1 - O Paradigma da Programação por Objectos

1.1 Introdução

Robert Floyd, um dos grandes cientistas da computação, recebeu em 1979 o *ACM Turing Award*, tendo então proferido uma palestra intitulada "Os Paradigmas de Programação", na qual **paradigmas de programação** são definidos como modelos e abordagens que permitem conceptualizar o que se deve entender por computações, bem como devem ser estruturadas e organizadas as tarefas que se pretendem ver realizadas por um computador.

Contrariamente ao que muitos pensam, o paradigma da **Programação Por Objectos** (PPO) ou Programação Orientada aos Objectos, as duas traduções mais comuns da expressão em língua inglesa *"object-oriented programming"*, não é um paradigma de programação nascido nos anos 90, sendo na verdade décadas mais antigo.

Este paradigma, que serve de base às tecnologias de objectos hoje em dia usadas em praticamente todas as áreas da Informática, tem na sua génese um conjunto de desenvolvimentos realizados ao longo de muitos anos em áreas do conhecimento bastante distintas, designadamente, a Simulação e a Engenharia de *Software*.

Pela via da Simulação, a maioria dos conceitos fundamentais do paradigma da PPO surge nos anos 60. Pela via da Engenharia de *Software*, surgem nos anos 70 as primeiras implementações de relevo tendo por base tal paradigma, ainda que, por várias razões, apenas nos anos 90 tenham de facto tido impacto tecnológico.

No entanto, curiosamente, alguns dos conceitos fundamentais do paradigma dos objectos remontam à filosofia grega, em particular a Sócrates e ao seu discípulo Platão, nos séculos V e IV a.C., a propósito da forma ideal de realizar a catalogação ou classificação do conhecimento, já então visto como podendo ser dividido em "ideias individuais" e em "classes de ideias". Estas duas correntes deram também origem, no início, a linguagens de PPO baseadas em modelos diferentes, tal como se irá referir mais adiante.

1.1.1 A via da simulação

De facto, a primeira linguagem a ser desenvolvida tendo por base a maior parte dos conceitos que iremos posteriormente estudar, e que são os conceitos principais que caracterizam e tornam diferente este paradigma de todos os outros, surgiu no final dos anos 60, foi desenvolvida por Dahl em Oslo e designava-se SIMULA-67.

Como o próprio nome indicia, a linguagem destinava-se a ser não propriamente uma linguagem de programação, mas antes uma linguagem na qual pudessem ser desenvolvidos modelos do mundo real e sobre estes executadas simulações.

O problema a resolver nesta área consiste em encontrar formas simples de representar (modelizar) em computador, para mais fácil estudo, entidades do mundo real cujo "comportamento" se pretende analisar, sendo reconhecido que tais entidades possuem atributos caracterizadores muito próprios, isto é, valores associados a propriedades que representam a sua "estrutura" interna.

As entidades do mundo real são, neste contexto da simulação, consideradas modeláveis, desde que sobre as mesmas se possuam as seguintes informações:

- **Identidade** (única);
- **Estrutura** (via atributos, ou seja, as propriedades estáticas);
- **Comportamento** (as acções e reacções respectivas);
- **Interacção** (forma de relacionamento com outras entidades).

A estas entidades abstractas, modelos de entidades reais através de um processo de abstracção (simplificação), caracterizadas pela informação anteriormente indicada, a linguagem SIMULA-67 deu o nome de "objectos". Dada a necessidade de modelar por vezes inúmeros objectos diferindo apenas na sua identidade, mas que em tudo o resto possuíam características iguais, por exemplo, pontos no plano 2D, logo, todos com coordenadas X e Y e com iguais comportamentos (por exemplo todos podem ser incrementados ou decrementados em X e Y), SIMULA-67 introduz a noção de "classe" como entidade definidora e geradora de todos os "indivíduos" que obedecem a um dado padrão comum de estrutura e comportamento. Seguindo o exemplo, teríamos então que definir uma Classe Ponto2D, a partir da qual todos os pontos individuais poderiam ser criados, só desta forma podendo haver a garantia de que todos possuem a mesma estrutura e comportamento, ainda que, naturalmente, possuindo valores diferentes nos seus atributos, para além de terem, naturalmente, diferentes identificadores.

Os objectos assim definidos em SIMULA-67 eram considerados, do ponto de vista da simulação, como "entidades reactivas", ou seja, entidades capazes de responder a solicitações exteriores realizando operações sobre o seu estado interno, e enviando uma resposta à entidade solicitadora. As "classes" funcionavam como "fábricas de …".

Sendo programas em computador simulações digitais quer de modelos conceptuais quer de modelos físicos, os resultados obtidos na área da Simulação usando tal abordagem foram sempre de grande interesse para a Engenharia de *Software*.

1.1.2 A VIA DA ENGENHARIA DE *SOFTWARE*

Nos anos 60 e 70, a Engenharia de *Software* havia adoptado um conjunto de princípios relativos ao processo de desenvolvimento de aplicações e construção de linguagens, genericamente designados como análise e programação **estruturadas** e **procedimentais**.

A ideia geral consistia em considerar-se que, quer nas tarefas de análise quer nas tarefas de programação, as entidades de interesse deveriam ser inicialmente pouco definidas, sendo quer estrutural quer funcionalmente pouco concretas, por forma controlar-se a sua intrínseca complexidade. Em seguida, em passos sucessivos, cada um destes níveis de abstracção funcional, ou seja, de processos (na análise) ou instruções (na programação), era individual e separadamente refinado até se atingir o limite do seu detalhe. Porém, sempre com a máxima atenção focada na componente funcional, ou seja, processos, procedimentos e instruções. Os dados acompanhavam o processo de refinamento quase que por simpatia, sendo claramente entidades de segundo plano.

O processo era designado por **Análise Estruturada** ou **Programação Estruturada**, conforme a fase do desenvolvimento e o produto final pretendido, tratando-se de facto de um processo em árvore de desenvolvimento de cima (nível mais abstracto) para baixo (nível mais concreto), por isso por muitos designado por metodologia *"top-down"* (Figura 1.1).

Este modelo estruturado funcional e *top-down*, em que as acções representavam as entidades computacionais de 1ª classe e os dados as entidades computacionais de 2ª, quer pelas suas virtudes quer pelas suas ineficiências, é um modelo muito importante dado que esteve na base de todos os grandes desenvolvimentos na área da Engenharia de *Software* dos últimos 35 anos, e também porque foi com base no mesmo que a Informática foi mundialmente sendo "realizada" nas organizações mais diversas.

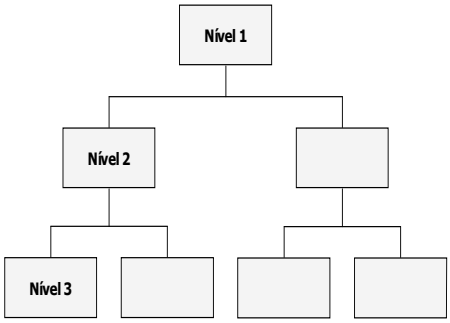

Figura 1.1 – Refinamento progressivo dos processos

Como prova fundamental desta tendência orientada às instruções e aos algoritmos, Niklaus Wirth, que em 1971 havia concebido a linguagem PASCAL desenvolvendo ideias inovadoras apresentadas na linguagem ALGOL de 1968, entre elas a necessidade de os dados serem rigorosamente associados a tipos verificáveis pelo compilador, usa, em 1975, a linguagem PASCAL para apresentar, em obra de programação célebre de igual nome, a fórmula em tal momento fundamental: Algoritmos + Estruturas de Dados = Programas. Note-se que os Algoritmos vêm primeiro e em segundo lugar surgem as Estruturas de Dados. Note-se ainda que a semântica do + nunca foi então claramente definida, ou seja, como deveria ser realizada estruturadamente a junção entre algoritmos e dados. Mas o mais importante a realçar é o facto de que instruções e dados foram durante muitos anos vistos como entidades importantes, mas com tratamentos separados.

Estes princípios serviram igualmente de guião ao desenvolvimento das principais linguagens então largamente utilizadas, designadamente, o Basic, o Fortran e o Cobol, linguagens caracterizadas por possuírem um conjunto de instruções com poucas formas de estruturação do controlo da execução, o que causava muita dificuldade de leitura e compreensão do código dos programas. Além disso, os programas eram muito inseguros por não existir qualquer **verificação estática de tipos** (*static type checking*).

De facto, ao contrário do que se verifica actualmente com linguagens mais modernas, como PASCAL, MODULA, C, C++, JAVA, etc., muitas variáveis não eram declaradas, isto é, associadas a tipos de dados. Não o sendo, o compilador não poderia realizar uma verificação estática de tipos, isto é, verificar em tempo de compilação – e não em tempo de execução – se todas as variáveis estavam a ser bem usadas, no sentido de lhes serem associados os valores compatíveis com o seu tipo.

As primeiras tentativas de melhoria destas linguagens de programação surgiram naturalmente à medida que os projectos de *software* se iam tornando de maior dimensão, onde passou a ser inaceitável, até pelos custos envolvidos, que todo o projecto de *software* tivesse que começar sempre "do zero", ou seja, que nada do que havia sido feito em projectos anteriores pudesse ser reaproveitado. Surge assim a noção muito importante de **reutilização de software**, ou seja, de garantir que qualquer programador deveria desenvolver código posteriormente reutilizável noutros projectos. Porém, nos anos 70, a única solução prática existente para garantir tal possibilidade, dada a inexistência de construções nas linguagens de programação que auxiliassem a tal objectivo, consistia em instruir e solicitar aos programadores que documentassem de forma clara o seu código, usando o que as linguagens de então aceitavam para documentação dos programas. Sentindo que o seu poder poderia estar em jogo nas organizações, os programadores adoptaram a prática contrária, ou seja, ou não documentavam programas ou os documentavam de forma dúbia, de forma a garantirem que as organizações, do ponto de vista informático, deles continuavam dependentes.

O constante aumento da complexidade dos problemas a resolver, entretanto sentida por todos os envolvidos em Informática, conduziu à necessidade de criação de mecanismos de abstracção capazes de facilitar as tarefas quer de análise quer de programação, ainda que sob um ponto de vista procedimental, ou seja, procurando sempre responder à questão de como especificar a funcionalidade das aplicações de forma a que a mesma ou partes delas possam ser tão claras que possam ser facilmente corrigida (erros), mantida (mudanças de requisitos) e reutilizada (noutras aplicações). Este tem sido o grande e final desiderato da Engenharia de *Software*.

As linguagens de programação, ou seja, as tecnologias de base da programação, acabaram por mais tarde dar

resposta a algumas exigências, não apenas desenvolvendo esquemas de controlo da execução dos programas muito mais compreensíveis, tais como as estruturas *for, repeat, while* e *case*, como também diferentes mecanismos de abstracção de controlo (ou de instruções), tais como *as subrotinas* (*subroutines*), as funções (*functions*) e também os procedimentos (*procedures*).

Estes primeiros mecanismos de **abstracção de controlo** implementam uma forma simples de **reutilização de software**, ao permitirem que um bloco de instruções seja escrito independentemente do programa principal, tendo a si associado um identificador único, pelo qual tal conjunto passa a ser referido (ou invocado). Em geral, este bloco de instruções é relativamente genérico, pois aceita parâmetros de entrada sobre os valores dos quais as instruções são executadas, sendo produzidos resultados que são comunicados para o exterior.

A estrutura de um programa deixa assim de ser monolítica, passando cada vez mais a ser composta por um programa principal e por um conjunto de procedimentos e funções por este utilizáveis através de mecanismos simples de invocação.

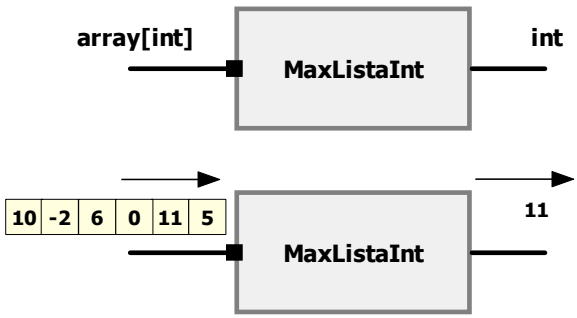

Figura 1.2 – Abstracção de controlo ou procedimental

A abstracção de controlo associada aos procedimentos ou funções resulta do facto de que, tal como se pretende ilustrar na Figura 1.2, para que os mesmos sejam utilizados não é necessário que se conheça o seu interior (o seu código), mas apenas a sua forma correcta de invocação e o resultado por este eventualmente devolvido. Ou seja, os procedimentos são vistos como *black boxes*, cujo interior é desconhecido, o que não impede a sua utilização desde que se compreendam as relações das entradas com as saídas, ou seja, a transformação realizada.

Por exemplo, uma simples função Maior, que dados dois valores de entrada de um dado tipo dá como resultado o maior dos valores parâmetro, teria que ter tantas diferentes versões quantos os possíveis tipos de dados simples dos seus parâmetros de entrada. A solução seria criar uma função de nome diferente para cada caso, como MaiorInt, MaiorReal ou MaiorFloat, o que, como se compreende, se tornava pouco eficaz. Qualquer outra solução mais "inteligente" teria que ser completamente programada.

Por outro lado, ao ser definido pelo programador um novo tipo de dados, havia que ter em atenção que, a menos que o programador definisse igualmente um conjunto de procedimentos e funções que implementassem as operações que com tal tipo de dados se pretendiam realizar, por exemplo, operações básicas, tais como a igualdade de valores do tipo, a comparação entre valores do tipo e outras, muito pouco se poderia fazer com os valores de tal tipo. Por exemplo, se o programador necessitasse de desenvolver uma *Stack* de inteiros, não só deveria criar uma representação da *stack*, por exemplo usando um *array* de inteiros, como também programar os procedimentos de criação, *pop*, *push*, etc.. Se tal não fosse feito, a *stack* teria uma representação mas não poderia ser manipulável.

De facto, e de um ponto de vista matemático, definir um tipo de dados com sendo apenas um conjunto de valores que variáveis de tal tipo podem assumir é redutor, dado que associa um tipo a um conjunto matemático. Definir um tipo de dados como tal conjunto de valores, bem como todas as operações que sobre o mesmo se podem realizar, associa um tipo de dados a uma álgebra, noção matemática muito diferente da noção de conjunto. Esta distinção definicional trouxe às Ciências da Computação conceitos novos, sendo a noção de *Abstract Data Type* ou **Tipo Abstracto de Dados** *(TAD)* uma das mais importantes, até porque, como veremos adiante, se liga directamente com algumas características da PPO. Não foi ainda este o passo

seguinte.

Cronologicamente, o passo seguinte das linguagens, ainda no sentido de melhorar os mecanismos de abstracção de controlo, foi o aparecimento em algumas delas, por exemplo, em UCSD PASCAL, MODULA, PL/1, etc., de unidades de computação designadas por **módulos**. Os módulos continham declarações de dados e declarações de diversos procedimentos e funções que podiam ser invocados do exterior, mas possuíam a importante propriedade de poderem ser compilados isoladamente e posteriormente poderem ser "ligados" a qualquer programa que deles necessitasse.

A abstracção continuava, portanto, a ser de controlo ou de funcionalidade, mas os módulos facilitavam largamente a tão desejável reutilização de código, ainda que, em geral, via importações completas dos mesmos.

Com este mecanismo, a maior parte das linguagens passou a ser fornecida com um enorme conjunto de módulos guardados em bibliotecas, que auxiliavam muito os programadores, já que, não sendo necessário conhecer o seu código, ofereciam uma funcionalidade, por vezes, de grande utilidade. Na Figura 1.3 apresenta-se um exemplo típico de um módulo que oferece funcionalidades relacionadas com cálculos matemáticos.

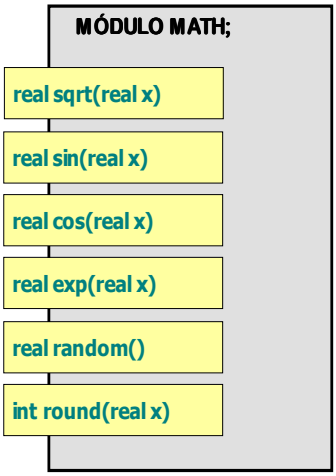

Figura 1.3 – Módulo como abstracção procedimental

Device drivers, módulos de gestão de ficheiros, módulos de tratamento de *strings*, etc., passaram a ser as novas *black boxes* da programação, desta vez porém com a evidente vantagem de serem facilmente reutilizáveis por serem unidades de compilação em separado.

No entanto, e para além dos anteriormente referidos problemas relacionados com a falta de generalidade dos procedimentos e funções por serem "tipados", tal como mais uma vez pode ser visto no módulo MATH, o outro problema que conduziu à falência relativa deste modelo, que já permitia uma "programação modular, mas ainda orientada às instruções", é que a utilização dos dados continuava a ser descurada, por continuar a ser considerado um problema de segunda ordem de importância.

Em resultado, os dados definidos num dado módulo poderiam ser acessíveis a vários outros módulos, como a Figura 1.4 procura ilustrar, ou seja, os procedimentos de um dado módulo poderiam ter acesso às estruturas de dados de outros módulos utilizados pelo programa.

Porém, admitindo tais acessos, um grave problema surge então. Se o objectivo dos módulos é encontrarmos unidades de programação que sejam independentes e reutilizáveis em qualquer contexto, isto é, em diferentes programas, então, tais módulos deveriam ser construídos em completa independência de todos os outros, ou seja, sendo completamente autónomos e, portanto, não devendo construir a sua funcionalidade à custa do acesso directo às variáveis de outros módulos, apenas realizando computações que utilizam unicamente o seu estado interno.

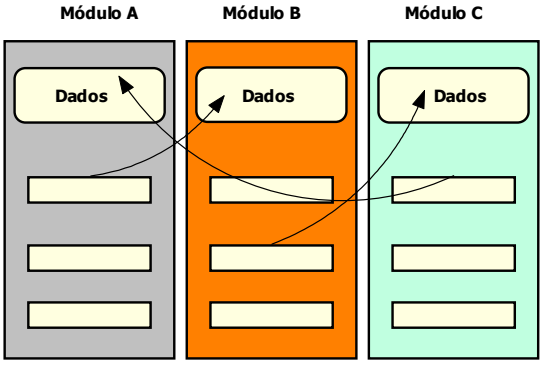

Figura 1.4 – Módulos interdependentes

De facto, se um módulo A depende das variáveis de um módulo B e, por sua vez, B depende de outros módulos (ver Figura 1.4), então, onde quer que se necessite de A, o módulo B e todos os de que B depende, e assim sucessivamente de forma transitiva, teriam que ser importados também para que A pudesse ser compilado e usado. A complexidade de controlar todas estas dependências em situações usuais de erro ou de simples manutenção seria para o programador uma tarefa de grande complexidade.

Assim, a única forma de se garantir que um módulo é completamente independente do contexto, ou seja, utilizável em qualquer programa, será torná-lo independente de todo e qualquer outro módulo, ou seja, torná-lo absolutamente autónomo. Para tal haverá que garantir que os seus procedimentos apenas acedem às variáveis que são locais ao módulo e, adicionalmente, o que é muito relevante, que no código dos procedimentos não existem instruções de *input/output*.

1.2 ABSTRACÇÃO DE DADOS E *DATA HIDING*

Com base em tais requisitos, os módulos passam a ser vistos como definindo uma estrutura de dados interna e contendo um conjunto de procedimentos e funções que devem ser o único código com acesso às suas variáveis internas, sendo igualmente de garantir que tais funções e procedimentos não acedem a quaisquer outras variáveis que não façam parte dos dados locais do módulo (Figura 1.5).

Deste modo, a entidade fundamental de um módulo, que passa a estar protegida e "escondida" (daí a expressão *data hiding*) e que, idealmente, deve ser inacessível do exterior, é a **estrutura de dados local** ao módulo. Os procedimentos e funções passam a representar o papel de "serviços" disponibilizados pelo módulo para que do exterior se possa aceder à estrutura de dados. Assim sendo, os módulos passam a ser vistos, finalmente, como **mecanismos de abstracção de dados** e não de abstracção de controlo.

A Figura 1.5 representa este tipo de modularidade baseada em mecanismos de abstracção de dados.

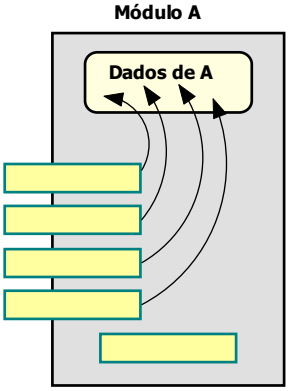

Figura 1.5 – Módulo independente do contexto

Se todos os módulos forem programados tendo por base estes princípios, o que apenas dependerá dos programadores dado que, em geral, as linguagens não fazem qualquer tipo de verificação e validação destas

regras, então, os módulos passam a ser não só unidades de programação completamente autónomas, portanto de facto independentes do contexto onde vão ser usados e assim sempre reutilizáveis, como ainda passam a poder ser vistos como "cápsulas", no sentido em que "escondem" do exterior detalhes de implementação, quer de dados quer de procedimentos, com benefícios para ambas as partes.

A Figura 1.6 ilustra o encapsulamento dos dados, tornados privados e apenas acessíveis aos métodos programados no interior do módulo, dos quais alguns serão invocáveis do exterior.

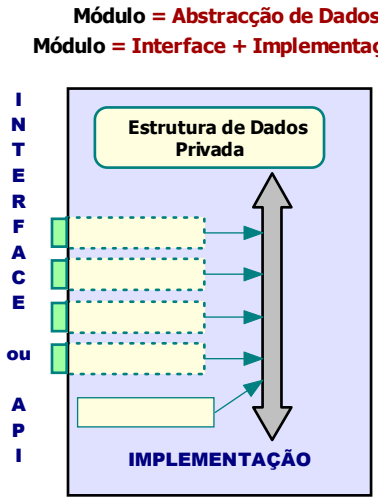

Figura 1.6 – Módulo como Abstracção de Dados

O acesso do exterior à funcionalidade do módulo é garantida, dado que o módulo declara como públicos, isto é, invocáveis do seu exterior, um conjunto de procedimentos e funções. A este conjunto de procedimentos e funções invocáveis do exterior de um módulo dá-se em geral o nome de **interface**, ou até de API, do inglês *application programmer´s interface*. Estes deverão ser, de facto, dentro deste conjunto de regras que temos procurado estabelecer com vista à obtenção de objectivos muito importantes no âmbito da programação, os únicos mecanismos de acesso exterior ao módulo. Assim, um módulo passa a ser uma **abstracção de dados** que se pode dividir em duas camadas distintas: **uma interface** e **uma implementação** (Figura 1.6).

Esta visão de um **módulo de software** como sendo uma "cápsula", ou seja, uma entidade apenas acessível do exterior pelo que de si define como sendo de uso "público", ou seja, os procedimentos que fazem parte da sua interface, necessita agora de ser analisada com base em exemplos concretos, para que melhor se compreendam tais vantagens. Vejamos um exemplo: consideremos o módulo seguinte programado numa linguagem fictícia semelhante a PASCAL.

```
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.
```

Este módulo, de nome **COMPLEXO**, em estreita obediência às regras apresentadas guarda um representação do tipo de dados `Complexo` como sendo um `RECORD` com dois `REAL`, e define um conjunto de funções disponibilizadas na sua interface para que do exterior do módulo possam ser criados e manipulados números complexos, designadamente, consultas, modificação da parte real, comparação e soma. Note-se que os números complexos a tratar não são interiores ao módulo. Existem em espaços de variáveis de programas que usam este módulo.

Torna-se agora fundamental compreender, com base neste exemplo, a importância do **encapsulamento** e da obediência a certas regras por parte de quem quem programa. O que está encapsulado neste módulo é, antes de mais, a representação de um número complexo. Por isso, é que é uma **abstracção de dados**, como o próprio nome do módulo indica. Vamos agora considerar dois exemplos distintos de utilização deste módulo `COMPLEXO` e verificar o que acontece em cada caso perante uma situação de alteração do código interno do módulo, em especial da representação do tipo `complexo`.

Consideremos um primeiro programa obedecendo rigorosamente às regras básicas de utilização de um módulo, ou seja, fazendo acesso ao módulo usando apenas o que este exporta, isto é, torna público (a sua API), que serão sempre procedimentos e nomes de tipos de dados, mas nunca representações internas.

Todo o código está escrito usando apenas as funções da API, sem nunca utilizar a representação interna do tipo de dados que, apesar de tudo, o programador sabe (basta-lhe ler o código fonte) ser um `RECORD` com dois campos `REAL`. No entanto, o programador respeitou as regras do encapsulamento tendo apenas usado a interface do módulo.

```
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.
```

Consideremos, em seguida, um segundo programa que realiza exactamente as mesmas operações básicas que este primeiro, mas que, no entanto, as vai realizar usando o conhecimento que o seu programador tem sobre a representação interna do tipo de dados `Complexo` e que usa explicitamente. Ou seja, conhecendo a representação interna do tipo `Complexo`, usa-a directamente no seu código para fazer acesso aos valores da mesma que pretende manipular (os campos do `RECORD`):

```
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);

  complx2.real = 5.99;
  preal = complx2.real; writeln("Real2 = ", preal);

  complx2.real = complx1.real + complx2.real;
  complx2.img = complx1.img + complx2.img;

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

END.
```

Naturalmente que o programa não vai gerar erros de compilação e vai executar de forma correcta, produzindo até os mesmos resultados que o primeiro. Porém, este segundo programa, contrariamente ao primeiro, não é independente do contexto e a qualquer momento poderá tornar-se num programa errado e que não executa.

De facto, porque usa directamente a representação interna do tipo de dados, que deveria ser protegida, mas que de facto não o é na maioria das linguagens pois o compilador aceita a sintaxe utilizada, este programa passará a conter erros caso a representação do tipo de dados no módulo seja modificada, numa próxima versão, por exemplo.

Imagine-se que o programador do módulo `COMPLEXO` altera a representação interna do tipo de dados `Complexo` para um *array* de reais com duas posições, e reprograma o código dos procedimentos em conformidade com esta nova representação. O que acontecerá a cada um dos dois programas anteriores?

O primeiro programa, porque apenas usou os mecanismos de acesso ao módulo que fazem parte da interface deste, não sofreria qualquer tipo de problema. Porém, o segundo programa deixaria de estar correcto porque usa a representação antiga de forma explícita e directa, e esta foi entretanto alterada. Todas as instruções a cinzento passariam a estar erradas, gerando erros de compilação.

Finalmente, uma outra regra de programação muito importante e que deve ser respeitada sempre que pretendermos escrever código independente do contexto e, portanto, reutilizável, é nunca introduzir código de leitura e escrita (*input/output*) junto do código que implementa a camada computacional.

Há muitos anos que na área das interfaces com o utilizador foi definido um princípio fundamental para o correcto desenvolvimento e articulação entre as duas camadas, designado por "princípio da separação", que advoga exactamente a completa separação das instruções que implementam a interface com o utilizador das que implementam a funcionalidade das aplicações, ou seja, a separação da camada interactiva da camada computacional.

A justificação é, cada vez mais, muito fácil de compreender. Antes de tudo, para a mesma camada computacional poderá ser necessário ter mais de um tipo de interface com o utilizador, dependendo dos ambientes de execução. Por exemplo, sem componentes gráficos, com componentes gráficos, como janelas e botões, de um dado sistema de janelas ou de outro, etc. Se introduzirmos código de *input* ou de *output* na camada computacional, é evidente que a funcionalidade desta ficará dependente do tipo de interface com o utilizador que tal código representa. Deixa de imediato de ser uma camada computacional independente do

contexto e, assim, não reutilizável ou, no mínimo, não facilmente reutilizável.

Em conclusão, tendo sido durante muitos anos uma das grandes preocupações da Engenharia de *Software* a criação de unidades de programação que oferecessem a quem desenvolve aplicações não apenas um grau de abstracção que permitisse que a complexidade dos projectos fosse diminuída, mas também que permitissem reduzir custos de projecto caso pudessem ser unidades reutilizáveis, e tendo-se pensado inicialmente que a solução estava nas abstracções de controlo ou procedimentais, a prática veio a demonstrar que a evolução de tais mecanismos se deveria centrar nos dados e não nas instruções, o que conduziu à definição, actualmente em vigor, de que tais unidades de abstracção, independentes do contexto e reutilizáveis, devem ser de facto **mecanismos de abstracção de dados**. Estes assumem a estrutura de uma cápsula cujo acesso exterior deverá apenas ser realizado através do que tal cápsula disponibiliza para o exterior pela definição da sua interface ou API.

Através de pequenos exemplos, foram apresentadas regras, quer para a construção quer para a utilização de tais módulos, regras que são essenciais para que tais mecanismos sejam não só bem programados como também bem utilizados. A noção de ***objecto***, crucial à PPO, que se apresenta na secção seguinte, assenta fundamentalmente em tais conceitos e regras, que são ainda complementadas em PPO pela existência de mecanismos adicionais de abstracção, de generalização e de extensibilidade.

1.3 O QUE É UM OBJECTO?

A noção de ***objecto*** é uma das noções cruciais ao paradigma da PPO, dado que tal conceito pretende em si concentrar todas as virtudes de um modelo de concepção e desenvolvimento de *software* baseado nas propriedades anteriormente estudadas e vistas como fundamentais, e que são:

- **a independência de contexto** (que permite reutilização);
- **a abstracção de dados** (que garante abstracção);
- **o encapsulamento** (que garante abstracção e protecção);
- **a modularidade** (que garante composição das partes).

Um objecto é, no contexto da PPO, o módulo computacional básico e único, e, por definição, corresponde à representação abstracta de uma entidade autónoma sob a forma de:

- Uma identidade única;
- Um conjunto de atributos privados (o **estado** interno do objecto);
- Um conjunto de operações que são as únicas que podem aceder de forma directa a tal estado interno. Destas operações algumas podem ser definidas como invocáveis a partir do exterior do objecto (públicas), constituindo a sua **interface**, enquanto que outras poderão ser declaradas como apenas acessíveis a partir de outras internas ao objecto (privadas); tais operações representam, no seu conjunto, o designado **comportamento** total do objecto.

As operações que um objecto é capaz de realizar e que são por si definidas como acessíveis ou invocáveis do seu exterior, constituem aquilo que normalmente se designa por **interface** do objecto, ou apenas **API**, no sentido de que quem pretender utilizar a funcionalidade oferecida pelo mesmo apenas o poderá fazer usando as operações definidas por tal objecto como públicas.

Dadas estas características e propriedades de acesso e visibilidade, **um objecto** é de facto uma entidade cuja estrutura interna **deve ser** desconhecida do exterior e, por isso, não directamente acessível, e que apenas divulga para o exterior um conjunto de operações que é capaz de executar quando externamente invocadas, operações essas que poderão, ou não, devolver a quem as invocou um resultado. É comum até associar este comportamento dos objectos à noção de "prestação de serviços" às entidades que os solicitem via interface, sendo tais entidades vistas como seus "clientes".

Assim, um objecto pode ser visto como sendo uma ***black box***, ou cápsula, que disponibiliza alguns "botões" que, quando são accionados, realizam uma computação interna no objecto e devolvem, ou não, um resultado. O conjunto de serviços que um objecto é capaz de prestar coincide com a sua **interface** ou API.

Passaremos a designar os identificadores que guardam os valores dos atributos de um dado objecto por **variáveis de instância**, e as operações que representam o seu comportamento, ou seja, as computações que é capaz de realizar internamente, por **métodos de instância**.

1.4 O ENCAPSULAMENTO: PROPRIEDADE FUNDAMENTAL

A Figura 1.7 procura reforçar visualmente esta perspectiva de que um objecto deve ser visto como uma **cápsula** (uma "capsulação de dados", ou ainda, usando um neologismo informático de origem anglo-saxónica, um **encapsulamento de dados**).

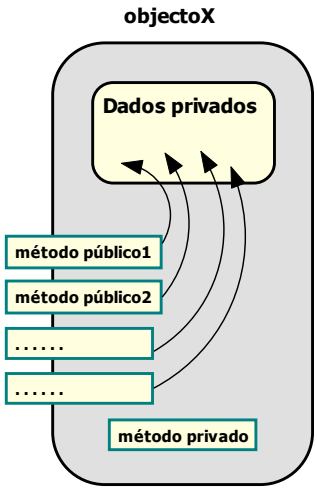

Figura 1.7 – Objectos como cápsulas

Em termos genéricos, ou seja, independentemente de qualquer paradigma e tendo apenas em atenção propriedades desejáveis do *software*, demonstrou-se na secção anterior que o encapsulamento dos dados é uma propriedade fundamental para que se possa atingir a tão desejada **independência de contexto**, que vai, por sua vez garantir propriedades importantes, tais como a facilidade de reutilização, a facilidade de detecção de erros e a modularidade.

Note-se portanto que esta noção de **objecto** está completamente de acordo com a noção de módulo de dados elaborada anteriormente. Tal como a figura permite analisar, um objecto em PPO possui uma estrutura de dados interna e privada, que corresponde à sua representação definida, em geral entre várias possíveis.

A Figura 1.8, que se apresenta a seguir, representa a estrutura e o comportamento de um objecto `complexo1`:

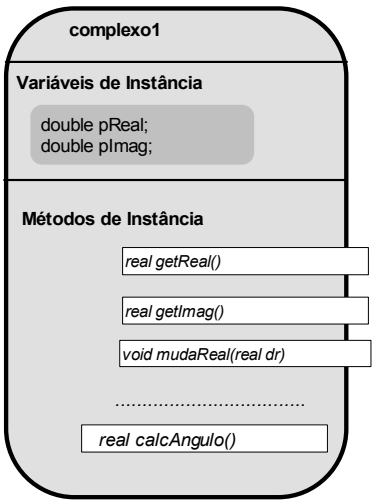

Figura 1.8 – Um objecto

Os métodos privados, dado apenas poderem ser invocados do código de outros métodos do objecto e não a partir do exterior do objecto, funcionam como métodos auxiliares.

Um objecto apresenta-se deste modo como uma **unidade computacional** fechada e autónoma, ou seja um **módulo,** capaz de realizar operações sobre o seu próprio estado interno e devolver respostas para o exterior sempre que os seus métodos definidos como públicos sejam solicitados, isto é, invocados. Ou seja, um objecto é capaz de prestar serviços através da activação dos métodos que foram tornados públicos, serviços que se traduzem no envio de respostas (ou seja resultados) às activações realizadas a partir do seu exterior.

1.5 MENSAGENS

Assim sendo, torna-se desde já importante analisar qual o mecanismo que, em linguagens de PPO, permite que uns objectos possam invocar métodos de outros, desta forma solicitando resultados do comportamento interno de outros objectos.

De facto, por exemplo, em JAVA, os métodos de um objecto não são invocados de forma directa, isto é, não é uma usual e directa invocação de um procedimento. Em PPO, a interacção entre diferentes objectos faz-se através de um mecanismo de **mensagens**. Quando um objecto pretende invocar um método de um outro objecto a que tem acesso, tem que a este enviar a **mensagem** adequada para que tal método seja executado.

Assim, em PPO, em cada computação, ou seja, em cada transição de estado do programa, existe um objecto que é **emissor** de uma mensagem e um outro objecto que é o **receptor** da mesma. A "computação" resulta do facto de que um objecto que recebe uma dada mensagem vai executar, caso tal seja possível, o método que a tal mensagem é por si associado, segundo regras bem definidas, sendo resultados possíveis de tal execução quer a sua alteração de estado interno, quer a devolução de um resultado ao objecto que lhe enviou tal mensagem (Figura 1.9).

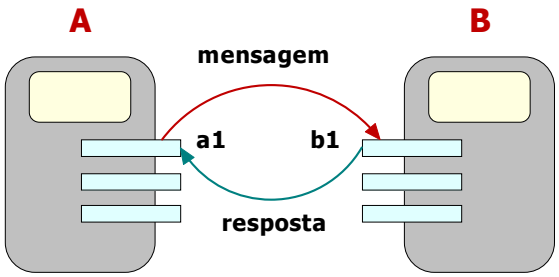

Figura 1.9 – Interacção entre objectos usando mensagens

Tal como a figura procura representar, o envio de uma mensagem de um objecto a outro é realizado durante a execução de um determinado método do emissor, dado que este necessita de um "serviço" particular do receptor para realizar a sua própria computação, que será certamente para ele próprio poder igualmente prestar um serviço solicitado por um outro qualquer objecto.

A Figura 1.9 mostra-nos uma situação comum em que um objecto **A** durante a execução do seu método ***a1()*** necessitou de enviar uma mensagem ao objecto **B**, invocando o método ***b1()***, cujo resultado é fundamental para a continuação da execução do seu método. O objecto **B** executa o método invocado correspondente à mensagem recebida e envia o resultado ao objecto **A** que só então retoma a execução do seu método ***a1()*** que entretanto havia ficado suspenso à espera da recepção de tal resultado.

Desta interacção entre objectos através do envio de mensagens entre si, resulta a computação e as necessárias transições de estado interno dos objectos, estados individuais que representam, no seu somatório, o estado global do programa, que é, portanto, neste paradigma, um estado claramente distribuído.

Na Figura 1.10 representa-se um objecto que implementa uma *stack* de inteiros e que torna públicos os

usuais métodos de inicialização de uma *stack*, introdução de um inteiro, remoção do elemento do topo, consulta do elemento no topo e de determinação da actual dimensão da *stack*.

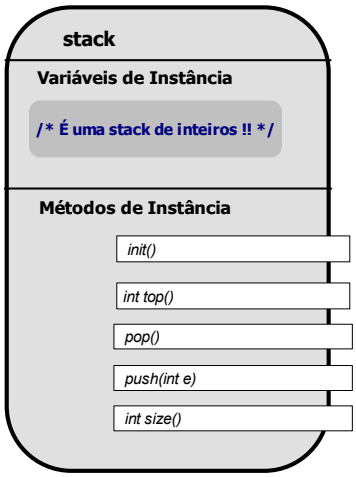

Figura 1.10 – Um objecto *stack* e a sua interface

As mensagens são, como vimos, um mecanismo de acesso indirecto ao código e estado de um dado objecto. Assim, para cada mensagem enviada a um objecto, e em função do identificador e parâmetros da mensagem, é activado, caso exista no objecto receptor, o método de igual identificador ao da mensagem, e compatível em termos do número e tipo dos parâmetros de tal mensagem.

Para além desta reacção de um objecto à recepção de uma mensagem, se da execução do método resultar um dado valor resultado, este é de imediato utilizável pelo objecto emissor da mensagem.

Ainda que de linguagem para linguagem de PPO possam existir ligeiras variações de sintaxe, a sintaxe geral para o envio de uma mensagem a um objecto assume sempre uma das seguintes formas simples e genéricas:

- Envio de uma mensagem sem argumentos a um objecto, sem que haja retorno de resultado pelo método correspondente:

  receptor.mensagem( );

- Envio de uma mensagem com argumentos a um objecto, sem que haja retorno de resultado pelo método correspondente:

  receptor.mensagem(arg1, arg2, ..., argn);

- Envio de uma mensagem sem argumentos a um objecto, havendo retorno de resultado pelo método correspondente:

  resultado = receptor.mensagem();

- Envio de uma mensagem com argumentos a um objecto, havendo retorno de resultado pelo método correspondente:

  resultado = receptor.mensagem(arg1, arg2, ..., argn);

Note-se que nas expressões anteriores, **resultado** e **receptor** representam de forma genérica identificadores de variáveis que, em C++ e em JAVA, que são linguagens de PPO com *type checking*, ao contrário, por exemplo, de Smalltalk, têm tipos de dados a si associados, ou seja, apenas podem referenciar valores de tais tipos. Quanto à expressão **mensagem(arg1, arg2, ..., argn)**, ela representa uma forma de identificação de um método a executar no receptor, método esse que, conforme se disse atrás, deve possuir *o* mesmo nome, ter igual número de parâmetros, sendo os seus parâmetros compatíveis em tipo com os tipos dos argumentos da mensagem, sejam estes valores explícitos (como `10` ou `"abc"`) ou valores contidos em variáveis (como *s*, *x*).

Assim, **mensagens e métodos são entidades distintas**, sendo os métodos invocados através do envio das correspondentes mensagens a um objecto.

Procurando agora concretizar um pouco mais, através de exemplos simples, todo o poder de um mecanismo como o mecanismo de mensagens, único em PPO, e a sua efectiva diferenciação da noção de método, consideremos um objecto *stack* tal como apresentado anteriormente na Figura 1.10.

Conforme publicitado na sua interface, um objecto *stack* de inteiros é capaz de responder às mensagens seguintes, as únicas compatíveis com os métodos tornados públicos:

- **push(int e);**
- **init();**
- **pop();**
- **int top();**
- **int size();**

Admitindo que um qualquer objecto emissor solicitou a um dos vários possíveis objectos que implementam uma *stack* a determinação do seu tamanho, seja no exemplo tal objecto associado à variável *stack1*, tal funcionalidade descrita na Figura 1.10, seria implementada pela expressão seguinte:

```
tam = stack1.size();
```

Caso um emissor apenas pretendesse determinar qual o topo actual de tal *stack*, então a mensagem a enviar ao objecto particular ***stack1***, deveria ser a seguinte:

```
elem = stack1.top();
```

sendo o resultado do envio de tal mensagem, ou seja, o resultado da execução do método **top()** guardado na variável designada por *elem*.

Admitindo que um dado objecto emissor possa ter acesso ao objecto *stack1*, caso o primeiro pretendesse durante a execução de um qualquer método seu modificar o estado interno de *stack1*, por exemplo, inserindo em *stack1* mais um inteiro, então, no código de tal método, deveria aparecer a expressão correspondente à efectiva execução de tal objectivo, ou seja:

```
stack1.push(12);
```

O facto de o **mecanismo de mensagens** ser independente dos próprios métodos dos objectos, confere a este mecanismo um grau de abstracção e generalização de grande interesse, dado que a mesma mensagem poderá ser reconhecida por vários objectos distintos, cada um executando depois o seu próprio método.

Tal confere uma flexibilidade invulgar, muito difícil de conseguir na programação imperativa. Por exemplo, se tivermos objectos distintos que são triângulos, rectângulos ou círculos, faz todo sentido que todos eles respondam à mensagem **desenha**():

```
triangulo1.desenha();
rectang1.desenha();
circulo2. desenha();
```

cada um deles activando o método específico para que se obtenha tal resultado. O valor da mensagem – a **invocação do método** –, é dado pelo receptor. Num paradigma imperativo deveríamos ter uma função para cada tipo de objecto desenhável.

Por outro lado, ao invés do modelo imperativo, se mais tarde criarmos objectos que são, por exemplo, trapézios, quadrados, hexágonos, etc., então estes objectos também poderão responder à mensagem **desenha()**, desde que nas suas classes seja implementado o *seu* método específico **desenha()**.

As mensagens são igualmente um mecanismo de suporte à abstracção, dado que do exterior de um objecto apenas a sua interface deverá ser visível, representando esta o conjunto de mensagens que o objecto

reconhece. No exemplo anterior da *stack*, seríamos capazes de a utilizar sem saber como está efectivamente representada.

O facto de as mensagens serem independentes dos métodos, para além da necessária coincidência nos nomes e argumentos, é também positivo para que os vocabulários das linguagens, ainda por cima de linguagens extensíveis pelo utilizador, sejam de certa forma reduzidos. Por exemplo, se, numa dada linguagem de PPO, for usual que a mensagem enviada a um objecto para se determinar a sua dimensão seja, por exemplo, *size()*, torna-se natural que o programador de novos objectos que possuam características dimensionais siga a regra, e programe um método *size()* que responda a tal mensagem. Desta forma, passa a existir não só uma redução do vocabulário de mensagens e métodos, como também uma natural normalização, o que tem grandes vantagens, principalmente tendo em consideração que estas são em geral linguagens com bibliotecas extensas e que, existindo um claro objectivo de reutilização, tal significa um esforço inicial de conhecimento do que já existe implementado, em geral, muito grande.

1.6 OBJECTOS EM PPO: INSTÂNCIAS E CLASSES

Introduzimos anteriormente a noção de objecto em PPO como sendo uma entidade encapsulada, protegida e unicamente acessível através do que torna público pela sua interface e cujo comportamento definido é acessível do exterior através de um mecanismo de envio de mensagens.

Racionalmente, as únicas formas de garantir que todos os objectos do tipo `Complexo` são realmente iguais em **estrutura** (possuem as mesmas variáveis de instância) e em **comportamento** (possuem o mesmo conjunto de métodos que implementam as eventuais respostas às mensagens recebidas), são as seguintes:

1. Qualquer novo objecto de um dado tipo que se pretenda criar é construído a partir de um outro objecto do mesmo tipo usando um mecanismo de "*copy & paste*", dando-se, em seguida, os valores desejados às variáveis de instância do novo objecto;
2. Objectos de um dado tipo possuem a sua representação estrutural, ou seja, as suas variáveis de instância, e comportamental, ou seja, os seus métodos, guardados num objecto especial que representa a definição de todos os objectos desse tipo, definição padrão que é reproduzida sempre que um novo objecto de tal tipo tem que ser criado.

Ainda que tenham existido linguagens de PPO que tenham implementado a primeira solução, de facto, as principais linguagens de PPO, como Smalltalk, C++ e JAVA, adoptaram a segunda solução.

Nestas linguagens, a forma encontrada para se garantir que todos os objectos de um dado tipo têm igual estrutura e comportamento, foi criar uma "espécie particular de objectos" que guardam a definição de tal estrutura e de tal comportamento.

Estes objectos especiais designam-se em PPO por **Classes**. Assim, as classes são, numa primeira definição, objectos particulares que servem para:

- Conter a descrição da estrutura e comportamento de objectos similares;
- Criar objectos particulares possuindo tal estrutura e comportamento.

Assim, passaremos a designar por **classe** todos os objectos que guardam a estrutura e o comportamento comuns a todos os objectos que a partir de si são identicamente criados, objectos individuais esses que designaremos por **instâncias** (tradução discutível mas estabelecida da respectiva palavra inglesa *"instance"*). Metaforicamente apenas, as classes serão de momento vistas como as "fábricas" ou "moldes" das instâncias, mas será com as instâncias que iremos realizar as computações.

Isto é, a única evolução relativamente a tudo o que até ao momento havia sido dito relativamente ao paradigma da PPO é que ficamos agora a saber que os **objectos** a que até agora nos havíamos referido são de facto **instâncias de uma classe**, sendo tal classe representada por um objecto especial que garante que todas as instâncias a partir de si criadas são coerentes, ou seja, têm igual estrutura e comportamento.

Portanto, se num dado programa necessitarmos de objectos do tipo Ponto2D, que possuem duas variáveis de instância para representar as suas coordenadas inteiras e alguns métodos para a sua manipulação, então,

deveremos construir a classe de nome Ponto2D e, a partir desta, as instâncias de Ponto2D que necessitarmos.

Será com estas instâncias, através do mecanismo de envio de mensagens que invocam os respectivos métodos de tipo público, que realizaremos as computações, quer modificando o estado internos de tais instâncias, quer enviando mensagens que activam métodos que consultam tal estado interno e devolvem resultados.

Tal como definida, numa notação já muito próxima de JAVA, esta classe Ponto2D especifica que qualquer objecto instância de Ponto2D que a partir da mesma possa vir a ser criado, deve conter duas variáveis de instância de nome **x** e **y**, variáveis que apenas poderão conter valores do tipo inteiro e, ainda, que qualquer instância de Ponto2D deverá ser capaz de responder às mensagens *coordX()* e *coordY()*, que terão como resultado, respectivamente, os valores inteiros das coordenadas em **x** e em **y** dos pontos receptores das mesmas, bem como serão capazes de incrementar os seus valores internos em **dx** e **dy** unidades ao receberem as mensagens *incX()* e *incY()*, em cujo caso apenas é alterado o estado interno do objecto, não sendo pois devolvido qualquer resultado, conforme se especifica usando a palavra reservada **`void`** (Figura 1.11).

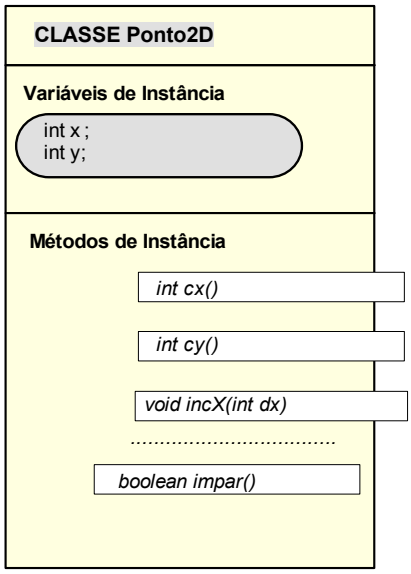

Figura 1.11 – A Classe Ponto2D

Definida a classe `Ponto2D`, poderemos a partir de agora criar objectos que são de facto as instâncias desta classe. Em JAVA tal é realizado usando expressões da forma seguinte:

```
Ponto2D pt1 = new Ponto2D();
Ponto2D pt2 = new Ponto2D();
```

que serão mais tarde completamente analisadas, mas que, conforme facilmente se pode compreender, criam duas instâncias de `Ponto2D` referenciadas por **`pt1`** e **`pt2`**, variáveis que foram declaradas como tendo tal tipo.

Possuindo um objecto do tipo `Ponto2D` referenciado pela variável `pt1`, poderemos, a partir de então, e tal como vimos atrás, enviar a tal objecto, que é uma instância da classe `Ponto2D`, as mensagens que na sua classe foram definidas como sendo as mensagens a que este é capaz de responder, por exemplo, as seguintes:

```
cx = pt1.coordX();
cy = pt1.coordY();
pt1.incX();
pt1.incY();
```

Pela estrutura de cada uma das expressões que se apresenta, torna-se imediato compreender se a mensagem corresponde a uma modificação do estado interno do objecto, tal como `incX()` e `incY()`, em que não há resultado devolvido, ou se se trata da invocação de um método de consulta, em cujo caso é devolvido um resultado (Figura 1.12).

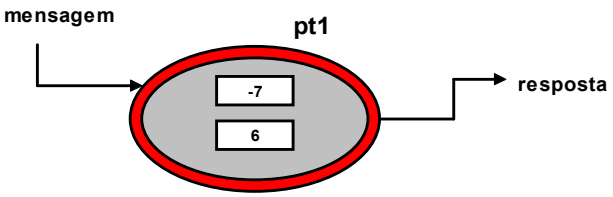

Figura 1.12 – Mensagem/resposta como computação

Torna-se também de imediato claro que, em PPO, qualquer instância possui um só tipo, isto é, é instância de uma e uma só classe.

Como veremos nos capítulos seguintes, as variáveis de instância definidas numa classe podem não só ser de tipo simples, mas também definidas como sendo instâncias de uma dada classe já existente, desta forma permitindo que classes possam ser usadas na criação de outras classes. Por exemplo, tendo a classe `Ponto2D` poderíamos definir uma classe `Circulo` à custa de duas variáveis de instância: uma do tipo `Ponto2D` (o centro) e outra do tipo `double`, o raio.

1.7 SÍNTESE DO CAPÍTULO

Apresentou-se neste capítulo a génese do paradigma da Programação Orientada aos Objectos, com especial incidência, dada a sua importância para a compreensão do paradigma, na evolução dos conceitos na Engenharia de *Software* que vieram a conduzir à noção de **abstracção de dados**, da qual deriva a definição de **objecto**, a entidade fundamental em PPO.

Foram introduzidos alguns princípios de programação que visam garantir que os objectos que desenvolvemos, bem como quem deles faz uso, são unidades que, por serem independentes do contexto, facilitam a manutenção e a reutilização. Duas regras foram consideradas como fundamentais:

- Um objecto não deve manipular directamente os dados internos de outro;
- Um objecto genérico não deve ter instruções de *input/output* no seu código.

Assim, a visão de criação e utilização de um objecto é a de uma **cápsula** contendo dados privados e que disponibiliza, através da sua **interface**, um conjunto de métodos que são accionáveis do exterior através de um mecanismo de **mensagens**.

Esta perspectiva favorece uma outra comum interpretação do que se deve entender por objectos, apresentando-os como "caixas pretas" que prestam um conjunto de "serviços" ao exterior ao receberem mensagens que lhes sejam apropriadas. Esta visão permite ainda que, por vezes, se refira o objecto que envia a mensagem como "objecto-emissor" ou "objecto-cliente", e o que a recebe e, eventualmente, responde à mesma como **receptor** ou "servidor".

A computação num sistema de objectos decorre da troca de mensagens entre os vários objectos, dessa troca resultando alterações nos seus estados internos e, consequentemente, do estado global do programa, que, neste modelo computacional, é um estado distribuído por todos os objectos que o constituem.

Finalmente, foi realizada a distinção entre dois tipos de objectos, **instâncias** e **classes**. **Classes** são objectos especiais onde se definem estrutural e comportamentalmente todos os objectos instância criados a partir de tal classe. As **instâncias** serão as nossas entidades activas, computacionais, pois respondem a mensagens activando métodos e realizando as computações programadas em tais métodos.

No próximo capítulo, introduziremos a tecnologia JAVA e o núcleo básico da linguagem JAVA5 que iremos posteriormente utilizar e enriquecer nos capítulos seguintes. No capítulo 3 estudaremos de forma aprofundada como criar classes e instâncias em JAVA.