UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INTRODUÇÃO À PROGRAMAÇÃO

ORIENTADA A OBJETOS

COM SMALLTALK

Miguel Jonathan

Departamento de Ciência da Computação - Inst. Matemática
e
Núcleo de Computação Eletrônica

Agosto 1994

INDICE

Pág

APRESENTAÇÃO

Nos últimos anos, o interesse pela Orientação a Objetos (O-O) de um modo geral cresceu de forma acentuada, especialmente em Banco de Dados, Sistemas Operacionais e Análise e Projeto de aplicações, havendo já um consenso de que o paradigma será dominante nos próximos anos.

A utilização de Smalltalk como linguagem base para esse curso decorre do fato de Smalltalk ser uma linguagem uniformemente orientada a objetos, e não uma linguagem híbrida, como C++ e Object Pascal. Dessa forma, o aluno é levado a se concentrar somente em mecanismos voltados para o paradigma, evitando a ilusão de estar programando de forma O-O, quando na realidade está apenas utilizando uma linguagem que pode também ser programada de forma O-O. Além disso, Smalltalk é a linguagem mais tradicional no campo, tendo sido pioneira na introdução dos principais conceitos relacionados com a programação O-O, como métodos, mensagens, polimorfismo, encapsulamento, além de re-introduzir o conceito de classes e hierarquia de classes, originários da linguagem Simula.

O aprendizado de linguagens O-O é mais demorado que o das chamadas linguagens algorítmicas convencionais, em parte porque é necessário dominar uma quantidade muito maior de conhecimento para fazer uso efetivo do paradigma. Isso porque uma das idéias centrais é a re-utilização maciça de código, através da utilização direta de classes pré-fabricadas, ou de sua especialização através da criação de sub-classes para adaptá-las a situações específicas. Existe aqui um contraste significativo com a tradição da programação convencional, onde as linguagens são restritas a alguns poucos tipos e comandos elementares, de rápida absorção, e o programador procura principalmente criar programas próprios para resolver cada situação particular, com baixo índice de re-utilização de *soluções* para sub-problemas específicos.

Esse texto procura apresentar algumas das características de uma linguagem O-O, sem a pretensão de oferecer um quadro completo, e muito menos a de formar programadores. O seu objetivo é essencialmente o de introduzir conceitos. O leitor interessado deve procurar as referências no final do texto para se aprofundar no assunto.

Rio de Janeiro, julho de 1994.

Miguel Jonathan
(jonathan@nce.ufrj.br)

CAPÍTULO I - Conceitos básicos.

1.1 Histórico - origens de Smalltalk.

Smalltalk é o nome da mais popular linguagem orientada para objetos existente. A expressão "orientada para objetos" contrasta com as linguagens de programação convencionais, que são orientadas para programas e dados.

A primeira linguagem a incorporar facilidades para definir classes de objetos genéricos na forma de uma hierarquia de classes e sub-classes foi a linguagem Simula [Dahl66], [Birtwistle et al.73]. Simula foi idealizada em 1966, na Noruega, como uma extensão da linguagem ALGOL 60.

Uma classe em Simula é um módulo englobando a definição da estrutura e do comportamento comuns a todas as suas instâncias (objetos). Como o nome indica, é uma linguagem adequada à programação de Simulações de sistemas que podem ser modelados pela interação de um grande número de objetos distintos.

As idéias de Simula serviram de base para as propostas de utilização de Tipos Abstratos de Dados [Liskov e Ziller 74], e também para Smalltalk. Smalltalk foi desenvolvida no Centro de Pesquisas da Xerox durante a década de 70 [Goldberg83], e incorporou, além das idéias de Simula, um outro conceito importante, devido a Alan Kay, um de seus idealizadores: o princípio de objetos ativos, prontos a "reagir" a "mensagens" que ativam "comportamentos" específicos do objeto. Ou seja, os objetos em Smalltalk deixam de ser meros "dados" manipulados por "programas", e passam a ser encarados como "processadores idealizados" individuais e independentes, aos quais podem ser transmitidos comandos em forma de "mensagens".

Outras linguagens orientadas para objetos tem sido desenvolvidas, notadamente C++ [Stroustrup 86], uma extensão de C, Objective-C, outra extensão de C, menos popular que a anterior [Cox86], Pascal orientado a objetos [Borland92], Eiffel [Meyer88] e mais recentemente, no Brasil, TOOL [TOOL94], [Carvalho 93].

Além da Xerox, que criou a ParcPlace Systems especialmente para comercializar Smalltalk-80 e seus sucedâneos (ObjectWorks), a Digitalk lançou em 1986 uma versão de Smalltalk para ambiente DOS, e mais recentemente a versão para Windows, o que contribuiu para uma maior difusão da linguagem [Digitalk].

Smalltalk, assim como outras linguagens orientadas para objetos, tem sido usada em aplicações variadas onde a ênfase está na Simulação de modêlos de sistemas, como automação de escritórios, animação gráfica, informática educativa, instrumentos virtuais, editores de texto e bancos de dados genéricos, etc. Tais aplicações diferem substancialmente daquelas em que a ênfase está na resolução de problemas através de algorítmos, tais como problemas de busca, otimização e resolução numérica de equações. Para essas aplicações, é mais adequado o uso de linguagens algorítmicas convencionais, como Pascal, Algol e Fortran.

1.2. Um novo paradigma de programação: classes, objetos, métodos e mensagens.

Em Smalltalk, assim como em outras Linguagens Orientadas para Objetos (LOO), não existe a mesma distinção conceitual entre "programas" e "dados", como nas linguagens algorítmicas tradicionais. Nestas, toda a ação concentra-se no programa, que contém a sequência de instruções, enquanto os dados são passivos e atuam como parâmetros ou argumentos das instruções.

Em Smalltalk a ação reside nos objetos do sistema, onde cada objeto constitui uma espécie de cápsula englobando não só uma estrutura de dados, mas também um conjunto de rotinas associadas.

Cada objeto simula um conceito ou abstração, e pode "reagir" a mensagens que recebe e que fazem parte do seu protocolo de comunicação com os demais objetos do sistema. Ao reconhecer uma mensagem, o objeto ativa uma das rotinas de seu repertório particular (um método), que descreve o comportamento do objeto associado a essa mensagem.

Mais exatamente, cada objeto é uma instância de sua classe. É a classe que contém a descrição da representação interna e dos métodos comuns a todas as suas instâncias. Cada instância da classe, por sua vez, possui sua própria memória privativa (seu estado interno) onde ficam armazenados os valores de seus componentes, que represntam suas características individuais.

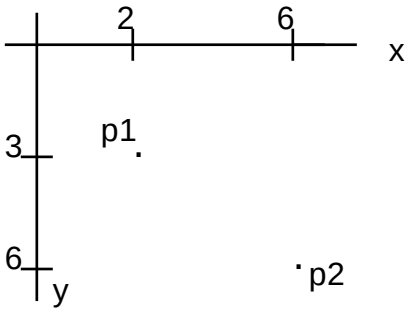
Estrutura de um objeto em Smalltalk - variáveis de instância

Um objeto é sempre uma instância de alguma classe. Cada instância possui uma estrutura onde seu *estado* pode ser armazenado. Em Smalltalk uma instância pode possuir componentes *nomeados* (semelhante a campos de registros, em Pascal), pode possuir componentes acessíveis por meio de *índices,* ou pode ter uma estrutura mista. Em todos os casos, os valores dos componenes são referências (apontadores) para outros objetos, a menos de objetos primitivos como números, caracteres e booleanos.

Os componentes nomeados são chamados de *variáveis de instância*. A classe Point, por exemplo, possui duas variáveis de instância para representar as suas coordenadas cartesianas, de nomes *x* e *y*. Já a classe Array só possui componentes indexáveis. Outras classes podem ter variáveis de instância e também ser acessadas por meio de índices

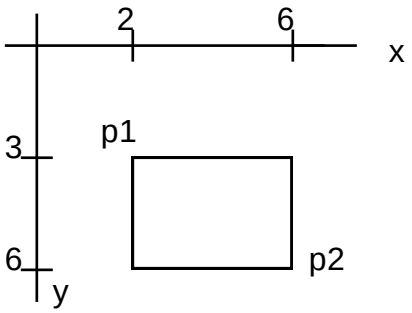
Para ilustrar, suponha que as classes Número, Ponto e Retângulo estejam implementadas (na verdade, essas classes são parte integrante de Smalltalk, de nomes Number, Point e Rectangle. Neste texto, por facilidade, serão usados os termos em português). Como visto acima, cada instância de Ponto é representada por duas instâncias de Número, suas coordenadas x e y (chamamos uma instância de Ponto por "um Ponto", e uma instância de Número por "um Número").

Sejam p1 e p2 dois Pontos, cujas coordenadas são, respectivamente, (2,3) e (6,6), como mostra a figura abaixo:



Na figura ao lado é adotada a convenção de Smalltalk, com a coordenada y aumentando para baixo, que é consistente com a disposição do texto numa página e com a varredura de tela

Na definição da classe Retângulo, a estrutura de dados de cada instância é formada por um par de Pontos, o canto superior esquerdo do Retângulo (origin) e o seu canto inferior direito (corner). Poderemos ter, portanto, um Retângulo onde a variável origin é o Ponto p1 e a variável corner é o Ponto p2, como abaixo:



Criando Pontos e Retângulos:

Um Ponto com coordenadas x e y pode ser criado enviando-se ao Numero x a mensagem @ y. A expressão abaixo cria um Ponto e faz a variável p1 referenciá-lo:

```
p1 := 2 @ 3
```

Dizemos que o Numero 2 reage à mensagem @ 3, e responde com uma nova instância de Ponto com as coordenadas 2 e 3.

Da mesma forma, um Retângulo como na figura acima pode ser criado enviando-se ao Ponto p1 a mensagem corner: p2. Na expressão abaixo, a variável ret passa a referenciar um novo Retângulo:

```
ret := p1 corner: p2
```

ou, diretamente,

```
ret := 2 @ 3 corner: 6 @ 6
```

Podemos a seguir enviar outras mensagens ao Retângulo ret que sejam próprias do protocolo da classe Retângulo. Por exemplo, Retângulos reagem à mensagem containsPoint: p, onde p é um Ponto, retornando o valor true (verdade) caso o Ponto p esteja contido no seu interior, ou false (falso) em caso contrário, como na expressão abaixo:

```
ret containsPoint: 4 @ 4
```

Nesse caso, o Retângulo ret retorna o objeto true. Note que, em Smalltalk, true é também um objeto, uma das duas instâncias da classe Boolean.

1.3 Uniformidade do ambiente Smalltalk - mensagens de classe.

A linguagem Smalltalk foi desenvolvida com a preocupação de criar um ambiente de desenvolvimento e operação de software totalmente uniforme. Essa uniformidade foi sendo implementada ao longo das diversas versões (Smalltalk-72, Smalltalk-76, Smalltalk-80) e foi completada com a versão Smalltalk-80.

Todo processamento em Smalltalk obedece ao mesmo paradigma de objetos e mensagens, seja para programar uma aplicação, utilizar funções do sistema, ou usar uma aplicação qualquer.

Seja, por exemplo, a criação de instâncias de uma determinada classe. Nos exemplos da sessão anterior, instâncias das classes Ponto e Retangulo foram criadas através do envio de mensagens a objetos. Mas a forma mais geral de criar instâncias de uma classe é enviar uma mensagem específica à própria classe. Isso só é possível porque classes são também objetos e, portanto, obedecem ao mesmo formalismo de reação a mensagens.

A mensagem padrão para se criar uma nova instância de uma classe C qualquer é new, como nas expressãoes abaixo:

```
C new
x := C new
```

A classe C reage à mensagem e cria uma nova instância de si mesma, retornando uma referência a ela. Na segunda expressão, a variável x passa a referenciar essa nova instância. Os componentes de instâncias criadas com essa mensagem são inicializados com uma referência ao objeto nulo nil. Esse objeto é a única instância da classe UndefinedObject.

Como objeto, toda classe é também instância de alguma outra classe. Essa classe é normalmente uma sub-classe da classe **Behavior**, ou seja, Behavior contém o protocolo e a descrição da estrutura de dados comuns a todas as classes do sistema. Por exemplo, a mensagem new faz parte do protocolo da classe Behavior.

Em muitos casos é conveniente para uma classe ter uma estrutura de dados, mensagens e métodos específicos para ela. Os métodos específicos referem-se normalmente à inicialização de suas instâncias de forma particular, e suas estruturas podem conter informações que sejam comuns a todas as suas instâncias.

Por exemplo, a classe Date é a classe que define o protocolo para instanciar, comparar e computar datas diversas. Essa classe reage à mensagem today respondendo com uma instância (uma data) com o valor da data de hoje, como em:

```
d := Date today
```

Instâncias de Date reagem, por sua vez, à mensagem dayName, respondendo com o nome do dia da semana a que correspondem. Portanto a sequência:

```
Date today dayName
```

produz como resultado o nome do dia da semana da data de hoje.

Note que, enquanto today é uma mensagem específica para *a classe* Date, dayName é uma mensagem específica para *instâncias* de Date.

1.4 Métodos primitivos e programados - a Virtual Machine e a Virtual Image.

Os métodos são as rotinas (algorítmos) que implementam a funcionalidade de cada mensagem. Métodos em Smalltalk são quase todos programados na própria linguagem Smalltalk.

Smalltalk é uma linguagem extensível, onde cada usuário pode criar novas classes, mensagens e métodos. O sistema Smalltalk-80 é fornecido já com centenas de métodos programados em Smalltalk e que, em seu conjunto, compõem a chamada Imagem Virtual (Virtual Image).

Alguns métodos, porém, precisam ser implementados diretamente em código objeto, específico para cada máquina, para fins de eficiência. Esses métodos são chamados de primitivos, e correspondem a uma pequena parcela do total dos métodos. Entre eles incluem-se as operações aritméticas, entrada e saída, e outras funções que atuam sobre o hardware do sistema. Esses métodos são utilizados da mesma forma que os demais, embora não possam ser alterados pelo programador. Em seu conjunto, formam a Máquina Virtual (Virtual Machine), que precisa ser implementada separadamente para cada modelo distinto de computador.

1.5 Revisão dos conceitos básicos.

As sessões anteriores apresentaram, informalmente, os principais conceitos que caracterizam a programação orientada para objetos em Smalltalk. A seguir, esses conceitos são resumidos para uma melhor apreciação em conjunto.

1. Todos os elementos do sistema são objetos. Todo objeto é uma instância de alguma classe.
2. Cada objeto, além de possuir uma estrutura de dados particular, é ativo, no sentido de que reage a mensagens que lhe são enviadas, retornando um valor, que também é um objeto.
3. Para cada classe, suas instâncias "compreendem" apenas as mensagens que fazem parte do seu protocolo de comunicação, rejeitando as demais. A cada mensagem corresponde um método específico, que fica localizado na classe.
4. Qualquer componente de um objeto só pode ser acessado, ou modificado, por meio de mensagens especialmente implementadas para tal fim.
5. Objetos podem ter estruturas bastante gerais, e podem ser construídos a partir de outros objetos previamente implementados. Este mecanismo permite definir, construir e manipular estruturas de grande complexidade.
6. Toda a computação em Smalltak obedece ao mesmo paradigma de classes, objetos e mensagens, formando um ambiente uniforme e integrado. Essa característica torna possível que uma classe, uma vez definida, possa ser utilizada em diversas aplicações, estimulando a criação de classes de utilidade ampla.
7. Smalltalk é uma linguagem extensível. Ela permite criar novas classes, e para cada classe existente, permite adicionar ou retirar métodos. Isso significa que o sistema pode ser ampliado e modificado, mantendo sempre sua uniformidade.

CAPÍTULO 2 - Hierarquia de Classes

2.1 Classes e sub-classes.

Neste capítulo estudaremos uma das mais poderosas características das linguagens orientadas para objetos, e de Smalltalk em particular. Trata-se da possibilidade de organizar a informação no computador em uma hierarquia de abstrações de forma semelhante à utilizada pela mente humana para lidar com a complexidade do mundo.

Um exemplo bem conhecido é a classificação dos animais e plantas em uma hierarquia de categorias. A cada categoria corresponde um conjunto de características específicas que a distingue das demais do mesmo nível. Além disso, cada categoria partilha também de todas as características específicas dos níveis acima do dela.

A figura abaixo ilustra a hierarquia do Reino Animal, organizada em sub-categorias chamadas filos, classes, ordens, famílias, gêneros e espécies. Ao lado de cada categoria, aparecem sublinhadas algumas de suas características específicas:

ANIMAL (reino) - sistema digestivo, movimento
VERTEBRADO (filo) - coluna vertebral
MAMÍFERO (classe) - fêmeas amamentam
ROEDOR (ordem) - incisivos compridos
CIURÍDEO (família) - cauda peluda
ESQUILO (gênero) - trepam em árvores
ESQUILO-VERMELHO (espécie) vermelho
ESQUILO-CINZA (espécie)- cinza
:
:
TÂMIA (gên)- possuem listas, fazem túneis

:
:
MURÍDEO (família) - cauda comprida
RATO (gênero) - peso 300 a 500 g
.....
.....
CAMUNDONGO (gênero) - peso 15 a 30 g
(instância: Mickey)

Note a diferença essencial entre os conceitos de categoria e instância de categoria. Um determinado camundongo (ex. Mickey), é uma instância do gênero camundongo. Como tal, Mickey é um objeto, e além das características próprias de sua categoria (ex. peso de 15 a 30 g), possui também todas as características das suas super-categorias, a saber, cauda comprida (murídeo), incisivos compridos (roedor), etc.

A forma de expressão **é um** é normalmente usada para dizer que um objeto é uma instância de uma determinada categoria. Por extensão, a mesma forma aplica-se a todas as super-categorias do objeto. Por exemplo, dizemos que Mickey é um

Camundongo, é um Murídeo, é um Roedor, é um Mamífero, é um Vertebrado e é um Animal.

O ato de classificar os conceitos em uma hieraquia é que permite que as características partilhadas por vários sub-conceitos sejam associadas a um conceito comum, o que simplifica enormemente a manipulação intelectual de uma grande quantidade de conhecimentos.

Em Smalltalk, todas as informações são organizadas de forma similar, em uma hierarquia de classes e sub-classes. Cada classe implementa as características estruturais, bem como o protocolo de comunicação (mensagens), comuns a todos os objetos da classe. Cada sub-classe, por sua vez, só precisa implementar aquelas características que a diferenciam das demais.

No topo da hierarquia está a classe Objeto, onde são definidos os protocolos comuns a todos os objetos do sistema. Quanto mais alta uma classe está na hierarquia, tanto mais gerais são as características que ela representa. Quanto mais baixa, mais específicas são suas características.

2.2 Herança de características pelas sub-classes.

Quando uma classe é definida em Smalltalk, ela é sempre declarada como uma sub-classe de alguma classe pré-existente, que passa a ser a sua super-classe. Toda classe herda (incorpora) automaticamente todas as características da sua super-classe, exatamente como a classe Roedor herda todas as características da classe Mamífero, e esta as da classe Vertebrado.

Além de herdar as características da super-classe, cada classe pode implementar características adicionais próprias para suas instâncias na forma de novas mensagens e métodos, e de novos componentes na sua estrutura de dados, e também redefinir métodos já existentes acima na hierarquia.

Como exemplo, considere a classe Collection (Coleção), que em Smalltalk é a super-classe de todas as classes cujas instâncias são grupos ou coleções de objetos. A hierarquia de Collection tem, parcialmente, a disposição abaixo: (em Smalltalk/V)

```
Object
    Collection
        Bag
        IndexedCollection
            FixedSizeCollection
                Array
                Interval
                String
            OrderedCollection
        Set
            Dictionary
```

Intuitivamente aceitamos que instâncias de Array (vetores), de String (cadeias de caracteres) ou de Set (conjuntos), são casos particulares (especializações) do conceito genérico de coleção de objetos.

A classe Collection define o protocolo de mensagens comum a todas as suas sub-classes. Collection não contém diretamente nenhuma instância, e sua função é

apenas a de reunir as propriedades comuns de todas as coleções. Por esse motivo, é uma classe dita *abstrata*. Como ilustração, as mensagens:

isEmpty	(responde com true, se o receptor não contém nenhum elemento, e false em caso contrário)
e	
add: umObjeto	(acrescenta o objeto dado pelo argumento, umObjeto, na coleção do receptor)

são gerais para qualquer sub-classe e, portanto, faz parte do protocolo de Collection.

As sub-classes de Collection possuem características que as diferenciam entre si.

Bag (literalmente, "saco") é uma classe cujas instâncias são coleções de elementos não ordenados, e onde são permitidas duplicações. Já **Set** (conjunto) é uma classe onde as instâncias são também coleções de elementos não-ordenados, mas onde não são permitidas duplicações.

Uma instância de **Dictionary,** sub-classe de Set, é um conjunto especializado, onde cada elemento é formado por uma associação entre um nome (argumento, chave) e um valor, sendo cada argumento único.

IndexedCollection é o nome de uma classe abstrata que fornece o protocolo comum para todas as coleções cujas instâncias são acessíveis através de índices externos. **FixedSizeCollection** é uma sub-classe desta para todas as coleções indexáveis que possuem tamanho fixo, isto é, não podem ter elementos removidos ou acrescentados (podem somente ter os valores de seus elementos modificados). Já **OrderedCollection** é outra sub-classe de IndexedCollection cujas instâncias aumentam dinâmicamnete de tamanho para acomodar novos elementos.

A classe Collection e suas subclasses serão vistas com maior detalhe no capítulo 4.

2.3. Determinação do método correspondente a uma mensagem: as variáveis especiais "self" e "super".

Como foi visto na seção anterior, uma classe herda os métodos definidos nas classes acima na hierarquia, mas pode também redefiní-los e acrescentar outros. Quando um objeto recebe uma mensagem, é iniciada uma busca no protocolo específico de sua classe. Caso a mensagem seja encontrada, o método correspondente é executado. Caso contrário, a busca continua no protocolo da sua super-classe, e assim sucessivamente até chegar ao topo, na classe Object.

Para ilustrar, seja a hierarquia de classes abaixo, onde são mostrados os protocolos de cada uma, e os nomes dos métodos correspondentes a cada mensagem:

classe A	mens1 (método1) mens2 (método2) mens3 (método3)
classe AA.....	mens4 (método4)

mens1 (método1a)

classe AAA mens5 (método5)
mens2 (método2a)

Note que a classe AA redefine o método para a mensagem mens1, e a classe AAA redefine o método para a mensagem mens2.

Sejam 3 objetos, referenciados pelas variáveis x, y, e z, tais que x é da classe A, y da classe AA e z da classe AAA.

Suponha a seguir a sequência de expressões abaixo. Os métodos que serão ativados são mostrados ao lado de cada mensagem, entre parênteses:

x	mens1	(método1)
y	mens2	(método2)
y	mens3	(método3)
y	mens1	(método1a)
z	mens1	(método1a)
z	mens2	(método2a)
z	mens3	(método3)
z	mens4	(método4)
x	mens4	(erro: mensagem não encontrada)
y	mens5	(erro: mensagem não encontrada)
x	mens2	(método2)

Note que a mensagem mens5 não pode ser compreendida por y, pois não foi implementada em sua classe, nem em nenhuma de suas super-classes. O mesmo ocorre com a mensagem mens4 enviada a x.

As variáveis especiais: **self** e **super**.

Quando um objeto recebe uma mensagem, o método correspondente é ativado. Esse método, por sua vez, é composto por uma outra sequência de mensagens. As variáveis self e super podem ser usadas em qualquer expressão de um método M para representar o próprio receptor da mensagem que ativou M. A diferença entre mensagens dirigidas a self ou a super está na forma como Smalltalk inicia a busca do método correspondente. Quando uma mensagem, dentro de um método M, é enviada a self, a pesquisa do método correspondente inicia-se na classe do receptor da mensagem que ativou M. Caso a mensagem seja enviada a super, então a pesquisa inicia-se na super-classe do receptor.
Voltando ao exemplo anterior, suponha que o objeto **z** recebeu a mensagem mens5, ativando o método método5. Suponha agora que, dentro de método5 ocorram as seguintes expressões:

```
método5
    :
    :
    self mens2.
    :
    super mens2.
```

:

Em ambas, self e super referem-se ao mesmo objeto referenciado por z, que é o receptor da mensagem que ativou método5. A primeira mensagem ativará o método método2a, enquanto a segunda ativará o método método2 pois, nesse último caso a pesquisa se iniciará na classe AA.

A sintaxe das expressões em Smalltalk será vista com maior detalhe no capítulo 3.

2.4 A classe Object e o protocolo comum a todos os objetos.

Object é uma classe abstrata que implementa o protocolo comum a todos os objetos de Smalltalk.

Existem várias mensagens a que qualquer objeto deve responder e que, por isso, se localizam na classe Object. Para qualquer instância de qualquer classe que receba uma dessas mensagens, um mesmo método será ativado. Algumas das situações comuns a todos os objetos são:

a. Testar a funcionalidade de um objeto (saber a que classe pertence, ou se responde a uma determinada mensagem).
b. Comparar dois objetos quanto à igualdade.
c. Tratar relações de dependência.
d. Fazer cópias de um objeto.
e. Tratar situações de erro que exigem um aviso ao usuário.

Os grupos de mensagens abaixo ilustram, de forma não exaustiva, o protocolo da classe Object:

Funcionalidade:

class retorna uma referência para a classe do receptor.

respondsTo: umSimbolo
retorna **true** caso exista no protocolo da classe a que o receptor pertence (ou de alguma super-classe) algum método cujo seletor seja o argumento umSimbolo, e **false** em caso contrário.

isNil retorna **true** caso o receptor seja o objeto nulo (**nil**)

notNil retorna **true** caso receptor não seja nil.

Igualdade de objetos. As mensagens = e == :

O conceito de igualdade em Smalltalk difere das linguagens convencionais, devido à complexidade das estruturas dos objetos.

Em Smalltalk, a mensagem == retorna true somente se o receptor e o argumento forem **o mesmo** objeto. Por exemplo,se x e y apontam para o mesmo objeto, então

x == y retorna o objeto **true**.

Já a mensagem **=** e' mais sutil. A sua implementação "default" na classe Object é idêntica a **==.** Mas essa mensagem é redefinida em algumas classes para significar uma igualdade "fraca": é o caso de dois objetos diferentes, mas com

componentes iguais. Por exemplo, na classe **Array**, duas instâncias diferentes com exatamente os mesmos valores são **=**, mas nao são **==.** Essa redefinição se aplica também para outras classes como **Date**, **Point**, **IndexedCollection** (Array incluída), **String** e **Time**.

Dependentes de um objeto:

Todo objeto em Smalltalk pode ter um ou mais objetos "dependentes" dele. Essa dependência é implementada da seguinte forma:

Existe um dicionário, chamado **Dependents**, que é implementado como uma variável de classe da classe Object. Ou seja, é uma variavel de classe de todas as classes do ambiente.

Sejam n objetos dependentes de um determinado objeto referenciado pela variável X. Então existe uma entrada no dicionário Dependents (i.e. um par chave-valor), onde a chave é uma referência a X, e o valor é uma OrderedCollection que contém os n objetos dependentes de X.

Note que X nao referencia diretamente seus dependentes. Esse artifício permite que um número indeterminado de objetos possam se tornar dependentes de algum outro, sem que seja necessário criar estruturas especiais para isso em cada instância. Essa facilidade é muito explorada na implementação de interfaces diferentes para uma mesma aplicação.

As mensagens abaixo do protocolo da classe Object permitem organizar as relações de dependência:

addDependent: umObjeto
: inclui o argumento, umObjeto, como dependente do receptor

allDependents
: retorna um Set com os dependentes diretos e indiretos do receptor, isto é, incluindo todos os dependentes de seus dependentes, etc.

dependents
: retorna uma OrderedCollection com os dependentes diretos do receptor

dependsOn: umObjeto
: inclui o receptor na coleção de dependentes do argumento, umObjeto.

release
: libera todos os dependentes diretos do receptor, caso existam, isto é, remove a entrada do dicionário Dependents cuja chave é (i.e. referencia) o receptor.

As mensagens abaixo servem para um objeto enviar uma mesma mensagem a todos os seus dependentes:

broadcast: umSimbolo
: envia a todos os dependentes do receptor uma mesma mensagem (sem argumentos) cujo seletor é umSimbolo.

broadcast: umSimbolo with: umObjeto
: envia a todos os dependentes do receptor uma mesma mensagem cujo seletor é umSimbolo.com um argumento, dado por umObjeto.

changed: umObjeto
: envia a todos os dependentes do receptor a mensagem update: umObjeto. A mensagem update: deve estar implementada nas classes dos dependentes,

possivelmente com funções diferentes em cada uma. O parâmetro umObjeto é em geral analisado por cada dependente para decidir se a mensagem interessa a ele.

Cópia de objetos:

Dois métodos básicos permitem obter cópias de instâncias de qualquer classe. Ambos criam uma nova instância e retornam uma referênca para ela:

shallowCopy
: (cópia rasa) retorna uma cópia do receptor, que partilha as variáveis de instância do receptor

deepCopy
: (cópia funda) retorna uma cópia do receptor onde cada variável de instância contém uma cópia rasa da variável de instância correspondente do receptor.

2.5 Metaclasses.

(Esta seção não é essencial para um primeiro contato com Smalltalk, mas é inserida neste ponto por coerência com o assunto deste capítulo, e pode ser deixada para leitura posterior).

Como foi descrito nas seções anteriores, toda classe em Smalltalk é também um objeto como qualquer outro e responde a mensagens que fazem parte de seu protocolo particular. Em geral essas *mensagens de classe* objetivam a criação de instâncias de si mesmas com uma inicialização específica, como a mensagem today para a classe Date, que cria uma nova instância com o valor da data de hoje.

Existe uma classe especial, chamada Behavior (literalmente, *comportamento*), onde fica localizado o protocolo comum a todas as classes do sistema. Todas as classes, enquanto objetos, são instâncias de Behavior , ou de alguma sub-classe dela. É importante compreender bem essa natureza **dual** das classes em Smalltalk: de um lado são classes, mas também são instâncias de outra classe. Em outras linguagens, como C++ e Eiffel, classes não são objetos, e não podem portanto receber mensagens.

A mensagem padrão para criar uma nova instância de qualquer classe é new, enviada à própria classe. Como essa mensagem é comum a todas, ela fica localizada na classe Behavior.

Por outro lado, mensagens específicas para cada classe, como today para a classe Date, não podem ficar em Behavior, pois não devem ser compreendidas pelas demais classes. Devem estar em alguma sub-classe de Behavior da qual Date seja uma instância. Essa sub-classe de Behavior é chamada a *metaclasse* de Date. A classe Date é a única instância de sua metaclasse.

A metaclasse de toda classe C é criada automaticamente, quando C é criada, e C é implementada como a única instância de sua metaclasse. Todas as mensagens específicas para a classe C ficam localizadas na metaclasse de C.

Metaclasses não recebem nomes particulares. Uma referência à metaclasse de uma classe C qualquer pode ser obtida enviando-se a C a mensagem class. A metaclasse de C é portanto referenciada como *C class*.

Para manter a coerência geral do modêlo, cada metaclasse é também um objeto. Mas, ao contrário das classes, uma metaclasse não é uma instância de alguma outra metaclasse (pois, se assim fosse, haveria uma cadeia infinita de metaclasses de metaclasses de metaclasses, etc!) Cada metaclasse é uma instância de uma classe especial chamada **MetaClass**.

Além de MetaClass, Behavior possui a sub-classe **Class**, que é a super-classe de todas as classes que não são meta-classes. A hierarquia de classes fica portanto: (parte)

Object
.....Behavior
....Metaclass (*instâncias: Date class, Point class, Class class....*)
....Class
....Object class (*única instância: a classe Object*)
....Behavior class
.... MetaClass class
.... Class class (*única instância: a classe Class*)
....Magnitude class
.... Date class (*única instância:a classe Date*)
.... :
....Point class (*única instância: Point)*
....Rectangle class
....etc.
.....Magnitude
....Date (*instância: 17 July 1994*)
....Number
....Integer (*instância: 23*)
....:
.....Point (*instância: 2 @ 3)*
.....Rectangle (*instância: 2 @ 3 corner: 10 @ 20*)
.....:
..... etc.

Note que a classe Class também tem sua metaclasse, *Class class*, da qual Class é instância.

Portanto, quando se deseja, em Smalltalk, criar uma mensagem específica para inicializar instâncias de uma classe C qualquer, essa mensagem é incorporada, juntamente com o método correspondente, no protocolo de *C class*, a metaclasse de C.

Hierarquia das metaclasses:

Cada classe é automaticamente uma instância de sua metaclasse, como foi visto acima. Note que as metaclasses formam também uma hierarquia a partir da classe *Object class*, subclasse imediata de Class. Se uma classe A é super-classe de outra classe AA), então *A class* é também super-classe de *AA class*. Isto significa que podemos criar mensagens de inicialização que valem para toda uma sub-hierarquia de classes bastando incorporar essas mensagens na metaclasse da classe mais elevada dessa hierarquia. Por exemplo, uma mensagem incorporada ao protocolo da

metaclasse de Collection pode ser enviada a qualquer sub-classe de Collection para inicializar instâncias.

CAPÍTULO 3 - EXPRESSÕES E MÉTODOS EM SMALLTALK.

3.1 Métodos e Expressões

Como vimos nos capítulos anteriores, toda computação em Smalltalk é ativada pelo envio de mensagens aos objetos do sistema. Ao receber uma determinada mensagem, um objeto ativa o **método** específico que realiza a computação correspondente.

Neste texto não usaremos qualquer notação formal para descrever a sintaxe ou a semântica da linguagem - para cada conceito será apresentada uma descrição informal seguida de alguns exemplos.

Métodos são algorítmos onde as ações são especificadas por uma *sequência de expressões Smalltalk* separadas entre si por um ponto, como abaixo (os símbolos entre colchetes são opcionais):

```
        expressão.
        expressão.
        :
[^] expressão [.]
```

Cada expressão retorna sempre um único objeto como resultado. Expressões dentro de métodos podem ser precedidas opcionalmente pelo caractere **^** (*caret)*. A execução do método termina quando uma expressão precedida do *caret* (^) é executada, ou após a execução da última expressão da sequência. Note que não existe em Smalltalk a noção de *comando* diferenciada da noção de *expressão*.

Como será visto adiante, um método é sempre ativado pela execução de uma *expressão de mensagem*, que consiste do envio de uma mensagem a um determinado objeto, que é chamado de *receptor* da mensagem. Essa expressão se reduzirá ao valor retornado pelo método após ele terminar sua execução. O valor que um método retorna será o valor da última expressão executada, caso esta seja precedida do *caret.* Caso a última expressão executada no método não seja precedida do *caret*, então o método retornará uma referência ao *receptor.*

3.2 Categorias de expressões - Atribuição.

Uma expressão pode ser de uma das 4 categorias abaixo, e pode ser precedida por zero ou mais *prefixos de atribuição*. Um prefixo de atribuição é formado por um nome de variável seguido pelos símbolos **:=** (Smalltalk/V) ou **<-** (Smalltalk-80), e tem por efeito atribuir às variáveis o valor do resultado da expressão, como no exemplo a seguir:

```
p1 := p2 := (10 @ 10   corner: 50 @ 50) center
```

A expressão acima faz p1 e p2 referenciarem o Ponto situado no centro do Retângulo com canto superior esquerdo em 10 @ 10 e canto inferior direito em 50 @ 50.

Categorias de expressão em Smalltalk

a. **Literal**: descreve um objeto constante. Um literal pode ser um *número*, um *caráter*, uma *cadeia de caracteres*, um *símbolo* ou um *vetor de literais*. Cada um representa instâncias de classes básicas do ambiente, e tem sintaxe própria, como será descrito adiante.
   Por exemplo: 25 $? #abc #(10 20 $c)

b. **Identificador de variável**: denota uma referência a um objeto. E uma sequência de letras e números, iniciando por letra.
   Por exemplo: umConjunto lista umNúmero x Retângulo

c. **Expressão de mensagem**: denota uma mensagem para um objeto receptor. Todo o processamento em Smalltalk é causado pela execução de expressões de mensagens.
   Por exemplo: lista removeLast
   (mensagem "removeLast" enviada ao objeto referenciado pela variável "lista")

d. **Expressões de Bloco**:
   Blocos são objetos especiais da classe **Context**, e contêm uma sequência de expressões delimitadas por colchetes, cuja execução poderá ser ativada oportunamente. São usados para construir estruturas de controle.
   Por exemplo:
   ```
   [indice := indice + 1.
    lista at: indice put: 0]
   ```
   Expressões de Bloco são apresentadas abaixo com maior detalhe na seção 3.5.

3.3 Sintaxe dos literais

3.3.1. Números:

Todo número em Smalltalk pertence a alguma sub-classe da classe Number (sub-classe de Magnitude), que contém o protocolo geral para todos os objetos numéricos. Number tem tres sub-classes: *Float* (números com ponto flutuante), *Integer* (inteiros) e *Fraction* (números racionais, frações). Inteiros e Frações são sempre representações exatas. Números em ponto flutuante são representações nem sempre exatas de números reais, com a precisão limitada pela quantidade disponível de algarismos significativos.

A sintaxe dos números de base decimal é semelhante à das linguagens convencionais. Frações, quando não dão resultado inteiro, são representadas pela expressão de divisão de dois inteiros.

```
Exemplos:   Float:      235.7   2.357e2   -45.7564e-7
            Integer:    425   1001
            Fraction:   3/7
```

Literais numéricos em bases diferentes de 10 podem ser representados bastando adicionar um prefixo indicando a base. O prefixo é formado pelo inteiro que representa a base seguido da letra r, como mostram os exemplos abaixo:

```
2r1001      (9, na base 2)
2r1001e3    (72, na base 2)
```

3r120	(15, na base 3)
8r34.1	(28.125, na base 8)
8r10e2	(512, na base 8)
16r1A	(26, na base 16)

Note que quando números são expressos em notação científica, (mantissa e expoente, separados pela letra e), o valor do número é o resultado da mantissa multiplicada pela base elevada ao expoente.

3.3.2. Caracteres e cadeias de caracteres:

Caracteres são constantes da classe **Character**. Um caractere é representado por um cifrão $ seguido do caractere, como nos exemplos abaixo:

```
$a    $C    $5    $*    $(    $$
```

Cadeias de caracteres são objetos da classe **String**, e são formados por uma sequência de objetos da classe Character. Cadeias são representadas pela sequência de caracteres delimitada por apóstrofos. Como em outras linguagens, a ocorrência de um apóstrofo é representada por dois apóstrofos. Por exemplo:

```
'Smalltalk-80'    'copo d''agua'
```

3.3.3. Símbolos:

Símbolos são objetos usados para representar identificadores únicos no ambiente Smalltalk como, por exemplo, nomes de classes, de seletores de mensagem, e de variáveis globais. Um símbolo é formado por uma cadeia de caracteres alfanuméricos iniciando por uma letra. Símbolos são únicos, isto é, não podem existir dois Símbolos com exatamente os mesmos caracteres. Além disso, seus caracteres são fixos, e não podem ser alterados por meio de mensagens, como ocorre com as cadeias. Símbolos são instâncias da classe **Symbol** (uma subclasse de String), e são representados pelo caractere # seguido pelos caracteres do símbolo.

Por exemplo:

```
#Dictionary
#at:put:
```

3.3.4 Vetores de literais:

Vetores são objetos da classe **Array**. Um vetor de literais é representado por uma sequência de literais delimitada por um par de parênteses, e precedida pelo caractere #. Símbolos e vetores que ocorrem no interior de um vetor não são precedidos por #.

Por exemplo, o vetor de literais abaixo contém 6 objetos: as cadeias de caracteres *'vermelho'* e *'azul'*, o Array com 3 inteiros *#(1 2 3)*, o caractere *$a*, o inteiro *32* e o símbolo *#at:put:* :A classe Array será estudada com maior detalhe no capítulo 4

```
#('vermelho'    'azul'    (1   2   3)    $a    32    at:put:)
```

3.4 Nomes de variáveis - variáveis privativas e partilhadas

Um nome de variável é uma sequência de letras e dígitos começando por uma letra, como abaixo:

umNumero R35 indice Set

Variáveis representam referências a objetos do sistema. Devido à estrutura modular de Smalltalk, cada objeto possui variáveis internas (variáveis de instância) a que só ele tem acesso, isto é, só são acessíves através de métodos da classe do objeto. Essas variáveis são chamadas privativas ("private"). Outras variáveis são acessíveis a mais de um objeto e são chamadas variáveis partilhadas ("shared").

Por convenção, os nomes das variáveis privativas iniciam-se sempre por uma letra minúscula, enquanto os nomes das variáveis partihadas iniciam-se por uma letra maiúscula.

Classes são objetos partilhados e, portanto, o nome de uma classe inicia-se sempre por uma letra maiúscula.

Pseudo-variáveis: alguns identificadores especiais referem-se a objetos mas, ao contrário dos nomes de variáveis, não podem ter o seu valor alterado por meio de atribuição. São as chamadas pseudo-variáveis, abaixo relacionadas:

nil representa o objeto nulo. Toda variável não inicializada refere-se a nil.
true refere-se ao objeto da classe Boolean que representa o valor lógico Verdadeiro.
false idem para o valor lógico Falso.
self e super
referem-se ao receptor da mensagem que ativou o método em que são usadas. São descritas adiante na seção 2.3

3.5 Expressões de mensagem

Uma expressão de mensagem representa o envio de uma solicitação a um objeto receptor para realizar uma determinada operação. A cada expressão de mensagem corresponde uma resposta do receptor.

As expressões de mensagem em Smalltalk equivalem à chamada de procedimento ou função em linguagens convencionais, onde o receptor é um parâmetro privilegiado. O seletor da expressão especifica qual a operação a ser realizada e permite localizar o método que implementa a operação. Uma expressão de mensagem pode conter zero ou mais argumentos. A sintaxe de uma expressão de mensagem é

<receptor> <mensagem>

Existem 3 tipos básicos de expressão de mensagem, a saber, *unárias*, *binárias* e *mensagens com palavras-chave*.

a. Mensagens unárias: não contém argumentos, e são formadas apenas pelo seletor. Por exemplo:
   beta sin — mensagem sin enviada a um número referenciado pela variável beta. Responde com o valor do seno do ângulo em radianos representado pelo valor de beta.
   3 factorial — mensagem factorial enviada ao número 3. Responde com o fatorial de 3 (número 6).

b. Mensagens binárias: são mensagens com apenas um argumento, nas quais o seletor é representado por um ou dois caracteres não alfa-numéricos. Equivalem

às expressões com operadores nas linguagens convencionais. São usadas em operações aritméticas e relacionais, como nos exemplos abaixo:

3 + 2	O receptor é o número 3, o seletor é + e o argumento da mensagem é o número 2. Responde com o objeto 5.
indice <= 2	O receptor é o objeto referenciado pela variável indice, o seletor é <= , e o argumento é o número 2. Responde o objeto *true* ou o objeto *false*.

c. Mensagens com palavras-chave: ("keyword messages") - são mensagens com um ou mais argumentos, onde o seletor é representado por uma sequência de uma ou mais palavras-chave, cada uma precedendo um dos argumentos. Cada palavra-chave é, nesse caso, um identificador seguido do caractere "**:**". Por exemplo:

Array new: 3	O receptor é a classe Array, o seletor é new: (uma palavra-chave), e o argumento é 3. Responde com uma nova instância de Array com 3 elementos, inicialmente nulos.
list at: 1 put: 'um'	O receptor é o objeto referenciado pela variável list, o seletor é at:put: (duas palavras-chave), e os dois argumentos são 1 e 'um'. Faz o primeiro elemento de list referenciar a cadeia 'um', e retorna uma referência a list.

Por meio de prefixos de atribuição é possível associar o objeto resultado da expressão a uma variável , como na expressão abaixo:

```
list := Array new: 3
```

Nesse caso, após a criação da instância de Array, a variável list passará a referenciá-la.

Parsing das expressões de mensagem: regras de precedência.

Difere das linguagens convencionais para possibilitar a uniformidade de tratamento de qualquer mensagem. Mensagens podem ser *encadeadas* a qualquer nível em uma mesma expressão, e o resultado de cada mensagem pode servir de receptor ou argumento para outras mensagens.

As regras gerais para a avaliação de expressões de mensagens encadeadas são dadas abaixo. Parênteses podem ser usados para alterar a ordem de avaliação:

a. Mensagens unárias tem precedência sobre todas as demais. Todos os seletores unários tem a mesma precedência.

   Exemplo: 15 + list size é equivalente a 15 + (list size)

b. Mensagens binárias são avaliadas da esquerda para a direita. Todos os seletores binários tem a mesma precedência, (ao contrário das linguagens convencionais, onde multiplicação e divisão tem precedência sobre soma e subtração). Mensagens binárias tem precedência sobre mensagens com palavras-chave.

   Exemplo: 3 + 2 * 4 max:10 é equivalente a ((3 + 2) * 4) max:10

c. Mensagens com palavras-chave, sempre que forem usadas como receptor ou como argumento de outra mensagem, deverão ser colocadas entre parênteses.

Exemplo 1: r1 s1: a s2: b s3: c
é interpretada como uma mensagem ao objeto r1 com seletor **s1:s2:s3:** e argumentos a, b e c.

Exemplo 2: r1 s1: (a s2: b s3: c)
é interpretada como uma mensagem a r1 com seletor **s1:** e com argumento igual ao objeto resultante da mensagem ao objeto a com seletor **s2:s3:** e argumentos b e c.

Exemplo 3: (r1 s1: a) s2: b s3: c
é interpretada como uma mensagem ao objeto resultante da mensagem **r1 s1: a**, com seletor **s2:s3:** e argumentos b e c.

Como ilustração, seja a expressão abaixo:

```
list size + 5 * 10 - 50 max: limite sqrt
```

onde list é um Array de 3 elementos, e limite tem o valor 100. Arrays respondem à mensagem unária size com seu tamanho. Números respondem à mensagem max: umNumero com o maior entre os valores do receptor e do argumento. Números também respondem à mensagem unária sqrt com a raiz quadrada de si mesmos.

De acordo com as regras de avaliação, a expressão acima é equivalente à seguinte expressão com parênteses:

```
((((list size) + 5) * 10) - 50) max: (limite sqrt)
```

A expressão acima será reduzida de acordo com os passos seguintes:

```
(((3 + 5) * 10) - 50) max: (100 sqrt)
(((3 + 5) * 10) - 50) max: 10
((8 * 10) - 50) max: 10
(80 - 50)max: 10
30 max: 10
30
```

3.6 Expressões de Bloco.

Um Bloco é um objeto formado por uma sequência de expressões separadas por um ponto, e delimitada por um par de colchetes. Por exemplo:

```
[indice := indice + 1.
 lista at: indice put: 'vermelho']
```

Como objeto, um bloco pode ser referenciado por variáveis, e pode reagir a mensagens específicas. A expressão

```
x := [indice := indice + 1.
      lista at: indice put: 'vermelho']
```

faz a variável x referenciar o bloco acima.

Um bloco ativa as expressões em seu interior ao receber a mensagem unária value. Por exemplo, após a atribuição acima, a expressão

x value

terá como efeito a execução das expressões do bloco.

Como outro exemplo, se E é uma expressão qualquer, então as expressões **E** e **[E] value** são equivalentes.

Quando um bloco recebe a mensagem value, o valor resultante, ou resposta, é o resultado da última expressão da sequência que compõe o bloco.

Bloco vazio: o resultado do envio da mensagem value a um bloco vazio é nil, ou seja, [ ] value tem o valor nil.

Argumentos de blocos: blocos podem ter argumentos, que funcionam como variáveis locais. A forma de um bloco com argumentos é o seguinte:

[:v1 :v2 .... :vn| sequência de expressões]

onde v1, v2, ...,vn são nomes de argumentos locais do bloco. Blocos com argumentos são usados em estruturas de controle que realizam iterações sobre coleções de objetos, e são examinados no capítulo 4, na seção sobre Coleções.

3.7 Estruturas de contrôle em Smalltalk.

As três estruturas básicas de contrôle, *repetição simples*, *seleção condicional,* e *repetição condicional* são implementadas através do uso de blocos. Embora o efeito seja semelhante ao obtido por comandos tipo *if-then-else*, *for* e *while* em linguagens como Pascal, a implementação utiliza princípios próprios da programação orientada para objetos. As estruturas de controle em Smalltalk não tem uma sintaxe particular, e são implementadas através de mensagens com palavras-chave apropriadas, como será visto a seguir.

3.7.1 Seleção condicional - ifTrue:ifFalse:

É utilizada quando desejamos selecionar qual, entre duas sequências de expressões, será executada, dependendo de uma condição ser verdadeira ou falsa. Smalltalk provê uma mensagem especial para realizar esse tipo de seleção, que tem seletor **ifTrue:ifFalse:** e cujos argumentos são ambos da classe Bloco. Essa mensagem faz parte do protocolo da classe Boolean, cujas duas únicas instâncias são os objetos *true* e *false*.

Seja a mensagem: **ifTrue: bloco1 ifFalse: bloco2** enviada tanto para *true* como para *false*, onde as variáveis *bloco1* e *bloco2* são blocos.

Quando o objeto *true* recebe essa mensagem, ele envia para o primeiro argumento (bloco1) a mensagem value, provocando a execução de suas expressões. Quando

false recebe essa mensagem, ele envia a mensagem value para o segundo argumento, provocando a execução das expressões de bloco2. Isso é possível porque, graças ao mecanismo do *polimorfismo*, a mesma mensagem recebida por objetos de classes diferentes, pode provocar efeitos diferentes.

Esse mecanismo ilustra mais uma vez como o paradigma de objetos e mensagens é mantido uniformemente em todo o ambiente Smalltalk, mesmo para a implantação das estruturas básicas da linguagem.

Como exemplo, seja a expressão abaixo que atribui à variável paridade o valor 'par' ou 'impar' dependendo da paridade de outra variável numero:

```
(numero \\ 2) = 0      ifTrue: [paridade := 'par']      ifFalse:[paridade := 'impar']
```

Nota: Numeros reagem à mensagem \\ respondendo com o valor de si mesmos módulo argumento. Note que os parênteses acima são desnecessários, pelas regras de precedência. O mesmo efeito da expressão acima pode ser obtido com a seguinte expressão:

```
paridade :=   numero \\ 2 = 0      ifTrue: ['par']      ifFalse: ['impar']
```

Quando só se deseja testar a veracidade ou a falsidade de uma condição, podem ser usadas as mensagens com um único argumento, ifTrue: ou ifFalse:.

A mensagem: **ifTrue: bloco** é equivalente à mensagem

```
ifTrue: bloco  ifFalse: [ ]
```

Da mesma forma, a mensagem: **ifFalse: bloco** é equivalente à mensagem

```
ifTrue: [ ]  ifFalse: bloco
```

3.7.2 Repetição simples - timesRepeat:

Para repetir um número determinado de vezes uma sequência de expressões, o mecanismo usado em Smalltalk consiste em enviar a um Inteiro, cujo valor é o número desejado de repetições, a mensagem **timesRepeat:**, onde o argumento é um bloco que contém a sequência de expressões a repetir.

Ao receber a mensagem, o Inteiro responde enviando ao bloco um número de mensagens value sucessivas igual ao seu próprio valor. Por exemplo, a expressão abaixo

```
3  timesRepeat: [n := n * n]
```

faz com que o valor de n seja elevado à quarta potência.

3.7.3 Repetição condicional - whileTrue: e whileFalse:

Smalltalk implementa a repetição condicional de um bloco de expressões através do envio da mensagem whileTrue: (ou whileFalse:) a outro bloco, cujo valor retornado deve ser um Booleano. O argumento da mensagem é o bloco que contém a sequência de expressões a serem repetidas.

Quando um bloco recebe a mensagem **whileTrue:,** ele envia a si mesmo a mensagem value. Caso a resposta seja *true*, ele envia a seguir a mensagem value ao bloco do argumento, e torna a enviar a si mesmo a mensagem value, reiniciando o ciclo. Caso a resposta seja *false*, o processo termina.

Por exemplo, para zerar todos os elementos de um Array de nome lista, pode ser usada a sequência abaixo:

```
i := 1.
[i <= lista size]
    whileTrue:    [lista at: i   put: 0.
                                i := i + 1]
```

A mensagem **whileFalse:** produz efeito inverso, fazendo com que as expressões do bloco argumento sejam repetidas enquanto o valor do receptor for igual a *false*.

3.8 Sintaxe dos métodos.

Um método é uma sequência de expressões precedida pelo *padrão da mensagem* que o ativa e, opcionalmente, por uma declaração de variáveis temporárias. Variáveis temporárias são acessíveis apenas para as expressões do método, e desaparecem após o término de sua execução. A declaração das variáveis temporárias consiste em um conjunto de nomes de variáveis delimitado por 2 barras verticais.

Um método é sempre implementado dentro de uma determinada classe e será ativado sempre que alguma instância de sua classe, ou de uma sub-classe desta, receber uma mensagem com o seu padrão de mensagem (desde que não tenha sido redefinido para a sub-classe).

Comentários podem ser incluidos livremente no interior dos métodos, sob a forma de textos delimitados por aspas duplas.

Como exemplo, seja o método abaixo, que poderia ser incluído na classe Number para obter a potência de ordem n (inteiro não negativo) de um número qualquer:

```
potência: expoente
"responde com o valor do receptor elevado à potência dada por expoente"
  |result|
  result := 1.
  expoente timesRepeat: [result := self * result].
  ^ result
```

A primeira linha é o padrão da mensagem, onde o seletor é potência:, com um argumento formal, expoente. A variável temporária result é usada somente para apoiar o algoritmo. A pseudo-variável self refere-se ao objeto receptor da mensagem que é, no caso, o número a ser elevado à potência expoente. A última linha indica que o método retornará o valor final de result.

Após implementado na classe Number, a expressão

```
5 potência: 2          retornará o valor 25.
```

3.9 Métodos recursivos.

Uma mensagem pode ser utilizada recursivamente na definição de qualquer método. Por exemplo, o método abaixo, que poderia implementar a mensagem fatorial na classe Integer, utiliza essa mensagem recursivamente:

```
fatorial
    self = 0 ifTrue: [^1].
    self < 0
      ifTrue: [self error: 'receptor não deve ser negativo']
      ifFalse: [^self * (self - 1) fatorial]
```

A mensagem error: pertence ao protocolo da classe Object, e é portanto acessível a objetos de todas as classes, como descrito na seção 2.4. O seu efeito é retornar uma mensagem de erro.

CAPÍTULO 4 - CLASSES BÁSICAS DA LINGUAGEM SMALLTALK

Smalltalk é uma linguagem extensivel e uniforme onde todos os conceitos são implementados como objetos de alguma classe. Podemos dividir o conjunto das classes em dois sub-conjuntos maiores: o das classes que são normalmente fornecidas com a linguagem, e o das classes que são implementadas pelos usuários para suas aplicações específicas. Do primeiro grupo fazem parte cerca de 100 classes, que incluem desde as classes mais fundamentais, como **Number** e **Boolean**, até classes sofisticadas, como **Animation**, que representa objetos gráficos animados. O conjunto das classes que são fornecidas com a linguagem forma a chamada Imagem Virtual do sistema ("Virtual Image").

Neste capítulo, algumas das classes básicas da Imagem Virtual são apresentadas de forma reduzida, com a finalidade de ilustrar a variedade e diversidade de objetos que podem ser manipulados neste ambiente, bem como para possibilitar a compreensão dos exemplos de aplicação.

4.1 Magnitude.

Magnitude é uma classe abstrata que fornece o protocolo comum para os objetos que podem ser comparados dentro de uma ordenação linear. Por exemplo, duas datas podem ser comparadas, assim como dois caracteres alfabéticos, para se determinar se são iguais, ou se um antecede ou precede o outro.

As mensagens do protocolo de Magnitude possuem como seletores os operadores relacionais convencionais, como >, <, >= e <=. Os operadores de teste de igualdade e desigualdade são implementados diretamente na classe Object, uma vez que são mais gerais, pois dois objetos quaisquer podem ser comparados quanto à igualdade. Magnitude inclui ainda as mensagens **between:and:,** para testar se uma grandeza está dentro de um intervalo, além de **max:** e **min:** para determinar a maior e a menor entre duas grandezas, respectivamente.

São sub-classes de Magnitude as classes Association, Character, Date, Number e Time.

Association:
Cada instância desta classe representa a associação entre um par de objetos, onde o primeiro é uma chave e o segundo um valor associado. Associações são normalmente usadas como elementos da classe Dictionary (Dicionário). A comparação entre duas associações é feita com referência às chaves das duas.

Character:
Cada instância representa um caractere ASCII.

Date:
Cada instância representa uma data específica, a partir do início do calendário Juliano.

Time:
Cada instância representa um determinado segundo, dentro de um dia de 24 horas.

Protocolos adicionais de Date e de Time permitem a conversão entre formas equivalentes de datas e horários, bem como incluem mensagens para contagem do tempo de execução de blocos.

As classes Date e Time possuem um protocolo próprio (*mensagens à classe*) bastante variado, além do protocolo para suas instâncias. Alguns exemplos ilustrativos são dados a seguir.

Time totalSeconds	retorna o número de segundos decorridos desde 1/1/1901 até a data de hoje.
Date fromDays: n	retorna a data correspondente a n dias decorridos a partir de 1/1/1901. O valor de n pode ser negativo, para datas anteriores a 1/1/1901.
Date today	retorna (uma instância com) a data de hoje (do sistema operacional).

O protocolo das *instâncias* de Date permite adicionar e subtrair quantidades de dias de datas, para obter novas datas, bem como subtrair uma data de outra para obter o número de dias entre elas. Por exemplo:

data1 subtractDate: data2 retorna o número de dias decorridos entre as datas data1 e data2.

Como exemplo de aplicação, suponha um sistema de empréstimo de livros em uma biblioteca que deve verificar se um livro está em atraso ao ser devolvido e, caso esteja, calcular a multa devida. A multa é definida como, digamos, m reais por dia de atraso. Assumindo que a variável dataDevolução contém o valor da data limite para devolver o livro, o trecho de programa abaixo atribui à variável multa o valor devido:

```
multa := Date today > dataDevolução
        ifTrue: [^ m * (Date today subtractDate: dataDevolução)]
        ifFalse: [^0]
```

4.2 Coleções de objetos.

4.2.1 A classe **Collection**.

A classe Collection foi apresentada na seção 2.2, e contém o protocolo comum a todos os objetos que representam grupos ou coleções de objetos. Todas as estruturas de dados básicas são instâncias de alguma classe descendente de Collection. A hierarquia da classe Collection em Smalltalk-80 é apresentada abaixo, à esquerda, em comparação com a de Smalltalk/V, à direita, onde podem ser notadas algumas diferenças.

```
Collection                          Collection
    Bag                                 Bag
    SequenceableCollection          IndexedCollection
        ArrayedCollection                   FixedSizeCollection
            Array                               Array
            Bitmap                              Bitmap
            ByteArray                           ByteArray
            RunArray                            Interval
            String                              String
                Symbol                              Symbol
            Text                                OrderedCollection
        Interval                                SortedCollection
        OrderedCollection               Set
    Set                                         Dictionary
        Dictionary
```

O protocolo de Collection inclui diversas mensagens, que são em geral implementadas de forma diversa em cada subclasse. Seguem alguns exemplos de várias categorias de mensagens:

Acrescentar ou remover elementos de uma coleção:

add: umObjeto	inclui o argumento no receptor.
addAll: umaColeção	inclui todos os elementos do argumento no receptor.
remove: umObjeto	remove o argumento do receptor.
removeAll: umaColeção	remove do receptor todos os elementos do argumento.

Verificar o conteúdo de uma coleção:

isEmpty	responde *true* se o receptor não contém elementos (coleção vazia).
includes: umObjeto	responde true se o argumento está incluído entre os elementos do receptor.
size	responde o número de elementos do receptor.

Converter uma variedade de coleção em outra, desde que compatível:

asSet	responde com uma instância de Set contendo os mesmos elementos do receptor, com as duplicações removidas.
asBag	responde com uma instância de Bag contendo os mesmos elementos do receptor.
asSortedCollection	responde com uma instância de SortedCollection, contendo os elementos do receptor ordenados de forma ascendente.
asSortedCollection:[:a :b| a > b]	idem, em forma descendente.

Realizar iterações sobre os elementos de uma coleção.

Esse protocolo permite realizar processamentos específicos com cada elemento do receptor. As mensagens abaixo implementam os iteradores principais em Smalltalk. O argumento é sempre um bloco de expressões que deve conter uma variável

temporária que assumirá sucessivamente os valores dos elementos da coleção do receptor a cada iteração.

do: umBloco	executa as expressões do argumento para cada um dos elementos do receptor.
select: umBloco	seleciona os elementos do receptor que satisfazem uma condição expressa pelo argumento. Retorna uma nova coleção, semelhante à do receptor, contendo os elementos do receptor para os quais o argumento retorna o valor lógico *true*.
reject: umBloco	idem, para os elementos do receptor para os quais o argumento retorna o valor lógico *false*.
collect: umBloco	executa as expressões do bloco argumento para cada um dos elementos do receptor. Responde com uma nova coleção, semelhante à do receptor, contendo os objetos retornados pelo bloco a cada iteração.
detect: umBloco	responde com o primeiro elemento do receptor para o qual o bloco argumento retorna *true*. Caso não exista nenhum, responde uma mensagem de erro.

Exemplo de aplicação:

Seja um conjunto de alunos de uma escola, na forma de uma instância da classe Set, referenciada pela variável alunos. Suponha que a classe Aluno foi implementada, e que seu protocolo inclui as mensagens abaixo:

nota1	retorna a nota da primeira prova.
nota2	retorna a nota da segunda prova.
nome	retorna o nome do aluno.
turma	retorna a turma do aluno.

Suponha que serão aprovados os alunos que alcançarem média aritmética entre as duas provas maior ou igual a 5.0. Então:

a. Para obter o número de alunos aprovados:

```
|aprovados|
aprovados := 0.
alunos do: [:cada | cada nota1 + cada nota2 / 2 >= 5.0
                ifTrue: [aprovados := aprovados + 1]].
^aprovados
```

Note o uso da variável auxiliar cada, no bloco da mensagem do:. A cada iteração, a variável recebe o valor do próximo elemento do receptor.

b. Para obter um Set com apenas os nomes de todos os alunos:

```
|nomes|
nomes := alunos collect: [:cada | cada nome].
^nomes
```

c. Para obter esses nomes em ordem alfabética:

```
^nomes asSortedCollection.
```

d. Para obter um Set com os nomes dos alunos aprovados:

```
|nomesAprovados|
nomesAprovados:= (alunos select:
                    [:cada| cada nota1 +cada nota2 / 2 >= 5])
                    collect: [:cada | cada nome].
^nomesAprovados
```

Note que o Set alunos responde à mensagem select: acima com o sub-conjunto dos alunos aprovados. A mensagem collect: é então enviada a esse novo conjunto para extrair apenas os nomes desses alunos.

e. Para obter o número de alunos reprovados:

```
^ alunos size - aprovados size
"onde aprovados é o conjunto obtido no ítem a acima."
```

Alternativamente, pode-se fazer:

```
^ (alunos reject: [:a | a nota1 + a nota2 / 2 >= 5]) size
```

f. Para obter o número de alunos da turma A:

```
^ (alunos select: [:a| a turma = 'A']) size
```

ou, alternativamente,

```
|numA|
numA := 0.
alunos do: [:a| a turma = 'A'   ifTrue:[ numA := numA+1]].
^ numA
```

g. Para obter um conjunto com os nomes dos alunos reprovados:
Supondo que os ítens b e d acima foram executados:

```
^ nomes removeAll: nomesAprovados
```

4.2.2 Sub-classes de Collection.

As principais sub-classes de Collection são apresentadas nesta seção de forma sumária, através de alguns exemplos.

Set

Instâncias dessa classe representam coleções não-ordenadas de objetos, sem elementos duplicados.

Dictionary

Sub-classe de Set, ou seja, é uma especialização de Set. Cada elemento de um dicionário é uma instância da classe Association, que contém um par chave/valor, e

pode ser acessado pela chave. Não podem existir dois elementos com a mesma chave. Para checar igualdade usa-se o protocolo de =, e não de ==. As mensagens do protocolo geral de Collection aplicam-se aos valores de seus elementos. Mensagens adicionais permitem manipular com as chaves, como mostram os exemplos abaixo:

at: chave ifAbsent: umBloco
retorna o valor associado ao primeiro argumento, chave. Caso não seja encontrado nenhum elemento com chave igual ao argumento, executa o bloco (segundo argumento)

at: chave put: umObjeto
mensagem herdada do protocolo da classe Object. Associa o elemento de chave igual ao primeiro argumento, um valor igual ao segundo argumento.

Bag

Instâncias dessa classe representam coleções não ordenadas com duplicação permitida.

Interval

Coleções de números que representam uma progressão aritmética. Podem ser criados com a mensagem: **to: limite by: passo** enviada para um Numero.

Por exemplo, a mensagem: 100 to: 200 by: 20

cria um Intervalo contendo os números 100 120 140 160 180 200

Instâncias de Interval podem também ser criadas enviando-se à classe a mensagem: **from: inicio to: limite by: passo**

Por exemplo, a mensagem:: Interval from: 0 to: 10 by: 2.5

cria o intervalo formado pelos números 0 2.5 5 7.5 10

O protocolo geral de Collection aplica-se naturalmente a Interval, permitindo que iterações comuns em outras linguagens possam ser realizadas com auxílio de intervalos. Por exemplo:

```
1 to: 5 do: [:i| expressões diversas]
```

equivale ao seguinte texto em Pascal:

```
for i:= 1 to 10 do
    begin
    expressões diversas
    end;
```

IndexedCollection

(Essa classe denomina-se SequenceableCollection em Smalltalk-80, com pequenas diferenças)

Esta classe é também abstrata, e provê o protocolo comum a todas as coleções acessíveis externamente por índices inteiros. Seu protocolo inclui mensagens para determinar o primeiro e o último elementos, obter o índice de um elemento, copiar um trecho da coleção entre 2 índices, etc.

FixedSizeCollection

Sub-classe abstrata de IndexedColection. Todas as suas sub-classes tem em comum o número fixo de elementos, acessíveis por índices inteiros, ou seja não podem ser expandidos. Entre suas sub-classes estão:

Array

elementos de um Array podem ser quaisquer objetos.

String

elementos de uma String são caracteres, formando uma cadeia.

Symbol

subclasse de String. Os elementos de um Symbol são caracteres, e representam cadeias únicas em todo o sistema.

OrderedCollection

Instâncias dessa classe são coleções ordenadas que podem crescer e se contrair automaticamente, à medida que mais ou menos elementos são colocados ou removidos da estrutura. São usadas na construção de diversas estruturas de dados, como pilhas e filas. O protocolo das instâncias dessa classe inclui mensagens como:

addLast: umObjeto	inclui o argumento no final do receptor (última posição).
removeLast	remove o último elemento do receptor.
removeFirst	remove o primeiro elemento.
last	retorna o último elemento.
first	retorna o primeiro elemento.
add: x after: y	inclui o primeiro argumento (x) logo após o segundo (y).

SortedCollection

Sub-classe de OrderedCollection. Essas coleções são mantidas automaticamente ordenadas segundo um critério pré-fixado de ordenação de seus elementos. Portanto, mensagens como add:x after:y não são aceitas e produzem uma mensagem de erro.

O critério de ordenação dos elementos é estabelecido de um modo geral pela mensagem sortBlock: blocoBooleano, que pode ser enviada tanto para a classe, para criar uma nova instância, ou para uma instância, para modificar um critério anterior de ordenação. A mensagem deve ter o formato abaixo:

sortBlock: [:a :b | sequência de expressões]

Os elementos da coleção serão mantidos automaticamente ordenados de forma tal que, dados dois elementos quaisquer a e b da coleção, a última expressão da sequência do bloco retornará sempre *true.*

Por exemplo, a expressão abaixo cria uma SortedCollection S cujos elementos serão sempre mantidos ordenados de forma descendente:

```
S := SortedColection sortBlock: [:a   :b | a >= b]
```

Evidentemente, para que o critério acima possa ser aplicado, quaisquer dois objetos da coleção devem poder ser comparados por meio da relação **>=**. No caso, S poderia conter Strings, ou Numeros, por exemplo. Caso os objetos não sejam comparáveis diretamente, as expressões do bloco devem definir um critério em função de seus componentes, como no próximo exemplo.

Seja SCA uma SortedCollection de instâncias de uma classe Aluno, e sejam nome, da classe String, e idade, da classe Inteiro, componentes de Aluno, acessíveis respectivamente pelas mensagens *nome* e *idade.* A mensagem abaixo reordena, e passa a manter, seus elementos em ordem decrescente de idade, e em ordem crescente de nome dentro dos grupos de mesma idade:

```
SCA sortBlock: [:a :b |   (a   idade = b   idade and: [a nome <= b nome] )
                                                    or: [a   idade >   b idade] ]
```

A mensagem de classe **new** cria uma nova instância com o critério de ordenação padrão, que é manter os elementos em ordem crescente. Ou seja, a expressão:

```
SortedCollection new
```

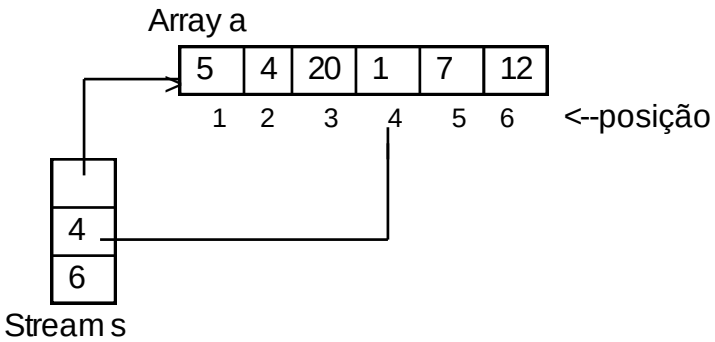
produz o mesmo efeito que: `SortedCollection sortBlock: [:a   :b| a <= b]`

4.3 Streams

Stream (literalmente "fluxo" ou "corrente") é uma classe que implementa o protocolo para realizar o acesso sequencial aos elementos de coleções indexadas, tais como Arrays, Strings e OrderedCollections.

Cada instância de Stream contém 3 componentes principais:
1 Uma referência à coleção que a Stream acessa.
2 Um marcador da última posição acessada na coleção.
3 Um contador com o número de elementos da coleção.

Na ilustração abaixo, é mostrada uma Stream s, que acessa um Array a de 6 elementos. O marcador de posição indica que a última posição acessada foi a de número 4:



Streams são úteis para implementar algorítmos que percorrem coleções sequenciais. A classe Stream é, na realidade, uma classe abstrata, pois são as suas sub-classes, abaixo descritas, que possuem instâncias diretamente:.

1 ReadStream - objetos desta classe são usados para acessar coleções de objetos exclusivamente para fins de leitura.

2 WriteStream - objetos desta classe são usados para acessar coleções de objetos exclusivamente para fins de gravação ou saída. O protocolo de WriteStream permite que as coleções acessadas sejam automaticamente expandidas para acomodar mais elementos. Para isso, cada instância mantém, além dos tres componentes básicos, um marcador da posição de maior ordem já gravada na coleção que ela acessa. O protocolo de WriteStream não aceita acesso para leitura.

3 ReadWriteStream - sub-classe de WriteStream, incorpora ambos os protocolos de ReadStream e de WriteStream.

4 FileStream - sub-classe de ReadWriteStream. Objetos desta classe são usados para acessar arquivos externos vistos como uma longa sequência de bytes ou caracteres. O protocolo é modificado em relação aos anteriores, pois não se trata mais de acessar objetos genéricos. O primeiro componente, isto é, a coleção acessada, é nesse caso uma referência ao "buffer" de página do arquivo, que contém uma String. Outros componentes são acrescentados para armazenar dados específicos.

5 TerminalStream - sub-classe de ReadWriteStream, usada em Smalltalk/V, para definir o protocolo de leitura e escrita em um terminal.

Utilização de Streams: Streams são usadas em todos os métodos que implelmentam acessos de leitura e/ou gravação em qualquer coleção indexável de objetos. Uma mesma coleção pode ter várias Streams a ela associadas. Streams podem ser criadas através da mensagem abaixo enviada à sub-classe desejada:

```
on: umaColeçãoIndexada
```

Por exemplo, uma ReadStream s como a da figura acima poderia ser criada executando a expressão abaixo:

```
s := ReadStream on: #(5  4  20  1  7  12)
```

O marcador de posição, neste caso, é inicializado com valor zero.

4.4 Classes gráficas.

As classes gráficas permitem manipular objetos visuais, e a construção de interfaces gráficas. Essas classes variam muito de acordo com a versão de Smalltalk usada. A versão Smalltalk/V Windows, por exemplo, faz uso intensivo dos objetos visuais da interface do Windows. A versão DOS já possui classes diferentes para manipular com janelas e bitmaps. Smalltalk-80 apresenta por sua vez um conjunto bastante diferenciado de classes gráficas.

Como o seu uso é razoavelmente sofisticado, e este é um texto introdutório, as classes gráficas não serão discutidas em detalhe.

Imagens na tela são criadas e manipuladas por meio de mensagens a instâncias das classes gráficas de Smalltalk. Os objetos básicos utilizados para a formação de imagens são Pontos e Retângulos.

Para ilustrar essas diferenças, as hierarquias abaixo apresentam as classes gráficas dos ambientes Smalltalk/V para DOS(à esquerda) , e Smalltalk/V para Windows:

```
Object
   BitBlt
         Pen
            Commander
   Collection
         FixedSizeCollection
            Bitmap
   DisplayObject
         DisplayMedium
            Form
```

```
Object
   Font
   GraphicsMedium
         Bitmap
         Printer
         Screen
         StoredPicture
   GraphicsTool
         TextTool
            Pen
               RecordingPen
                  Commander
    Icon
```

4.5 A Interface Gráfica

Smalltalk é uma linguagem projetada para uso interativo em estações de trabalho individuais. Devido à grande quantidade de classes e mensagens disponíveis, é utilizada uma interface gráfica que facilita a consulta e edição das classes e métodos. Esta interface antecipou-se em muitos anos ao estilo atualmente difundido pelas interfaces do MacIntosh e do Windows, que utiliza um sistema de janelas múltiplas na tela, que podem apresentar simultaneamente informações diversas. As janelas podem ser superpostas, sem perda de informação, e o operador pode selecionar qual a janela ativa, que fica sobre as demais, e na qual informações podem ser digitadas. A manipulação da interface é feita por meio de um dispositivo de controle do cursor ("mouse") juntamente com o teclado normal do terminal.

Existem vários tipos de janelas com funcionalidades diferentes. Janelas podem ser sub-divididas em "divisórias" ("panes", em ingles). A cada divisória corresponde um "menu" de funções que pode ser ativado pelo "mouse" (ou teclado), e cada função pode ser selecionada. O uso combinado do cursor na tela, dos menus de função e da operação do "mouse", permite que uma parcela significativa da interação com o sistema seja feita sem necessidade de digitação de instruções e sem necessidade de memorizar a sintaxe dos comandos, o que aumenta a velocidade da interação.

Os tipos principais de janelas são:

Área de Trabalho: ("Workspace") - são usadas para editar textos. O menu de funções inclui todo os comandos de edição, e permite executar diretamente trechos selecionados de métodos editados, com a resposta sendo enviada para o mesmo local.

Folheador: ("Browser) - é um tipo especial de janela utilizada para a consulta e edição de informações contidas em dicionários. Algumas divisórias são definidas como "divisórias de lista" ("list panes") onde são apresentadas as chaves de consulta que podem ser roladas para cima ou para baixo, até localizar a chave desejada. Ao

selecionar uma chave, a informação associada é apresentada em outra divisória própria para a edição de textos ("text pane"). Folheadores especializados são usados para editar os métodos das diversas classes do sistema. Dois folheadores principais da interface são o Folheador de Classe ("Class Browser") e o Folheador da Hierarquia das Classes ("Class Hierarchy Browser") O primeiro é usado para consultar e editar os métodos de uma determinada classe. O segundo é usado para consultar e editar todas as classes do sistema. Outro folheador importante é o Folheador de Disco ("Disk Browswer") que permite consultar o conteúdo do disco do sistema e realizar diversas funções de manutenção de arquivos.

"Prompter" - não há uma tradução ideal para esse termo, que significa "solicitador de resposta". É usada para solicitar do usuário alguma resposta específica, podendo já apresentar o valor padrão de resposta dentro de uma divisória de texto. O usuário pode alterar ou não a resposta padrão e enviar sua resposta.

A interface gráfica de Smalltalk é bastante sofisticada, e varia também com a versão e fabricante, não cabendo discutí-la aqui neste texto, que se propõe a ser apenas uma introdução à linguagem. O leitor interessado dever consultar o manual da versão de interesse.

CAPÍTULO 5 - CONTRIBUIÇÕES PARA A ENGENHARIA DE SOFTWARE

Neste capítulo serão examinadas as contribuições que o paradigma da programação orientada para objetos pode trazer para a solução de alguns problemas da Engenharia de Software, mostrando que muitos desses problemas podem advir do uso de modelos inadequados de programação. Esas considerações foram apresentadas anteriormente em [Jonathan 87].

5.1 Re-utilização de software.

Smalltalk (e outras LOO's) estimula de forma natural a reutilização de software. Sendo um ambiente integrado de programação, todos os recursos do sistema estão disponíveis de maneira uniforme para qualquer aplicação. A partir do momento em que uma classe é definida e implementada, qualquer aplicação pode utilizá-la diretamente, ou definir subclasses específicas para a aplicação, com custo marginal. Por exemplo, basta definir uma vez uma classe *ArvoreBinaria*, e qualquer aplicação pode gerar quantas instâncias dessa classe forem necessárias, e utilizar suas propriedades, sem necessidade de escrever uma única linha de código.

Essa facilidade permite abordar um projeto de software de forma semelhante à construção de sistemas físicos, isto é, como um objeto complexo constituído pela reunião de diversos objetos pré-fabricados de uso geral, cada qual com estrutura e funcionalidade bem definidas.

5.2 Confiabilidade, Custos e Manutenibilidade.

Como consequência, os custos de produção de cada classe são divididos entre todas as aplicações que dela se utilizam, com redução considerável do custo por aplicação. Por outro lado, aplicações podem ser implementadas em menor prazo devido à redução do esforço de especificação e programação, e os produtos resultantes apresentam um maior grau de confiabilidade, já que utilizam classes previamente testadas. Por fim, a manutenção, tanto corretiva como evolutiva, fica grandemente simplificada como resultado do uso de componentes padronizados, e pela total modularidade da arquitetura do sistema.

5.3 Facilidades para construção de protótipos rápidos.

Embora as implementações de Smalltalk e outras LOO's ainda não sejam eficientes o bastante para competir com as linguagens convencionais, são por outro lado potencialmente úteis para a construção de protótipos rápidos, para uso na depuração das especificações de aplicações. Sistemas funcionalmente corretos podem ser implementados rapidamente em Smalltalk e colocados à disposição dos usuários para utilização experimental. Modificações nas especificações podem ser facilmente implementadas e testadas realisticamente em muitos casos. Dessa forma torna-se possível convergir mais rapidamente para um produto idealmente próximo das expectativas do usuário

Referências

[Birtwistle et al.73] Birtwistle, G et al., *Simula Begin*, Auerbach, Philadelphia, 1973.

[Borland92] Borland International, Inc, *Borland Pascal with Objects - version 7.0 - User's Guide,* Scotts Valley, CA, 1992.

[Carvalho 93] Carvalho, S E R ,*The Object and Event Oriented LanguageTOOL*, Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 6/93.

[Cox86] Cox, B J, *Object-Oriented Programming*, Addison-Wesley, Reading, MA, 1986.

[Dahl66] Dahl, O J e K. Nygaard, *Simula - an Algol-based Simulation Language*, Comm.ACM 9 (9), Sept. 66, pp.671-678.`

[Digitalk] Digitalk, Inc. *Smalltalk/V 286 Tutorial and Programming Handbook* e *Smalltalk /V Windows Tutorial and Programming Handbook*, Digitalk, Los Angeles, 1988 e 1992.

[Goldberg83] Goldberg, Adele e D Robson, *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

[Ingalls 86] Ingalls D H H, *A Simple Technique for Handling Multiple Polymorphism*, ACM SIGPLAN Notices, 21 (11), Nov. 1986, pp. 347-49.`

[Jonathan 87] Jonathan, M, *Orientação para Objetos em Smalltalk - 80: uma abordagem eficaz para a construção de sistemas de software*, Anais do I Simpósio Brasileiro de Engenharia de Software, Petrópolis, RJ, out. 1967.

[LaLonde90] LaLonde, W R e J R Pugh, *Inside Smalltalk, vol I e II*, Prentice-Hall, New Jersey, 1990.

[Liskov e Ziller 74] Liskov, B e S. Ziller, *Programming with Abstract Data Types*, SIGPLAN Notices, April 1974, pp. 50-59.

[Meyer88] Meyer, B, *Object-Oriented Software Construction*, Prentice-Hall International (UK), Cambridge, 1988.

[Pohl91] Pohl, I , *C++ para Programadores de Pascal*, Berkeley Brasil Editora, Rio de Janeiro, 1991.

[Stroustrup 86] Stroustrup, B, *The C++ Programming Language*, Addison-Wesley, 1986.

[TAK90] Takahashi, T e H K E Liesenberg, *Programação Orientada a Objetos*, VII Escola de Computação, São Paulo, 1990.

[TOOL94] *TOOL - Manuais de Referência*, SPA - Sistemas, Planejamento e Análise Ltda, Rio de Janeiro, 1994.