3.2. Classe e Objeto

Nesta seção serão descritos dois componentes básicos de uma LP OO: classes e objetos. Esta terminologia teve origem em *Simula* e *Smalltalk* e não é usual em linguagens imperativas ou funcionais, por isso parte do objetivo desta seção é relacionar estes termos com os construtores das linguagens mais conhecidas.

Como já visto, simplificadamente, um **objeto** é algo que ocupa memória e possui um estado (modificável). Esta é, essencialmente, a definição de um objeto numa linguagem orientada a objetos, mas com algumas modificações. Primeira, o estado de um objeto é inicialmente interno ou local ao próprio objeto. Em outras palavras, o estado de um objeto é representado por **atributos** locais declarados como parte do objeto e inacessíveis para componentes fora do objeto. Além disso, cada objeto inclui um conjunto de operações através dos quais o estado local pode ser acessado e modificado (chamados de **métodos**). A diferença entre métodos e procedimentos/funções, é que eles podem automaticamente acessar o objeto de dado, ao contrário de procedimentos e funções externas, e podem ser vistos como contendo um parâmetro implícito representando o próprio objeto. A chamada de um método corresponde a enviar mensagens a objetos, isto é, mensagens são chamadas a métodos. A terminologia "invocar um método" ou "chamar um método" também é utilizada. Toda coleção de métodos de um objeto pode ser chamada de interface de mensagens do objeto.

Objetos podem ser declarados através da criação de um padrão para os seus métodos e estados locais. Este padrão é chamado de **classe**, e é essencialmente igual a um tipo de dado. Em algumas LP OO, como por exemplo C++, classes são incorporadas no sistema de tipos da linguagem. Objetos podem, então, ser declarados como pertencendo a uma classe particular, exatamente como as variáveis são declaradas em uma linguagem como *C* e *Pascal*.

Um objeto também é chamado de **instância** de uma classe. Membros de uma classe, isto é, seus atributos e métodos, podem ser classificados como sendo públicos, protegidos ou privados. Através destes mecanismos de proteção, o programador pode determinar quais partes de uma classe vão ficar escondidas, ou protegidas. Em outras palavras, quando os membros forem declarados como privados, eles poderão ser usados apenas por outros membros da mesma classe; membros protegidos podem ser usados por membros nesta classe e em qualquer subclasse; membros públicos podem ser utilizados por qualquer instrução interna ou externa à classe (figura 3.7).

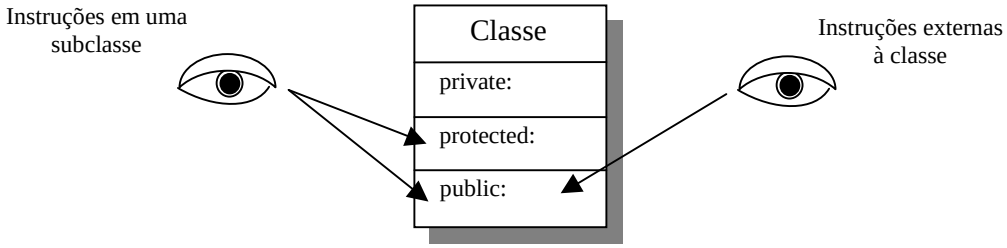

Figura 3.7 – Membros privados são usados exclusivamente por suas classes

3.3. Hierarquia de classes: Herança e Reusabilidade

De meados até o final da década de 80, tornou-se claro para muitos desenvolvedores de sistemas que uma das melhores maneiras de aumentar a produtividade era a reutilização de *software*. Os tipos abstratos de dados, com o encapsulamento e controle de acesso, eram as unidades ideais para serem reutilizadas. Porém, ocorria o problema de que as características e capacidades dos tipos existentes não eram adequadas para o novo uso. O tipo antigo requeria pelo menos algumas pequenas modificações. Tais modificações podem ser difíceis em um código existente. Além disso, em muitos casos as modificações requerem alterações em todos programas clientes. Um segundo problema é que todas as definições de tipos abstratos de dados são independentes e estão no mesmo nível. Isto geralmente impossibilita a estruturação do programa para solucionar o problema proposto. Em muitos casos, os

problemas possuem categorias de objetos relacionados, tais como "irmãos" (similares entre si) e "pais e filhos" (com algum tipo de relacionamento subordinado).

A herança oferece uma solução para estes problemas de reutilização e organização dos programas, permitindo o compartilhamento de dados e operações entre classes. Se um novo tipo abstrato de dado pode herdar os dados e a funcionalidade de alguns tipos existentes, e também permite modificar algumas destas entidades, a reutilização é facilitada sem haver necessidade de fazer alterações. É possível pegar um tipo abstrato de dado existente e "moldá-lo" para preencher as novas necessidades.

Os tipos abstratos de dados, seguindo a linguagem *Simula 67* que deu origem à programação orientada a objetos, são geralmente chamados de classes, e instâncias das classes são chamadas de objetos. Uma classe que é definida através da herança a partir de outra classe é chamada de **classe derivada** ou **subclasse**. Um classe a partir da qual novas classes são derivadas é chamada de **classe pai**, **classe base** ou **superclasse**. Por exemplo, em *Simula*, uma classe B pode herdar alguns ou todos os atributos de objeto e métodos de outra classe A através da declaração desta classe na sua definição, o que é feito listando-se A antes de B:

```
A class B;
begin
        ...
end;
```

Neste caso, B, que é subclasse de A, herda todos os atributos e métodos de A, que é a superclasse de B [LOU 93, SEB 99].

No caso mais simples, uma classe herda todas as entidades (atributos e métodos) da classe pai. Entretanto, pode haver uma restrição através de mecanismos de controle de acesso às entidades da classe pai [SEB 99]. Para exemplificar o uso destes mecanismos, em *C++* declarar a superclasse como pública em uma subclasse significa que as propriedades herdadas devem reter o seu *status* original. Em outras palavras, se a superclasse é *public*, membros públicos desta classe permanecem públicos dentro da subclasse e podem ser acessados por qualquer outra função do programa. Da mesma forma, membros privados da superclasse permanecem privados, isto é, só podem ser acessados na própria superclasse, e membros protegidos permanecem protegidos na subclasse. Declarar a superclasse como *protected* em uma subclasse significa que seus membros públicos e protegidos podem ser usados pela subclasse, e outras classes derivadas da subclasse. Porém, os membros protegidos não podem ser usados por funções externas a classe. Em outras palavras todos os membros públicos e protegidos da superclasse tornam-se membros protegidos da subclasse. Também pode-se declarar a superclasse como *private*, o que tornaria todos os membros herdados privados não importando qual era o seu *status* original. Neste caso, todos os membros públicos e protegidos da superclasse tornam-se membros privados da subclasse, ou seja, uma classe que se derive da classe recém derivada não pode usar os membros da superclasse original, mesmo que esses campos fossem públicos originalmente [SWA 93]. Já em *Simula*, ao contrário de *C++*, não há mecanismos de proteção para métodos ou atributos, ou seja, todos os métodos estão disponíveis para classes derivadas, e classes derivadas são sempre subtipos (esta é uma justificativa para o uso dos termos subclasse e superclasse) [LOU 93].

O trecho de programa a seguir, cujo código equivalente na linguagem de programação *Java* é apresentado posteriormente, ilustra a utilização de herança em *C++*.

```
// Exemplo de Herança em C++

class VeiculoEstrada {        // Superclasse
        int rodas;
        int passageiros;
   public:
        void DefineRodas(int num);
        int PegaRodas();
        void DefineNumPassageiros(int num);
        int PegaNumPassageiros();
};
```

```
class Caminhao : public VeiculoEstrada {          // Subclasse
        int carga;
    public:
        void DefineCarga(int tamanho);
        int PegaCarga();
        void Mostrar();
};

class Automovel : public VeiculoEstrada {         // Subclasse
        int carro_tipo;
    public:
        void DefineTipo(int t);
        int PegaTipo();
        void Mostrar();
};

main() {
        Caminhao c1;        // Instância da classe Caminhao
        Automovel a1;       // Instância da classe Automovel
        c1. DefineRodas(18);
        c1. DefineNumPassageiros(2);
        c1. DefineCarga(int tamanho);
        c1.Mostrar();
        a1. DefineRodas(4);
        a1. DefineNumPassageiros(5);
        a1.DefineTipo(2);
        a1.Mostrar();
}
```

Torna-se interessante comentar que existe um tipo de classe, chamada **classe abstrata**, que tem como função agrupar classes relacionadas, provendo uma "raiz comum" a partir da qual uma série de classes mais especializadas podem ser derivadas. As classes abstratas servem para agrupar os métodos e elementos de dados que são comuns a todas as subclasses, mas elas não trabalham realmente e não são completas o suficiente para operar como entidades independentes. Em outras palavras não é possível gerar um objeto a partir de uma classe abstrata. A maneira mais fácil de identificar se uma classe é abstrata é se ela é usada. Se for necessário criar uma classe, mas são usadas apenas as suas subclasses, então esta classe é uma classe abstrata. *Ada* suporta esta característica através do uso da palavra chave *abstract*, e *C++* através do uso de funções puramente virtuais [MUL 89, GHE 97]. Em *Java* também é usada a palavra chave *abstract*, e toda classe que contiver pelo menos um método abstrato é considerada uma classe abstrata e deve ser declarada como tal.

Além dos membros herdados da superclasse, uma subclasse pode tanto adicionar novos membros, como modificar os métodos herdados. Um método modificado tem o mesmo nome, e freqüentemente os mesmos parâmetros, do método original. Diz-se, então, que o novo método sobrescreve a versão herdada (*overloading*), geralmente com o objetivo de fornecer uma operação que é específica para objetos da subclasse, mas não é apropriada para objetos da classe pai.

Se uma classe criada através de herança possui uma única superclasse, então o processo é chamado de **herança simples**. Se a classe possui mais do que uma superclasse, como em *C++* e *Eiffel*, o processo é chamado de **herança múltipla**. Quando um número de classes são relacionadas através de herança simples, seus relacionamentos são mostrados através de uma árvore de derivação. Os relacionamentos entre as classes com herança múltipla podem ser mostrados em um grafo de derivação (figura 3.8).

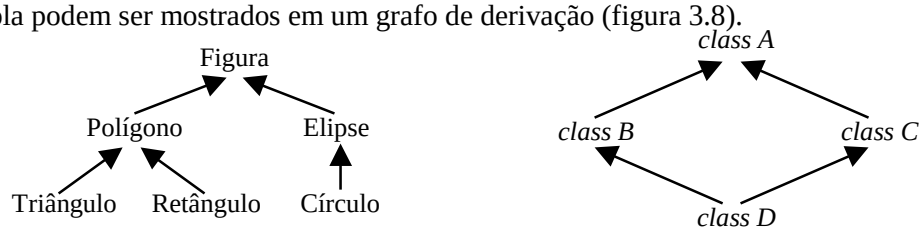

Figura 3.8 – Gráficos de herança (setas apontam da subclasse para a superclasse) [LOU 93]

A herança múltipla não é incluída em muitas linguagens de programação, como por exemplo *Java*, basicamente, por dois motivos: complexidade e eficiência. A complexidade adicional é ilustrada por diversos problemas, tal como a colisão de nomes. Isto ocorre, por exemplo, quando uma subclasse D herda tanto da classe B como da C, e tanto B como C incluem um atributo ou método com o mesmo nome. Um outro problema ocorre se tanto B como C forem derivados da mesma superclasse A (figura 3.8 à direita). Assim, tanto B como C têm atributos hereditários de A e D herda duas versões de cada um deles (de B e C). A questão da eficiência é mais "visual" do que real. No C++, por exemplo, o suporte à herança múltipla exige apenas uma operação extra para cada método vinculado dinamicamente. Não obstante isso seja necessário mesmo que o programa não use herança múltipla, é um pequeno custo adicional. Além disso, a manutenção de sistemas que usam herança múltipla pode ser um grande problema, uma vez que leva a dependências complexas entre classes.

O projeto de um sistema orientado a objetos começa com a definição de uma hierarquia de classes que descreva as relações dos objetos que irão fazer parte do programa. Uma desvantagem da herança como meio de aumentar a possibilidade de reutilização é que ela cria uma dependência entre as classes em uma hierarquia. Isto trabalha contra uma das vantagens dos tipos abstratos de dados, que é a independência entre eles. É claro que nem todos os tipos de dados abstratos devem ser completamente independentes, mas em geral a independência é uma das suas principais características. Porém, pode ser difícil, se não impossível, aumentar a capacidade de reutilização de tipos de dados abstratos sem criar dependências entre alguns deles [SEB 00].