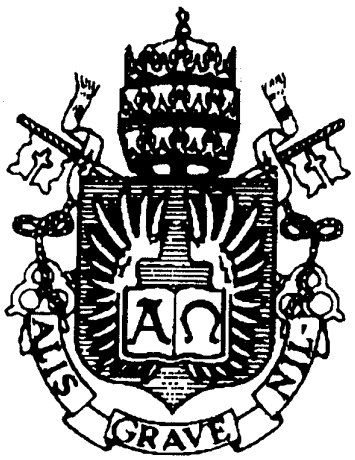

PUC

Monografias em Ciência da Computação
nº 26/92

X-NET: uma Linguagem, Três Paradigmas

Marcos A. R. Dantas
Mário Gheiner
Alberto V. Sarmento
Sérgio E. R. Carvalho

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22454-970
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, Nº 26/92
Editor: Carlos J. P. Lucena Julho, 1992

X-NET: uma Linguagem, Três Paradigmas

Marcos A. R. Dantas
Mário Gheiner
Alberto V. Sarmento
Sérgio E. R. Carvalho

* Este trabalho foi patrocinado pela Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

X-NET: Uma Linguagem, Tres Paradigmas

Marco A. R. Dantas
Mario Gheiner
Alberto V. Sarmento
Sergio E. R. Carvalho
(PUC Rio-BR, Depto. Informática)

PUCRioInf-MCC26/92

Abstract: X-NET is an object oriented programming language covering also the modular (conventional) and distributed programming paradigms. This report summarizes the main language features designed to support those three paradigms. In particular, the programming units program, module, model, class and script are described.

Resumo: X_NET é uma linguagem de programação orientada a objetos que cobre tambem os paradigmas de programação modular e distribuida em rede. Este relatório apresenta, de maneira sucinta, os aspectos principais de X_NET para o atendimento aos tres paradigmas. Em particular são apresentadas as unidades program, module, model, class e script.

Palavras-chave: orientação a objetos, programação distribuida, programação modular.

X-NET : Uma Linguagem, Três Paradigmas

Alberto Sarmento, Marco A. R. Dantas, Mario Gheiner, Sergio Carvalho

1 - Introdução

A rápida evolução do hardware é um forte incentivo para a busca de soluções de software que permitam e facilitem o uso efetivo das novas conquistas tecnológicas. X-Net representa mais um passo nesta direção.

Esta linguagem pretende atender a três paradigmas de programação: orientada a procedimentos, orientada a objetos e distribuída em rede. Ela foi projetada para permitir aos programadores uma evolução gradual através dos paradigmas atendidos.

Programadores experientes em programação modular, ao migrarem para esta linguagem, identificar-se-ão de imediato com o paradigma orientado a procedimentos, onde encontrarão as ferramentas com as quais está habituado. Aos poucos, poderá incursionar no paradigma da orientação a objetos e, finalmente, estender seus conhecimentos de forma que possa usufruir dos recursos de programação distribuída presentes na linguagem.

Este relatório contém uma rápida introdução à linguagem X-Net, cobrindo os aspectos mais importantes da mesma.

2 - Programação Orientada a Procedimentos

A programação procedural é suportada na linguagem pelas unidades sintáticas **program** e **module**. A principal unidade sintática da linguagem é um **program**. Toda aplicação escrita em X-Net possui uma e somente uma unidade programa, que marca o início da execução da aplicação.

A unidade **module** é a facilidade de modularização e encapsulamento oferecida pela linguagem dentro do paradigma de orientação a procedimentos. Há também usos muito interessantes desta unidade dentro de outros paradigmas, como, por exemplo, a reunião de classes utilizadas por uma aplicação.

Construções típicas de linguagens tais como Pascal, C ou Módula 2, estão presentes em X-Net, de forma que programadores egressos destas linguagens e suas correlatas se sentirão inteiramente confortáveis para desenvolver aplicações em X-Net. Isto suaviza a migração para a linguagem.

3 - Programação Orientada a Objetos

Esta secção mostra como as características fundamentais da orientação a objetos aparecem em X-Net_: herança, polimorfismo, genericidade, etc. Mais detalhes sobre estes mecanismos são descritos no apêndice A.

3.1 - Classes e Objetos

Classes em X-Net são descritores de tipos abstratos de dados acrescidos de características típicas das linguagens orientadas a objetos tais como polimorfismo, herança, etc.

As instâncias de classes, conhecidas como *objetos*, são implementadas dentro do mesmo espaço de endereçamento de seus usuários, estando sujeitas às regras de escopo convencionais. A área de armazenamento dos estados dos objetos é sempre alocada em heap, o que significa que objetos são implementados como ponteiros implícitos.

A comunicação entre objetos é realizada por chamadas de procedimento dentro do mesmo espaço de endereçamento do processo onde o serviço foi solicitado, embora tenha a semântica de troca de mensagens.

3.2 - Herança

Herança é um mecanismo corrente em linguagens orientadas a objetos que permite a definição incremental de tipos, facilitando a reutilização de código. O emprego deste mecanismo permite a extensão e/ou especialização (herança simples) e a combinação de classes (herança múltipla). Embora X-Net preveja o uso de ambas as formas de herança na sintaxe, a versão corrente contempla apenas a herança simples.

A linguagem permite ainda *renomear* e/ou *redefinir* identificadores. Renomear significa criar um sinônimo (**alias**) para um identificador, enquanto que redefinir significa alterar sua implementação. A renomeação é útil para resolver possíveis conflitos de nomes em métodos, operadores e atributos devido à herança múltipla. A redefinição é mandatória para os identificadores de métodos e operadores cujas implementações se encontrem redefinidas em uma subclasse.

Ao se utilizar do mecanismo de herança, é possível redefinir apenas métodos e operadores. A renomeação, entretanto, é possível para todos os identificadores herdados.

3.3 - Polimorfismo e Aliasing

X-Net admite *polimorfismo* direto e reverso (no sentido de classe para subclasse e vice-versa). Polimorfismo direto significa que um identificador associado a uma classe A pode fazer referência a instâncias de qualquer uma das sub-classes de A. O reverso é mais complicado: um identificador associado a uma sub-classe A pode referenciar um identificador associado a uma de suas super-classes apenas se o objeto corrente for, de fato, modelado pela classe A ou por alguma de suas sub-classes.

Uma vez que o conceito usual de atribuição de objetos não é rico o bastante para expressar o polimorfismo, X-Net opta por usar o conceito de sinônimo (também **alias**, expresso pelo

operador "<-"), no qual o objeto à esquerda do operador recebe uma cópia do descritor do objeto gerado (e possivelmente operado) na expressão à direita do operador.

O conceito de sinônimo é estendido linear e uniformemente à passagem de argumentos para métodos em X-Net.

3.4 - Tipos

X-Net pode ser considerada uma linguagem fortemente tipada. Embora o polimorfismo reverso fira o conceito de tipagem forte (pois parte da verificação de tipos precisa ser delegada à execução), todas as chamadas de métodos necessitam ser resolvidas, de forma direta ou indireta, em tempo de compilação.

A equivalência de tipos é decidida por nome em X-Net. Os tipos anônimos são considerados diferentes entre si.

3.5 - Genericidade

Uma classe de X-Net pode ser *genérica*, ou, em outras palavras, pode ser um construtor de classes.

Uma classes genérica recebe parâmetros que são, por sua vez, classes. Dentro da classe genérica, podem ser declaradas instâncias das classes parametrizadas, e o conjunto de métodos e operadores que são comuns a todas as classes podem ser aplicados a estas instâncias: **Destroy**, **DeepClone**, **ShallowClone**, sinônimo ("<-"), igualdade ("=="), e desigualdade ("=/=").

Outras operações envolvendo instâncias dos tipos parametrizados podem ser feitas mas, para tal, os operadores necessários precisam ser especificados na cláusula de requisição de operadores da classe genérica e, naturalmente, ser declarados, implementados e exportados pelas classes que servirem de parâmetros correntes.

Uma unidade sintática usuária de uma classe genérica precisa especificar apenas o identificador da mesma. A explicitação das classes que serão os parâmetros atuais é adiada até o uso efetivo da classe genérica. Sub-classes que herdam de classes genéricas podem manter a genericidade e deixar alguns (ou todos) os parâmetros ainda por especificar.

3.6 - Métodos

Métodos são os serviços oferecidos pelas classes. Métodos cujos identificadores não constarem na cláusula de exportação de uma classe são privativos dela e são visíveis apenas no seu escopo e no escopo de suas sub-classes.

Os métodos possuem, opcionalmente, argumentos de entrada e/ou saída e podem retornar valores para o chamador. Métodos possuem um corpo de comandos e opcionalmente declarações locais e tratadores de exceções. Identificadores declarados localmente possuem visibilidade restrita ao próprio método.

Existem métodos pré-definidos, presentes em todas as classes, para definir o ciclo de vida de instâncias (**Create** e **Destroy**), e clonear instâncias (**ShallowClone** e **DeepClone**). O método **Create** deve ser aplicado a classes (é um caso especial), ao contrário de qualquer outro, que deve ser aplicado a instâncias.

Métodos podem ser declarados com os qualificadores (palavras reservadas) **deferred**, **virtual** ou **unique**. O qualificador **deferred** significa que a semântica do método não se encontra implementada na classe onde foi definido. A classe que contém um método **deferred** precisa ser declarada como **abstract**. Este método será obrigatoriamente implementado nas subclasses que possam ser instanciadas.

O qualificador **virtual** implica que o método poderá vir a ser redefinido nas subclasses. Esta é a qualificação "default". A *assinatura* de um método (tipo, nome, número e ordem

dos argumentos, se houver, e tipo do resultado, se houver) definido em uma classe não pode ser alterada nas suas subclasses, embora o método possa ser renomeado.

Finalmente, o qualificador **unique** indica que o método em questão não poderá sofrer redefinição em nenhuma subclasse (isto não impede a renomeação do método). O uso do mesmo identificador em uma subclasse é inteiramente dissociado do seu significado na superclasse.

Em uma determinada classe, um método que seja redefinido nas cláusulas de herança ou de importação não poderá ser qualificado como **unique**. Uma classe que possui um método **deferred** só permite a redefinição deste método em seus decendentes por um método **virtual**.

Os argumentos e formas de passagem são descritos mais adiante.

3.7 - Operadores

X-Net permite admite também a criação de *operadores* unários ou binários. Operadores (notação pré-fixada ou infixada, conforme tipo do operador) funcionam como alternativa à forma de solicitação de serviços de uma classe, e podem ser identificados tanto por símbolos quanto por palavras. A classe do resultado da operação deve ser sempre aquela onde o operador foi declarado mas os operandos podem ser de qualquer tipo.

Operadores nativos da linguagem ou criados pelo usuário podem ser sobrecarregados, desde que não gerem ambigüidades para a máquina de expressões da linguagens. A renomeação sempre está disponível para remover ambigüidades.

4 - Programação Concorrente

O paradima da orientação a objetos foi estendido na linguagem X-Net, de forma a suportar também a programação concorrente. Para tanto, foram introduzidos os conceitos de *scripts* e *atores*.

Scripts e atores correspondem, respectivamente, à classes e objetos: um ator é uma instância de um script. Cada ator constitui-se num processo com espaço de endereçamento próprio. Sua comunicação com os demais processos se faz através da troca de mensagens interprocessos com suporte para comunicação síncrona e assíncrona. Em ambientes distribuídos os atores podem ser instanciados em nodos distintos da rede.

Atores são uma forma mais completa de implementação de objetos (do ponto de vista teórico), pois o comportamento está, de fato, associado ao mesmo. Como consequência imediata, atores podem ser passados como argumentos de serviços mas objetos não podem, porque o processo servidor não tem a "obrigação" de conhecer o seu comportamento (devido ao polimorfismo, o parâmetro formal pode ser declarado como uma super-classe do parâmetro atual).

Há dois tipos de atores : os dirigidos por uma aplicação (atores dependentes) e os servidores (prestadores de serviços a diversas aplicações). Enquanto os primeiros são destruídos quando do encerramento da aplicação à qual estão associados, os servidores sobrevivem.

5 - Modelos : A União de Dois Paradigmas

À primeira vista, o que difere sintaticamente uma classe de um script é o nome da unidade sintática. Na verdade, existem outras diferenças, como por exemplo, a forma de passagem e tipo de argumentos de métodos, assincronismo no envio de mensagens, etc. Para os códigos isentos destas particularidades, X-Net provê a unidade **model**, a qual possui os mesmos componentes sintáticos de **class** e **script**.

Quando um modelo é compilado, é gerado código para ambas as especializações. Para utilizar um modelo (declarar instâncias), é necessário especializá-lo antes. Isto pode ser feito na cláusula de uso (**uses**), nas cláusulas de especialização (**class** e **script**, que são semelhantes à cláusula **type**) ou adiada até a definição das variáveis. É importante observar que uma classe ou um script

definem um tipo, mas um modelo só define um tipo quando é especializado (a palavra *tipo* aqui designa o conjunto de informações necessárias ao compilador para alocação de uma instância e verificação de correção de expressões).

6 - Tratamento de Exceções

O mecanismo de *exceções* está presente em todas as unidades sintáticas da linguagem X-Net. O seu tratamento pode ser feito no escopo do procedimento, da unidade sintática ou delegado ao cliente.

Existem identificadores pré-definidos na linguagem que são usados para tratamento de exceções de hardware, que são levantadas pelo mecanismo de interrupções.

Identificadores para tratamento de exceções por software podem ser definidos pelos projetista das unidades sintáticas. Estas interrupções são levantadas pelo comando **raise**, como no seguinte exemplo:

if <expressão-booleana>
    raise <identificador-tratamento-exceção>

Dizemos que um identificador associado à uma exceção está declarado se este está especificado na cláusula de exceção, embora não possua código para tratamento da exceção. Um identificador está definido se está declarado e possui código para tratamento da exceção.

O tratamento de exceções pode ser delegado às unidades importadoras de um modelo ou módulo onde foi declarado. Isto permite que o projetista especifique o tratamento desejado para uma determinada exceção.

Nos módulos, os tratadores de exceção que não se encontram definidos, tem que ser delegados na sua interface. Nos modelos os tratadores delegados são especificados na cláusula de delegação de exceções.

Quando é levantada uma exceção, se o identificador em causa estiver definido no escopo do método, operador ou função onde ocorreu a exceção, ou ainda na cláusula de exceção global da unidade sintática, a execução será desviada para o código associado a este identificador, desaparecendo assim a causa da exceção. Normalmente, ao final do tratamento, a execução retorna para o comando seguinte ao ponto onde foi levantada a exceção.

Se o identificador for delegado para tratamento de exceção em outra unidade sintática, a execução do método, operador ou função em curso é desviada para o código associado ao identificador no cliente.

O código de tratamento de uma exceção pode levantar uma nova exceção.

7 - Coleções e Persistência

X-Net permite criar *coleções de instâncias* através do construtor **collection**. Coleções podem ou não ser polimórficas. Coleções polimórficas podem armazenar instâncias de uma determinada classe ou de seus descendentes. Estes descendentes devem estar registrados na coleção.

As coleções podem ser uma boa ferramenta para o mapeamento de metodologias tais como entidade-relacionamento para modelagem de dados, e suas extensões ou modelagens orientadas a objetos.

A abordagem do tema foi feita em X-Net com o intuito de fornecer facilidades ao construtor de aplicações, utilizando conceitos que, por fazerem parte do senso comum, certamente já serão parte do repertório do mesmo.

A linguagem fornece formas para utilizar uma coleção, inserir, obter e remover componentes, iteradores e indexação de componentes, etc.

Todo este aparato pode ser montado apenas em memória. Entretanto, a associação da palavra reservada **persistent** a uma coleção instrui o compilador a gerar uma correspondência em disco (acrescentando-se os nomes físicos dos arquivos envolvidos, obviamente).

Se uma classe C for colecionada, todos os seus estados que são objetos não terão seus próprios estados colecionados junto com a classe C. Deverão ter sua própria coleção. Esta regra fomenta o uso de coleções persistentes em aplicações de produção mas não permite a construção de coleções cujos estados sejam armazenados fisicamente no mesmo arquivo.

Tal simplicidade só foi possível pela flexibilização consciente de algumas regras:

1. Quando se está definindo uma coleção de C,D,E,F (onde D, E e F fazem parte da hierarquia de C), obtém-se acesso a todos os estados de C, D, E e F que são objetos e é obrigatória a determinação das instâncias de coleções onde se encontram os objetos referidos. Esta regra contradiz diretamente o encapsulamento.

2. Coleções de tipos não-classe só são permitidas se o tipo envolvido não contiver objetos. Ponteiros (que só fazem sentido em memória) são gravados como lixo se as coleções são feitas persistentes. Isto conflita com a ortogonalidade da linguagem.

Em suma, concessões foram feitas, em nome da simplicidade de uso pelo construtor de software.

8 - Argumentos de Funções, de Métodos e de Operadores

Argumentos são conjuntos de identificadores associados a tipos e podem ser passados de quatro formas distintas:

. **in** (valor)
. **out** (resultado)
. **inout** (valor-resultado)
. **ref** (referência)

São feitas as seguintes observações sobre argumentos e parâmetros:

1. Objetos e atores só podem ser passados como argumento de uma única forma, a qual tem a semática de um alias (operador <-). Portanto, se o argumento for um objeto ou ator, constitui erro especificar a forma de passagem.

2. Objetos não podem ser argumentos de serviços fornecidos por atores, porque, devido ao polimorfismo, não há forma de garantir que o código necessário para operar com os estados do objeto esteja no ator que fornece o serviço. Esta restrição não se aplica a atores e estes podem ser passados como argumentos livremente.

3. Argumentos dos demais tipos podem ser passados das quatro formas já mencionadas, que tem a seguinte semântica:

in	cópia de valor para o servidor *antes* da execução do serviço (valor)
out	cópia de valor para o cliente *depois* da execução do serviço (resultado)
inout	combinação de **in** e **out** (valor-resultado)
ref	endereço do argumento (referência)

4. Métodos oferecidos como serviços de atores não admitem a forma **ref** (os endereços só fazem sentido no mesmo processo).

5. Constitui erro passar ponteiros explícitos, ou arrays ou records que contenham ponteiros explícitos, ou objetos como argumentos de serviço fornecido por um ator, mas este erro não é detectado.

9 - Interface com o Usuário

X-Net possui uma hierarquia de modelos que facilitam a construção de interfaces. Esta hierarquia baseia-se na biblioteca de rotinas do software X-Windows.

10 - Portatilidade da Linguagem

O compilador (escrito em C) gera código em C com o intuito de facilitar a portatilidade do produto. A primeira versão do compilador será implementada para o ambiente UNIX V utilizando TCP/IP para suportar a parte de concorrência.

Apêndice A: Resumo das Unidades Sintáticas da Linguagem

A linguagem X-Net possui três unidades sintáticas de encapsulamento:

A.1 - Programa

É a unidade principal de uma aplicação, onde o processamento é iniciado. Toda aplicação tem uma e somente uma unidade programa. A unidade possui a seguinte estrutura:

programa -> **program** <identificador-programa>
[[<lista-argumento-formal-programa>]] ;
[<cláusula-importação>]
{
[<cláusula-definição-constante>]
[<cláusula-definição-tipo>]
[<cláusula-definição-instância>]
[<cláusula-definição-função>]
...}*
<bloco>
[<cláusula-definição-exceção>]
end [**program**] .

* A especificação de um programa permite a declaração de parâmetros formais, condicionada às normas do sistema operacional sobre o qual a linguagem está implementada;

* A cláusula de importação disponibiliza outras unidades sintáticas. Somente os identificadores exportados são visíveis. Esta regra é quebrada nas coleções;

* A cláusula de definição de tipos permite especilizar modelos em scripts e classes e criar estruturas de dados. Objetos e atores podem ser membros destas estruturas;

* A cláusula de definição de constantes permite definir constantes;

* A cláusula de definição de instâncias permite declarar identificadores de variáveis, objetos e atores que estarão associados respectivamente a tipos, classes e scripts;

* Funções podem ser declaradas e implementadas na unidade programa;

* O início de execução de uma aplicação começa pelos comandos existentes no comando composto do programa;

* A unidade programa pode fazer tratamento de exceções.

A.2 - Módulos

São unidades sintáticas usadas para dominar a complexidade de projeto, quer seja como elemento de segmentação, quer seja como elemento de encapsulamento. Os módulos oferecem construções lingüísticas adequadas para a programação orientada a procedimentos.

Sua estrutura apresenta o seguinte formato:

módulo -> **module** <identificador-módulo> ;
[<interface-módulo>]
[<implementação-módulo>]
[<inicialização-módulo>]
end [**module**] .

interface-módulo -> **interface**
[<cláusula-importação>]
{
[<cláusula-definição-constante>]
[<cláusula-definição-tipo>]
[<cláusula-definição-instância>]
[<cláusula-especificação-função>]
...}*
[<cláusula-delegação>]

implementação-modulo -> **implementation**

[<cláusula-importação>]

{

[<cláusula-definição-constante>]

[<cláusula-definição-tipo>]

[<cláusula-definição-instância>]

[<cláusula-definição-função>]

...}*

[<cláusula-definição-exceção>]

inicialização-módulo -> **initialization**

<bloco>

* Cada módulo pode ser compilado separadamente e ser utilizado em qualquer outra unidade (inclusive outros módulos);

* Módulos são constituidos de três partes: **interface**, **implementation** e **initialization**;

* Módulos possuem identificadores que são usados para referenciá-los dentro de outras unidades sintáticas;

* Módulos podem importar outras unidades sintáticas, especializar modelos em classes ou scripts, definir tipos, declarar constantes, instâncias e funções. Estas declarações podem aparecer tanto na interface como na implementação;

* Os identificadores declarados na interface são visíveis pelas unidades sintáticas que importam este módulo, enquanto que aqueles declarados na implementação são privados, não podendo ser acessados pelos módulos usuários;

* Funções tem seus códigos especificados na implementação;

* A inicialização contém código que é executado antes do início da unidade usuária;

* O mecanismo de tratamento de exceções também está presente nos módulos.

A.3 - Modelos

São unidades sintáticas usadas para representar tipos abstratos de dados. Cada modelo pode representar uma classe e um script. Os conceitos de herança, polimorfismo, encapsulamento, troca de mensagens e genericidade (modelos genéricos) estão presentes nos modelos. Modelos oferecem suporte para o paradigma da orientação a objetos e também para o processamento distribuído.

Para que um modêlo possa ser instanciado é preciso especializá-lo em classe ou script. A etapa de especialização pode ser suprimida se o modêlo for declarado diretamente como classe ou como script (às vezes isto é necessário, pois a classe/script que está sendo definida pode possuir particularidades que não permitam a generalidade requerida por um modêlo).

Modelos possuem a seguinte estrutura:

modelo -> [**abstract**]
(script | class | model)
<identificador-modelo>
[[<lista-parâmetro-formal-modelo>]];
[<cláusula-herança>]
[<cláusula-importação>]
[<cláusula-exportação>]
[<cláusula-requisição>]

[<cláusula-delegação>]

[<cláusula-estado-comportamento>]

[<cláusula-definição-exceção>]

end [**model**].

cláusula-estado-comportamento ->

{ <cláusula-definição-constante> |

<cláusula-definição-tipo> |

<cláusula-definição-instância> |

<cláusula-definição-método> |

<cláusula-definição-operador> ...}*

* Modelos podem ser importados por outras unidades sintáticas. Uma vez importados, classes e scripts podem ser definidos a partir destes, conforme as necessidades de projeto;

* Modelos qualificados como **abstract** não podem ser instanciados;

* Modelos podem ser escritos diretamente como classe ou script e, neste caso, não há necessidade de especialização;

* Modelos podem ser genéricos, bastando para isto que se definam parâmetros formais;

* A extensão e/ou especialização de modelos pode ser conseguida através do mecanismo de herança. A linguagem suporta herança simples e mùltipla;

* A cláusula de importação torna visível os identificadores exportados por outras unidades sintáticas;

* A genericidade em um modelo pode requerer operadores que não podem ser definidos a *priori* pois o parâmetro genérico ainda não é conhecido no momento de sua compilação. As assinaturas desses operadores são especificadas na cláusula de requisição de operadores;

* A cláusula de exportação especifica todos os identificadores de um modelo que são exportados;

* As cláusulas de definição de classes, scripts e tipos são semelhantes aos seus correspondentes nas demais unidades sintáticas;

* A cláusula de definição de estados define os atributos que compõem um modelo. Estes atributos podem ser constantes ou não;

* A cláusula de definição de comportamento especifica a semântica e implementação dos métodos e operadores do modelo.