

# **Modelagem de Software Orientada a Objetos com UML**

CIV 2802 – Sistemas Gráficos para Engenharia

André Maués Brabo Pereira (UFF)  
Luiz Fernando Martha (PUC-Rio)

# Conteúdo

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

# 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

```
#ifndef STACK_H
#define STACK_H

class Stack
{
public:
    Stack();
    ~Stack();
    void push(double _n);
    double pop();
    bool isEmpty();
    void show();

private:
    int m_top;
    double *m_elems;
};

#endif
```

```
#ifndef REAL_H
#define REAL_H

class Real
{
public:
    Real(double _val);
    ~Real();
    Real sum(Real _n);
    Real sub(Real _n);
    Real mul(Real _n);
    Real div(Real _n);

private:
    double m_value;
};

#endif
```

# Orientação a Objetos

## Classe, Objeto e Encapsulamento

```
#ifndef STACK_H
#define STACK_H

class Stack
{
public:
    Stack();
    ~Stack();
    void push(double _n);
    double pop();
    bool isEmpty();
    void show();

private:
    int m_top;
    double *m_elems;
};

#endif
```

```
#ifndef STACK_H
#define STACK_H

#include "real.h"

class Stack
{
public:
    Stack();
    ~Stack();
    void push(Real _n);
    Real pop();
    bool isEmpty();
    void show();

private:
    int m_top;
    Real* m_elems;
};

#endif
```

```
#ifndef REAL_H
#define REAL_H

class Real
{
public:
    Real(double _val);
    ~Real();
    Real sum(Real _n);
    Real sub(Real _n);
    Real mul(Real _n);
    Real div(Real _n);

private:
    double m_value;
};

#endif
```

# 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 e Polimorfismo

```
#ifndef REAL_H
#define REAL_H

class Real
{
public:
    Real(double _val);
    ~Real();
    Real sum(Real _n);
    Real sub(Real _n);
    Real mul(Real _n);
    Real div(Real _n);

private:
    double m_value;
};

#endif
```

```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex
{
public:
    Complex(double _re,
            double _im);
    ~Complex();
    Complex sum(Complex _n);
    Complex sub(Complex _n);
    Complex mul(Complex _n);
    Complex div(Complex _n);

private:
    double m_real;
    double m_imag;
};

#endif
```

# Orientação a Objetos

## Herança e Polimorfismo

```
#ifndef REAL_H
#define REAL_H

class Real
{
public:
    Real(double _val);
    ~Real();
    Real sum(Real _n);
    Real sub(Real _n);
    Real mul(Real _n);
    Real div(Real _n);

private:
    double m_value;
};

#endif
```

```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex
{
public:
    Complex(double _re,
            double _im);
    ~Complex();
    Complex sum(Complex _n);
    Complex sub(Complex _n);
    Complex mul(Complex _n);
    Complex div(Complex _n);

private:
    double m_real;
    double m_imag;
};

#endif
```

```
#ifndef NUMBER_H
#define NUMBER_H

class Number
{
public:
    Number(int _type);
    ~Number();
    Number sum(Number _n);
    Number sub(Number _n);
    Number mul(Number _n);
    Number div(Number _n);

private:
    int m_type;
};

#endif
```

# Orientação a Objetos

## Herança e Polimorfismo

```
#ifndef REAL_H
#define REAL_H

#include "number.h"

class Real : Number
{
public:
    Real(double _val);
    ~Real();
    Number sum(Number _n);
    Number sub(Number _n);
    Number mul(Number _n);
    Number div(Number _n);

private:
    double m_value;
};

#endif
```

```
#ifndef COMPLEX_H
#define COMPLEX_H

#include "number.h"

class Complex : Number
{
public:
    Complex(double _re,
            double _im);
    ~Complex();
    Number sum(Number _n);
    Number sub(Number _n);
    Number mul(Number _n);
    Number div(Number _n);

private:
    double m_real;
    double m_imag;
};

#endif
```

```
#ifndef NUMBER_H
#define NUMBER_H

class Number
{
public:
    Number(int _type);
    ~Number();
    Number sum(Number _n);
    Number sub(Number _n);
    Number mul(Number _n);
    Number div(Number _n);

protected:
    int m_type;
};

#endif
```

# Orientação a Objetos

## Herança e Polimorfismo

```
#ifndef REAL_H
#define REAL_H

#include "number.h"

class Real : Number
{
public:
    Real(double _val);
    ~Real();
    Number sum(Number _n);
    Number sub(Number _n);
    Number mul(Number _n);
    Number div(Number _n);

private:
    double m_value;
};

#endif
```

```
#ifndef COMPLEX_H
#define COMPLEX_H

#include "number.h"

class Complex : Number
{
public:
    Complex(double _re,
            double _im);
    ~Complex();
    Number sum(Number _n);
    Number sub(Number _n);
    Number mul(Number _n);
    Number div(Number _n);

private:
    double m_real;
    double m_imag;
};

#endif
```

```
#ifndef NUMBER_H
#define NUMBER_H

class Number
{
public:
    Number(int _type);
    ~Number();
    virtual Number sum(Number _n) = 0;
    virtual Number sub(Number _n) = 0;
    virtual Number mul(Number _n) = 0;
    virtual Number div(Number _n) = 0;

protected:
    int m_type;
};

#endif
```

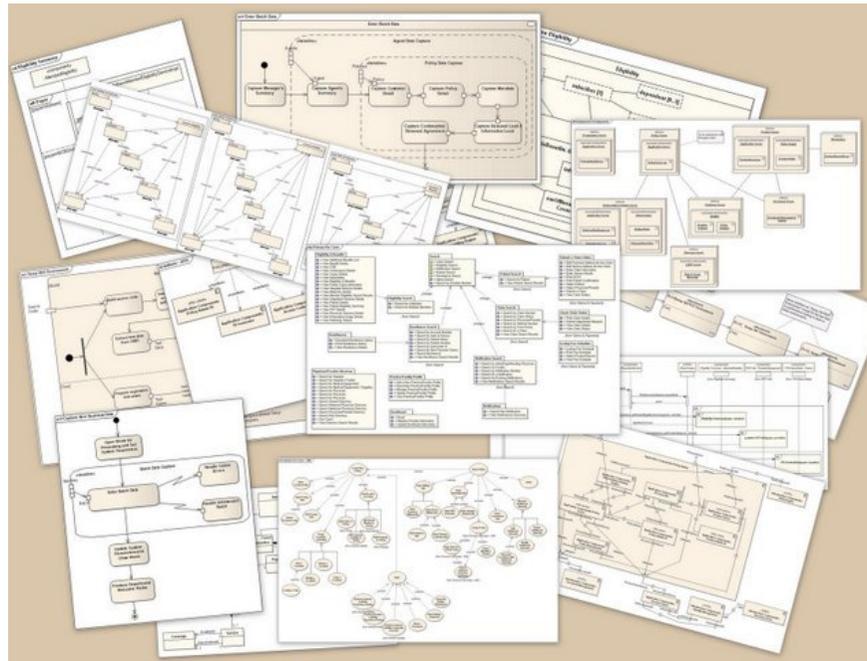
# UML



## 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

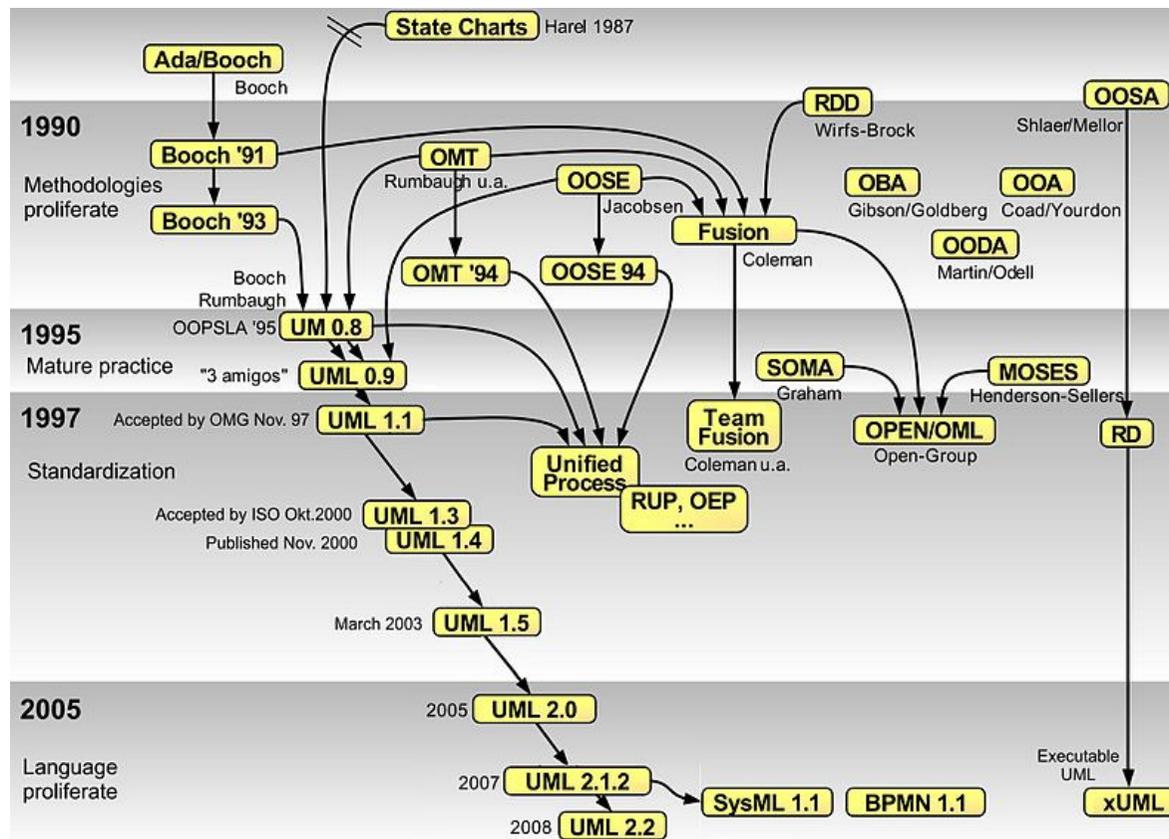
### **Vantagens:**

- 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.

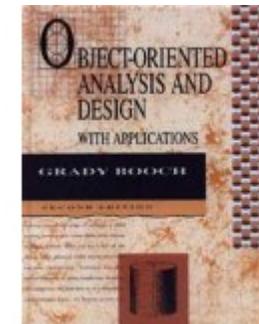


# UML

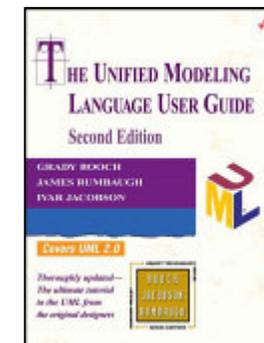
## Breve Histórico



1993

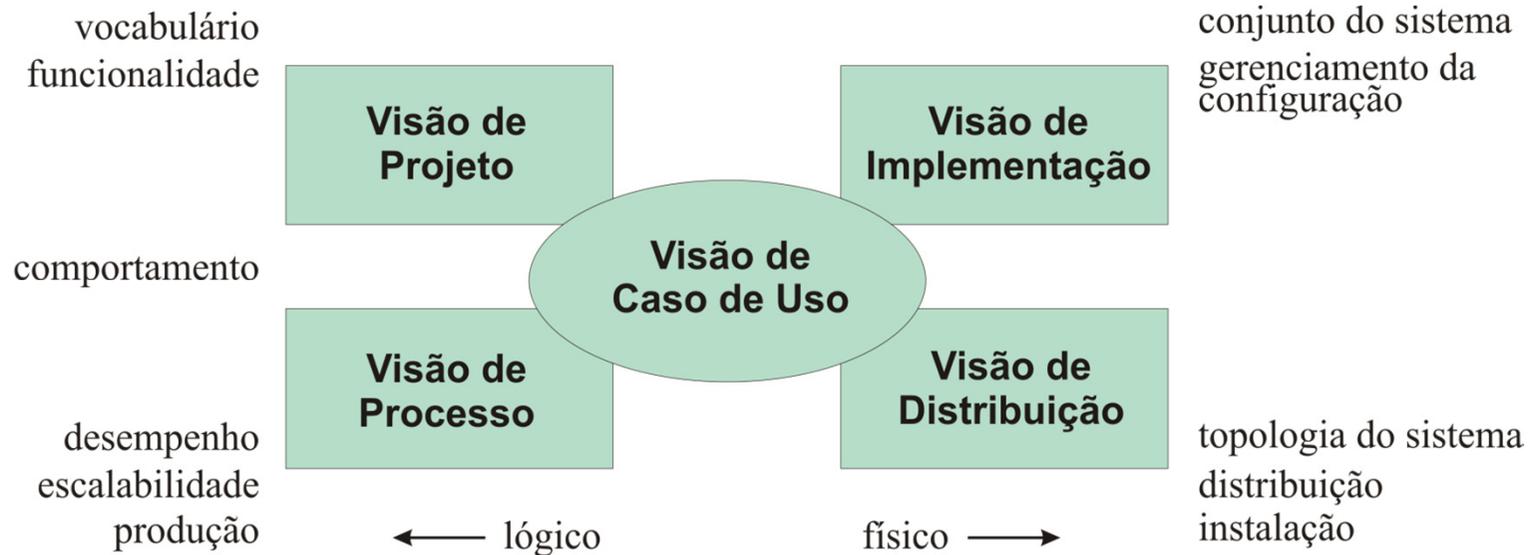


2005



# UML

## Visões (Arquitetura de um Sistema OO)

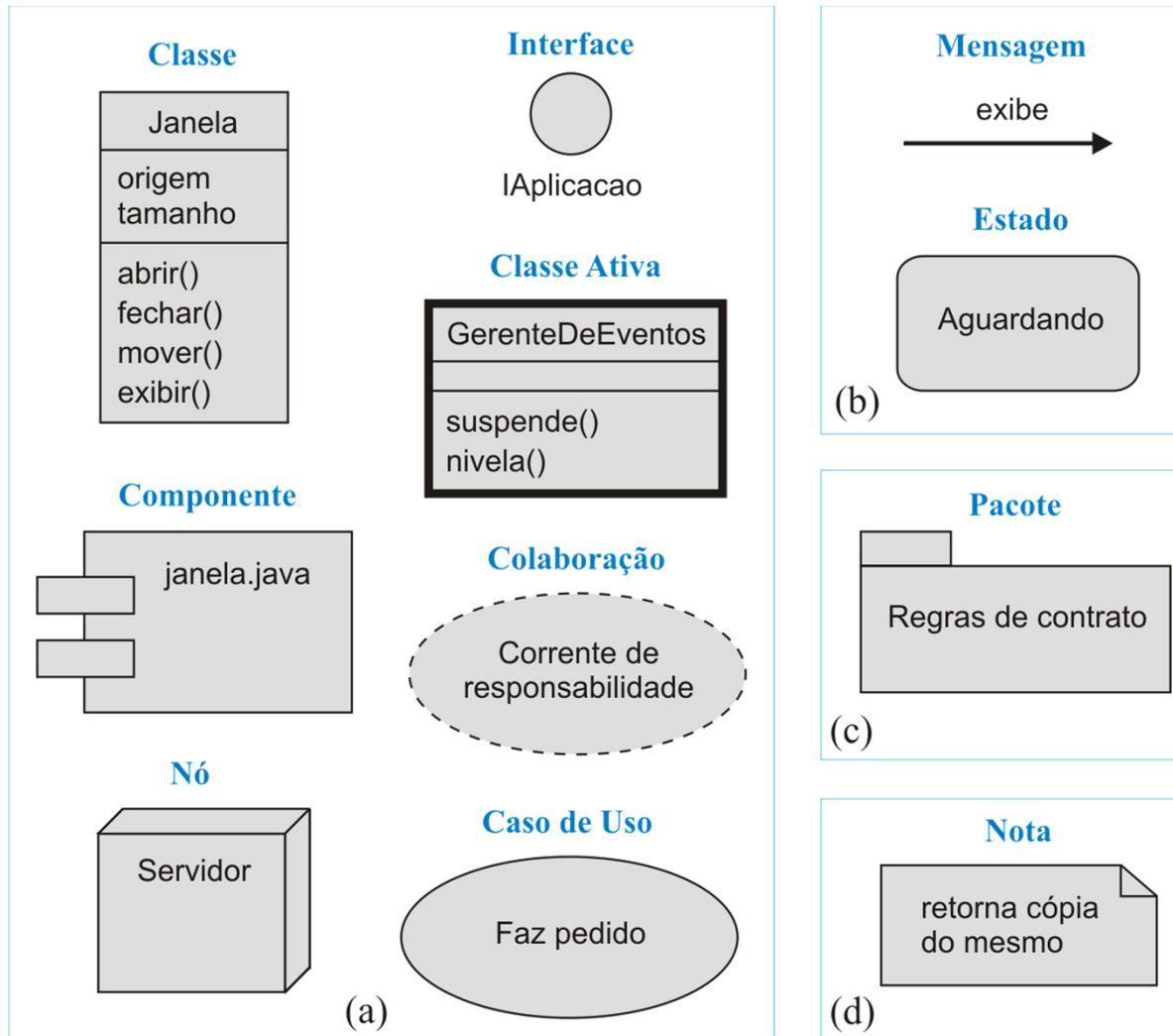


De acordo com a UML, deve-se ter uma **visão de casos de uso**, expondo as exigências do sistema; uma **visão de projeto**, capturando o vocabulário do espaço do problema e do espaço da solução; uma **visão do processo**, modelando a distribuição dos processos e linhas do sistema; uma **visão de implementação**,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

## Elementos em UML



## Blocos de Construção

# Elementos: Classes em UML

Como  
representar  
esta classe  
em UML?

```
#ifndef NUMBER_H
#define NUMBER_H

class Number
{
public:
    Number(int _type);
    ~Number();
    virtual Number sum(Number _n) = 0;
    virtual Number sub(Number _n) = 0;
    virtual Number mul(Number _n) = 0;
    virtual Number div(Number _n) = 0;

protected:
    int m_type;
};

#endif
```

# Blocos de Construção

## Elementos: Classes em UML

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

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

```
#ifndef NUMBER_H
#define NUMBER_H

class Number
{
public:
    Number(int _type);
    ~Number();
    virtual Number sum(Number _n) = 0;
    virtual Number sub(Number _n) = 0;
    virtual Number mul(Number _n) = 0;
    virtual Number div(Number _n) = 0;

protected:
    int m_type;
};

#endif
```

## 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:double - m_imag:double
+ sum(_n:Number) : Number + sub(_n:Number) : Number + mul(_n:Number) : Number + div(_n:Number) : Number

```
#ifndef NUMBER_H
#define NUMBER_H

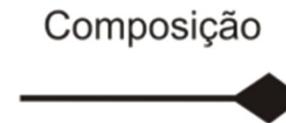
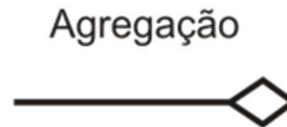
class Number
{
public:
    Number(int _type);
    ~Number();
    virtual Number sum(Number _n) = 0;
    virtual Number sub(Number _n) = 0;
    virtual Number mul(Number _n) = 0;
    virtual Number div(Number _n) = 0;

protected:
    int m_type;
};

#endif
```

# Blocos de Construção

## Relações em UML



**Dependência**



**Generalização**



**Realização**



## Blocos de Construção

# Relações: entre Classes UML

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

<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?

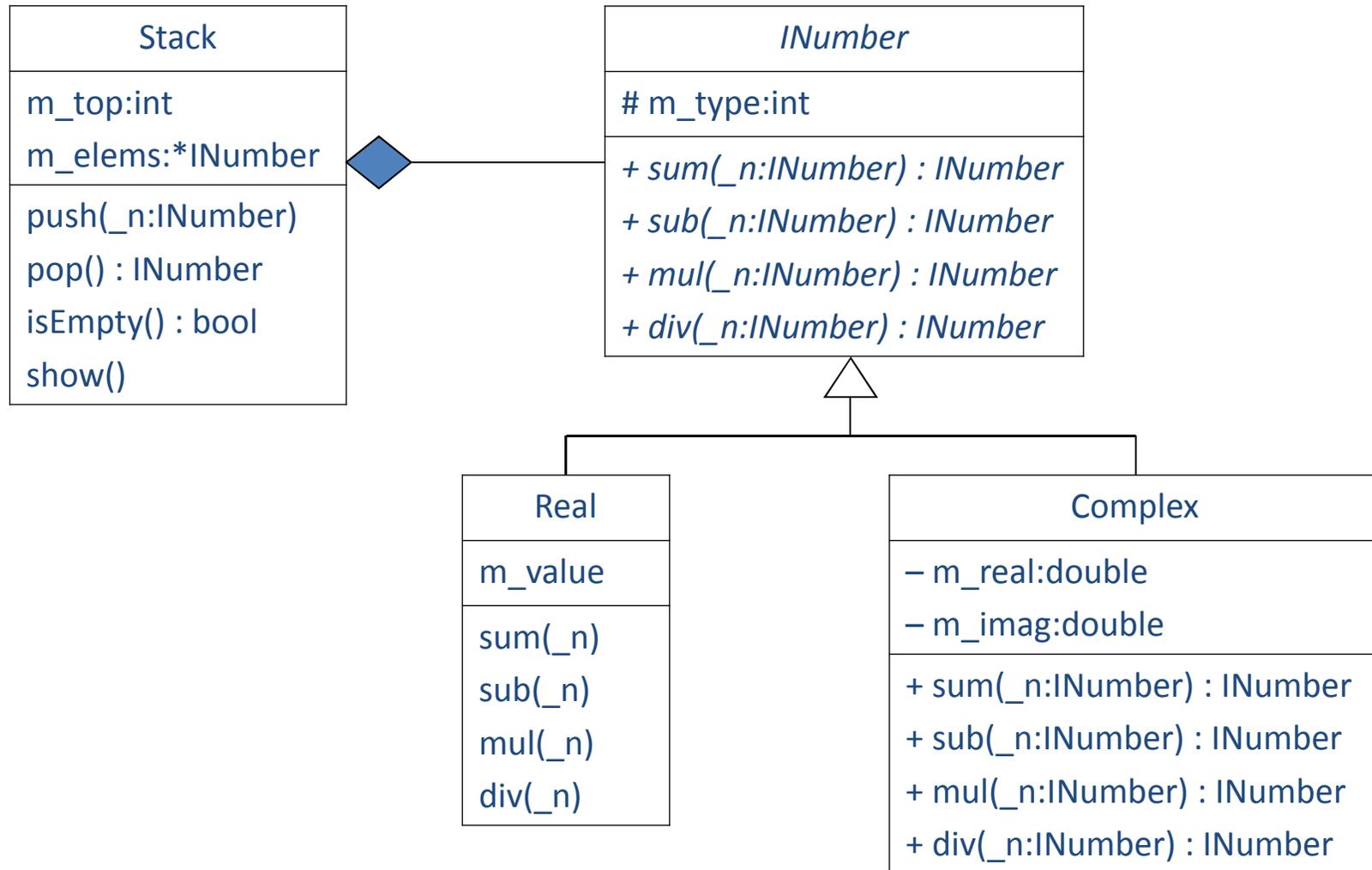
Como representar estas  
relações em UML?

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

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

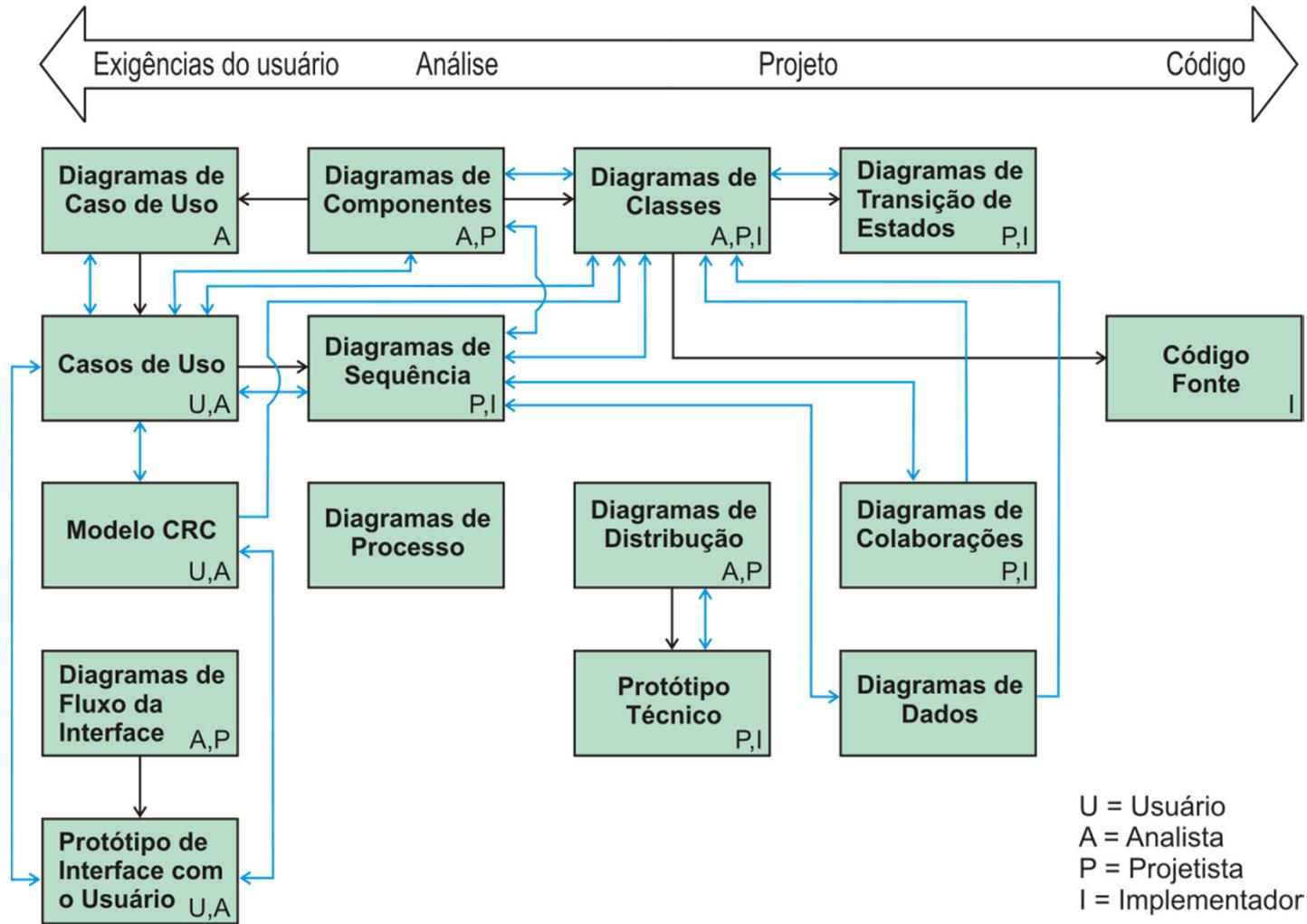
## Blocos de Construção

# Relações: entre Classes UML



# Blocos de Construção

## Diagramas em UML



CRC: classe, responsabilidade e colaboração

# Modelagem Orientada a Objetos

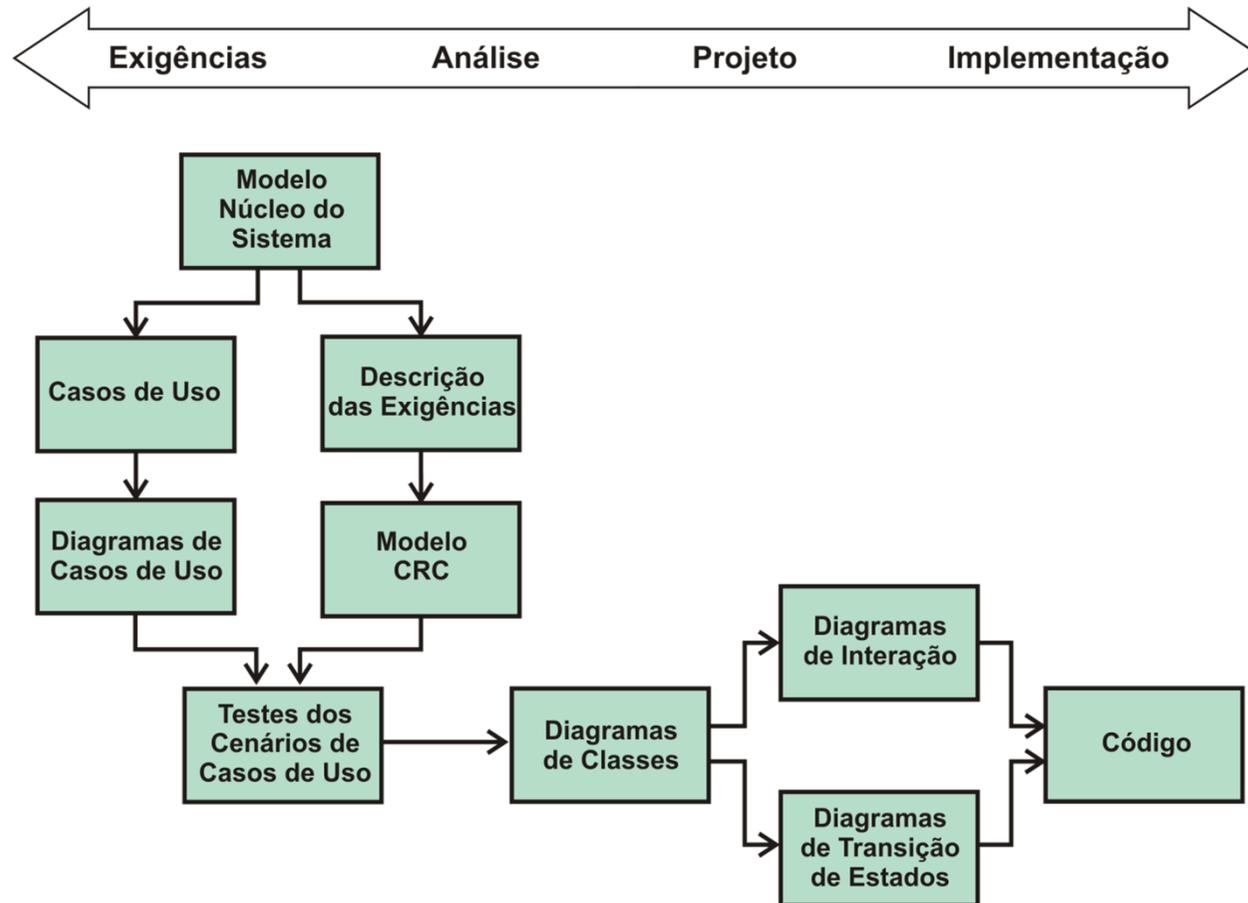
Uma metodologia é um processo organizado de produção de software, que utiliza técnicas predefinidas e notações convencionais.

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

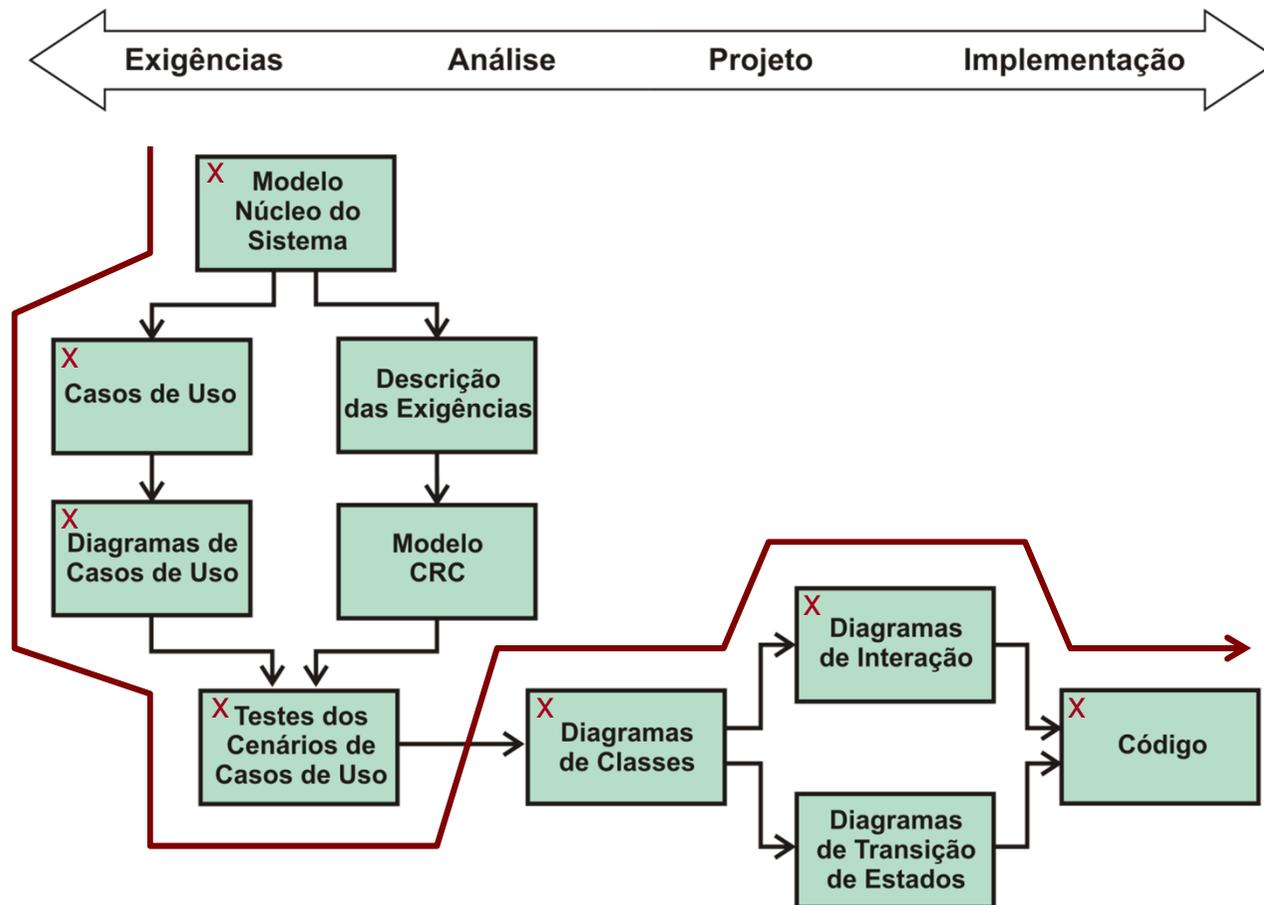
Tradicionalmente, a formulação inicial do problema, a análise, o projeto, a implementação, os testes e a operação (manutenção e aperfeiçoamento) compõem estas etapas do ciclo de vida.

“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* 

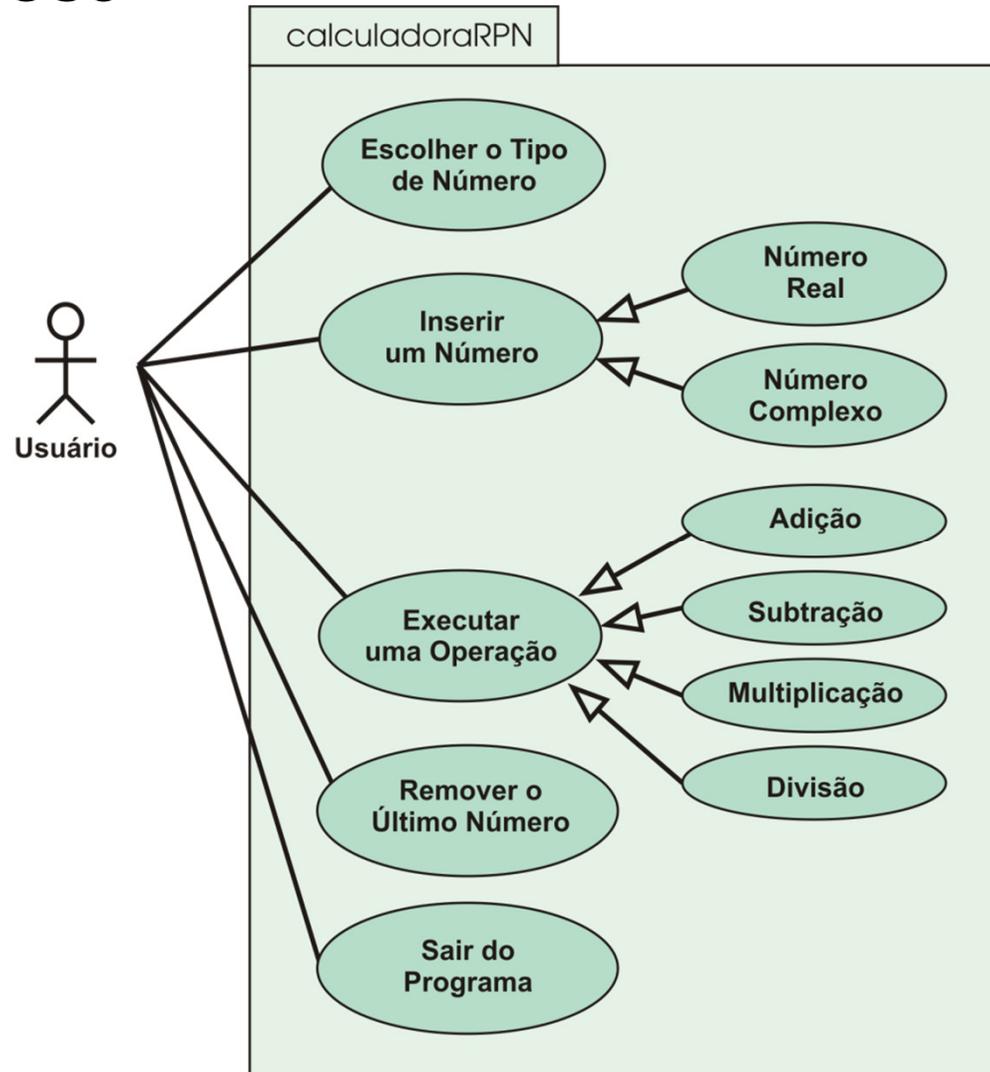
- apagar o último número inserido 

- 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:

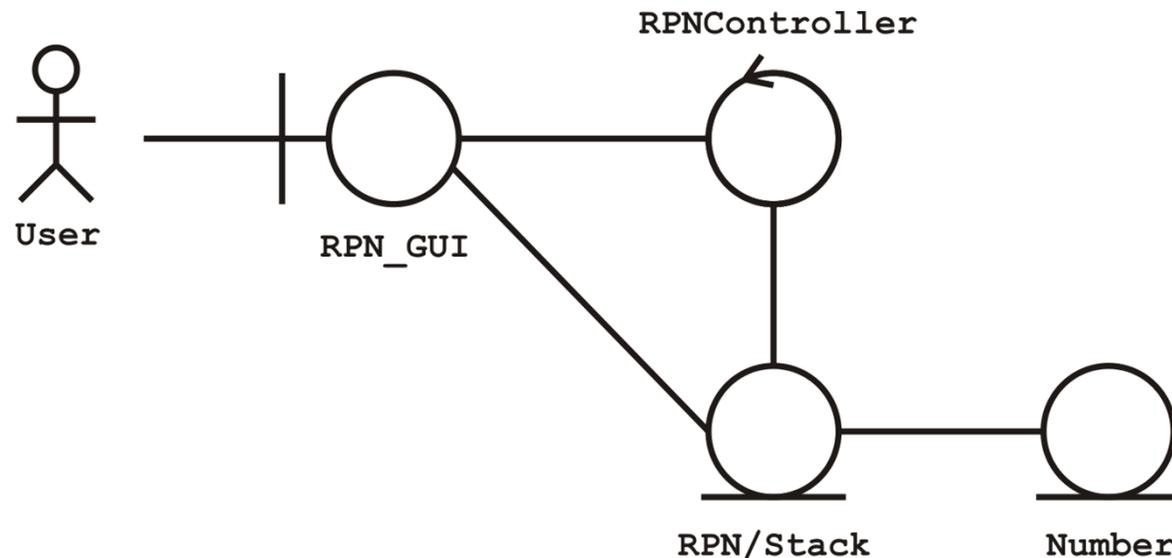
- Um objeto ou entidade única para representar a calculadora (RPN).
- Uma quantidade arbitrária de objetos, cada representando um determinado número (Number) Esse número ainda pode ser: inteiro (Integer), real (Real) ou complexo (Complex).
- Uma estrutura de dados especial para armazenar os números, 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**

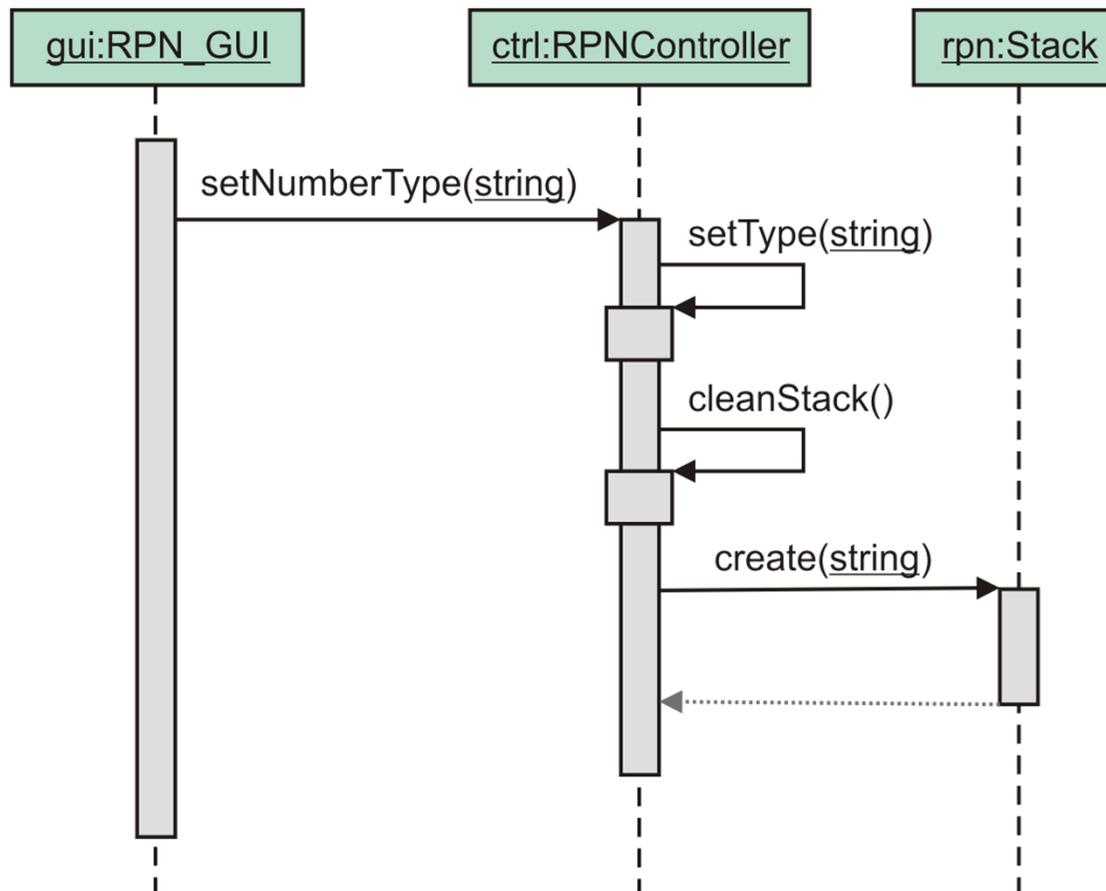
## da Calculadora RPN

### **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:

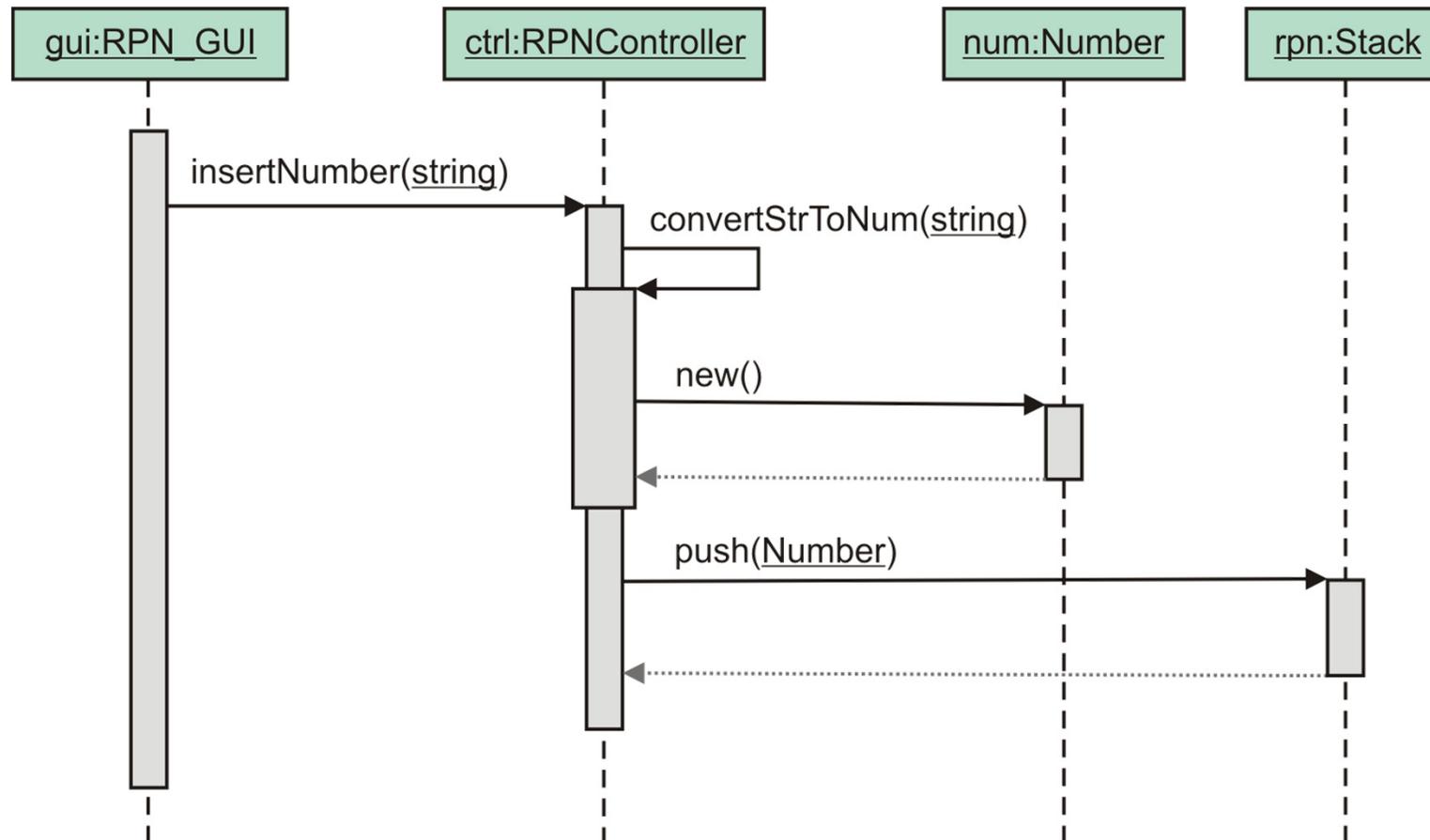
# Projeto Orientado a Objetos da Calculadora RPN

## Diagramas de Sequência



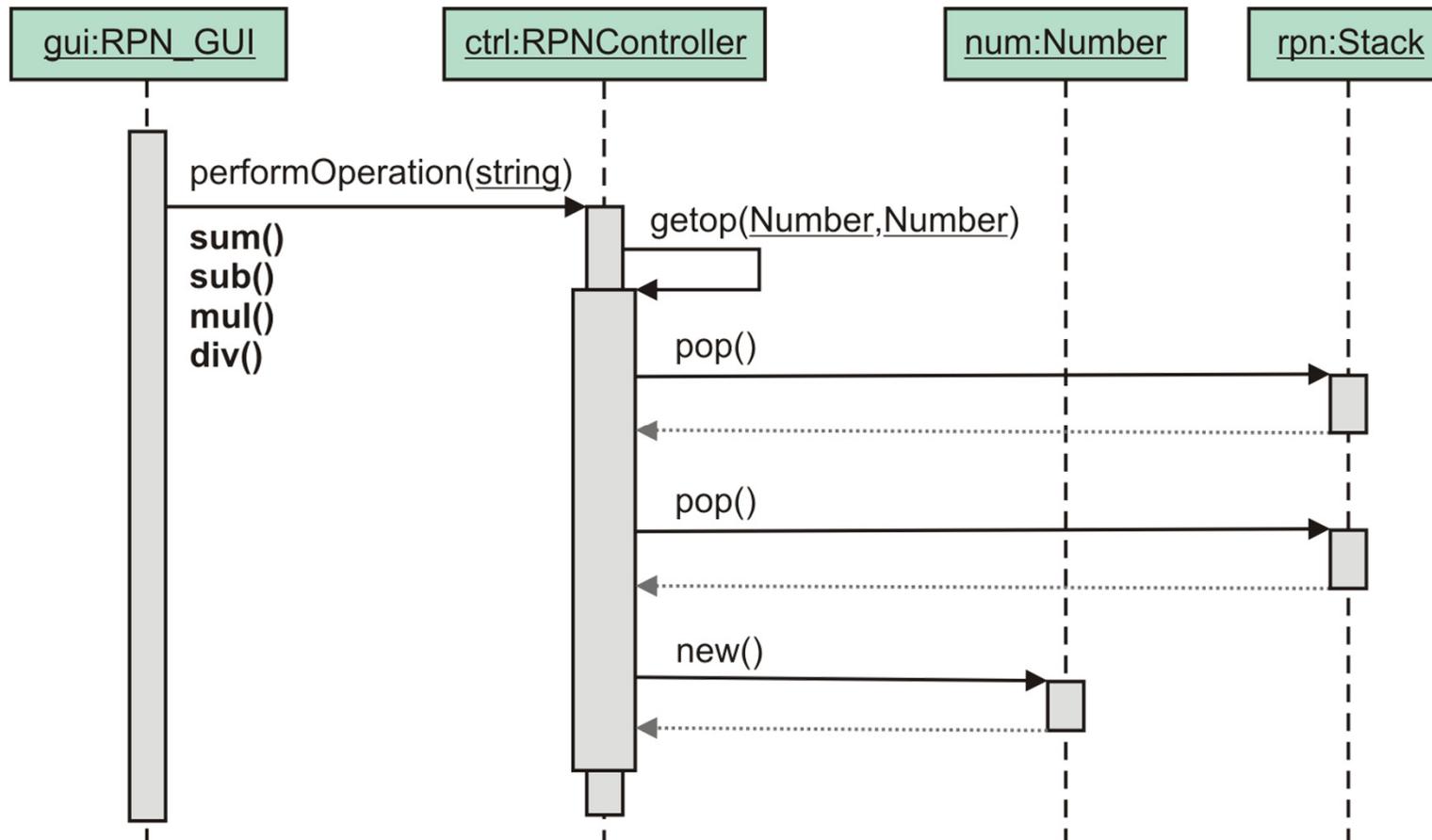
# Projeto Orientado a Objetos da Calculadora RPN

## Diagramas de Sequência



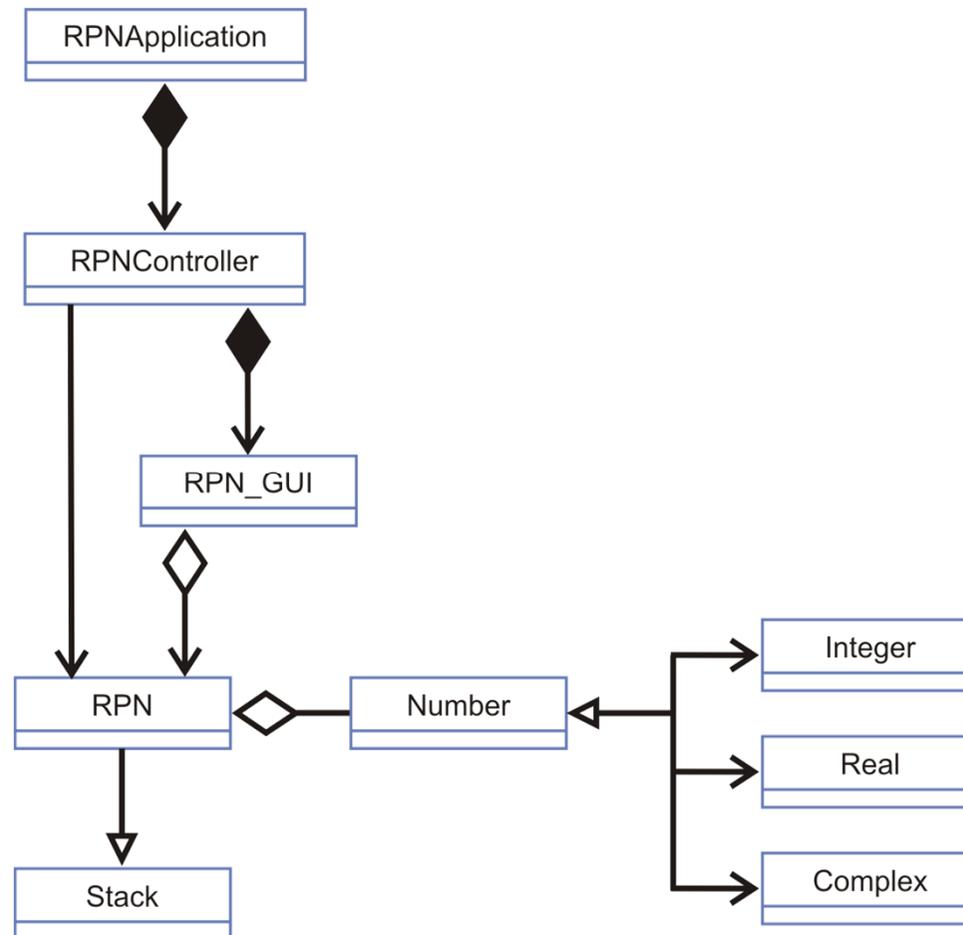
# Projeto Orientado a Objetos da Calculadora RPN

## Diagramas de Sequência



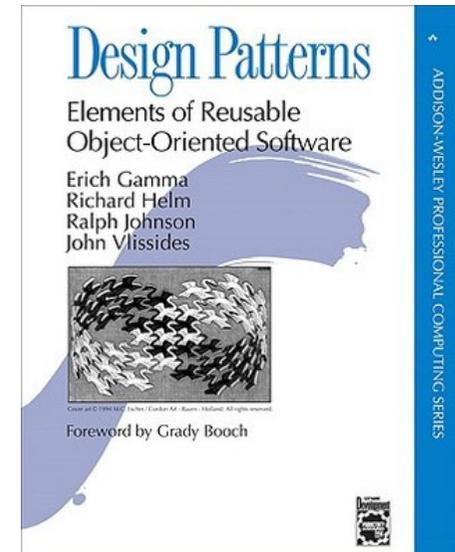
# Projeto Orientado a Objetos da Calculadora RPN

## Diagrama de Classe



# Padrões de Projeto

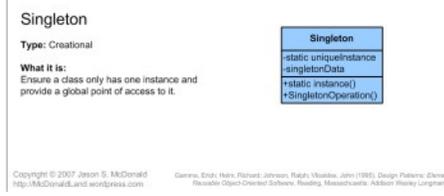
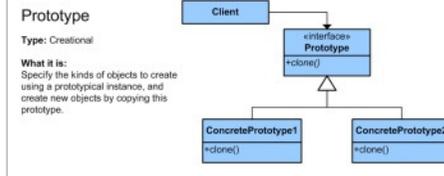
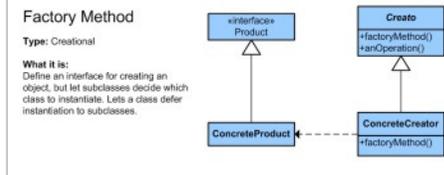
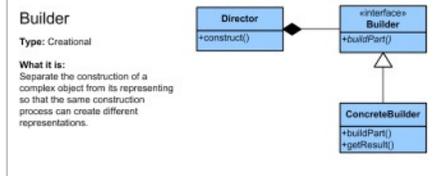
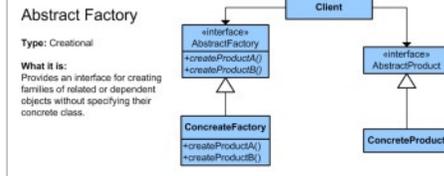
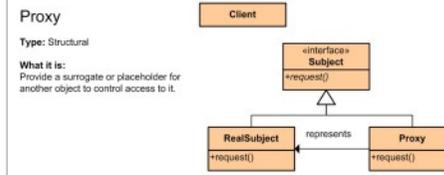
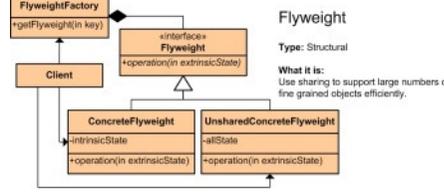
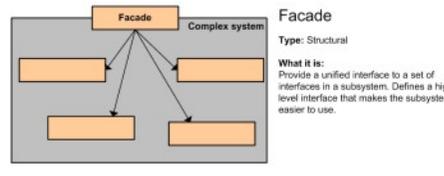
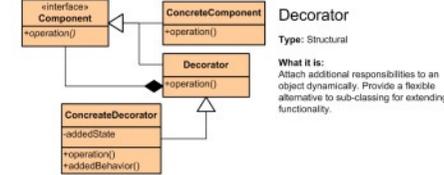
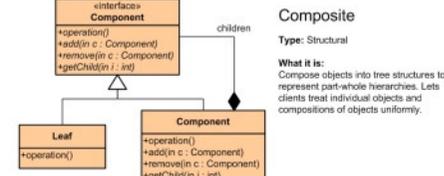
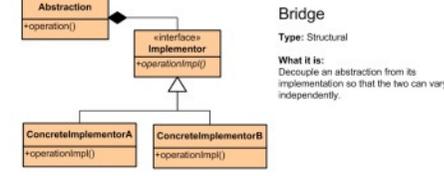
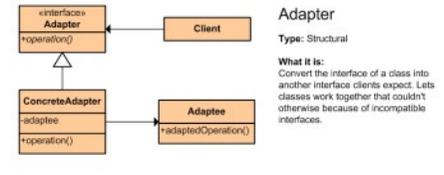
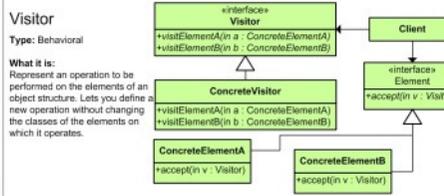
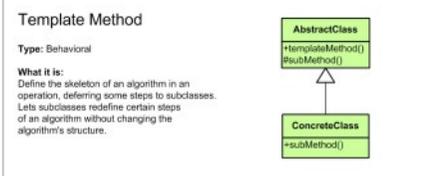
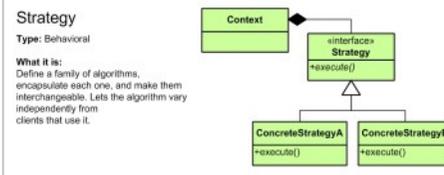
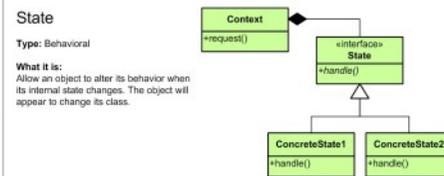
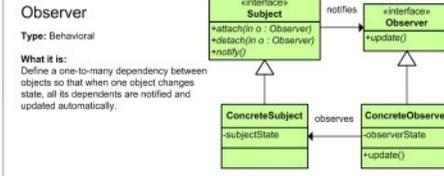
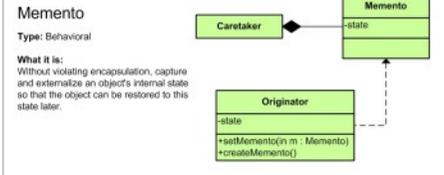
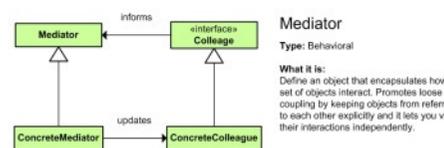
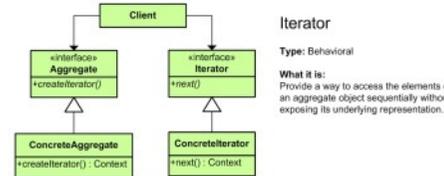
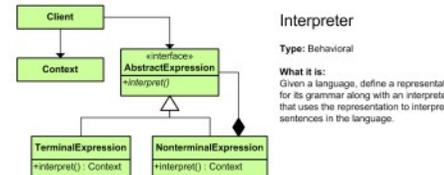
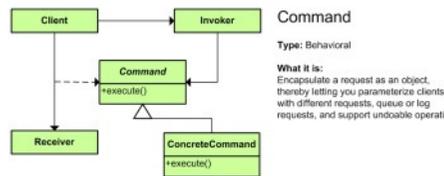
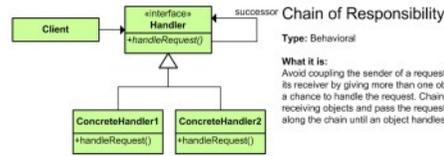
- Identificação de Objetos (tarefa difícil)
- Técnicas de Decomposição do Sistema em Objetos
- Identificação das Abstrações Menos Óbvias



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



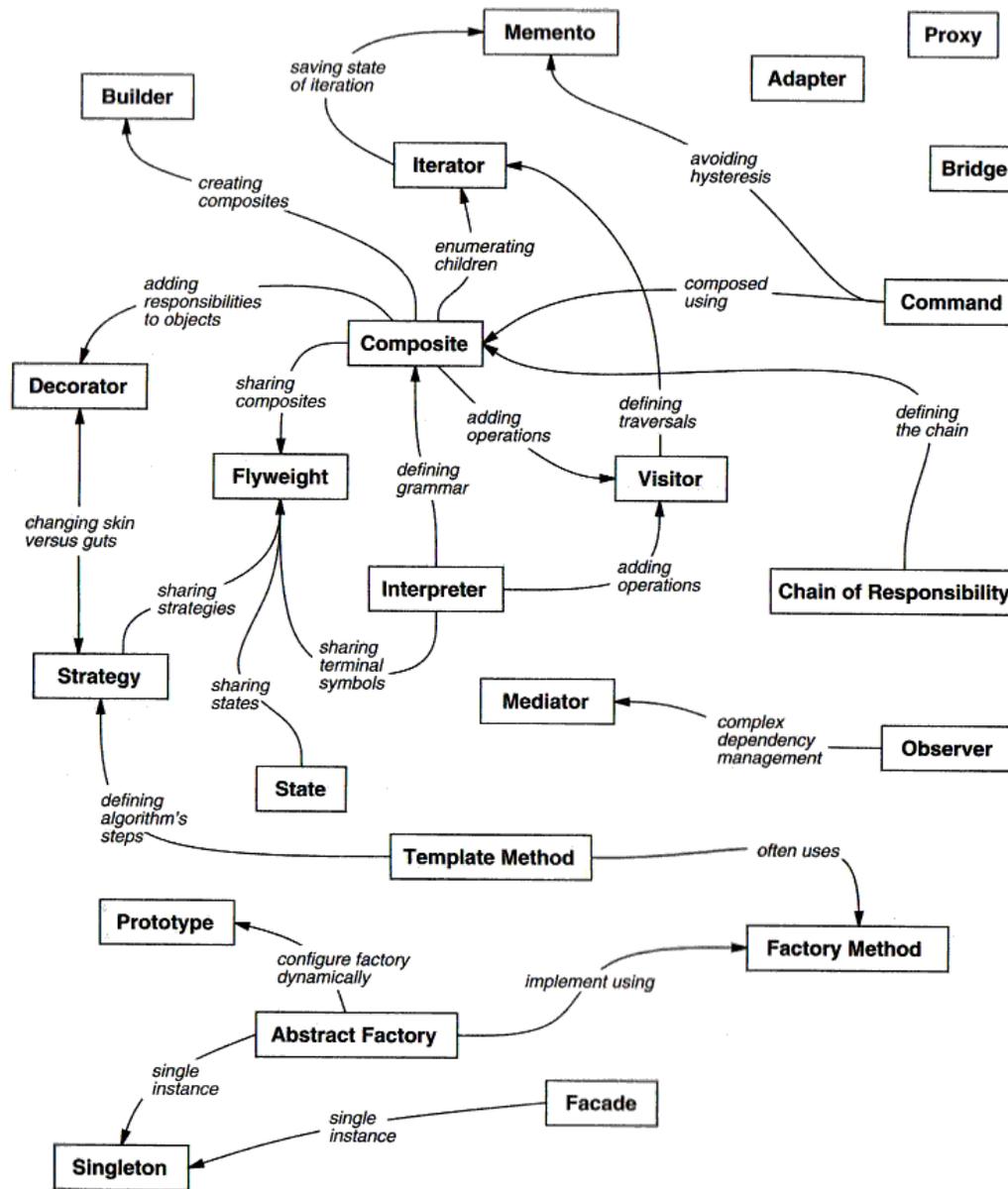


Figure 1.1: Design pattern relationships

