View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade do Minho: RepositoriUM

C++ é inadequado para ensinar OO

Miguel Pessoa Monteiro
Escola Superior de Tecnologia, Instituto Politécnico de Castelo Branco
Castelo Branco

João Miguel Fernandes
Dep. Informática, Universidade do Minho
Braga

1. Introdução

Num curso superior de Informática não deve haver preocupação em ensinar a última novidade tecnológica, mas antes os fundamentos que permitam aos diplomados uma capacidade para aprender continuamente [1]. No ensino da programação orientada ao objecto (POO) é importante utilizar-se uma linguagem com suporte coerente dos conceitos fundamentais do paradigma, sem características que entrem em contradição com o modelo central, sob pena da linguagem se tornar um obstáculo à aprendizagem.

Neste artigo argumenta-se precisamente que C++ é um obstáculo à aprendizagem da POO. Muitos alunos acreditam que aprender C++ lhes dará todas as competências na área da programação de que necessitarão no futuro. Esta visão errada tem como consequência que poderão ficar pior preparados para se adaptarem às mudanças a que a informática está sujeita [2].

C++ foi criado tendo em vista a compatibilidade com o C, pelo que todas as decisões de concepção do C++ estiveram dependentes das características, boas ou más, da linguagem anterior. Só este facto já permite questionar se tudo o que o C++ oferece será coerente, sobretudo se se considerar que o próprio C, por vezes classificado de "linguagem assembly estruturada", dá suporte a vários paradigmas da programação. Na verdade C++ não é uma linguagem orientada ao objecto (LOO), mas uma **linguagem multiparadigma**, dotada dum suporte imperfeito dos conceitos da OO, misturados com conceitos oriundos de modelos de programação mais antigos.

A seguir alude-se a alguns dos principais aspectos técnicos em que C++ apresenta imperfeições no suporte à POO. São feitas comparações com algumas das LOOs mais importantes, sobretudo Eiffel e Java.

2. Encapsulamento vs. ocultação

Há muita confusão entre os conceitos de **encapsulamento** e **ocultação** (*data hiding* e *implementation hiding*), e C++ não ajuda a desfazê-la. "Encapsulamento" tem sido considerado erradamente como sinónimo de ocultação de dados e implementação interna, mas na realidade significa que os componentes (definições, dados e rotinas) são "metidos numa cápsula" (classe, na terminologia da POO). Uma parte desses componentes tem que ser visível fora das instâncias dessa classe, pois caso contrário os objectos seriam como caixas pretas sem comunicação com o exterior. As partes visíveis constituem a **interface** do objectos.

Na POO não existem entidades globais a toda a aplicação – todos os componentes são encapsulados em classes, mesmo que alguns sejam públicos. Não é o que sucede em C++, onde tudo pode ser global – um programa em C++ pode ser completamente desprovido de encapsulamento.

Ocultação de implementação significa que a maneira como determinada propriedade é realizada não é visível fora das instâncias da classe. Assim, quando uma propriedade dum objecto é acessível apenas para leitura, não deve transparecer no texto fonte se esta é realizada através duma variável, uma constante ou um método sem argumentos. A propriedade pode ser pública, apenas a sua realização e mecanismo de acesso não o devem ser. Porém a sintaxe do C++ impede isso – por exemplo, o acesso a uma propriedade "total", realizada através de 3 maneiras – constante, variável e rotina – tem o seguinte texto:

```
obj.total          ou          obj->total
obj.TOTAL          ou          obj->TOTAL
obj.total()        ou          obj->total()
```

A 2ª linha resulta da convenção do C de representar constantes em maiúsculas. Na 3ª linha também há diferenças pois é obrigatório colocar parêntesis nas funções, mesmo quando não recebem argumentos. Distinções sintácticas deste género entram em conflito com o conceito de ocultação da POO.

Vemos 2 colunas porque C++ apresenta 2 mecanismos de acesso aos objectos, através de *alias* e de apontadores. Podemos constatar que estes mecanismos têm sintaxes diferentes: também aqui a realização não é ocultada, facto que constitui mais um obstáculo à reutilização de código.

Java não corrigiu estes aspectos sintácticos, mas fornece um mecanismo único para o acesso aos objectos, eliminando o uso de apontadores. Apenas a coluna da esquerda se lhe aplica. Infelizmente esta sintaxe induz muitos programadores a pensar que ocultação significa ocultar atributos e apenas permitir a visibilidade dos nomes e **assinaturas** (lista de argumentos) dos métodos.

3. Abstracção

Ao contrário das variáveis em linguagens procedimentais, geralmente os objectos não são processados através de nomes que representam directamente a zona de memória onde se encontram. Em lugar disso, é utilizada a **referência**, que dá o acesso **autorizado** ao objecto. Correspondem a apontadores cujo mecanismo de acesso se encontra oculto. Esta ocultação permite montar infra-estruturas de suporte aos objectos (incluindo um colector de lixo).

Cada referência possui um tipo associado, que condiciona os objectos que podem ser referenciados (apenas os da mesma classe ou das suas ascendentes) e determina as propriedades do objecto que podem ser usadas. Assim, é o tipo da referência que determina as propriedades do objecto que podem ser usadas, não o tipo do objecto. Se um objecto é usado através duma referência duma classe ascendente, então apenas podem ser usadas as propriedades do objecto definidas nessa classe ascendentes.

É importante frisar que os objectos não sofrem transformações pelo facto de serem referenciados por referências duma classe diferente: uma vez criado, um objecto nunca muda de classe ou de propriedades. No entanto os objectos podem ser referenciados por referências de diversas classes da mesma hierarquia de herança, que determinam a vista que podemos ter do objecto. É este o significado dado à abstracção – protecção do texto fonte contra uso de propriedades que não as declaradas no tipo da referência usada para acesso.

C++ não possui referências – mantém o mecanismo dos apontadores que herdou do C, e introduz um mecanismo diferente, o qual confusamente também chamou referência (ou *alias*, que é mais correcto).

4. Identidade dos objectos

Segundo a POO cada objecto possui identidade própria, independente dos valores do seu estado interno. Este conceito é um dos mais fundamentais da POO, através do qual é possível objectos conterem ou referenciarem outros objectos, ou um objecto ser simultaneamente referenciado por vários outros objectos.

Para dar suporte à identidade, uma LOO deve dar suporte às operações associadas, nomeadamente os testes **identidade** (se 2 objectos são o mesmo), **igualdade** (se os 2 objectos têm valores iguais), **cópia forte** (*deep copy* – produzir novo objecto cujo estado interno tem valores iguais ao do original) e **cópia fraca** (*shallow copy* – produzir nova referência ao mesmo objecto). Este suporte assenta no mecanismo de acesso aos objectos que é a referência.

Ambos os mecanismos do C++ para acesso a objectos, apontadores do C e *aliases*, têm problemas ou limitações importantes.

Um *alias* do C++ constitui um simples nome alternativo para um objecto, que funciona como o nome duma variável das linguagens procedimentais. Um *alias* não pode ser processado enquanto referência *per si*, ao contrário, por exemplo, duma referência do Java. Através do *alias* não obtemos suporte ao teste de identidade. Por exemplo, o teste seguinte tem semânticas diferentes em C++ e em Java:

```
if(obj1 == obj2) // ...
```

Em Java são as próprias referências que são comparadas, dando assim suporte ao teste da identidade. Em C++, obj1 e obj2 da expressão acima têm de ser apontadores. Usando *alias*, a expressão acima nem sequer é permitida, a menos que sobrecarregássemos expressamente a semântica do operador ==. Ao contrário do Java, C++ não fornece uma realização por defeito de ==. O principal benefício que os *alias* trazem ao C++ é a possibilidade de evitar apontadores na passagem de objectos a funções.

Em C++, o teste da identidade obtém-se comparando endereços de memória, um conceito de baixo nível e não intrínseco da identidade. Essa comparação é efectuada da seguinte forma (supondo que obj1 e obj2 são variáveis, e não apontadores):

```
if(&obj1 == &obj2) // ...
```

Assim, o suporte do C++ à identidade é limitado, e dificulta a aprendizagem do conceito – como pode um professor convencer alunos de que a POO introduz um conceito novo, quando tudo o que eles vêm é o mesmo mecanismo dos apontadores, que já conhecem do C, e um outro mecanismo que se limita a introduzir nomes alternativos?

5. Colector de lixo

O **colector de lixo** é um mecanismo que liberta o programador de quaisquer tarefas relacionadas com a libertação de memória, através da remoção implícita dos **objectos mortos** (objectos com zero referências no sistema), que são automaticamente varridos do sistema. Por ter um colector de lixo, Java simplificou consideravelmente a programação, em relação ao C++, e também lhe permitiu prescindir dos destrutores. Eiffel e Smalltalk também possuem colectores de lixo. Em C++ o programador é obrigado a efectuar manualmente tarefas que em outras linguagens são efectuadas automaticamente pelo colector de lixo ([3], pág. 34).

O funcionamento dum colector de lixo pressupõe a capacidade de detectar os objectos mortos. Como cada objecto pode referenciar vários outros objectos, o sistema tem de ser capaz de manter grafos de dependências dos diversos objectos e lidar com as referências circulares. Isto implica a manutenção duma tabela de referências que tem de ser actualizada sempre que um novo objecto é criado ou uma referência desaparece ou muda de valor.

Stroustrup (criador do C++) afirma que concebeu deliberadamente C++ para não depender dum colector de lixo, dadas as circunstâncias da altura. Porém admite que as circunstâncias mudaram, nomeadamente a maior dimensão dos projectos desenvolvidos em C++ e a melhor tecnologia dos colectores de lixo. Stroustrup conclui que C++ acabará fatalmente por incluir um colector de lixo **opcional** ([4], págs. 221 e 222), mas há dúvidas quanto à viabilidade dessa opção ([3], pág. 35; [5], pág. 1136).

Se o programador tiver acesso aos objectos sem ser por intermédio da infraestrutura de suporte ao colector de lixo, poderá causar incoerências graves no sistema. As estratégias para o funcionamento dum colector de lixo pressupõem que os objectos podem ser automaticamente deslocados duma zona de memória para outra, sem o programador se aperceber disso. Como lidar nesses casos com os valores dos apontadores para esses objectos? E como saberia o sistema detectar os apontadores que teriam de ser actualizados?

O exemplo do teste à identidade acima apresentado, usando endereços de memória, permite ilustrar as dificuldades que os apontadores colocam: num caso em que um colector de lixo seja realizado através dum fio de execução (*thread*) funcionando em ambiente preemptivo, o colector pode entrar em acção entre a obtenção dos 2 endereços a comparar, e transferir os objectos para outra

localização de memória, falseando o resultado da comparação. Já com referências é possível lidar com estas situações.

6. Sobrecarga

Em C++ a sobrecarga (*overloading*) e a sobreposição (*overriding*) são muitas vezes confundidas. A primeira consiste em permitir, dentro da mesma classe, métodos diferentes com o mesmo nome, mas necessariamente com assinaturas diferentes. A segunda consiste em permitir que uma classe descendente possa redefinir um método herdado duma das suas descendentes, mantendo o nome e assinatura. A redefinição é fundamental na POO, mas a sobrecarga é desnecessária, e é mesmo prejudicial pela confusão de conceitos que provoca. O próprio Stroustrup não aconselha o seu uso ([6], pág. 149).

Temos experiência concreta, em exames de programação, de que o conceito de sobrecarga interfere na compreensão de outros conceitos. Num caso concreto recorreu-se ao seguinte exemplo para justificar a utilidade da sobrecarga, para perguntar porque razão o exemplo não seguia o modelo de objectos e mensagens e como seria uma solução que o seguisse ([7], págs. 444-445):

```
int max(int, int);
int max(const vector<int> & );
int max(const matrix &);
```

A incongruência, que um número significativo de alunos não logrou detectar, é que no exemplo os dados principais são passados às funções como argumentos, quando tal deveria ser desnecessário, pois deveriam constituir o estado interno de objectos. As mensagens deveriam limitar-se a passar argumentos auxiliares, como o segundo inteiro no 1º exemplo acima. O exemplo acima, no qual a totalidade dos dados é passada como argumentos, segue o modelo procedimental.

Segundo a POO os dados constituem o estado interno do objecto e todos os métodos (excepto os globais à classe) recebem um argumento escondido (`this` em C++ e Java, `Current` em Eiffel, `self` em Smalltalk) que representa o objecto ao qual se destina a mensagem. Por este motivo a passagem de argumentos é menos frequente nas mensagens enviadas a objectos do que nos procedimentos. Os casos em que valores dos argumentos são alterados como efeito lateral são raros.

Java não eliminou a sobrecarga sintáctica de métodos, mas existe menos motivação para a usar (todos os dados se encontram encapsulados em classes). Eiffel não permite sobrecarga de rotinas.

7. Polimorfismo

O **polimorfismo** é um dos conceitos fundamentais da POO, mas não é fácil de apreender, e é no seu suporte que C++ tem a falha mais grave. Para que uma linguagem lhe dê suporte, tem de possuir 2 mecanismos associados – sobreposição (*overriding*) e selecção dinâmica (*dynamic binding* ou *late binding*). Pela maneira como dá suporte à selecção dinâmica C++ dificulta muito a aprendizagem desses conceitos.

Há **sobreposição** se as classes descendentes podem redefinir propriedades (geralmente métodos) herdadas de classes ascendentes, mantendo os mesmos nomes e assinaturas. A mesma operação pode assim ser realizada em várias classes da hierarquia de herança, sendo cada realização feita de acordo com as especificidades da respectiva classe.

Existe **selecção dinâmica** quando o método correspondente à mensagem que o objecto recebeu é seleccionado em tempo de execução, em lugar de ser previamente determinado pelo compilador, através do código gerado. É importante notar que o método a seleccionar depende do tipo do objecto, e não da referência usada para enviar a mensagem ao objecto.

As referências determinam, em função dos seus tipos, as propriedades dos objectos que são autorizadas (abstracção), mas nas situações em que existem várias realizações ao longo da hierarquia de herança não devem condicionar a selecção do método que realiza essa propriedade. O método a

seleccionar é sempre o definido para a classe do objecto, mesmo quando a referência usada é duma classe ascendente, e essa classe apresenta uma realização diferente desse método.

O polimorfismo é a capacidade dum sistema funcionar com objectos de diversas classes, seleccionando automaticamente a realização das propriedades do objecto concreto, consoante a classe desse objecto. Através do polimorfismo os programas podem ser **extensíveis**: se for posteriormente acrescentada na hierarquia de herança uma nova classe descendente, as partes anteriores do programa funcionarão correctamente também com a nova classe. As realizações das propriedades que a nova classe tenha redefinido são seleccionadas automaticamente, sem necessidade de nova compilação. Não havendo redefinição duma propriedade, é seleccionada a realização herdada.

Porém C++ usa, por defeito, a **selecção estática** – o método seleccionado não é o da classe do objecto, mas o da classe do *alias*/apontador. O compilador determina o método chamado logo na fase de compilação, e gera o código correspondente à sua chamada. O único método sobre o qual o compilador pode ter informação é o da classe associada ao apontador ou *alias*.

C++ permite a selecção dinâmica dum método apenas se, na primeira classe da hierarquia em que é declarado, este for precedido duma palavra reservada, **virtual**. A motivação para esta regra era a eficiência – a selecção dinâmica implica a consulta duma tabela em tempo de execução para a chamada do método certo, o que poderá ter algum impacto no desempenho do programa, enquanto que a selecção estática geralmente consiste num mero salto num endereço.

O facto de ser necessário incluir uma palavra extra para obter o resultado correcto é causador de inúmeras situações de confusão, particularmente entre estudantes. Esta decisão de concepção tem sido condenada em termos inequívocos por diversos especialistas ([3], pág. 9; [5], págs. 513 e 514).

8. Linguagens Alternativas

Dada a rápida evolução da informática, nenhuma linguagem tem assegurada durante muito tempo uma posição de domínio – quaisquer recomendações correm o risco de ficar rapidamente desactualizadas. Por exemplo, Java, apesar da posição que conquistou, e até por causa dela, já enfrenta o seu primeiro rival na figura de C#, uma linguagem criada pela Microsoft destinada a apoiar a sua plataforma proprietária de serviços na Internet, **.Net**.

Actualmente as LOOs que aconselhamos para serem usadas como ferramenta pedagógica do ensino da POO são Eiffel e Java.

Referências bibliográficas

[1] D. L. Parnas, *Education for Computing Professionals*. IEEE Computer, 23(1):17-22, 1990.

[2] E. A. Lee and D. G. Messerschmitt,. *Engineering an Education for the Future*. IEEE Computer, 31(1):77-85, 1998.

[3] I. Joyner, *C++?? A Critique of C++ and programming and Language Trends of the 1990s, 3rd edition*. www.elj.com, 1996

[4] B. Stroustrup, *The design and evolution of C++*, Addison Wesley, 1994.

[5] B. Meyer, *Object-Oriented Software Construction, 2nd Edition*, Prentice Hall, 2000.

[6] B. Stroustrup, *The C++ programming language*, Addison Wesley, 1997.

[7] S. B. Lippman, J. Lajoie, *C++ Primer, 3rd edition*, Addison Wesley, 1998.