

Introdução à Orientação a Objetos

Orientação a Objetos - DCC025

Prof. Edmar Welington Oliveira
edmar.oliveira@ufjf.edu.br

Universidade Federal de Juiz de Fora - UFJF
Departamento de Ciência da Computação - DCC

Desenvolvimento de Sistemas

- Aumento da complexidade de sistemas
 - Exigência: necessidade de desenvolvimento organizado e estruturado
 - Sistemas computacionais mal estruturados
 - Difícil manutenção
 - Integração inviável
 - Incorporação de novas funções, complexas
 - Reuso praticamente inexistente
 - ...

Necessidade da OO

- Justificativa para uso de OO
 - Minimizar os problemas na criação de softwares complexos
 - Grande problema
 - Não existência de encapsulamento lógico para operações e dados
 - Consequência: falta da **divisão de tarefas por responsabilidades**.
 - Qual o problema disto?
 - Construção de longos trechos de código, muitas vezes difíceis de compreender devido ao acúmulo de responsabilidade que lhe é atribuído - **problema de Coesão**.

Necessidade da OO

- Conclusão

- Efeito castata

- Quanto mais complexo o software
 - Mais difícil se torna sua manutenção
 - Aumenta-se os custos
 - Aumenta-se os riscos de confiabilidade do mesmo.

Objetivo Básico da OO

- Meta da OO
 - Identificar o melhor **conjunto de objetos** para descrever um sistema de software. O funcionamento deste sistema se dá através do relacionamento e troca de mensagens entre estes objetos.
 - Tentar aproximar mundo real de mundo virtual
 - Noção intuitiva: sistema modelado com base no mundo real
 - Objetos e relacionamentos entre objetos
 - Não em funções de sistema

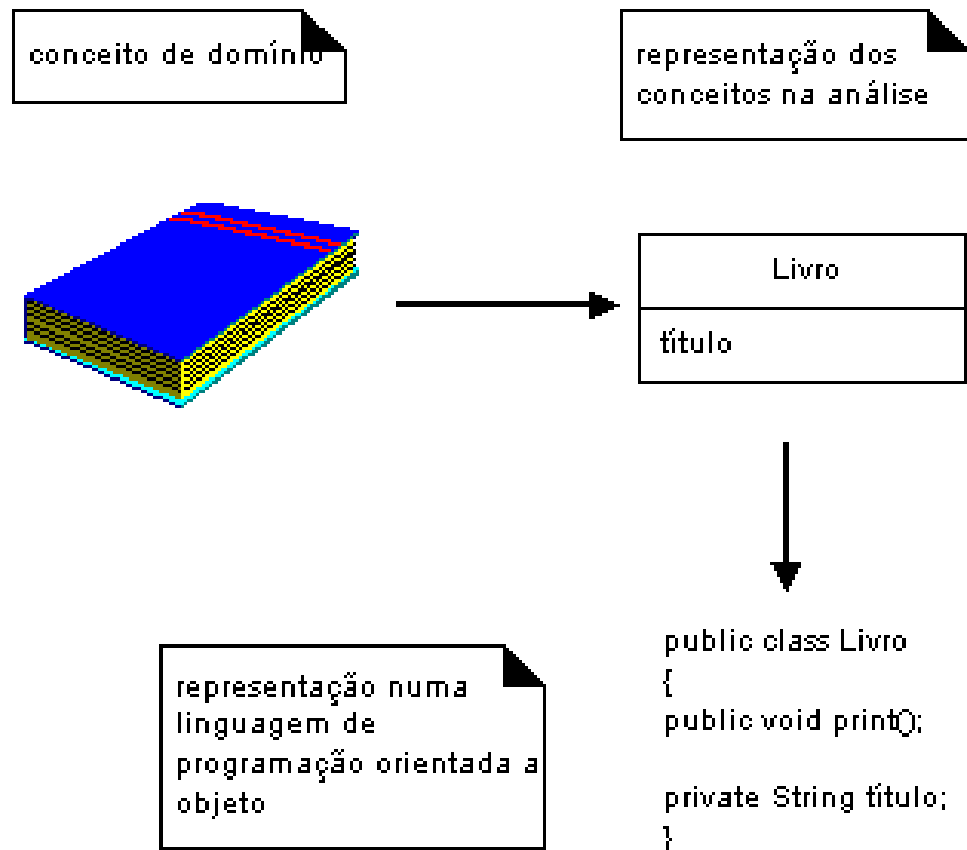
Surgimento da OO

- Qualidade de Software
 - Importância dos conceitos
 - Abstração
 - Modularidade
 - Proteção de informações

Vantagens da OO

- Exemplo
 - Manutenção e reuso de código mais fácil
- Manutenção
 - Na POO, existem certas características (herança, encapsulamento, etc.) que permitem que, quando necessária alguma alteração, modifique-se apenas o objeto que necessita desta alteração, e ela irá se propagar automaticamente as demais partes do software que utilizam este objeto.
- Reuso de Código
 - Uso de objetos desenvolvidos para um sistema em outros sistemas

Desenvolvimento OO



Desenvolvimento OO

- Resultado do uso de OO
 - Produtos de software mais estáveis e de melhor qualidade
 - Processo de desenvolvimento que permite
 - Melhor entendimento do sistema
 - Melhor entendimento do domínio da aplicação
 - Independência da implementação até estágios mais avançados

Conceituação de OO

■ Orientação a Objetos

- Abordagem de desenvolvimento que procura explorar o lado intuitivo dos desenvolvedores. Os objetos da computação são análogos aos objetos do mundo real.
- As estruturas centrais do processo são os Objetos
- Objetos trocam mensagens entre si
- Mensagens ativam métodos
- Métodos realizam as ações necessárias

Programação OO

- Programação Orientada a Objetos

- É um modelo de programação que baseia-se em conceitos como **classes**, **objetos**, **herança**, etc. Seu objetivo é a resolução de problemas baseada na identificação de objetos e o processamento requerido por estes objetos, e então na criação de simulações destes objetos.

Paradigma Estruturado

Paradigma Estruturado

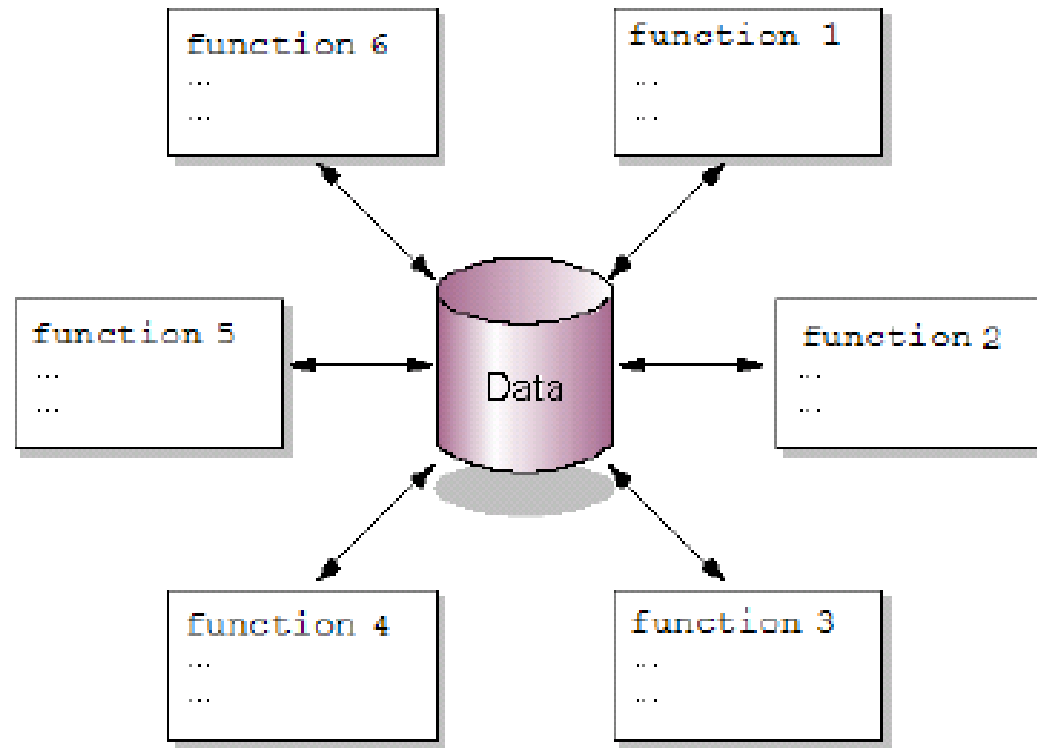
■ Análise Estruturada

- Foco principal nas funções e dados
- Informações desagrupadas
- Idéias e necessidades dos usuários não ficam claras
- Horas infindáveis de manutenção corretiva

■ Vantagens

- Simplicidade de termos
- Facilidades de aprendizagem
 - Razão por ser utilizada nas primeiras aulas de programação

Paradigma Estruturado



Paradigma Estruturado

- Programação Estruturada

- Programa

- Blocos elementares de código
 - Blocos se interligam através de três mecanismos básicos

- Mecanismos Básicos

- Seqüência, seleção e iteração.
 - Todas têm ponto de início e um ponto de término

Paradigma Estruturado

- Programa Estruturado

- Possui um único ponto de entrada e só um de saída
- Existem de "1 a n" caminhos desde o princípio até o fim do programa
- Todas as instruções são executáveis sem que apareçam loops infinitos.

- Resumindo

- Um programa tem um fluxo linear, seguindo por uma função principal (ou o corpo principal do programa) que invoca funções auxiliares para executar certas tarefas à medida que for necessário.

Codificação

Programa em C - Versão Inicial

```
#include <stdio.h>

int main()
{
    int combinacoes = 1;
    int i;

    for(i=1; i<=4; i++)
    {
        combinacoes = combinacoes * 2;
        printf("\n%d bits = %d combinacoes", i, combinacoes);
    }

    return 0;
}
```

1ª Modularização

```
#include <stdio.h>
int combinacoes;

void inicializa()
{
    combinacoes = 1;
}

int retornaCombinacao()
{
    combinacoes = combinacoes * 2;
    return combinacoes;
}

int main()
{
    int i;
    inicializa();

    for(i=1; i<=4; i++)
    {
        printf("\n%d bits = %d combinacoes", i, retornaCombinacao());
    }

    return 0;
}
```

Programa com modularização básica - apenas funções

Vantagens:

Módulos encapsulam a lógica do cálculo de combinações - possibilidade de reuso

1ª Modularização

■ Problemas

- Módulos dependem do programa principal
 - Ele mantém a variável “combinacoes”
- Isso é Péssimo
 - Programa principal se torna responsável por detalhes de implementação dos módulos, o que prejudica o reuso. O reuso é prejudicado pois toda vez que um programa quiser reusar os módulos, ele precisará declarar a variável “combinacoes”
 - Pode haver conflito de variáveis. A nova variável combinacoes poderá entrar em conflito com uma já existente. Isso irá forçar mudança de código

2ª Modularização

```
#include <stdio.h>

void inicializa()
{
    int combinacoes;
    combinacoes = 1;
}

int retornaCombinacao()
{
    int combinacoes;
    combinacoes = combinacoes * 2;
    return combinacoes;
}

int main()
{
    int i;
    inicializa();

    for(i=1; i<=4; i++)
    {
        printf("\n%d bits = %d combinacoes", i, retornaCombinacao());
    }

    return 0;
}
```

Tentativa de transmitir a variável
combinações para os módulos. Objetivo:
remover dependência

2ª Modularização

- Problema
 - A variável é criada e destruída a cada entrada/saída de cada um dos módulos
 - Impossibilita a continuidade desejada
 - Resultado incorreto no final do programa

3ª Modularização

```
#include <stdio.h>
int combinacoes;

void inicializa(int *combinacoes)
{
    *combinacoes = 1;
}

int retornaCombinacao(int *combinacoes)
{
    *combinacoes = *combinacoes * 2;
    return *combinacoes;
}

int main()
{
    int i;
    inicializa(&combinacoes);

    for(i=1; i<=4; i++)
    {
        printf("\n%d bits = %d combinacoes", i, retornaCombinacao(&combinacoes));
    }

    return 0;
}
```

Variável global é declarada e passada como parâmetro para os módulos

Vantagens:

A variável do programa principal se torna independente da variável dos módulos (os nomes podem ser diferentes)

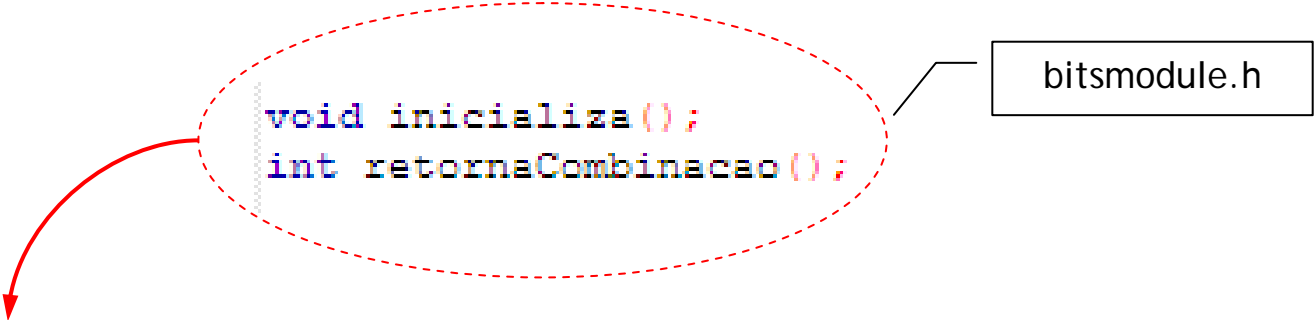
3ª Modularização

■ Problemas

- O programa principal continua precisando declarar e manter a variável “combinacoes”, o que ainda causa dependência dos módulos. Considerando as possibilidades de modularização, não há mais nada que se pode fazer. Consegue resolver algumas coisas, mas outros problemas continuam persistindo.

4ª Modularização

```
void inicializa();  
int retornaCombinacao();
```



bitsmodule.h

```
#include <bitsmodule.h>
```

```
static int combinacoes;
```

```
void inicializa()
```

```
{  
    combinacoes = 1;  
}
```

```
int retornaCombinacao()
```

```
{  
    combinacoes = combinacoes * 2;  
    return combinacoes;  
}
```

bitsmodule.c

4ª Modularização

```
#include <bitsmodule.h>
```

```
int main()
```

```
{
```

```
    int i;
```

```
    inicializa();
```

```
    for(i=1; i<=4; i++)
```

```
    {
```

```
        printf("\n%d bits = %d combinacoes",i,retornaCombinacao());
```

```
    }
```

```
    return 0;
```

```
}
```

bits.c

4ª Modularização

- Vantagens

- O módulo expõe apenas as interfaces das funções
 - Esconde-se detalhes de implementação
- O módulo controla e mantém o estado da variável combinações
 - A variável não é visível para o programa principal

4ª Modularização

- Problemas

- O módulo funciona apenas para uma instância
 - Problema em caso de dois cálculos de combinações em paralelo
 - Uso de múltiplas instâncias é possível, mas complicado

Java - Uso de Classes

- O que acontece
 - O módulo é transformado em uma **classe**

Classe BitsClass

```
public class BitsClass {  
  
    private int combinacoes;  
  
    public void inicializa() {  
        combinacoes = 1;  
    }  
  
    public int retornaCombinacao() {  
        combinacoes = combinacoes * 2;  
        return combinacoes;  
    }  
}
```

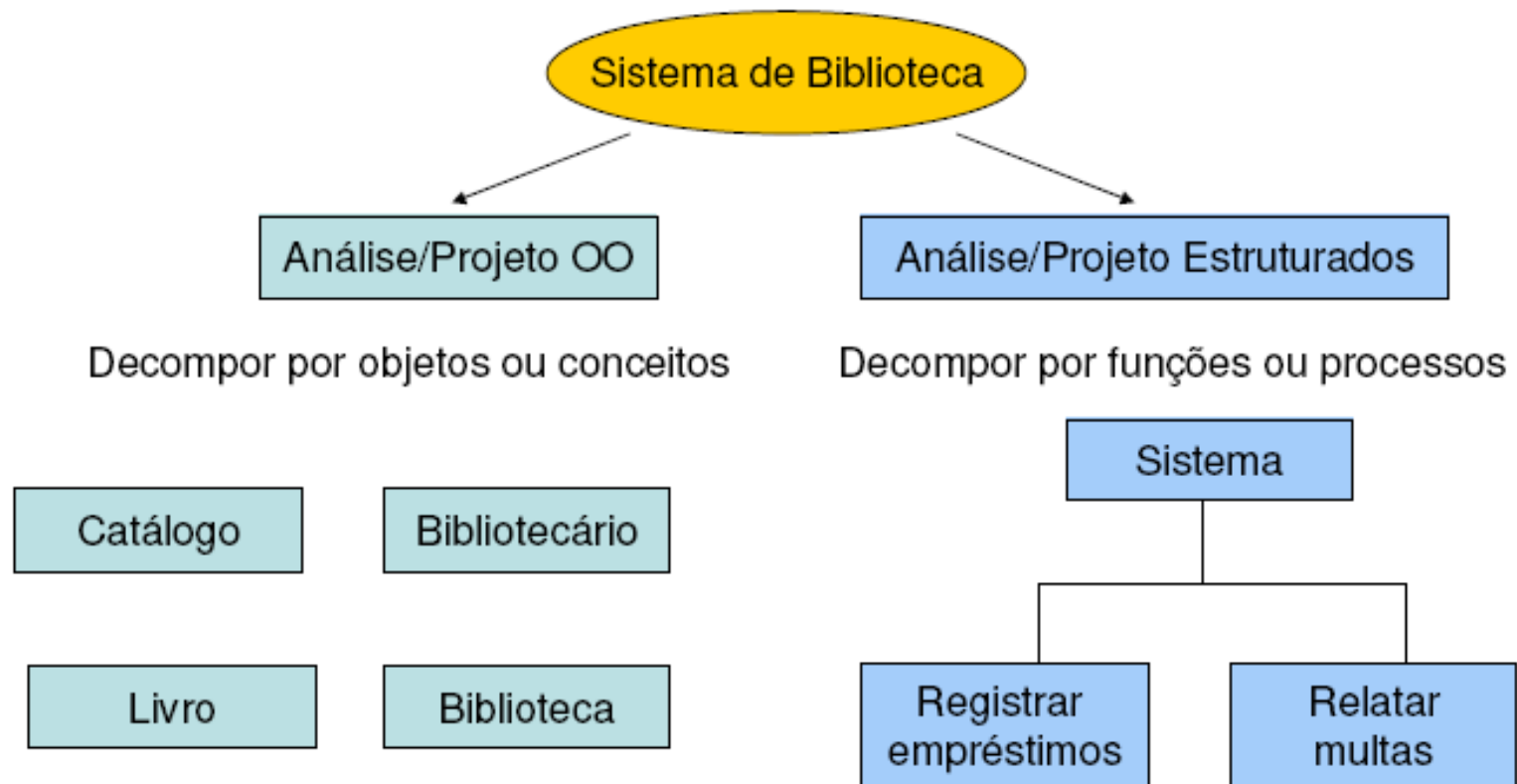
Java

Classe Bits

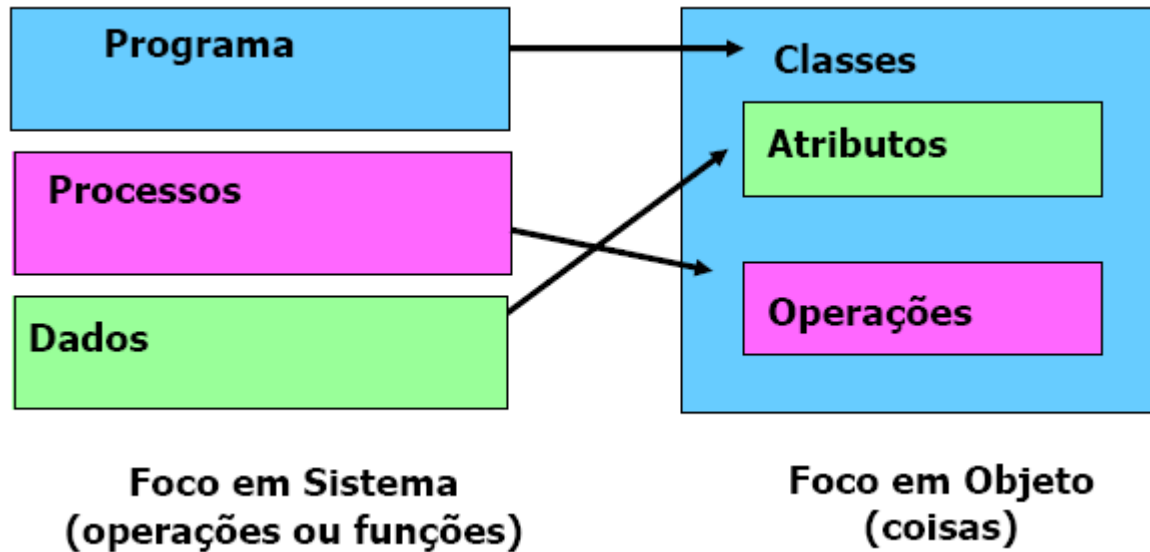
```
2 public class Bits {  
3  
4     public static void main (String args[]){  
5  
6         BitsClass bc = new BitsClass();  
7         bc.inicializa();  
8  
9         for(int i=1; i<=5; i++){  
10             System.out.println(i + "Bits = " + bc.retornaCombinacao() + "combinacoes");  
11         }  
12     }  
13 }
```

Estruturado vs OO

Desenvolvimento OO e Estruturado



Desenvolvimento OO e Estruturado



Introdução à Orientação a Objetos

Orientação a Objetos - DCC025

Prof. Edmar Welington Oliveira
edmar.oliveira@ufjf.edu.br

Universidade Federal de Juiz de Fora - UFJF
Departamento de Ciência da Computação - DCC