

# Programação Orientada aos Objectos

MiEI/LCC - 2º ano 2020/21

António Nestor Ribeiro

(editado por J.C. Campos em 2015/16)

## **Conteúdos baseados em elementos de:**

### **1. JAVA6 e Programação Orientada pelos Objectos**

F. Mário Martins, Editora FCA, Série Tecnologias de Informação, Julho de 2009. (e posteriores revisões, eg: Java 8 - POO + Construções Funcionais)

### **2. Objects First with Java - A Practical Introduction using BlueJ, Sixth edition**

David J. Barnes & Michael Kölling, Pearson, 2016.

### **3. Object Oriented Design with Applications**

G. Booch, The Benjamin Cummings Pub. Company, USA, 1991

### **4. Java Program Design - Principles, Polymorphism, and Patterns**

Edward Sciore, Apress Media, ISBN 978-1-4842-4142-4, 2019

# POO na Engenharia de Software

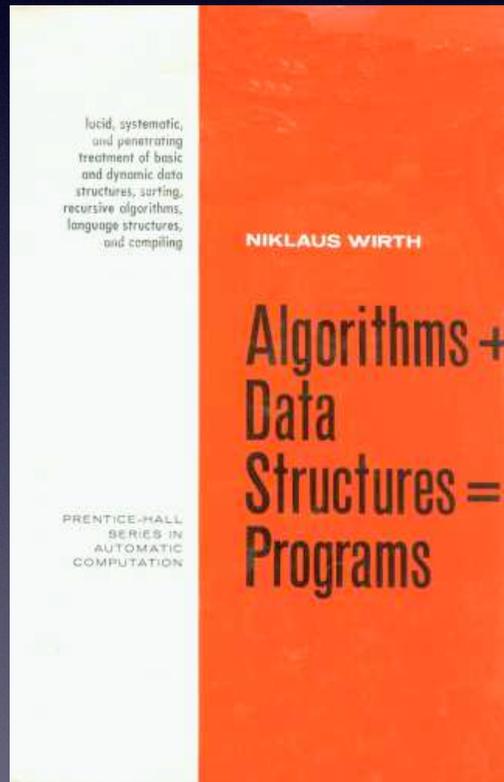
- nos anos 60 e 70 a Engenharia de Software adoptou uma base de trabalho que permitia ter um processo de desenvolvimento e construção de linguagens
- esses princípios de análise e programação designavam-se por estruturados e procedimentais

- a abordagem preconizada era do tipo “top-down”
- estratégia para lidar com a complexidade
- a princípio tudo é pouco definido e por refinamento vai-se encontrando mais detalhe
- neste modelo estruturado funcional e top-down:
  - as acções representam as entidades computacionais de 1ª classe, os algoritmos, a lógica computacional.
  - os dados são entidades secundárias, são as estruturas de dados que as funções e procedimentos “visitam”.

# Estratégia Top-Down

- Consiste num refinamento progressivo dos processos
  - assente numa lógica de dividir a complexidade, uma forma de “dividir para reinar”
  - na concepção de um sistema complexo é importante decompô-lo em partes mais pequenas e mais simples
  - esta divisão representa uma abordagem inteligente, na medida em que se pode trabalhar cada pedaço *per se*

- Niklaus Wirth escreve nos anos 70 o corolário desta abordagem no livro “Algoritmos + Estruturas de Dados = Programas”



- esta abordagem não apresentava grandes riscos em projectos de pequena dimensão
- contudo em projectos de dimensão superior começou a não ser possível ignorar as vantagens da reutilização que não eram evidentes na abordagem estruturada (o que é que se reutiliza? pedaços de código? funções?)
- É importante reter a noção de **reutilização** de software, como mecanismo de aproveitamento de código já desenvolvido e aplicado noutros projectos.

- Um exemplo bem conhecido: o “caso das lista ligadas”:
  - quantas vezes é que já fizemos, em contextos diferentes, código similar para implementar uma lista ligada de “coisas”?
  - porque é que não se reutiliza código? Código muito orientado aos dados: uma LL de Alunos é sempre diferente de uma LL de Carros.
    - porquê? o que muda?
    - porque é que não temos uma implementação genérica?

# Abstracção de controlo

- utilização de procedimentos e funções como mecanismos de incremento de reutilização
- não é necessário conhecer os detalhes do componente para que este seja utilizado
- procedimentos são vistos como caixas negras (black boxes), cujo interior é desconhecido, mas cujas entradas e saídas são conhecidas

- por exemplo:
  - função que dado um array de alunos os ordena por ordem crescente de nota
  - função que dados dois alunos devolve o menor (alfabeticamente) deles
- estes mecanismos suportam reutilização no contexto de um programa
- reutilização entre programas: “copy&paste”
- a reutilização está muito dependente dos tipos de dados de entrada e saída: quase sempre implica mexer nos tipos de dados

# Módulos

- como forma de aumentar o grão da reutilização várias linguagens criaram a noção de **módulos**
- os módulos possuem declarações de dados e declarações de funções e procedimentos invocáveis do exterior
- possuem a (grande) vantagem de poderem ser compilados de forma autónoma
- podem assim ser associados a diferentes programas (em C por exemplo os .o)

- o módulo como abstracção procedimental:

```
//--- ESTRUTURAS DE DADOS -----  
  
struct elemento {  
    void *dados;  
    struct elemento *proximo;  
};  
  
struct lista {  
    size_t tamanho_dados;  
    struct elemento *elementos;  
};  
  
//--- TYPEDEF -----  
  
typedef struct lista Lista;  
  
//--- FUNCOES -----  
  
void inicia_sll(struct lista *,size_t);  
int insere_cabeca_sll(struct lista *,void *);  
int insere_ord_sll(struct lista *,void *,int (void *,void *));  
int apaga_sll(struct lista *,void *,int (void *,void *));  
void destroi_sll(struct lista *);  
int procura_sll(struct lista ,void *,void *v,int (void *,void *));  
void aplica_sll(struct lista ,void (void *));  
void filtro_sll(struct lista *,void *,int (void *,void *));  
  
//--- FIM -----
```

- no entanto, este modelo não garante a estanquicidade dos dados
- os procedimentos de um módulo podem aceder aos dados de outros módulos

- Por vezes, apesar de termos módulos estes conhecem-se e acedem aos dados uns dos outros:
- problemas vários ao nível de dependência entre os módulos (até na compilação dos mesmos)
- a partilha de dados quebra as vantagens de uma possível reutilização
- numa situação de utilização de uma solução destas os diversos módulos teriam de ser todos compilados e importados para os programas cliente!!

# Tipos Abstractos de Dados

- os módulos para serem totalmente autónomos devem garantir que:
  - os procedimentos apenas acedem às variáveis locais ao módulo
  - não existem instruções de input/output no código dos procedimentos
  - não exibem publicamente informação que permita o conhecimento da sua implementação

- A estrutura de dados local passa a estar completamente escondida: **Data Hiding**
- Passamos a disponibilizar serviços (a que chamaremos de API - *Application Programming Interface*) que possibilitam que do exterior se possa obter informação acerca dos dados
- Desta forma, os módulos passam assim a ser vistos como mecanismos de **abstracção de dados**

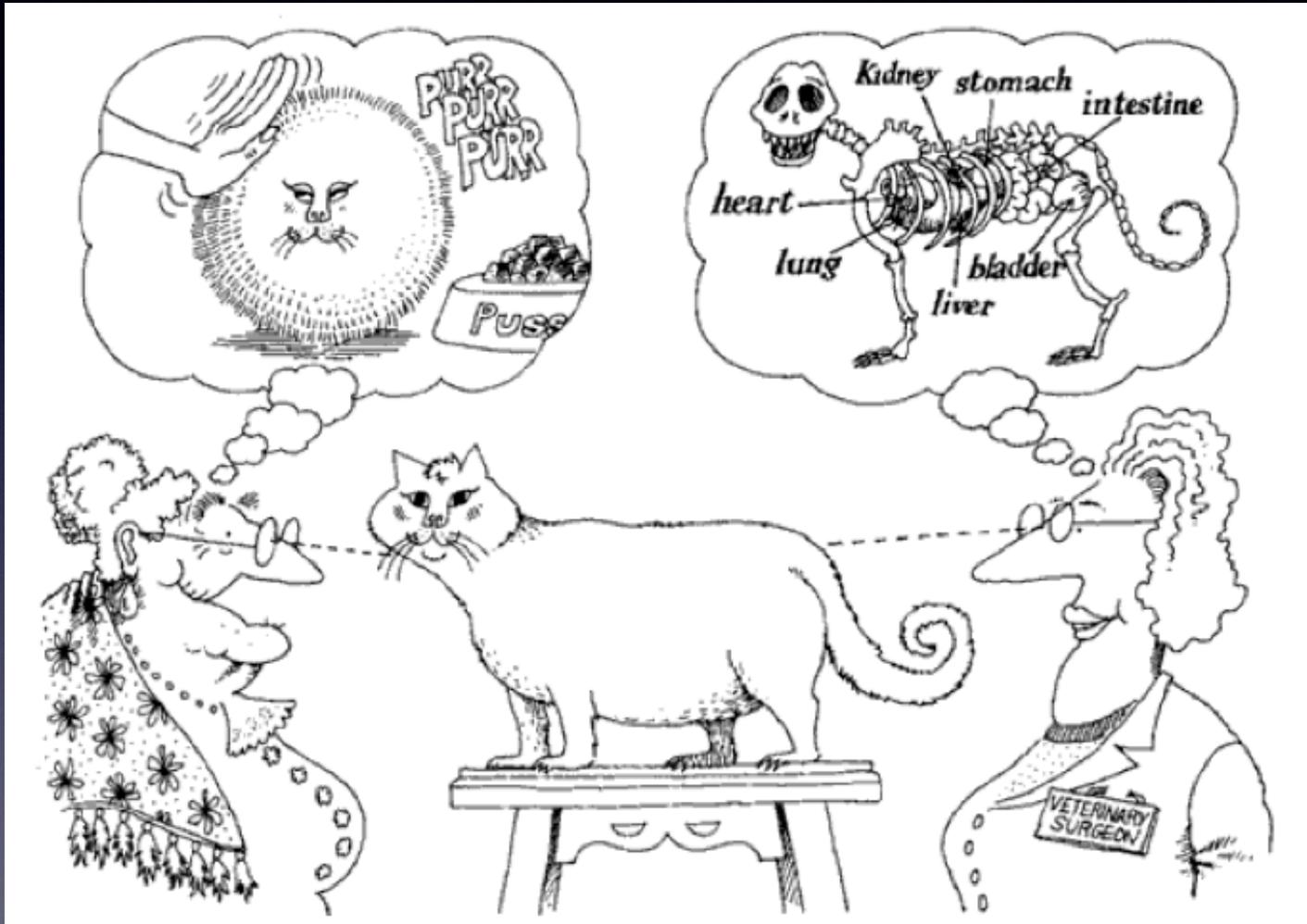
- Nesta abordagem o módulo passa a assumir apenas a informação necessária para que possa ser utilizado.

```
int init (int size, int (*compare)(void *, void *));  
int insert (int handle, void * data);  
int search (int handle, void *data);  
int remove (int handle, void *data);  
int clean (int handle);
```

a lista!!

- apenas se conhece a interface (API) e não se sabe mais nada da implementação
- neste caso, o programa cliente apenas tem acesso a um *handle* que é o apontador para o início da lista

# Abstracção



- cada actor tem a visão que mais lhe convém (interessa)  
(OOAD with Applications, Grady Booch)

- se os módulos forem construídos com estas preocupações, então passamos a ter:
  - capacidade de reutilização
  - encapsulamento de dados
  - possibilidade de termos alteração dos dados sem impacto nos programas clientes
    - ex: numa primeira fase podemos ter uma turma como sendo um array de alunos e depois (sem anúncio) mudar para uma lista ligada

# Programação com TAD

- Consideremos a seguinte definição de um TAD Aluno (por conveniência de escrita escrito em Java, e numa forma não completa)

```
public class Aluno {  
    String nome;  
    String numero;  
    String curso;  
    double media;  
  
    public Aluno(String nome, String numero, String curso, double media) {...}  
    public String getNome() {...}  
    public void atualizaNome(String novoNome) {...}  
    public String getNumero() {...}  
    public String getCurso() {...}  
    public void atualizaCurso(String novoCurso) {...}  
  
    ...  
}
```

- A utilização correcta deste tipo de dados é aquela que apenas utiliza a API para aceder à informação, cf:

```
public static void main(String[] args) {  
  
    Aluno a1 = new Aluno("alberto alves", "a55255", "MiEI", 12.5);  
    Aluno a2 = new Aluno("marisa pinto", "pg20255", "LMat", 15.3);  
  
    System.out.println("Curso do Aluno número:" a1.getNumero() + " = " + a1.getCurso());  
    System.out.println("Curso do Aluno número:" a2.getNumero() + " = " + a2.getCurso());  
  
    ....  
    ....  
    a2.actualizaCurso("MiEngCivil");  
    ...  
}
```

- o acesso ao tipo de dados Aluno é feito apenas via a API definida.
- alterações nas variáveis internas do tipo de dados não tem impacto no cliente

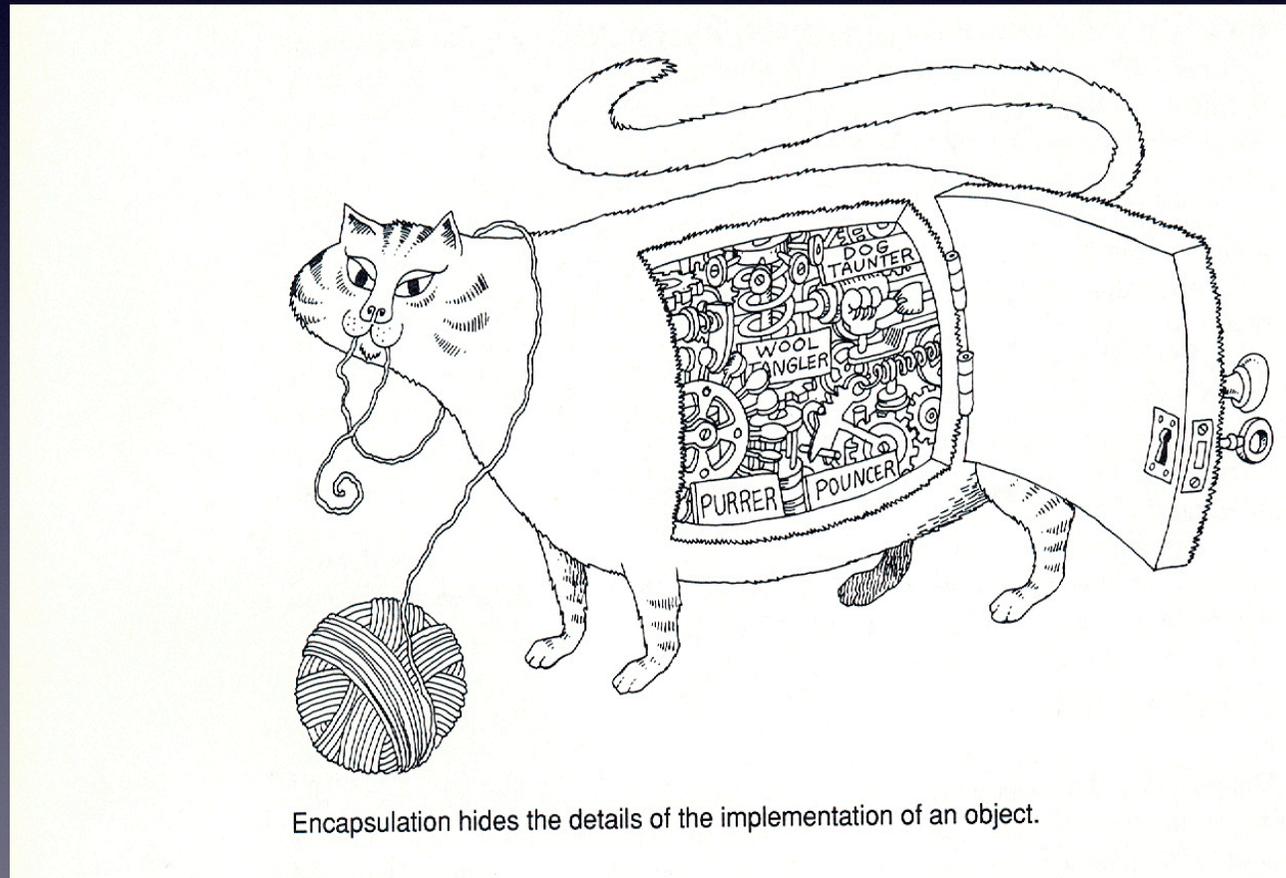
- Uma solução incorrecta, no que concerne à utilização dos módulos como tipos abstractos de dados, seria a que acederia directamente ao estado interno. Cf:

```
....  
....  
a2.curso = "MiEngCivil";  
...
```

- uma utilização destas torna a definição Aluno não reutilizável, visto que não existe a capacidade de a evoluir de forma autónoma das aplicações cliente.
- não se respeita o encapsulamento dos dados

# Encapsulamento

- apenas se conhece a interface e os detalhes de implementação estão escondidos



Encapsulation hides the details of the implementation of an object.

# Metodologia

- criar o módulo pensando no tipo de dados que se vai representar e manipular
- definir as estruturas de dados internas que se devem criar
- definir as operações de acesso e manipulação dos dados internos
- criar operações de acesso exterior aos dados
- não ter código de I/O nas diversas operações
- na utilização dos módulos utilizar apenas a API