

## Capítulo 5

### Aspectos de reutilização

*“Por que software não é como hardware? Por que todo desenvolvimento começa do nada? Deviam existir catálogos de módulos de software, assim como existem catálogos de chips: quando nós construímos um novo sistema, nós deveríamos estar usando os componentes destes catálogos e combinando-os, em vez de sempre reinventar a roda. Nós escreveríamos menos software, a talvez faríamos um desenvolvimento melhor. Será que alguns problemas dos quais todo mundo reclama - custos altos, prazos insuficientes, pouca confiabilidade - não desapareceriam? Por que não é assim?”*

Talvez você já tenha ouvido esta argumentação antes; talvez você próprio já tenha pensado nisso. Em 1968, no famoso workshop da OTAN sobre a crise de software, D. McIlroy já estava pregando a produção de componentes de software em massa. A reutilização, como um sonho, não é nova.

Qualquer pessoa que lide com o desenvolvimento de software se impressiona com seu caráter repetitivo. Diversas vezes, os programadores constroem funções e programas com o mesmo padrão: ordenação, busca de um elemento, comparação, etc. Uma maneira de interessante de avaliar esta situação é responder a seguinte pergunta: Quantas vezes, nos últimos seis meses, você escreveu uma rotina de busca de um elemento x em uma tabela t?

As dificuldades técnicas de reutilização se tornam mais visíveis quando se observa a natureza das repetições no desenvolvimento de sistemas. Esta análise revela que apesar dos programadores tenderem a escrever os mesmos tipos de rotinas diversas vezes, estas não são exatamente iguais. Se fosse, a solução mais simples teoricamente; na prática, porém, muitos detalhes mudam de implementação para implementação (tipos dos elementos, estrutura de dados associada, etc.).

Apesar das dificuldades, algumas soluções foram propostas com relativos sucessos:

**Reutilização de código-fonte:** Muito comum no meio científico. Muito da cultura UNIX foi difundida pelos laboratórios e universidades do mundo graças à disponibilidade de código-fonte ajudando estudantes a estudarem, imitarem e estenderem o sistema. No entanto, esta não é a forma mais utilizada nos meios industrial e comercial além de que esta técnica não suporta ocultação de informação (information hiding).

**Reutilização de pessoal:** É uma forma muito comum na indústria. Consiste na transferência de engenheiros de software de projetos a projetos fazendo a permanência de know-how na companhia e assegurando a aplicação de experiências passadas em novos projetos. Obviamente, esta é uma maneira não-técnica e limitada.

**Reutilização de design:** A idéia por trás desta técnica é que as companhias devem acumular repositórios de idéias descrevendo designs utilizados para os tipos de aplicação mais comuns.

### Requisitos para reuso

As idéias apresentadas anteriormente, apesar de limitadas, mostram aspectos importantes para a reutilização de código:

A noção de reuso de código-fonte lembra que software é definido pelo seu código. Uma política satisfatória de reutilização deve produzir programas (códigos) reutilizáveis.

A reutilização de pessoal é fundamental pois os componentes de software são inúteis sem profissionais bem treinados e com experiência para reconhecer as situações com possibilidade de reuso.

A reutilização de design enfatiza a necessidade de componentes reutilizáveis que estejam em um nível conceitual e de generalidade alto — não somente com soluções para problemas específicos. Neste aspecto, poderá ser visto como o conceito de classes nas linguagens orientadas por objetos pode ser visto como módulos de design e de implementação.

A maneira de produzir módulos que permitem boa possibilidade de reuso é descrita abaixo. Neste caso, o nosso exemplo de procurar um elemento x em uma tabela t é bem ilustrativo.

### Variação no tipo

Um módulo que implemente uma determinada funcionalidade deve ser capaz de fazê-lo sobre qualquer tipo a ele atribuído. Por exemplo, no caso da busca de um elemento x, o módulo deve ser aplicável a diferentes instâncias de tipo para x. É interessante a utilização do mesmo módulo para procurar um inteiro numa tabela de inteiros ou o registro de um empregado na sua tabela correspondente, etc.

## ***Variação nas estruturas de dados e algoritmos***

No caso da busca de um elemento  $x$ , o modo de busca pode ser adaptado para diversos tipos de estruturas de dados e algoritmos de busca associados: tabelas sequenciais (ordenadas ou não), vetores, listas, árvores binárias, B-trees, diferentes estruturas de arquivos, etc. Neste sentido, o módulo deve suportar variações nas estruturas a ele associado.

## ***Existência de rotinas relacionadas***

De forma a fazer uma pesquisa em uma tabela, deve-se saber como esta é criada, como os elementos podem ser inseridos, retirados, etc. Deste modo, uma rotina de busca não é por si só suficiente; há a necessidade do acoplamento de diversas rotinas primitivas e relacionadas entre si.

## ***Independência de representação***

Uma estrutura modular verdadeiramente flexível habilita seus clientes uma operação sem o conhecimento de modo pelo qual o módulo foi implementado. Por exemplo, deve ser possível ao cliente escrever a seguinte chamada para a busca de  $x$ :

```
esta_presente = BUSCA( x, T );
```

sem saber qual o tipo da tabela  $T$  no momento da chamada. Se vários algoritmos de busca foram implementados, os mecanismos internos do módulo são responsáveis de saber qual o apropriado sem a intervenção do cliente. De maneira simplificada, isto é uma extensão do princípio de ocultação de informação pois havendo a necessidade de mudança na implementação, os clientes estão protegidos.

No entanto, a idéia vai mais além. O princípio da independência de representação não significa somente que mudanças na representação devem ser invisíveis para os clientes durante o ciclo de desenvolvimento do sistema: os clientes devem ser imunes também a mudanças durante a execução. No exemplo acima, é desejável que a rotina `BUSCA` se adapte automaticamente para a forma de  $T$  em tempo de execução mesmo que esta forma tenha sido alterada do instante da última chamada.

Este requisito é importante não somente pela questão do reuso mas também pela extensibilidade. Se  $T$  pode mudar de forma em tempo de execução, então uma decisão no sistema deve ser tomada para a utilização da versão de `BUSCA` correta. Em outras palavras, se não houver esse mecanismo automático o código, em algum local, deve conter um controle do tipo:

```
se T é do tipo A então "mecanismo A"  
se T é do tipo B então "mecanismo B"  
se T é do tipo C então "mecanismo C"
```

A estrutura de decisão deve estar ou no módulo ou no cliente. Ambos os casos não são satisfatórios. Se a decisão estiver no módulo, este módulo deve saber sobre todas as possibilidades existentes. Tal política pode levar a construção de módulos difíceis de gerenciar e sujeitos a constantes manutenções. Deixar a decisão para o cliente não é melhor. Desta forma, o cliente é obrigado a especificar que  $T$  é uma tabela do tipo A, B, etc. mas não é requerido a dizer mais nada: esta informação é suficiente para determinar que variante de `BUSCA` deve ser utilizada.

A solução para o problema é introduzida pelas linguagens orientadas por objetos com o mecanismo de herança em que o desenvolvimento é feito através da descentralização da arquitetura modular. Esta é construída por sucessivos incrementos e modificações conectadas por relações bem definidas que definem as versões corretas de `BUSCA`. Este mecanismo é chamado de Amarração Dinâmica (Late-Binding ou Dynamic binding ).

## ***Semelhanças nos subcasos***

Este último item em reuso afeta o design e a construção dos módulos e não seus clientes. Este é fundamental pois determina a possibilidade da construção de módulos bem construídos sem repetições indesejáveis. O aparecimento de repetições excessivas nos módulos compromete suas consistências internas e torna-os difíceis de fazer manutenção.

O problema surge como os programadores podem se aproveitar e tomar vantagem de semelhanças em subcasos. Para tal, deve existir no conjunto de possibilidades de implementação subgrupos de soluções com a mesma estrutura. No exemplo de busca na tabela, um exemplo típico é o aparecimento de implementações relacionadas com tabelas sequenciais. Neste caso, o algoritmo pode ser descrito da mesma forma para todos os subcasos diferindo apenas em um conjunto reduzido de rotinas primitivas. A figura abaixo ilustra o algoritmo sequencial genérico e algumas implementações de operações primitivas.

```

int BUSCA( Elemento x; TabelaSequencial T )
{
    int Pos;
    COMEÇA();
    while not FINAL() && not ACHOU( pos, x, T ) do
        MOVE_PROXIMO();
    return not FINAL();
}

```

	Vetor	Lista Encadeada	Arquivo Seq.
COMEÇA( )	i := 1	l := ponta_lista	rewind( )
MOVE_PROXIMO( )	i := i + 1	l := l.next	read( )
FINAL( )	i > tamanho	l == NULL	eof( )

Neste caso, evita-se a repetição do método de BUSCA nas implementações de busca seqüencial. Tem-se uma única função de pesquisa que se difere em quais funções específicas de COMEÇA, MOVE\_PROXIMO e FINAL serão chamadas.

Os mecanismos descritos acima são compreendidos nas linguagens orientadas por objetos pelo mecanismo de herança descrito a seguir.

## Herança

Provavelmente herança é o recurso que torna o conceito de classe mais poderoso. Em C++, o termo herança se aplica apenas às classes. Variáveis não podem herdar de outras variáveis e funções não podem herdar de outras funções.

Herança permite que se construa e estenda continuamente classes desenvolvidas por você mesmo ou por outras pessoas, sem nenhum limite. Começando da classe mais simples, pode-se derivar classes cada vez mais complexas que não são apenas mais fáceis de debuggar, mas elas próprias são mais simples.

O objetivo de um projeto em C++ é desenvolver classes que resolvam um determinado problema. Estas classes são geralmente construídas incrementalmente começando de uma classe básica simples, através de herança. Cada vez que se deriva uma nova classe começando de uma já existente, pode-se herdar algumas ou todas as características da classe pai, adicionando novas quando for necessário. Um projeto completo pode ter centenas de classes, mas normalmente estas classes são derivadas de algumas poucas classes básicas. C++ permite não apenas herança simples, mas também múltipla, permitindo que uma classe incorpore comportamentos de todas as suas classes bases.

Reutilização em C++ se dá através do uso de uma classe já existente ou da construção de uma nova classe a partir de uma já existente.

## Classes derivadas

A descrição anterior pode ser interessante, mas um exemplo é a melhor forma de mostrar o que é herança e como ela funciona. Aqui está um exemplo de duas classes, a segunda herdando as propriedades da primeira:

```

class Caixa {
public:
    int altura, largura;
    void Altura(int a) { altura=a; }
    void Largura(int l) { largura=l; }
};

class CaixaColorida : public Caixa {
public:
    int cor;
    void Cor(int c) { cor=c; }
};

```

Usando a terminologia de C++, a classe Caixa é chamada classe base para a classe *CaixaColorida*, que é chamada classe derivada. Classes base são também chamadas de classes pai. A classe *CaixaColorida* foi declarada com apenas uma função, mas ela herda duas funções e duas variáveis da classe base. Sendo assim, o seguinte código é possível:

```
void main()
{
    CaixaColorida cc;
    cc.Cor(5);
    cc.Largura(3); // herdada
    cc.Altura(50); // herdada
}
```

Note que as funções herdadas são usadas exatamente como as não herdadas. A classe Colorida não precisou sequer mencionar o fato de que as funções *Caixa::Altura()* e *Caixa::Largura()* foram herdadas. Esta uniformidade de expressão é um grande recurso de C++. Usar um recurso de uma classe não requer que se saiba se este recurso foi herdado ou não, já que a notação é invariante. Em muitas classes pode existir uma cadeia de classes base derivadas de outras classes base. Uma classe herdada de uma árvore de herança como esta herdaria características de muitas classes pai diferentes. Entretanto, em C++, não é preciso se preocupar onde ou quando um recurso foi introduzido na árvore.

Derivar uma classe de outra aumenta a flexibilidade a um custo baixo. Uma vez que já existe uma classe base sólida, apenas as mudanças feitas nas classes derivadas precisam ser depuradas. Mas quando exatamente se usa uma classe base, e que tipos de modificações precisam ser feitas? Quando se herda características de uma classe base, a classe derivada pode estender, restringir, modificar, eliminar ou usar qualquer dos recursos sem qualquer modificação.

## O que não é herdado

Nem tudo é herdado quando se declara uma classe derivada. Alguns casos são inconsistentes com herança por definição:

- Construtores
- Destrutores
- Operadores new
- Operadores de atribuição (=)
- Relacionamentos friend
- Atributos privados

Classes derivadas invocam o construtor da classe base automaticamente, assim que são instanciadas.

## Membros de classes protected

Na seção de controle de acesso, vimos como deixar disponíveis ou ocultar atributos das classes, usando os especificadores public e private. Além desses dois, existe um outro especificador, protected. Do ponto de vista de fora da classe, um atributo protected funciona como private: não é acessível fora da classe; a diferença está na herança. Enquanto um atributo private de uma classe base não é visível na classe derivada, um protected é, e continua sendo protected na classe derivada. Por exemplo:

```
class A {
    private:
        int a;
    protected:
        int b;
    public:
        int c;
};

class B : public A {
    public:
        int geta() { return a; } // ERRO!! a não é visível
        int getb() { return b; } // válido (b protected)
        int getc() { return c; } // válido (c public)
};

void main()
```

```

{
    A ca;
    B cb;

    ca.a = 1; // ERRO! a não é visível (private)
    ca.b = 2; // ERRO! b não é visível de fora (protected)
    ca.c = 3; // válido (c é public)

    cb.a = 4; // ERRO! a não é visível nem internamente em B
    cb.b = 5; // ERRO! b continua protected em B
    cb.c = 6; // válido (c continua public em B)
}

```

## Construtores e destrutores

Quando uma classe é instanciada, seu construtor é chamado. Se a classe foi derivada de alguma outra, o construtor da classe base também precisa ser chamado. A ordem de chamada dos construtores é fixa em C++. Primeiro a classe base é construída, para depois a derivada ser construída. Se a classe base também deriva de alguma outra, o processo se repete recursivamente até que uma classe não derivada é alcançada.

Desta forma, quando um construtor para uma classe derivada é chamado, todos os procedimentos efetuados pelo construtor da classe base já foram realizados. Considere a seguinte árvore de herança:

```

class Primeira {};
class Segunda: public Primeira {};
class Terceira: public Segunda {};

```

Quando a classe Terceira é instanciada, os construtores são chamados da seguinte maneira:

```

Primeira::Primeira();
Segunda::Segunda();
Terceira::Terceira();

```

Esta ordem faz sentido, já que uma classe derivada é uma especialização de uma classe mais genérica. Isto significa que o construtor de uma classe derivada pode usar atributos herdados.

Os destrutores são chamados na ordem inversa dos construtores. Primeiro, os atributos mais especializados são destruídos, depois os mais gerais. Então a ordem de chamada dos destrutores quando Terceira sai do escopo é:

```

Terceira::~~Terceira();
Segunda::~~Segunda();
Primeira::~~Primeira();

```

Como os construtores das classes base são chamados automaticamente, deve existir alguma maneira de passar os argumentos corretos para estes construtores, no caso de eles necessitarem de parâmetros. Existe uma notação especial para este caso, ilustrada abaixo, para funções inline e não inline:

```

class Primeira {
    int a, b, c;
public:
    Primeira(int x, int y, int z) { a=x; b=y; c=z; }
};

class Segunda : public Primeira {
    int valor;
public:
    Segunda(int d) : Primeira(d, d+1, d+2) { valor = d; }
    Segunda(int d, int e);
};

Segunda::Segunda(int d, int e) : Primeira(d, e, 13)
{
    valor = d + e;
}

```

A partir do exemplo acima, não é difícil perceber que, se uma classe base não possui um construtor sem parâmetros, a classe derivada tem que, obrigatoriamente, declarar um construtor, mesmo que este construtor seja vazio:

```
class Base {
    protected:
        int valor;
    public:
        Base(int a) { valor = a; }
        // esta classe não possui um construtor
        // sem parâmetros
};

class DerivadaErrada : public Base{
    public:
        int pegaValor() { return valor; }
        // ERRO! classe não declarou construtor, compilador não
        // sabe que parâmetro passar para Base
};

class DerivadaCerta: public Base {
    public:
        int pegaValor() { return valor; }
        DerivadaCerta() : Base(0) {}
        // CERTO: mesmo que não haja nada a fazer
        // para inicializar a classe,
        // é necessário declarar um construtor
        // para dizer com que parâmetro
        // construir a classe Base
};
```

## Herança pública x herança privada

Nos exemplos acima, em toda declaração de uma classe derivada, usou-se a palavra public:

```
class B : public A { ...
```

Na realidade, os especificadores de acesso private e public podem ser usados na declaração de uma herança. Por default, as heranças são private, por isso usou-se public nos exemplos acima.

Estes especificadores afetam o nível de acesso que os atributos terão na classe derivada:

private: todos os atributos herdados (public, protected) tornam-se private na classe derivada;

public: todos os atributos public são public na classe derivada, e todos os protected também continuam protected.

Na realidade, isto é uma consequência da finalidade real de heranças public e protected, que voltará a ser discutida em compatibilidade de tipos.

## Exercício 9 - Calculadora como um objeto. Classe RPN.

Alterar a calculadora para transformar a própria calculadora em um objeto

