Programação Orientada aos Objectos

MiEI/LCC - 2º ano 2016/17

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

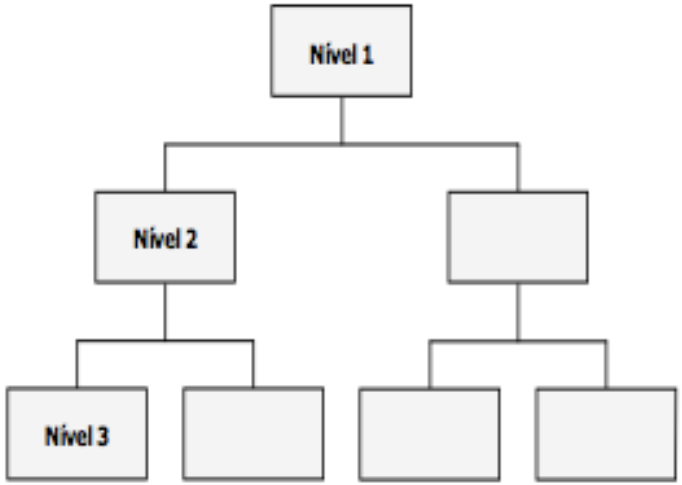
POO na Engenharia de Software

- nos anos 60 e 70 a Engenharia de Software havia adoptado 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

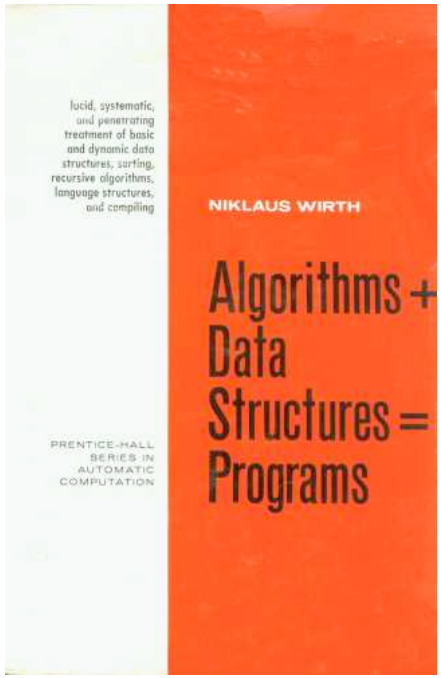
- a abordagem preconizada era do tipo "top-down"
  - estratégia para lidar com a complexidade
  - a princípio tudo é pouco definido e por refinamento vai-se encontrando mais detalhe
- neste modelo estruturado funcional e top-down:
  - as acções representam as entidades computacionais de 1ª classe
  - os dados são entidades de 2ª classe

Estratégia Top-Down

- refinamento progressivo dos processos



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



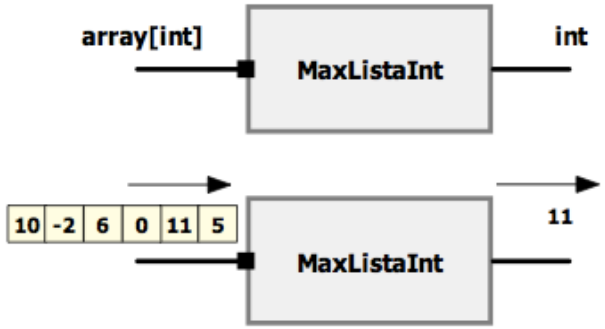
- esta abordagem não apresentava grandes riscos em projectos de pequena dimensão
- contudo em projectos de dimensão superior começou a não ser possível ignorar as vantagens da reutilização que não eram evidentes na abordagem estruturada
- É importante reter a noção de **reutilização** de software, como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

- Mas, como é que isto se faz numa programação estruturada?...
  - documentação, guia de estilo de programação, etc.
  - através da utilização dos mecanismos das linguagens:
    - procedimentos
    - funções
    - rotinas

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: ter uma função que dado um array de inteiros devolve o maior deles

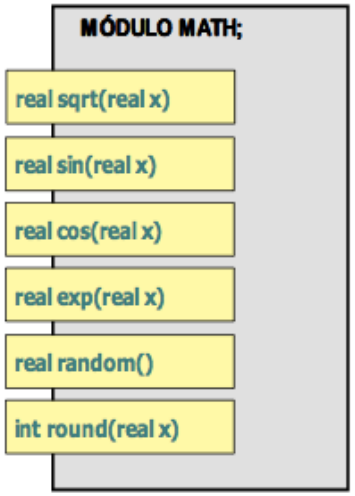


- estes mecanismos suportam reutilização no contexto de um programa
- reutilização entre programas: “copy&paste”
- a reutilização está muito dependente dos tipos de dados de entrada e saída

Módulos

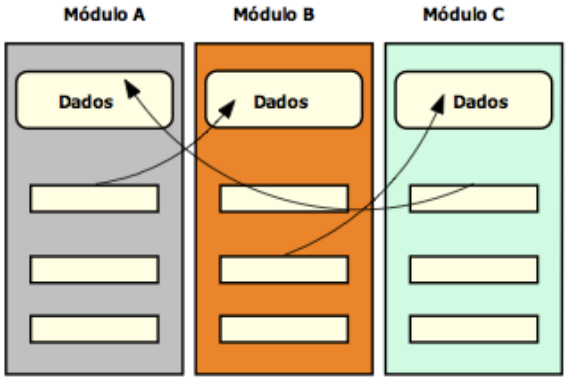
- como forma de aumentar o grão da reutilização várias linguagens criaram a noção de **módulos**
- os módulos possuem declarações de dados e declarações de funções e procedimentos invocáveis do exterior
- possuem a (grande) vantagem de poderem ser compilados de forma autónoma
  - podem assim ser associados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

- módulo como abstracção procedimental:



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

- módulos interdependentes:

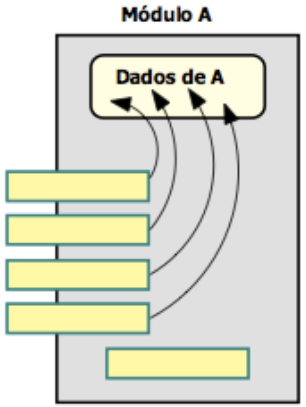


- a partilha de dados quebra as vantagens de uma possível reutilização
- num cenário mais real os diversos módulos interdependentes teriam de ser todos compilados e importados para os programas cliente

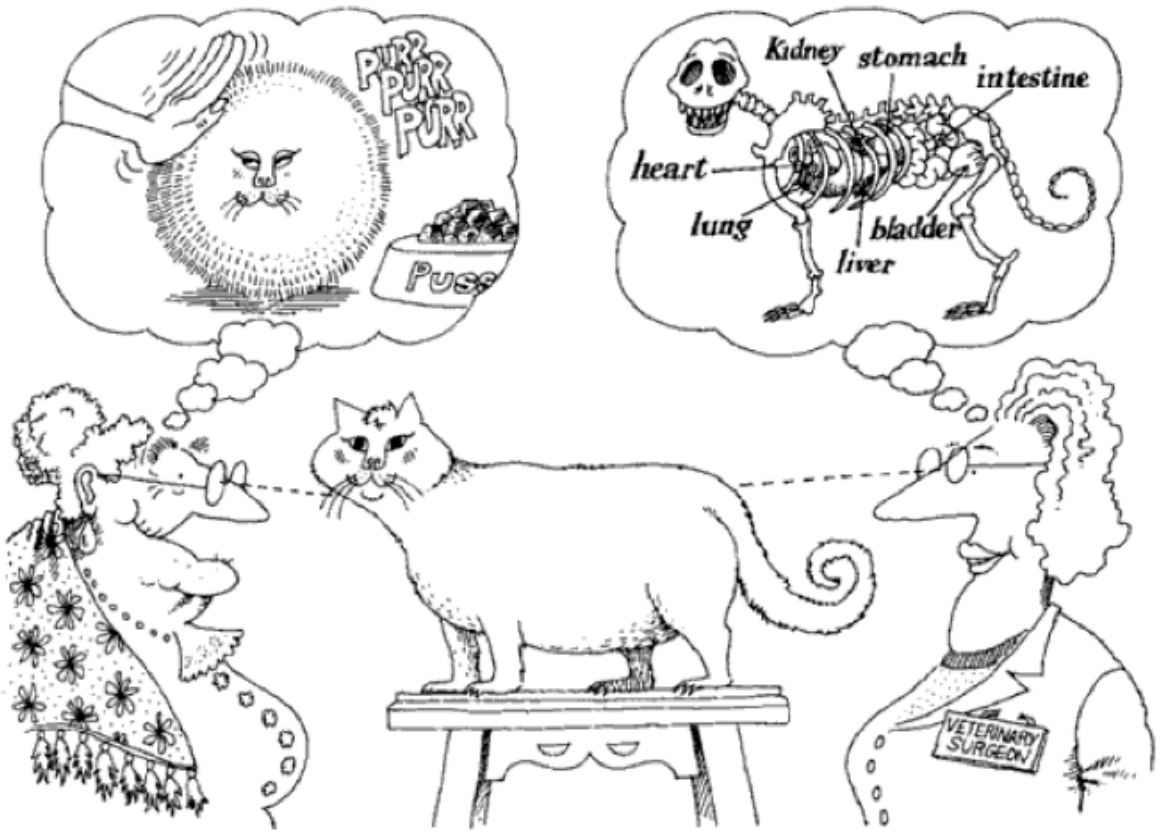
Tipos Abstractos de Dados

- 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

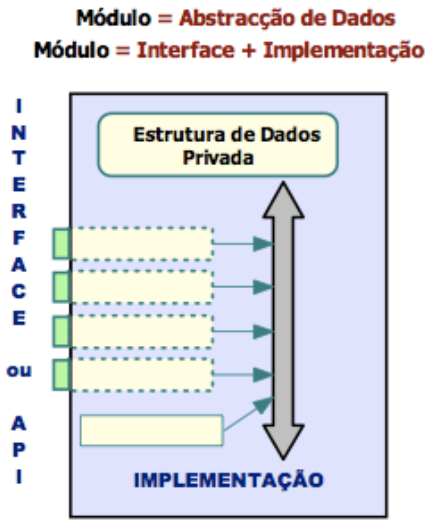
- A estrutura de dados local passa a estar completamente escondida: **Data Hiding**
- Os procedimentos e funções são serviços (API) que possibilitam que do exterior se possa obter informação acerca dos dados
- Módulos passam assim a ser vistos como mecanismos de **abstracção de dados**



Abstracção


PURR PURR PURR
Puss
Kidney
stomach
intestine
heart
lung
bladder
liver
VETERINARY SURGEON

- se os módulos forem construídos com estas preocupações, então passamos a ter:
  - capacidade de reutilização
  - encapsulamento de dados



Exemplo de um TAD

```
MODULE COMPLEXO;

TYPE
  COMPLEXO = RECORD
               real: REAL; // parte real
               img : REAL; // parte imaginária
             END;

(* --- Procedimentos e Funções ---*)

PROCEDURE criaCmplx(r: REAL; i: REAL) : COMPLEXO

PROCEDURE getReal(c: COMPLEXO): REAL;

PROCEDURE getImag(c: COMPLEXO) : REAL;

PROCEDURE mudaReal(dr: REAL; c: COMPLEXO) : COMPLEXO;

PROCEDURE iguais(c1: COMPLEXO; c2: COMPLEXO) : BOOLEAN;

PROCEDURE somaComplx(c: COMPLEXO; c1: COMPLEXO) : COMPLEXO;

END MODULE.
```

- Vejamos como é que este módulo pode ser utilizado pelos diversos programas...

- Exemplo de um programa que respeita os princípios de utilização de módulos

```
IMPORT COMPLEXO;       // PROGRAMA A

VAR complx1, complx2 : COMPLEXO;
    preal, pimg : REAL;

BEGIN
  complx1 = criaComplx(2.5, 3.6);

  preal = getReal(complx1); writeln("Real1 = ", preal);
  pimg  = getImag(complx1); writeln("Imag1 = ", pimg);

  complx2 = criaComplx(5.1, -3.4);

  complx2 = mudaReal(5.99, complx2);
  preal = getReal(complx2); writeln("Real2 = ", preal);

  complx2 = somaComplx(complx1, complx2);

  preal = getReal(complx2); writeln("Real2 = ", preal);
  pimg  = getImag(complx2); writeln("Imag2 = ", pimg);

END.
```

- todo o código está feito utilizando apenas a API (interface) do módulo Complexo

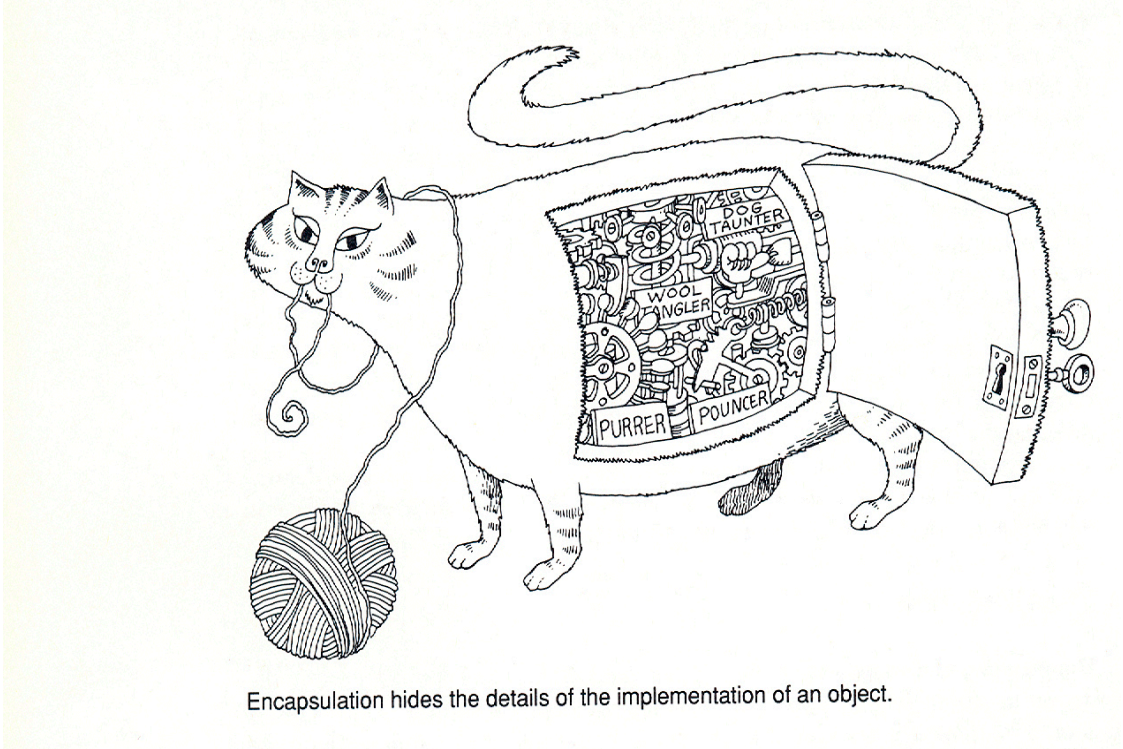
- é possível utilizar o mesmo módulo, mas de forma menos correcta e não respeitando o encapsulamento dos dados

```
IMPORT COMPLEXO;       // PROGRAMA B

VAR complx1, complx2 : COMPLEXO;
    preal, pimg : REAL;

BEGIN
  complx1 = criaComplx(2.5, 3.6);

  preal = complx1.real; writeln("Real1 = ", preal);
  pimg  = complx1.img; writeln("Imag1 = ", pimg);

  complx2 = criaComplx(5.1, -3.4);
```

```
  preal = getReal(complx2); writeln("Real2 = ", preal);
  pimg  = getImag(complx2); writeln("Imag2 = ", pimg);

END.
```

Encapsulamento

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



Encapsulation hides the details of the implementation of an object.

Desenvolvimento em larga escala

- desta forma estamos a favorecer as metodologias de desenvolvimento para sistemas de larga escala
- Factores decisivos:
  - data hiding
  - implementation hiding
  - encapsulamento
  - abstracção de dados
  - independência contextual

Metodologia

- criar o módulo pensando no tipo de dados que se vai representar e manipular
- 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

Evolução das abordagens

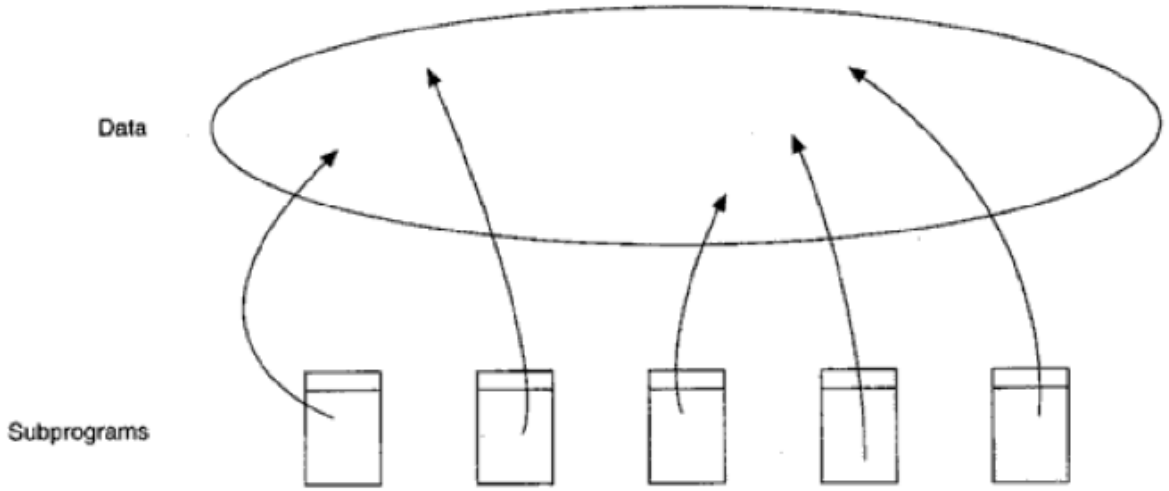


Figure 2-1
The Topology of First- and Early Second-Generation Programming Languages

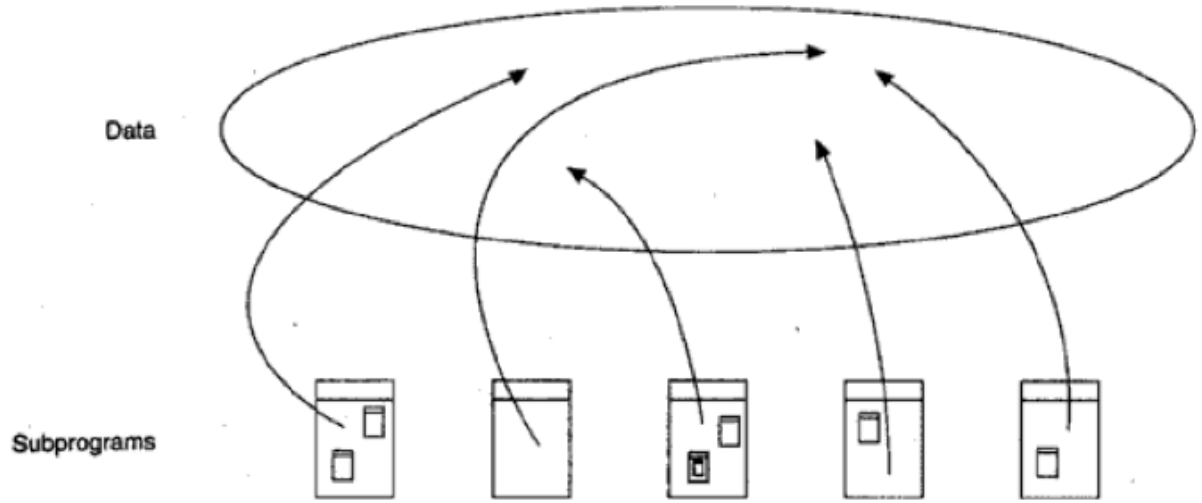


Figure 2-2
The Topology of Late Second- and Early Third-Generation Programming Languages

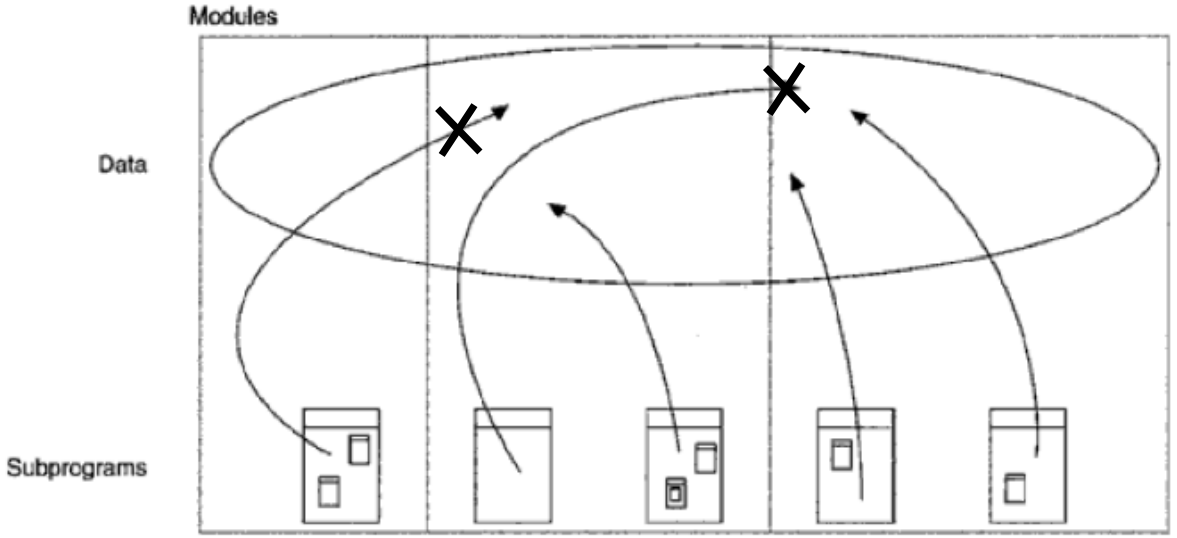


Figure 2-3
The Topology of Late Third-Generation Programming Languages

Onde queremos chegar

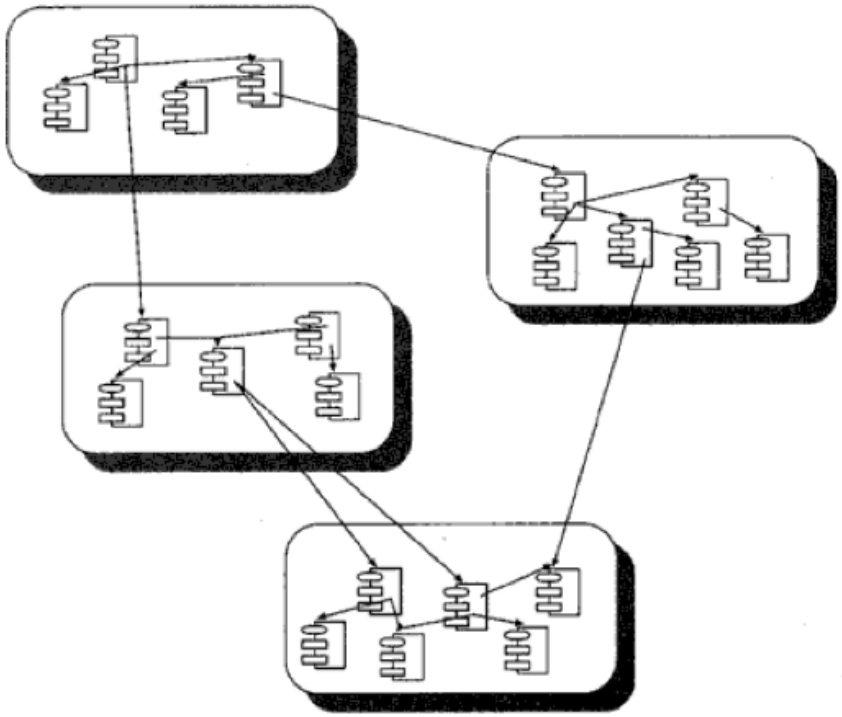


Figure 2-5
The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

As origens do Paradigma dos Objectos

- a maioria dos conceitos fundamentais da POO aparece nos anos 60 ligado a ambientes e linguagens de simulação
- a primeira linguagem a utilizar os conceitos da POO foi o SIMULA-67
  - era uma linguagem de modelação
  - permitia registar modelos do mundo real

- o objectivo era representar entidades do mundo real:
  - identidade (única)
  - estrutura (atributos)
  - comportamento (acções e reacções)
  - interacção (com outras entidades)

- Simula-67 introduz o conceito de "classe" como a entidade definidora e geradora de todos os "indivíduos" que obedecem a um dado padrão de:
  - estrutura
  - comportamento
- Classes são fábricas de *indivíduos*
  - a que chamaremos de "objectos"!

Passagem para POO

- Um objecto é a representação de uma entidade do mundo real, com:
  - atributos privados
  - operações

- **Objecto** = Dados Privados (variáveis de instância) + Operações (métodos)