**, modelando a distribuição dos processos e linhas do sistema; uma **visão de implementação**, dirigindo-se à realização física do sistema; e uma **visão de distribuição**, focando na edição da engenharia de sistema.

Cada uma dessas visões pode ter aspectos estruturais, assim como comportamentais. Juntas, essas visões representam a especificação completa de um sistema computacional.

Blocos de Construção da UML



Elementos (*Things*):

São as entidades básicas no modelo.

Relações (*Relationships*):

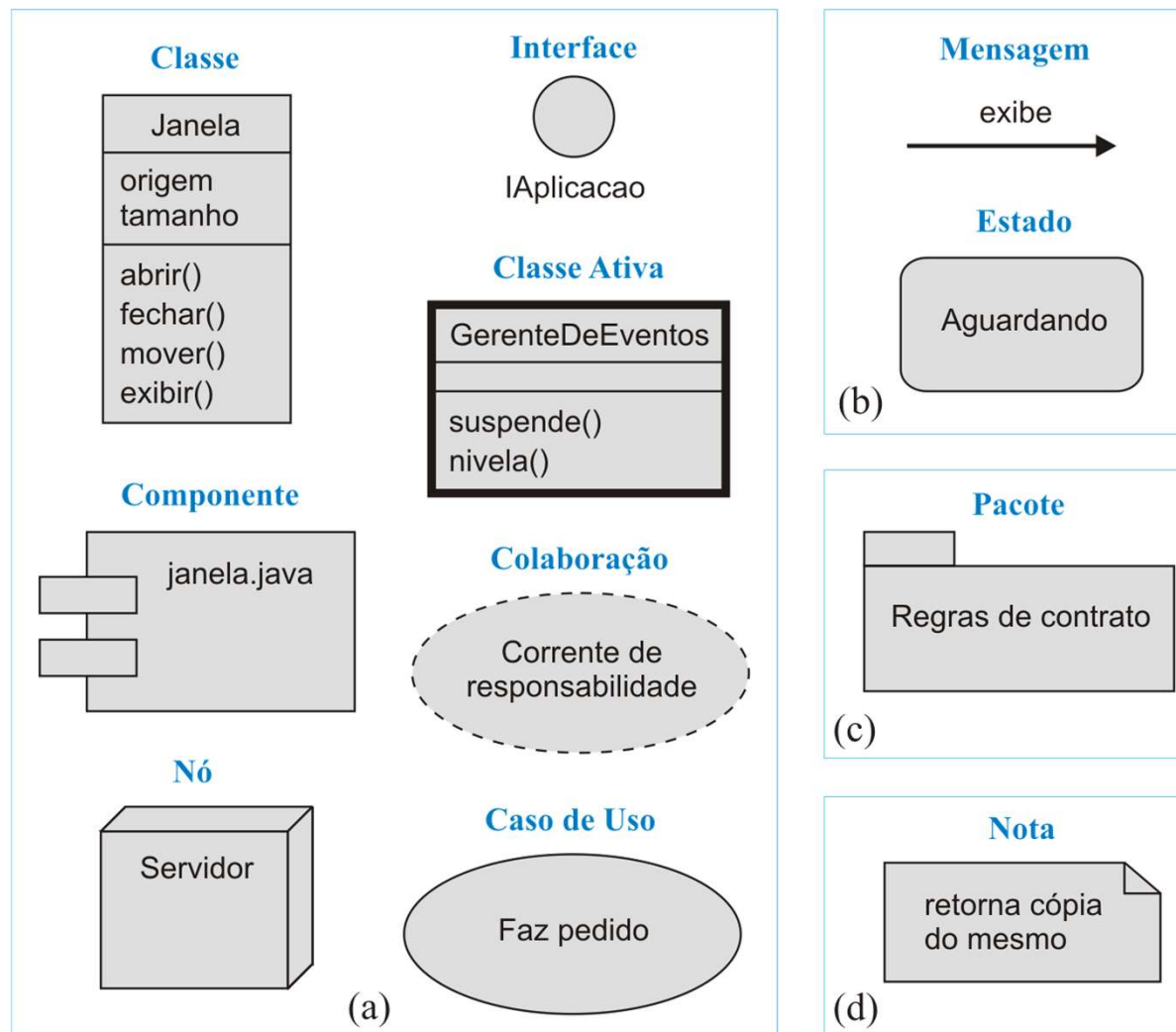
Conecta e amarra os elementos.

Diagramas (*Diagrams*):

Eles são os gráficos dos elementos e suas relações.

Blocos de Construção

Elementos em UML



Blocos de Construção

Elementos: Classes em UML

Como representar
esta classe em UML?

```
class Number:  
    def __init__(self):  
        self.m_type = 0  
  
    def sum(self, _n):  
        pass  
  
    def sub(self, _n):  
        pass  
  
    def mul(self, _n):  
        pass  
  
    def div(self, _n):  
        pass
```

Blocos de Construção

Elementos: Classes em UML

<i>Number</i>
m_type
sum(_n)
sub(_n)
mul(_n)
div(_n)

<i>Number</i>
m_type:int
+ sum(_n:Number) : Number
+ sub(_n:Number) : Number
+ mul(_n:Number) : Number
+ div(_n:Number) : Number

```
class Number:  
    def __init__(self):  
        self.m_type = 0  
  
    def sum(self, _n):  
        pass  
  
    def sub(self, _n):  
        pass  
  
    def mul(self, _n):  
        pass  
  
    def div(self, _n):  
        pass
```

Blocos de Construção

Elementos: Classes em UML

<i>Number</i>
m_type
sum(_n)
sub(_n)
mul(_n)
div(_n)

<i>Number</i>
m_type:int
+ sum(_n:Number) : Number
+ sub(_n:Number) : Number
+ mul(_n:Number) : Number
+ div(_n:Number) : Number

Real
m_value
sum(_n)
sub(_n)
mul(_n)
div(_n)

Complex
- m_real:float
- m_imag:float
+ sum(_n:Number) : Number
+ sub(_n:Number) : Number
+ mul(_n:Number) : Number
+ div(_n:Number) : Number

```
class Number:
    def __init__(self):
        self.m_type = 0

    def sum(self, _n):
        pass

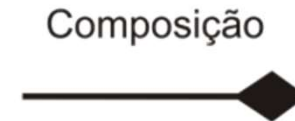
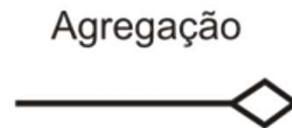
    def sub(self, _n):
        pass

    def mul(self, _n):
        pass

    def div(self, _n):
        pass
```

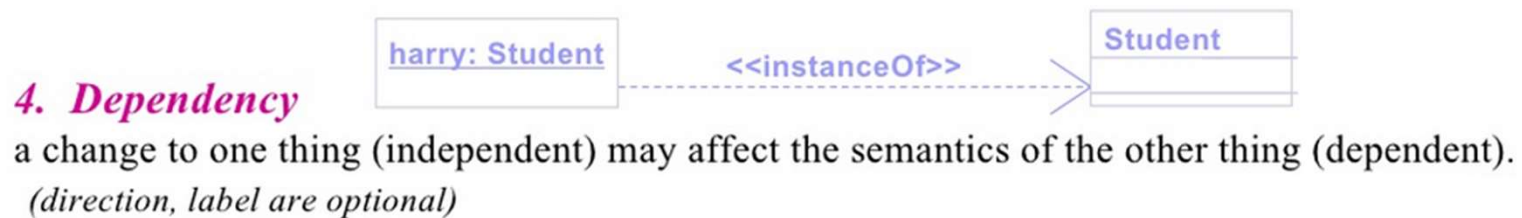
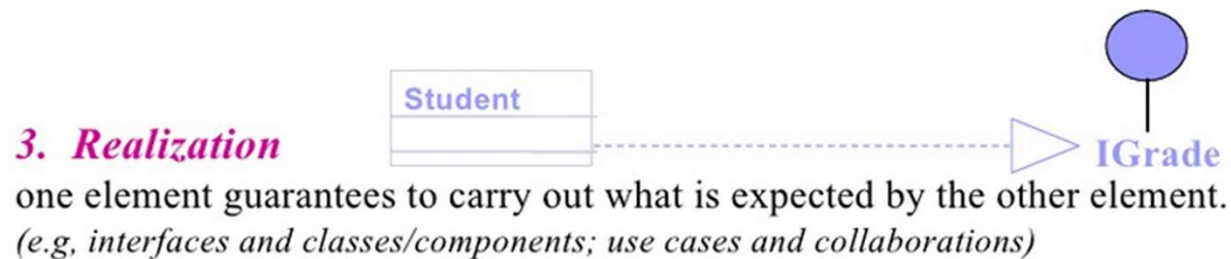
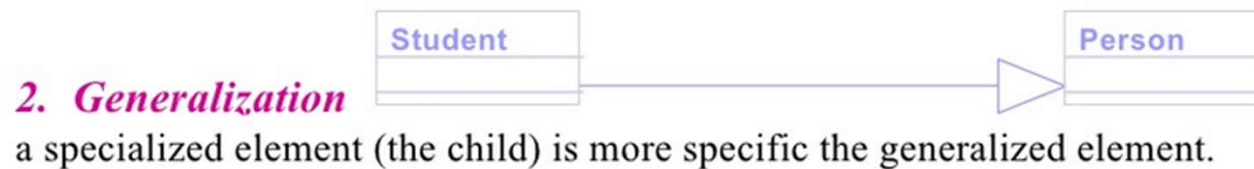
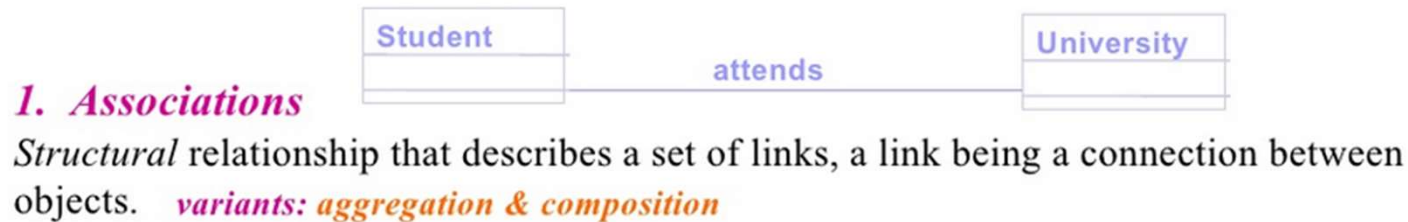
Blocos de Construção

Relações em UML



Blocos de Construção

Relações em UML



Blocos de Construção

Relações: entre Classes UML

Stack
m_top:int m_elems:*INumber
push(_n:INumber) pop() : INumber isEmpty() : bool dump()

<i>INumber</i>
m_type:int
+ sum(_n:INumber) : INumber + sub(_n:INumber) : INumber + mul(_n:INumber) : INumber + div(_n:INumber) : INumber

Como estas classes
estão relacionadas?

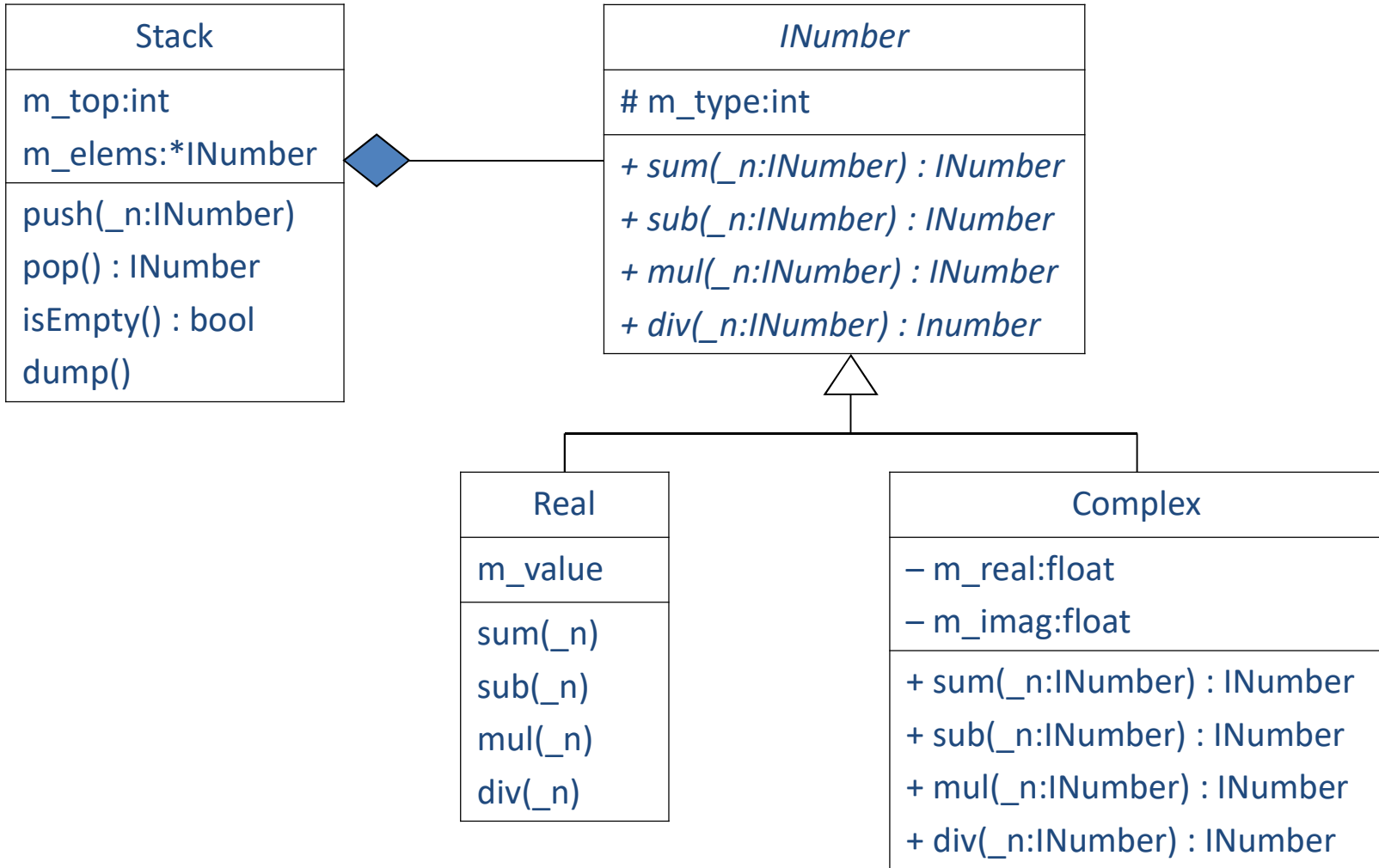
Como representar estas
relações em UML?

Real
m_value
sum(_n) sub(_n) mul(_n) div(_n)

Complex
- m_real:float - m_imag:float
+ sum(_n:INumber) : INumber + sub(_n:INumber) : INumber + mul(_n:INumber) : INumber + div(_n:INumber) : INumber

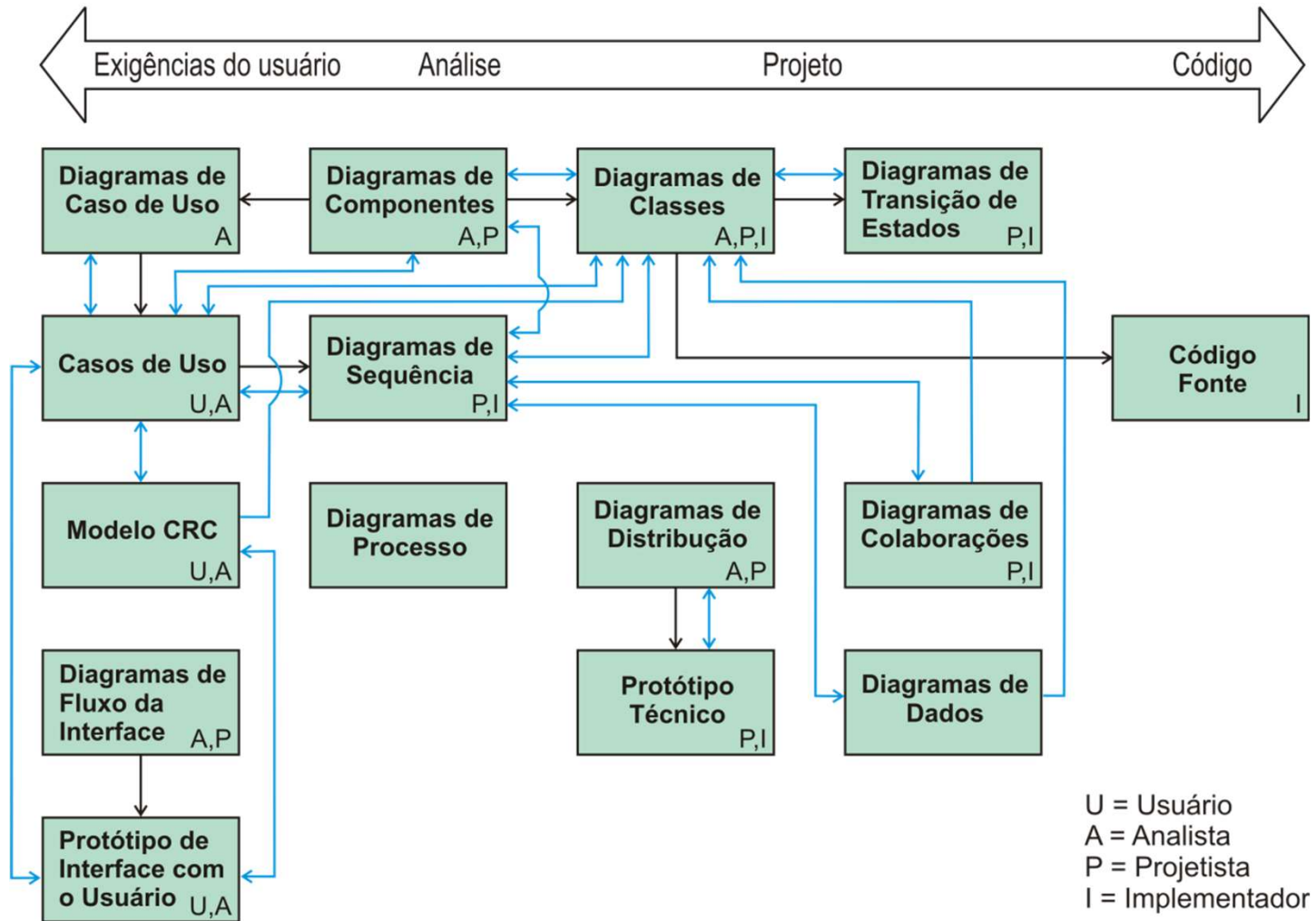
Blocos de Construção

Relações: entre Classes UML



Blocos de Construção

Diagramas em UML



CRC: classe, responsabilidade e colaboração

Modelagem de Software Orientada a Objetos

Modelagem Orientada a Objetos

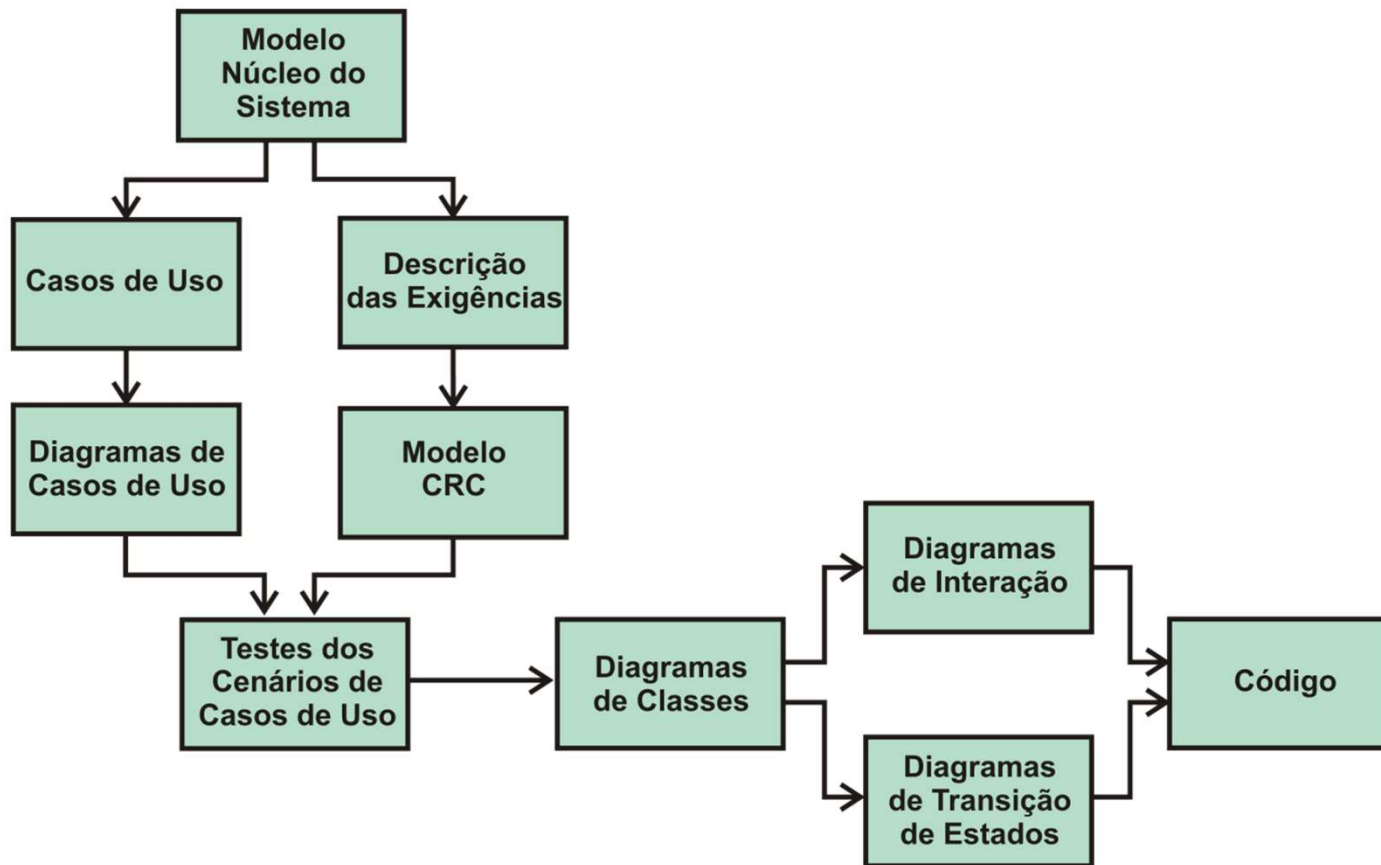
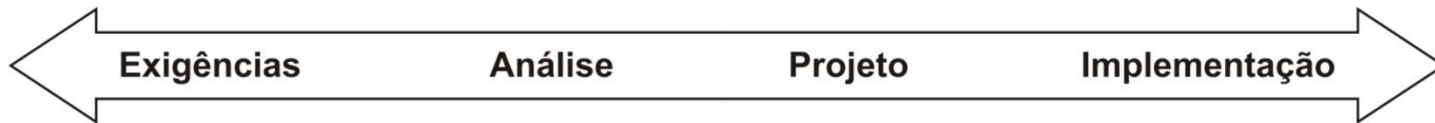
Uma metodologia é um processo organizado de produção de software, que utiliza técnicas predefinidas e notações convencionais.

As etapas que compõem este processo correspondem ao ciclo de vida do software.

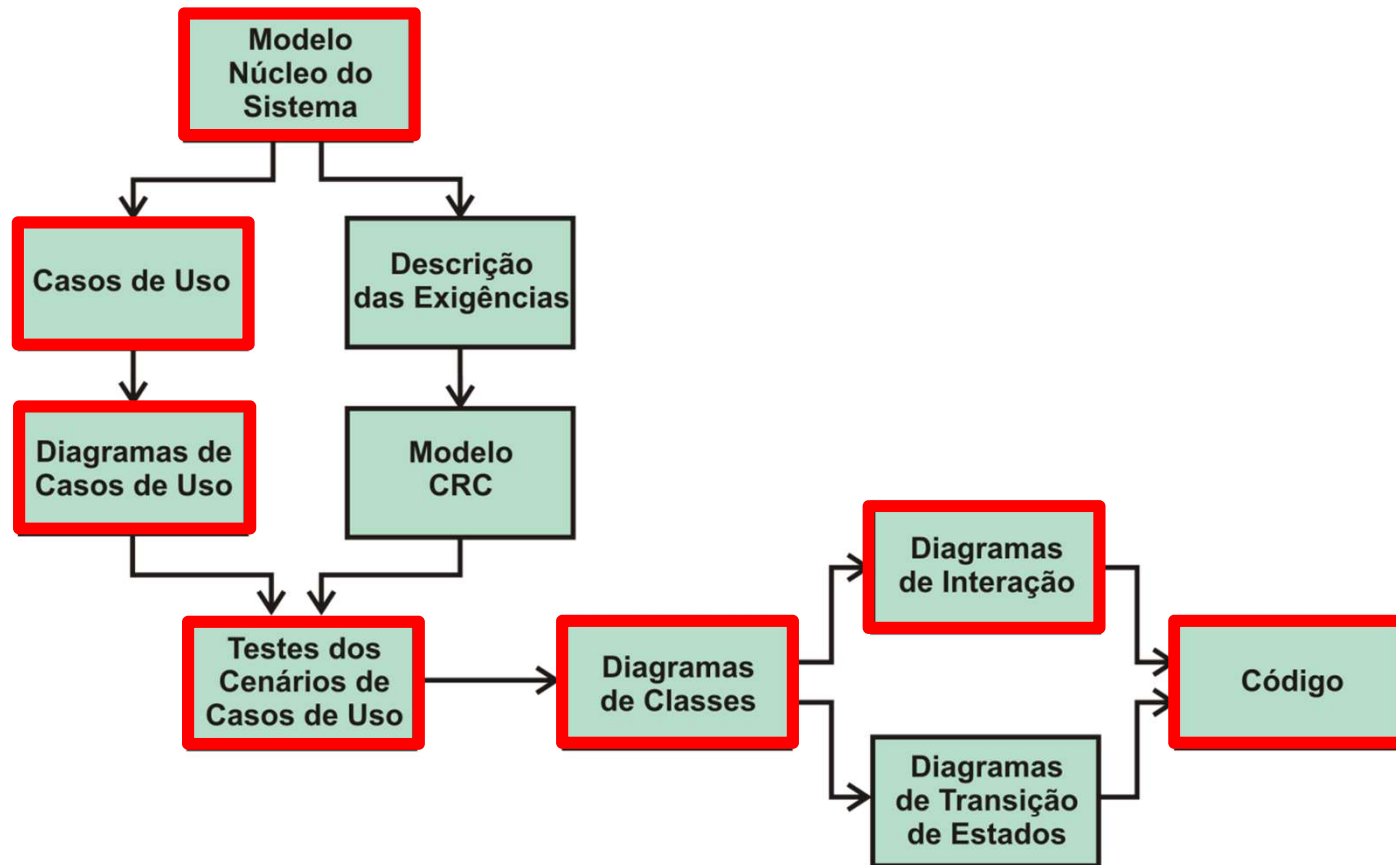
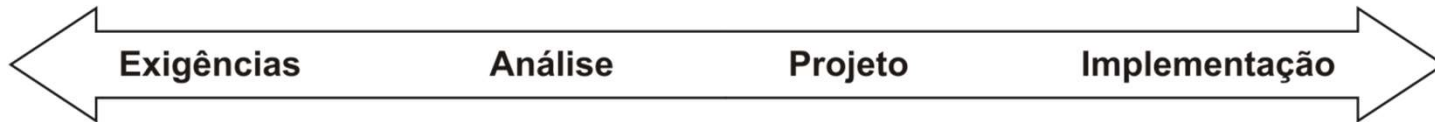
Tradicionalmente, a formulação inicial do problema, a análise, o projeto, a implementação, os testes e a operação (manutenção e aperfeiçoamento) compõem estas etapas do ciclo de vida.

“Um modelo é uma abstração de alguma coisa, cujo propósito é permitir que se conheça essa coisa antes de se construí-la” (Rumbaugh, 1994).

Modelagem Orientada a Objetos



Modelagem Orientada a Objetos



Modelagem Orientada a Objetos

- **Exigências**
 - Pré-requisitos / Requerimentos
 - Interface com o Usuário
- **Análise Orientada a Objetos**
 - Casos de Uso
 - Diagrama de Robustez
- **Projeto Orientado a Objetos**
 - Diagramas de Sequência
 - Diagramas de Classe
- **Programação Orientada a Objetos**

Modelagem Orientada a Objetos de uma Calculadora RPN

Exigências / Pré-requisitos

Deve ser possível inserir vários números na calculadora. Os números podem ser inteiros, reais e complexos. Os números reais têm duas casas decimais e os complexos têm duas casas decimais nas partes real e imaginária.

Deve ser possível realizar as quatro operações básicas: adição, subtração, multiplicação e divisão.

As operações devem ser realizadas com os dois últimos números que entraram na calculadora. Portanto, o pré-requisito para fazer uma operação é ter entrado com pelo menos dois números. O resultado de cada operação é um novo número criado, que substitui os dois números utilizados na operação. O restante dos números fica inalterado.

Devem ser visualizados apenas os quatro últimos números entrados.

Modelagem Orientada a Objetos de uma Calculadora RPN

Interface com o Usuário

Esboço da Interface gráfica do programa.

Estão faltando no esboço os seguintes botões:

- *enter*



- apagar o último número inserido

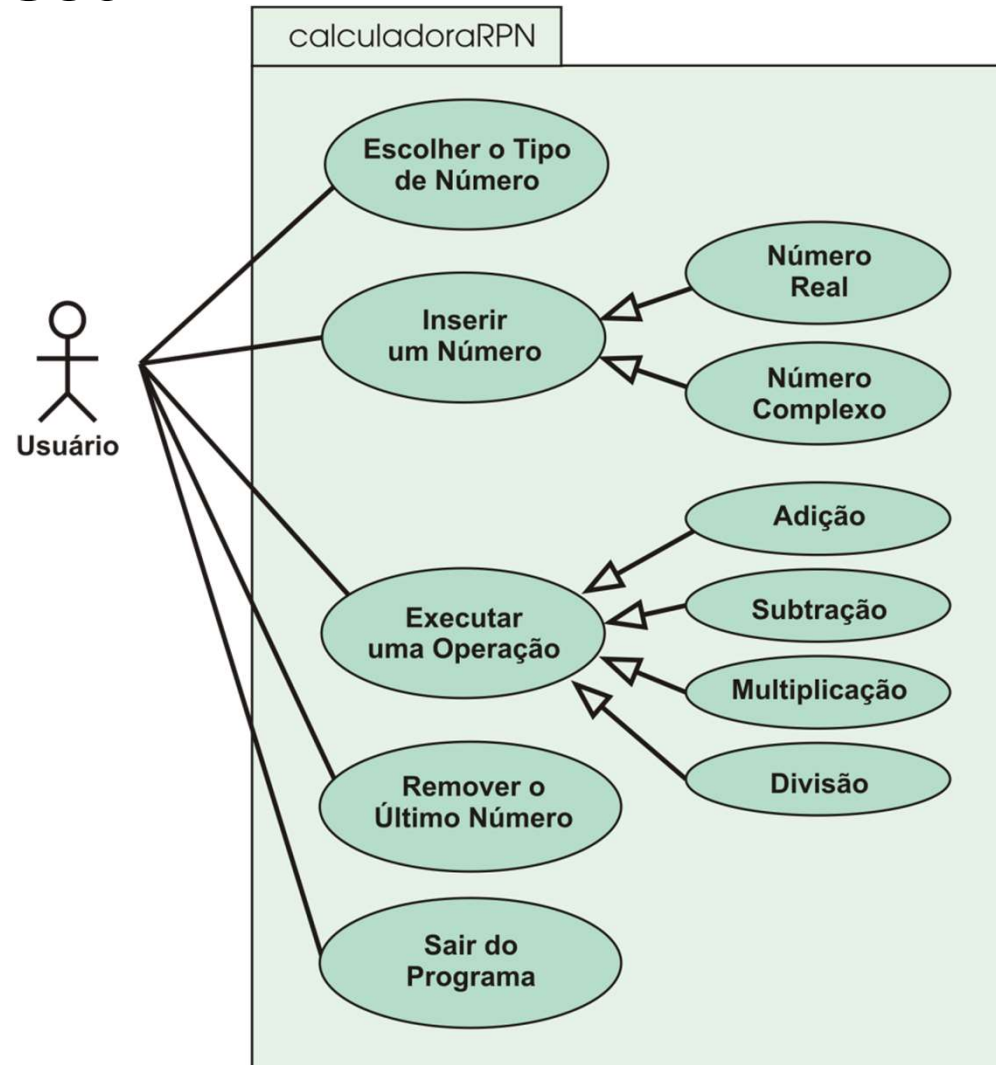


- chavear diferente tipo de número



Análise Orientada a Objetos da Calculadora RPN

Casos de Uso



Análise Orientada a Objetos da Calculadora RPN

Casos de Uso

- Escolher o Tipo de Número

Pode ser uma opção realizada no início da execução do programa, que irá definir o comportamento da calculadora. Durante a execução do programa, o usuário também pode pressionar um botão para escolher o tipo de número que ele quer trabalhar. Os números que já estão na calculadora, devem ser automaticamente convertidos para o novo formato.

- Inserir um Número

O caso de uso "Inserir um Número" é inicializado quando o usuário pressiona um botão correspondente ao número que ele deseja inserir na calculadora. Se o número for do tipo Inteiro ou Real basta ele clicar no botão com o número, porém se o tipo for complexo ele precisa inserir primeiro a parte real e em seguida, após um espaço, a parte imaginária.

- Executar uma Operação

Esse caso de uso é inicializado quando o usuário pressiona o botão correspondente a operação que ele deseja realizar. Qualquer operação é realizada com os dois últimos números que entraram na calculadora, porém o resultado depende da operação.

- Remover o último número

Remove o último número sem fazer nenhuma operação. O penúltimo passa a ser o último.

- Sair do Programa

Esse caso de uso é inicializado quando o usuário clica a caixa de fechamento do programa na janela principal do aplicativo. Os valores que estão na calculadora são perdidos.

Análise Orientada a Objetos da Calculadora RPN

Diagrama de Robustez

Um diagrama de robustez é basicamente um diagrama de colaboração UML simplificado.

Uma leitura inicial dos casos de uso sugere que o seguinte será parte do sistema:

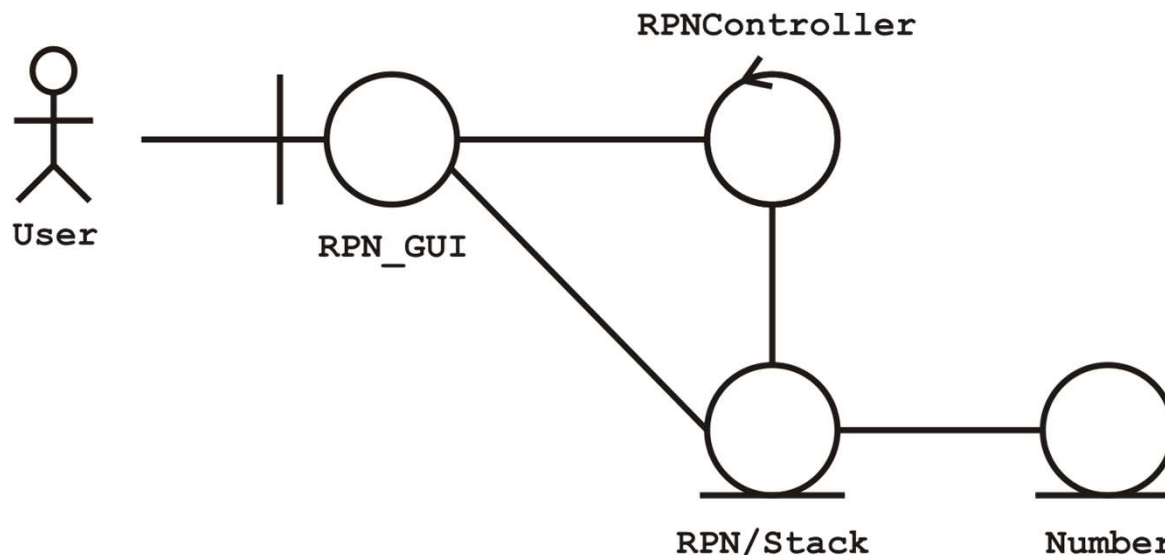
- Um objeto ou entidade única para representar a calculadora (`RPN`).
- Uma quantidade arbitrária de objetos, cada representando um determinado número (`Number`). Esse número ainda pode ser: inteiro (`Integer`), real (`Real`) ou complexo (`Complex`).
- Uma estrutura de dados especial para armazenar os números, sendo que o último número entrado é o primeiro a ser operado ou removido. Logo, a estrutura que demonstra ser a mais adequada para essa aplicação é a pilha (`Stack`).
- Um objeto gráfico representando a interface entre o sistema calculadora e o usuário (`RPN_GUI`).
- A controller object that carries out the use cases in response to user gestures on the GUI (`RPNController`). (Para um problema pequeno como esse, um único controle é suficiente.)

Análise Orientada a Objetos da Calculadora RPN

Diagrama de Robustez

Os vários casos de uso trabalham com esses objetos, como se segue:

- *Inserir um número* envolve pegar a nova informação do usuário, e então dizer ao objeto RPN para adicionar um novo número com essa informação na sua coleção.
- *Executar uma operação* envolve retirar os dois últimos números guardados no objeto RPN, executar a operação com esses números e mostrar na tela o seu resultado, o qual é adicionado como um novo número na sua coleção.
- etc...



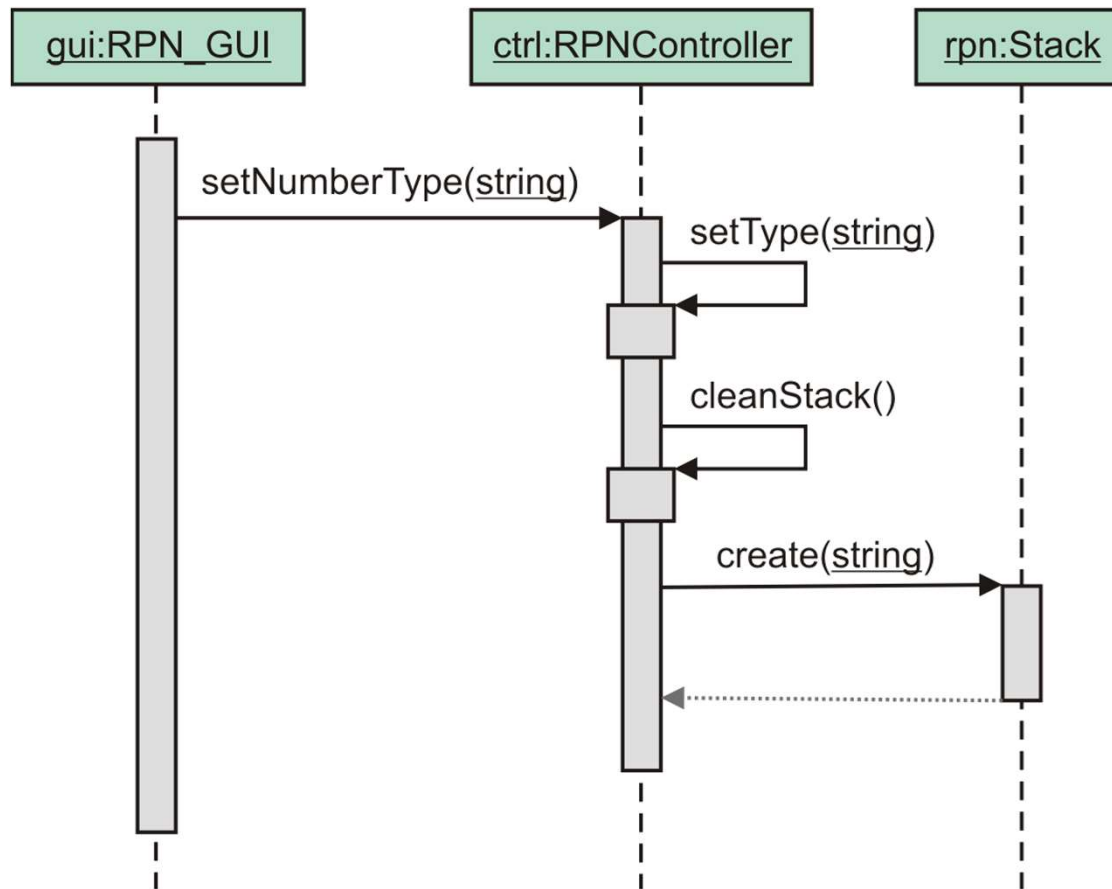
Projeto Orientado a Objetos da Calculadora RPN

Diagramas de Sequência

Cada um dos casos de uso descobertos na Análise do sistema será realizado por uma sequência de operações envolvendo os vários objetos que constituem o sistema:

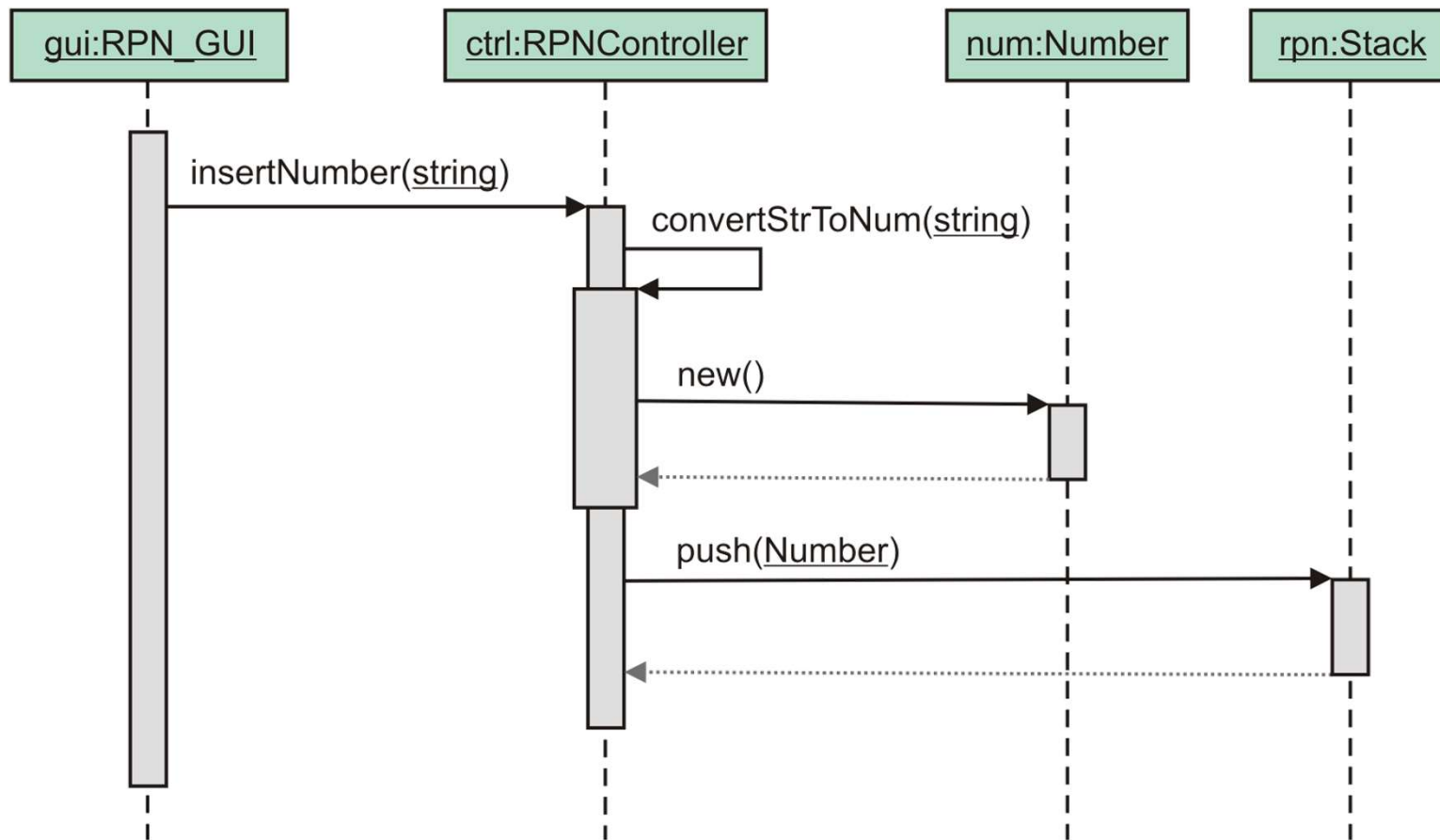
Projeto Orientado a Objetos da Calculadora RPN

Diagramas de Sequência



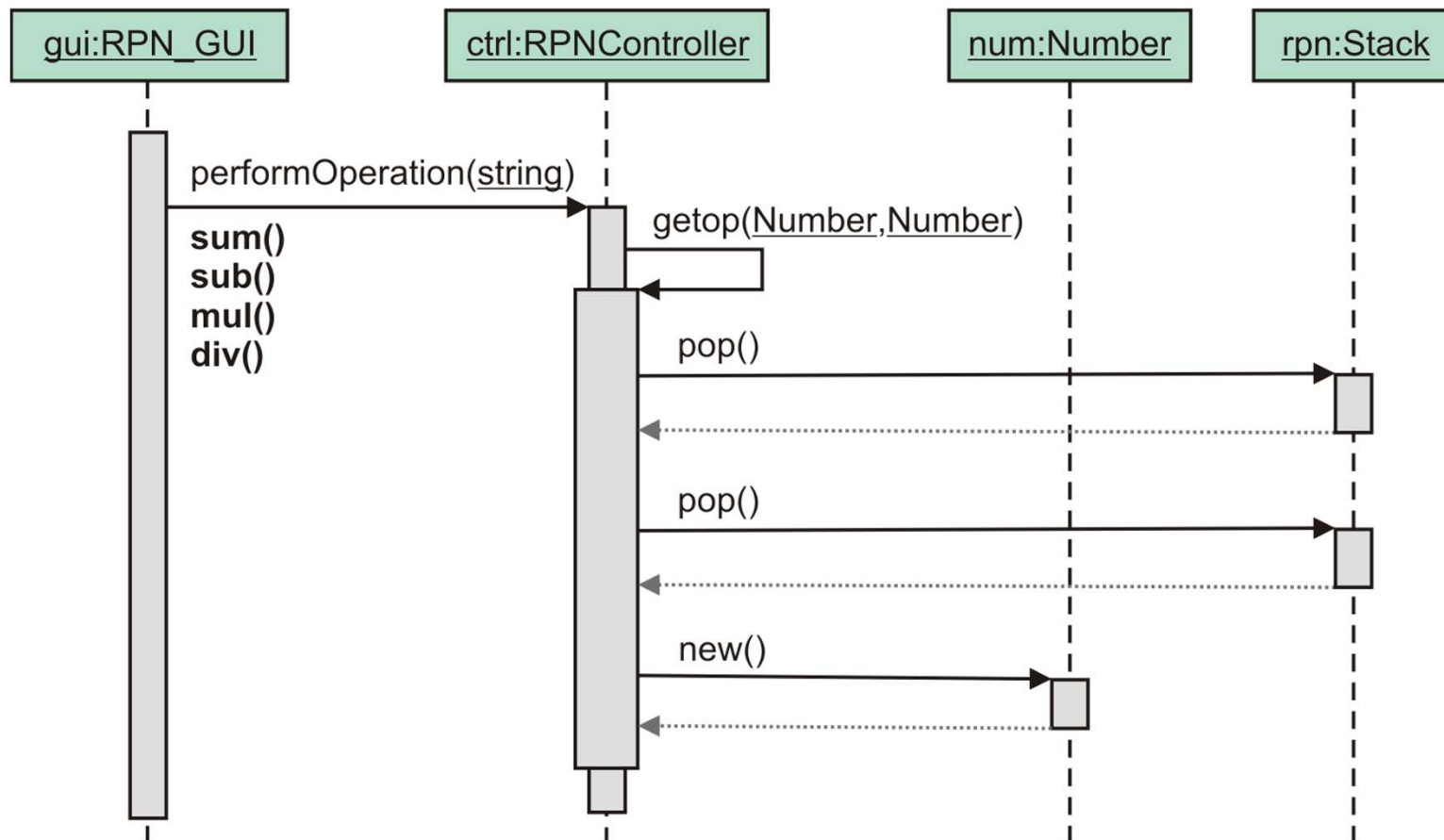
Projeto Orientado a Objetos da Calculadora RPN

Diagramas de Sequência



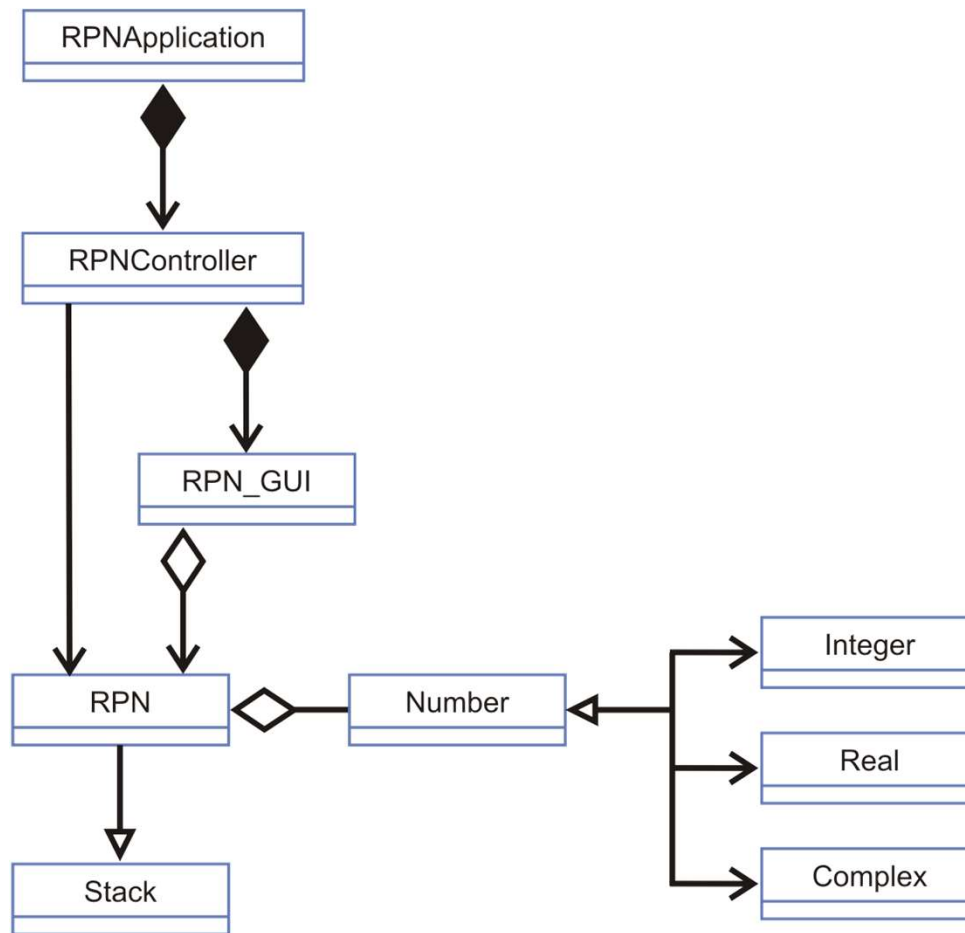
Projeto Orientado a Objetos da Calculadora RPN

Diagramas de Sequência



Projeto Orientado a Objetos da Calculadora RPN

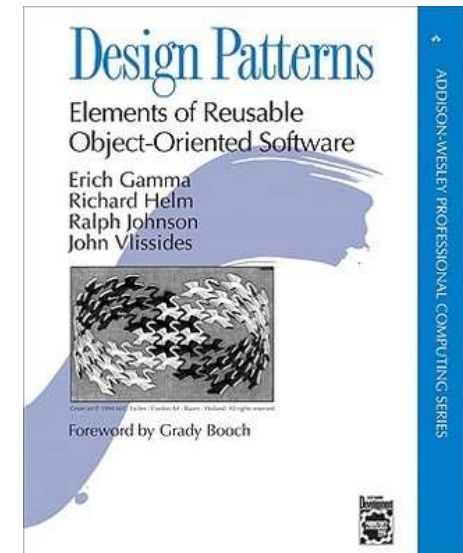
Diagrama de Classe



Introdução a Padrões de Projeto ***(Design Patterns)***

Padrões de Projeto

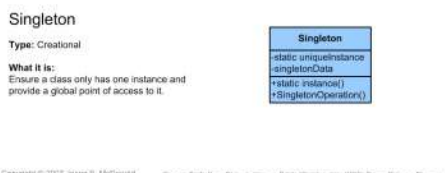
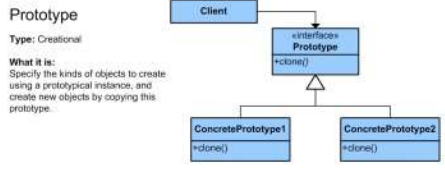
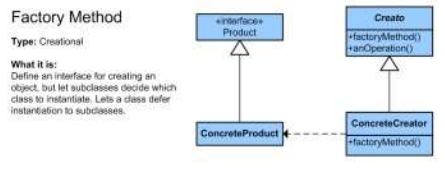
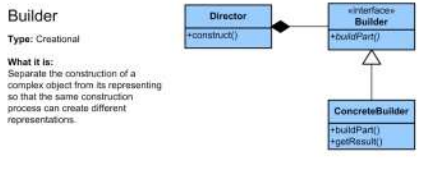
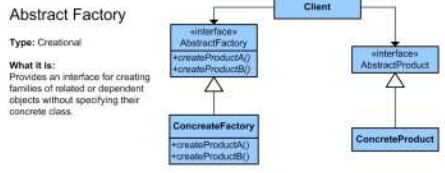
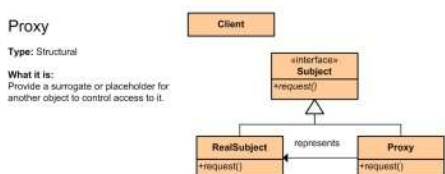
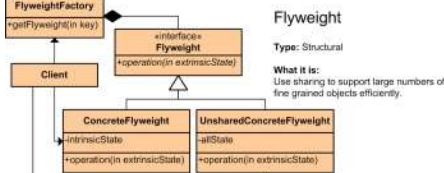
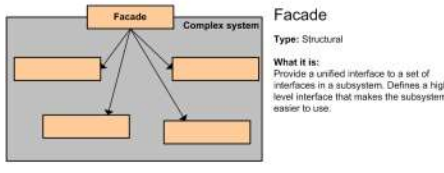
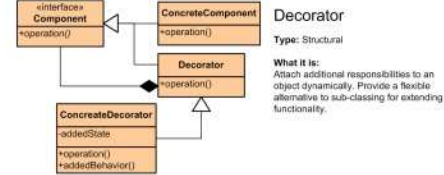
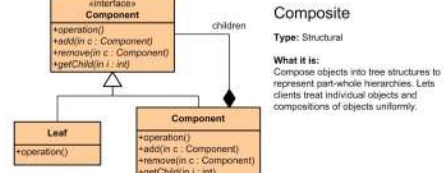
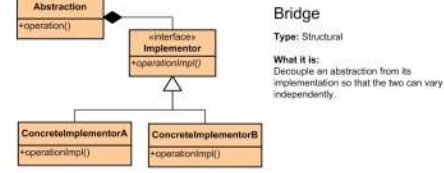
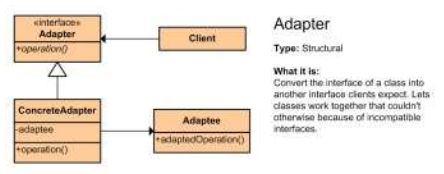
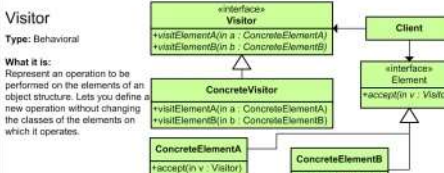
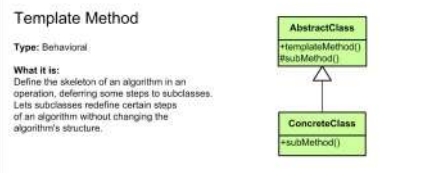
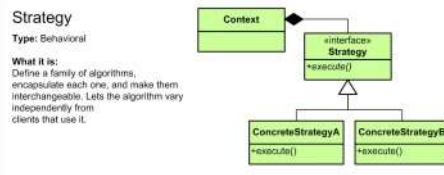
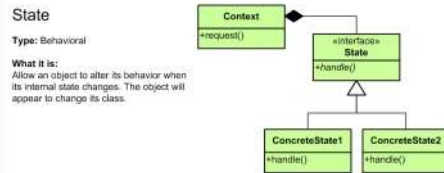
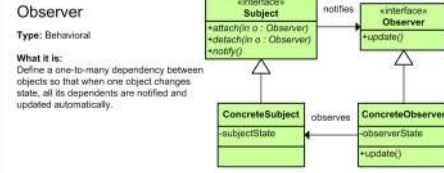
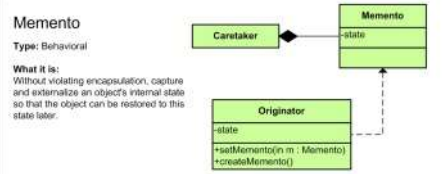
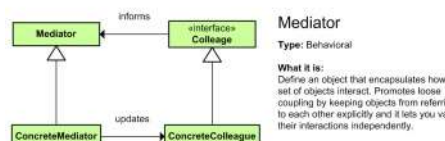
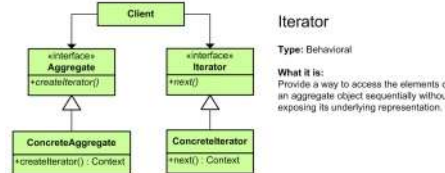
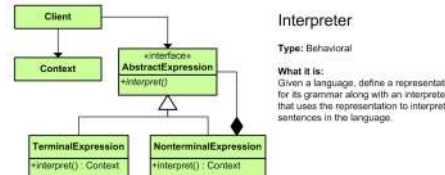
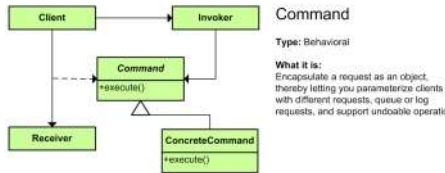
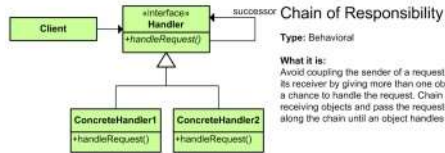
- Identificação de Objetos (tarefa difícil)
- Técnicas de Decomposição do Sistema em Objetos
- Identificação das Abstrações Menos Óbvias



Padrões de Projeto

		Propósito		
		Criação	Estrutura	Comportamento
Escopo	Classe	<i>Factory Method</i>	<i>Class Adapter</i>	<i>Interpreter</i> <i>Template Method</i>
	Objeto	<i>Abstract Method</i> <i>Builder</i> <i>Prototype</i> <i>Singleton</i>	<i>Object Adapter</i> <i>Bridge</i> <i>Composite</i> <i>Decorator</i> <i>Facade</i> <i>Flyweight</i> <i>Proxy</i>	<i>Chain of Responsibility</i> <i>Command</i> <i>Iterator</i> <i>Mediator</i> <i>Memento</i> <i>Observer</i> <i>State</i> <i>Strategy</i> <i>Visitor</i>

- C Abstract Factory
- B Adapter
- S Bridge
- C Builder
- B Chain of Responsibility
- B Command
- S Composite
- S Decorator
- S Facade
- C Factory Method
- S Flyweight
- B Interpreter
- B Iterator
- B Mediator
- B Memento
- C Prototype
- S Proxy
- B Observer
- C Singleton
- B State
- B Strategy
- B Template Method
- B Visitor



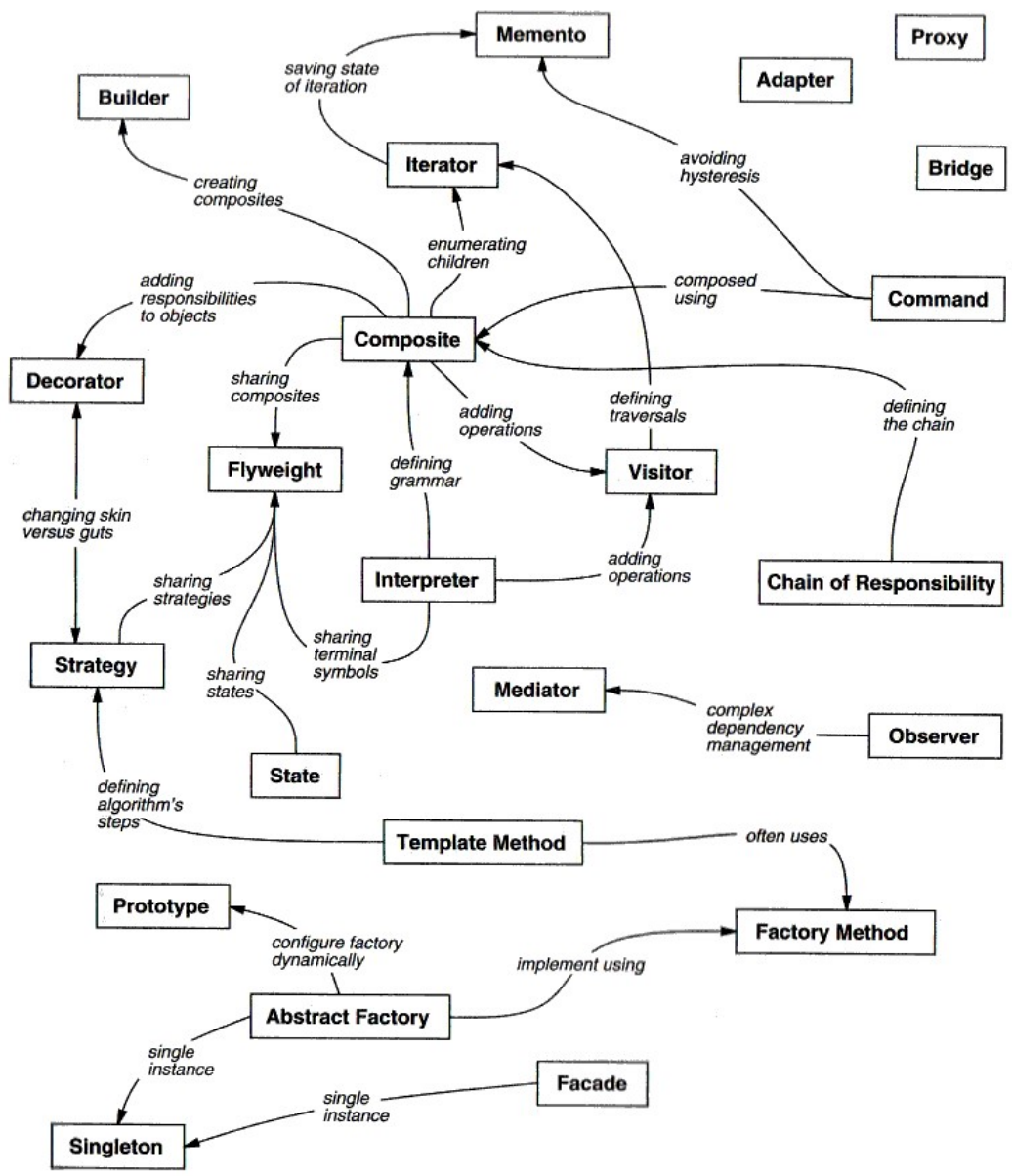


Figure 1.1: Design pattern relationships

